R: A Language and Environment for Statistical Computing

Reference Index

The R Development Core Team

Version 2.4.1 (2006-12-18)

ISBN 3-900051-07-0

Contents

Part I

Chapter 1

The `base` package

`base-package` *The R Base Package*

Description

Base R functions

Details

This package contains the basic functions which let R function as a language: arithmetic, input/output, basic programming support, etc. Its contents are available through inheritance from any environment.

For a complete list of functions, use `library(help="base")`.

`.Device` *Lists of Open Graphics Devices*

Description

A list of the names of the open graphics devices is stored in `.Devices`. The name of the active device is stored in `.Device`.

`.Machine` *Numerical Characteristics of the Machine*

Description

`.Machine` is a variable holding information on the numerical characteristics of the machine R is running on, such as the largest double or integer and the machine's precision.

Usage

```
.Machine
```

Details

The algorithm is based on Cody's (1988) subroutine MACHAR.

Note that on most platforms smaller positive values than `.Machine$double.xmin` can occur. On a typical R platform the smallest positive double is about `5e-324`.

Value

A list with components (for simplicity, the prefix "double" is omitted in the explanations)

`double.eps` the smallest positive floating-point number `x` such that `1 + x != 1`. It equals `base^ulp.digits` if either `base` is 2 or `rounding` is 0; otherwise, it is `(base^ulp.digits) / 2`.

`double.neg.eps`
a small positive floating-point number `x` such that `1 - x != 1`. It equals `base^neg.ulp.digits` if `base` is 2 or `round` is 0; otherwise, it is `(base^neg.ulp.digits) / 2`. As `neg.ulp.digits` is bounded below by `-(digits + 3)`, `neg.eps` may not be the smallest number that can alter 1 by subtraction.

`double.xmin` the smallest non-vanishing normalized floating-point power of the radix, i.e., `base^min.exp`.

`double.xmax` the largest finite floating-point number. Typically, it is equal to `(1 - neg.eps) * base^max.exp`, but on some machines it is only the second, or perhaps third, largest number, being too small by 1 or 2 units in the last digit of the significand.

`double.base` the radix for the floating-point representation

`double.digits`
the number of base digits in the floating-point significand

`double.rounding`
the rounding action.
0 if floating-point addition chops;
1 if floating-point addition rounds, but not in the IEEE style;
2 if floating-point addition rounds in the IEEE style;
3 if floating-point addition chops, and there is partial underflow;
4 if floating-point addition rounds, but not in the IEEE style, and there is partial underflow;
5 if floating-point addition rounds in the IEEE style, and there is partial underflow

`double.guard` the number of guard digits for multiplication with truncating arithmetic. It is 1 if floating-point arithmetic truncates and more than `digits` base `base` digits participate in the post-normalization shift of the floating-point significand in multiplication, and 0 otherwise.

`double.ulp.digits`
the largest negative integer `i` such that `1 + base^i != 1`, except that it is bounded below by `-(digits + 3)`.

`double.neg.ulp.digits`
the largest negative integer `i` such that `1 - base^i != 1`, except that it is bounded below by `-(digits + 3)`.

`double.exponent`
the number of bits (decimal places if `base` is 10) reserved for the representation of the exponent (including the bias or sign) of a floating-point number

`double.min.exp`
the largest in magnitude negative integer `i` such that `base ^ i` is positive and normalized.

`double.max.exp`
the smallest positive power of `base` that overflows.

`integer.max` the largest integer which can be represented.

`sizeof.long` the number of bytes in a C `long` type.

`sizeof.longlong`
the number of bytes in a C `long long` type. Will be zero if there is no such type.

`sizeof.longdouble`
the number of bytes in a C `long double` type. Will be zero if there is no such type.

`sizeof.pointer`
the number of bytes in a C `SEXP` type.

References

Cody, W. J. (1988) MACHAR: A subroutine to dynamically determine machine parameters. *Transactions on Mathematical Software*, **14**, 4, 303–311.

See Also

`.Platform` for details of the platform.

Examples

```
.Machine
## or for a neat printout
noquote(unlist(format(.Machine)))
```

`.Platform` *Platform Specific Variables*

Description

`.Platform` is a list with some details of the platform under which R was built. This provides means to write OS-portable R code.

Usage

```
.Platform
```

Value

A list with at least the following components:

`OS.type`	character, giving the **O**perating **S**ystem (family) of the computer. One of `"unix"` or `"windows"`.
`file.sep`	character, giving the **file sep**arator used on your platform: `"/"` on both Unix-alikes *and* on Windows (but not on the now abandoned port to Classic MacOS).
`dynlib.ext`	character, giving the file name **ext**ension of **dyn**amically loadable **lib**raries, e.g., `".dll"` on Windows and `".so"` or `".sl"` on Unix-alikes. (Note for MacOS X users: these are shared objects as loaded by `dyn.load` and not dylibs.)
`GUI`	character, giving the type of GUI in use, or `"unknown"` if no GUI can be assumed.
`endian`	character, `"big"` or `"little"`, giving the endianness of the processor in use. This is relevant when it is necessary to know the order to read/write bytes of e.g. an integer or double from/to a connection: see `readBin`.
`pkgType`	character, the preferred setting for `options("pkgType")`. Values `"source"`, `"mac.binary"` and `"win.binary"` are currently in use.
`path.sep`	character, giving the **path sep**arator, used on your platform, e.g., `":"` on Unix-alikes and `";"` on Windows. Used to separate paths in variables such as `PATH` and `TEXINPUTS`.
`r_arch`	character, posssibly `""`. The name of the architecture-specific directories used in this build of R.

See Also

`R.version` and `Sys.info` give more details about the OS. In particular, `R.version$platform` is the canonical name of the platform under which R was compiled.

`.Machine` for details of the arithmetic used, and `system` for invoking platform-specific system commands.

Examples

```
## Note: this can be done in a system-independent way by file.info()$isdir
if(.Platform$OS.type == "unix") {
   system.test <- function(...) { system(paste("test", ...)) == 0 }
   dir.exists <- function(dir) sapply(dir, function(d)system.test("-d", d))
   dir.exists(c(R.home(), "/tmp", "~", "/NO"))# > T T T F
}
```

.Script *Scripting Language Interface*

Description

Run a script through its interpreter with given arguments.

Usage

```
.Script(interpreter, script, args, ...)
```

Arguments

`interpreter`	a character string naming the interpreter for the script.
`script`	a character string with the base file name of the script, which must be located in the '`interpreter`' subdirectory of 'R_SHARE_DIR' (normally 'R_HOME/share').
`args`	a character string giving the arguments to pass to the script.
`...`	further arguments to be passed to `system` when invoking the interpreter on the script.

Note

This function is for R internal use only.

Examples

```
## not useful on Windows, where the help is zipped.
.Script("perl", "massage-Examples.pl",
        paste("tools", system.file("R-ex", package = "tools")))
```

abbreviate *Abbreviate Strings*

Description

Abbreviate strings to at least `minlength` characters, such that they remain *unique* (if they were).

Usage

```
abbreviate(names.arg, minlength = 4, use.classes = TRUE,
           dot = FALSE)
```

Arguments

`names.arg`	a character vector of names to be abbreviated, or an object to be coerced to a character vector by `as.character`.
`minlength`	the minimum length of the abbreviations.
`use.classes`	logical (currently ignored by R).
`dot`	logical; should a dot (`"."`) be appended?

Details

The algorithm used is similar to that of S. First spaces at the beginning of the word are stripped. Then any other spaces are stripped. Next lower case vowels are removed followed by lower case consonants. Finally if the abbreviation is still longer than `minlength` upper case letters are stripped.

Letters are always stripped from the end of the word first. If an element of `names.arg` contains more than one word (words are separated by space) then at least one letter from each word will be retained. If a single string is passed it is abbreviated in the same manner as a vector of strings.

Missing (`NA`) values are not abbreviated.

If `use.classes` is `FALSE` then the only distinction is to be between letters and space. This has NOT been implemented.

There is a limit on 8191 bytes on the elements of `names.arg` (after possible coercion).

Value

A character vector containing abbreviations for the strings in its first argument. Duplicates in the original `names.arg` will be given identical abbreviations. If any non-duplicated elements have the same `minlength` abbreviations then `minlength` is incremented by one and new abbreviations are found for those elements only. This process is repeated until all unique elements of `names.arg` have unique abbreviations.

The character version of `names.arg` is attached to the returned value as a names argument: no other attributes are retained.

Warning

This is really only suitable for English, and does not work correctly with non-ASCII characters in UTF-8 locales. It will warns if used with non-ASCII characters.

See Also

`substr`.

Examples

```
x <- c("abcd", "efgh", "abce")
abbreviate(x, 2)

(st.abb <- abbreviate(state.name, 2))
table(nchar(st.abb))# out of 50, 3 need 4 letters
```

---

abs *Miscellaneous Mathematical Functions*

Description

These functions compute miscellaneous mathematical functions. The naming follows the standard for computer languages such as C or Fortran.

Usage

```
abs(x)
sqrt(x)
```

Arguments

`x`	a numeric or `complex` vector or array.

Details

These are generic functions: methods can be defined for them individually or via the `Math` group generic. For complex arguments (and the default method), `z`, `abs(z) == Mod(z)` and `sqrt(z) == z^0.5`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`Arithmetic` for simple, `log` for logarithmic, `sin` for trigonometric, and `Special` for special mathematical functions.

Examples

```
require(stats) # for spline
xx <- -9:9
plot(xx, sqrt(abs(xx)),  col = "red")
lines(spline(xx, sqrt(abs(xx)), n=101), col = "pink")
```

agrep *Approximate String Matching (Fuzzy Matching)*

Description

Searches for approximate matches to `pattern` (the first argument) within the string `x` (the second argument) using the Levenshtein edit distance.

Usage

```
agrep(pattern, x, ignore.case = FALSE, value = FALSE,
      max.distance = 0.1)
```

Arguments

`pattern`	a non-empty character string to be matched (*not* a regular expression!). Coerced by `as.character` to a string if possible.
`x`	character vector where matches are sought. Coerced by `as.character` to a character vector if possible.
`ignore.case`	if `FALSE`, the pattern matching is *case sensitive* and if `TRUE`, case is ignored during matching.
`value`	if `FALSE`, a vector containing the (integer) indices of the matches determined is returned and if `TRUE`, a vector containing the matching elements themselves is returned.
`max.distance`	Maximum distance allowed for a match. Expressed either as integer, or as a fraction of the pattern length (will be replaced by the smallest integer not less than the corresponding fraction), or a list with possible components **`all:`** maximal (overall) distance **`insertions:`** maximum number/fraction of insertions **`deletions:`** maximum number/fraction of deletions **`substitutions:`** maximum number/fraction of substitutions If `all` is missing, it is set to 10%, the other components default to `all`. The component names can be abbreviated.

Details

The Levenshtein edit distance is used as measure of approximateness: it is the total number of insertions, deletions and substitutions required to transform one string into another.

The function is a simple interface to the `apse` library developed by Jarkko Hietaniemi (also used in the Perl String::Approx module).

Value

Either a vector giving the indices of the elements that yielded a match, of, if `value` is `TRUE`, the matched elements (after coercion, preserving names but no other attributes).

Author(s)

David Meyer ⟨David.Meyer@wu-wien.ac.at⟩ (based on C code by Jarkko Hietaniemi); modifications by Kurt Hornik.

See Also

grep

Examples

```
agrep("lasy", "1 lazy 2")
agrep("lasy", "1 lazy 2", max = list(sub = 0))
agrep("laysy", c("1 lazy", "1", "1 LAZY"), max = 2)
agrep("laysy", c("1 lazy", "1", "1 LAZY"), max = 2, value = TRUE)
agrep("laysy", c("1 lazy", "1", "1 LAZY"), max = 2, ignore.case = TRUE)
```

---

all *Are All Values True?*

Description

Given a set of logical vectors, are all of the values true?

Usage

```
all(..., na.rm = FALSE)
```

Arguments

...	one or more logical vectors. Other objects are coerced in a similar way as as.logical.default.
na.rm	logical. If true NA values are removed before the result is computed.

Details

This is a generic function: methods can be defined for it directly or via the Summary group generic. For this to work properly, the arguments ... should be unnamed, and dispatch is on the first argument.

Value

Given a sequence of logical arguments, a logical value indicating whether or not all of the elements of x are TRUE.

The value returned is TRUE if all of the values in x are TRUE, and FALSE if any of the values in x are FALSE.

If na.rm = FALSE and x consists of a mix of TRUE and NA values, the value is NA.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

any, the "complement" of all, and stopifnot(*) which is an all(*) "insurance".

Examples

```
range(x <- sort(round(rnorm(10) - 1.2, 1)))
if(all(x < 0)) cat("all x values are negative\n")
```

---

all.equal *Test if Two Objects are (Nearly) Equal*

Description

`all.equal(x,y)` is a utility to compare R objects `x` and `y` testing "near equality". If they are different, comparison is still made to some extent, and a report of the differences is returned. Don't use `all.equal` directly in `if` expressions—either use `isTRUE(all.equal(....))` or `identical` if appropriate.

Usage

```
all.equal(target, current, ...)

## S3 method for class 'numeric':
all.equal(target, current,
          tolerance = .Machine$double.eps ^ 0.5,
          scale = NULL, check.attributes = TRUE, ...)

attr.all.equal(target, current,
               check.attributes = TRUE, check.names = TRUE, ...)
```

Arguments

`target`	R object.
`current`	other R object, to be compared with `target`.
`...`	Further arguments for different methods, notably the following two, for numerical comparison:
`tolerance`	numeric ≥ 0. Differences smaller than `tolerance` are not considered.
`scale`	numeric scalar > 0 (or `NULL`). See Details.
`check.attributes`	logical indicating if the `attributes(.)` of `target` and `current` should be compared as well.
`check.names`	logical indicating if the `names(.)` of `target` and `current` should be compared as well (and separately from the `attributes`).

Details

There are several methods available, most of which are dispatched by the default method, see `methods("all.equal")`. `all.equal.list` and `all.equal.language` provide comparison of recursive objects.

Numerical comparisons for `scale = NULL` (the default) are done by first computing the mean absolute difference of the two numerical vectors. If this is smaller than `tolerance` or not finite, absolute differences are used, otherwise relative differences scaled by the mean absolute difference.

If `scale` is positive, absolute comparisons are made after scaling (dividing) by `scale`.

For complex arguments, the modulus Mod of the difference is used: all.equal.numeric is called so arguments tolerance and scale are available.

attr.all.equal is used for comparing attributes, returning NULL or character.

Value

Either TRUE or a vector of mode "character" describing the differences between target and current.

References

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer (for =).

See Also

identical, isTRUE, ==, and all for exact equality testing.

Examples

```
all.equal(pi, 355/113) # not precise enough (default tol) > relative error

d45 <- pi*(1/4 + 1:10)
stopifnot(
all.equal(tan(d45), rep(1,10)))         # TRUE, but
all      (tan(d45) == rep(1,10))        # FALSE, since not exactly
all.equal(tan(d45), rep(1,10), tol=0)   # to see difference
```

---

all.names *Find All Names in an Expression*

Description

Return a character vector containing all the names which occur in an expression or call.

Usage

```
all.names(expr, functions = TRUE, max.names = 200, unique = FALSE)

all.vars(expr, functions = FALSE, max.names = 200, unique = TRUE)
```

Arguments

expr	an expression or call from which the names are to be extracted.
functions	a logical value indicating whether function names should be included in the result.
max.names	the maximum number of names to be returned.
unique	a logical value which indicates whether duplicate names should be removed from the value.

Details

These functions differ only in the default values for their arguments.

Value

A character vector with the extracted names.

Examples

```
all.names(expression(sin(x+y)))
all.vars(expression(sin(x+y)))
```

---

any *Are Some Values True?*

Description

Given a set of logical vectors, are any of the values true?

Usage

```
any(..., na.rm = FALSE)
```

Arguments

`...`	one or more logical vectors. Other objects are coerced in a similar way as `as.logical.default`.
`na.rm`	logical. If true `NA` values are removed before the result is computed.

Details

This is a generic function: methods can be defined for it directly or via the `Summary` group generic. For this to work properly, the arguments `...` should be unnamed, and dispatch is on the first argument.

Value

Given a sequence of logical arguments, a logical value indicating whether or not any of the elements of `x` are `TRUE`.

The value returned is `TRUE` if any of the values in `x` are `TRUE`, and `FALSE` if all of the values in `x` are `FALSE`.

If `na.rm = FALSE` and `x` consists of a mix of `FALSE` and `NA` values, the value is `NA`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`all`, the "complement" of `any`.

Examples

```
range(x <- sort(round(rnorm(10) - 1.2,1)))
if(any(x < 0)) cat("x contains negative values\n")
```

aperm *Array Transposition*

Description

Transpose an array by permuting its dimensions and optionally resizing it.

Usage

```
aperm(a, perm, resize = TRUE)
```

Arguments

a	the array to be transposed.
perm	the subscript permutation vector, which must be a permutation of the integers 1:n, where n is the number of dimensions of a. The default is to reverse the order of the dimensions.
resize	a flag indicating whether the vector should be resized as well as having its elements reordered (default TRUE).

Value

A transposed version of array a, with subscripts permuted as indicated by the array perm. If resize is TRUE, the array is reshaped as well as having its elements permuted, the dimnames are also permuted; if resize = FALSE then the returned object has the same dimensions as a, and the dimnames are dropped. In each case other attributes are copied from a.

The function t provides a faster and more convenient way of transposing matrices.

Author(s)

Jonathan Rougier, ⟨J.C.Rougier@durham.ac.uk⟩ did the faster C implementation.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

t, to transpose matrices.

Examples

```
# interchange the first two subscripts on a 3-way array x
x  <- array(1:24, 2:4)
xt <- aperm(x, c(2,1,3))
stopifnot(t(xt[,,2]) == x[,,2],
          t(xt[,,3]) == x[,,3],
          t(xt[,,4]) == x[,,4])
```

append *Vector Merging*

Description

Add elements to a vector.

Usage

```
append(x, values, after = length(x))
```

Arguments

x	the vector to be modified.
values	to be included in the modified vector.
after	a subscript, after which the values are to be appended.

Value

A vector containing the values in x with the elements of values appended after the specified element of x.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
append(1:5, 0:1, after=3)
```

apply *Apply Functions Over Array Margins*

Description

Returns a vector or array or list of values obtained by applying a function to margins of an array.

Usage

```
apply(X, MARGIN, FUN, ...)
```

Arguments

X	the array to be used.
MARGIN	a vector giving the subscripts which the function will be applied over. 1 indicates rows, 2 indicates columns, c(1,2) indicates rows and columns.
FUN	the function to be applied: see Details. In the case of functions like +, %*%, etc., the function name must be backquoted or quoted.
...	optional arguments to FUN.

Details

If `X` is not an array but has a dimension attribute, `apply` attempts to coerce it to an array via `as.matrix` if it is two-dimensional (e.g., data frames) or via `as.array`.

`FUN` is found by a call to `match.fun` and typically is either a function or a symbol (e.g. a backquoted name) or a character string specifying a function to be searched for from the environment of the call to `apply`.

Value

If each call to `FUN` returns a vector of length `n`, then `apply` returns an array of dimension `c(n, dim(X)[MARGIN])` if `n > 1`. If `n` equals `1`, `apply` returns a vector if `MARGIN` has length 1 and an array of dimension `dim(X)[MARGIN]` otherwise. If `n` is `0`, the result has length 0 but not necessarily the "correct" dimension.

If the calls to `FUN` return vectors of different lengths, `apply` returns a list of length `prod(dim(X)[MARGIN])` with `dim` set to `MARGIN` if this has length greater than one.

In all cases the result is coerced by `as.vector` to one of the basic vector types before the dimensions are set, so that (for example) factor results will be coerced to a character array.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`lapply`, `tapply`, and convenience functions `sweep` and `aggregate`.

Examples

```
## Compute row and column sums for a matrix:
x <- cbind(x1 = 3, x2 = c(4:1, 2:5))
dimnames(x)[[1]] <- letters[1:8]
apply(x, 2, mean, trim = .2)
col.sums <- apply(x, 2, sum)
row.sums <- apply(x, 1, sum)
rbind(cbind(x, Rtot = row.sums), Ctot = c(col.sums, sum(col.sums)))

stopifnot( apply(x, 2, is.vector)) # not ok in R <= 0.63.2

## Sort the columns of a matrix
apply(x, 2, sort)

##- function with extra args:
cave <- function(x, c1, c2) c(mean(x[c1]), mean(x[c2]))
apply(x,1, cave,  c1="x1", c2=c("x1","x2"))

ma <- matrix(c(1:4, 1, 6:8), nr = 2)
ma
apply(ma, 1, table)  #--> a list of length 2
apply(ma, 1, quantile)# 5 x n matrix with rownames

stopifnot(dim(ma) == dim(apply(ma, 1:2, sum)))## wasn't ok before R 0.63.1

## Example with different lengths for each call
```

```
z <- array(1:24, dim=2:4)
zseq <- apply(z, 1:2, function(x) seq(length=max(x)))
zseq         ## a 2 x 3 matrix
typeof(zseq) ## list
dim(zseq) ## 2 3
zseq[1,]
apply(z, 3, function(x) seq(length=max(x))) # a list without a dim attribute
```

`args` *Argument List of a Function*

Description

Displays the argument names and corresponding default values of a function.

Usage

```
args(name)
```

Arguments

`name`	an interpreted function. If `name` is a character string then the function with that name is found and used.

Details

This function is mainly used interactively. For programming, use `formals` instead.

Value

A function with identical formal argument list but an empty body if given an interpreted function; `NULL` in case of a variable or primitive (non-interpreted) function.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`formals`, `help`.

Examples

```
args(c)             # -> NULL (c is a 'primitive' function)
args(graphics::plot.default)
```

`Arithmetic` *Arithmetic Operators*

Description

These binary operators perform arithmetic on numeric or complex vectors (or objects which can be coerced to them).

Usage

```
x + y
x - y
x * y
x / y
x ^ y
x %% y
x %/% y
```

Arguments

`x, y`	numeric or complex vectors or objects which can be coerced to such, or other objects for which methods have been written.

Details

The binary arithmetic operators are generic functions: methods can be written for them individually or via the `Ops` group generic function. (See `Ops` for how dispatch is computed.)

If applied to arrays the result will be an array if this is sensible (for example it will not if the recycling rule has been invoked).

Logical vectors will be coerced to numeric vectors, `FALSE` having value 0 and `TRUE` having value one.

`1 ^ y` and `y ^ 0` are `1`, *always*. `x ^ y` should also give the proper "limit" result when either argument is infinite (i.e., `+-Inf`).

Objects such as arrays or time-series can be operated on this way provided they are conformable.

For real arguments, `%%` can be subject to catastrophic loss of accuracy if `x` is much larger than `y`, and a warning is given if this is detected.

Value

These operators return vectors containing the result of the element by element operations. The elements of shorter vectors are recycled as necessary (with a `warning` when they are recycled only *fractionally*). The operators are + for addition, − for subtraction, * for multiplication, / for division and ^ for exponentiation.

`%%` indicates `x mod y` and `%/%` indicates integer division. It is guaranteed that `x == (x %% y) + y * ( x %/% y )` (up to rounding error) unless `y == 0` where the result is `NA` or `NaN` (depending on the `typeof` of the arguments).

If either argument is complex the result will be complex, and if one or both arguments are numeric, the result will be numeric. If both arguments are integer, the result of / and ^ is numeric and of the other operators integer (with overflow returned as `NA` with a warning).

The rules for determining the attributes of the result are rather complicated. Most attributes are taken from the longer argument, the first if they are of the same length. Names will be copied from the first if it is the same length as the answer, otherwise from the second if that is. For time series, these operations are allowed only if the series are compatible, when the class and `tsp` attribute of whichever is a time series (the same, if both are) are used. For arrays (and an array result) the dimensions and dimnames are taken from first argument if it is an array, otherwise the second.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`sqrt` for miscellaneous and `Special` for special mathematical functions.

`Syntax` for operator precedence.

`%*%` for matrix multiplication.

Examples

```
x <- -1:12
x + 1
2 * x + 3
x %% 2 #-- is periodic
x %/% 5
```

`array` *Multi-way Arrays*

Description

Creates or tests for arrays.

Usage

```
array(data = NA, dim = length(data), dimnames = NULL)
as.array(x)
is.array(x)
```

Arguments

`data`	a vector (including a list) giving data to fill the array.
`dim`	the dim attribute for the array to be created, that is a vector of length one or more giving the maximal indices in each dimension.
`dimnames`	the names for the dimensions. This is a list with one component for each dimension, either NULL or a character vector of the length given by `dim` for that dimension. The list can be names, and the names will be used as names for the dimensions.
`x`	an R object.

Value

`array` returns an array with the extents specified in `dim` and naming information in `dimnames`. The values in `data` are taken to be those in the array with the leftmost subscript moving fastest. If there are too few elements in `data` to fill the array, then the elements in `data` are recycled. If `data` has length zero, `NA` of an appropriate type is used for atomic vectors (`0` for raw vectors) and `NULL` for lists.

`as.array()` coerces its argument to be an array by attaching a `dim` attribute to it. It also attaches `dimnames` if `x` has `names`. The sole purpose of this is to make it possible to access the `dim`[names] attribute at a later time.

`is.array` returns `TRUE` or `FALSE` depending on whether its argument is an array (i.e., has a `dim` attribute) or not. It is generic: you can write methods to handle specific classes of objects, see InternalMethods.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`aperm`, `matrix`, `dim`, `dimnames`.

Examples

```
dim(as.array(letters))
array(1:3, c(2,4)) # recycle 1:3 "2 2/3 times"
#     [,1] [,2] [,3] [,4]
#[1,]    1    3    2    1
#[2,]    2    1    3    2
```

---

as.data.frame *Coerce to a Data Frame*

Description

Functions to check if an object is a data frame, or coerce it if possible.

Usage

```
as.data.frame(x, row.names = NULL, optional = FALSE, ...)
## S3 method for class 'character':
as.data.frame(x, ...,
              stringsAsFactors = default.stringsAsFactors())
## S3 method for class 'matrix':
as.data.frame(x, row.names = NULL, optional = FALSE, ...,
                                   stringsAsFactors = default.stringsAsFactors())

is.data.frame(x)
```

Arguments

x	any R object.
row.names	NULL or a character vector giving the row names for the data frame. Missing values are not allowed.
optional	logical. If TRUE, setting row names and converting column names (to syntactic names) is optional.
...	additional arguments to be passed to or from methods.
stringsAsFactors	logical: should the character vector be converted to a factor?

Details

as.data.frame is a generic function with many methods, and users and packages can supply further methods.

If a list is supplied, each element is converted to a column in the data frame. Similarly, each column of a matrix is converted separately. This can be overridden if the object has a class which has a method for as.data.frame: two examples are matrices of class "model.matrix" (which are included as a single column) and list objects of class "POSIXlt" which are coerced to class "POSIXct".

Arrays can be converted to data frames. One-dimensional arrays are treated like vectors and two-dimensional arrays like matrices. Arrays with more than two dimensions are converted to matrices by 'flattening' all dimensions after the first and creating suitable column labels.

Character variables are converted to factor columns unless protected by I.

If a data frame is supplied, all classes preceding "data.frame" are stripped, and the row names are changed if that argument is supplied.

If row.names = NULL, row names are constructed from the names or dimnames of x, otherwise are the integer sequence starting at one. Few of the methods check for duplicated row names. Names are removed from vector columns unless I.

Value

as.data.frame returns a data frame, normally with all row names "" if optional = TRUE.

is.data.frame returns TRUE if its argument is a data frame (that is, has "data.frame" amongst its classes) and FALSE otherwise.

References

Chambers, J. M. (1992) *Data for models.* Chapter 3 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

data.frame, as.data.frame.table for the table method (which has additional arguments if called directly).

as.environment *Coerce to an Environment Object*

Description

Converts a number or a character string to the corresponding environment on the search path.

Usage

```
as.environment(object)
```

Arguments

object	the object to convert. If it is already an environment, just return it. If it is a number, return the environment corresponding to that position on the search list. If it is a character string, match the string to the names on the search list.

Value

The corresponding environment object.

Author(s)

John Chambers

See Also

environment for creation and manipulation, search.

Examples

```
as.environment(1) ## the global environment
identical(globalenv(), as.environment(1)) ## is TRUE
try(as.environment("package:stats"))      ## stats need not be loaded
```

as.function *Convert Object to Function*

Description

as.function is a generic function which is used to convert objects to functions.

as.function.default works on a list x, which should contain the concatenation of a formal argument list and an expression or an object of mode "call" which will become the function body. The function will be defined in a specified environment, by default that of the caller.

Usage

```
as.function(x, ...)

## Default S3 method:
as.function(x, envir = parent.frame(), ...)
```

Arguments

x	object to convert, a list for the default method.
...	additional arguments, depending on object
envir	environment in which the function should be defined

Value

The desired function.

Note

For ancient historical reasons, `envir = NULL` uses the global environment rather than the base environment. Please use `envir = globalenv()` instead if this is what you want, as the special handling of `NULL` may change in a future release.

Author(s)

Peter Dalgaard

See Also

`function`; `alist` which is handy for the construction of argument lists, etc.

Examples

```
as.function(alist(a=,b=2,a+b))
as.function(alist(a=,b=2,a+b))(3)
```

---

as.POSIX* *Date-time Conversion Functions*

Description

Functions to manipulate objects of classes `"POSIXlt"` and `"POSIXct"` representing calendar dates and times.

Usage

```
as.POSIXct(x, tz = "")
as.POSIXlt(x, tz = "")

## S3 method for class 'POSIXlt':
as.numeric(x)
```

Arguments

x	An object to be converted.
tz	A timezone specification to be used for the conversion, *if one is required*. System-specific, but `""` is the current timezone, and `"GMT"` is UTC (Coordinated Universal Time, in French).

Details

The as.POSIX* functions convert an object to one of the two classes used to represent date/times (calendar dates plus time to the nearest second). They can convert a wide variety of objects, including objects of the other class and of classes "Date", "date" (from package **date** or **survival**), "chron" and "dates" (from package **chron**) to these classes. Dates without times are treated as being at midnight UTC.

They can also convert character strings of the formats "2001-02-03" and "2001/02/03" optionally followed by white space and a time in the format "14:52" or "14:52:03". (Formats such as "01/02/03" are ambiguous but can be converted via a format specification by strptime.) (As from R 2.4.0 fractional seconds can be converted.)

Logical NAs can be converted to either of the classes, but no other logical vectors can be.

The as.numeric method converts "POSIXlt" objects to "POSIXct".

If you are given a numeric time as the number of seconds since an epoch, see the examples.

Where OSes describe their valid timezones can be obscure. The help for tzset (or _tzset on Windows) can be helpful, but it can also be inaccurate. There is a cumbersome POSIX specification, (listed under environment variable TZ at http://www.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap08.html), which is often at least partially supported, but there may be other more user-friendly ways to specify timezones. For most Unix-alikes (including MacOS X) this can be an optional colon prepended to the path to a file (by default under '/usr/share/zoneinfo' or '/usr/lib/zoneinfo' (or even '/usr/share/lib/zoneinfo' on Solaris)), for example 'EST5EDT' or 'GB' or 'Europe/Paris'. See http://www.twinsun.com/tz/tz-link.htm for more details and references.

Value

as.POSIXct and as.POSIXlt return an object of the appropriate class. If tz was specified, as.POSIXlt will give an appropriate "tzone" attribute.

Note

If you want to extract specific aspects of a time (such as the day of the week) just convert it to class "POSIXlt" and extract the relevant component(s) of the list, or if you want a character representation (such as a named day of the week) use format.POSIXlt or format.POSIXct.

If a timezone is needed and that specified is invalid on your system, what happens is system-specific but it will probably be ignored.

See Also

DateTimeClasses for details of the classes; strptime for conversion to and from character representations.

Examples

```
(z <- Sys.time())             # the current datetime, as class "POSIXct"
unclass(z)                    # a large integer
floor(unclass(z)/86400)       # the number of days since 1970-01-01
(z <- as.POSIXlt(Sys.time())) # the current datetime, as class "POSIXlt"
unlist(unclass(z))            # a list shown as a named vector

## suppose we have a time in seconds since 1960-01-01 00:00:00 GMT
z <- 1472562988
# two ways to convert this
```

```
ISOdatetime(1960,1,1,0,0,0) + z # late August 2006
strptime("1960-01-01", "%Y-%m-%d", tz="GMT") + z

as.POSIXlt(Sys.time(), "GMT") # the current time in GMT
## Not run:
## These may not be correct names on your system
as.POSIXlt(Sys.time(), "EST5EDT")  # the current time in New York
as.POSIXlt(Sys.time(), "EST" )     # ditto, ignoring DST
as.POSIXlt(Sys.time(), "HST")      # the current time in Hawaii
as.POSIXlt(Sys.time(), "Australia/Darwin")
## End(Not run)
```

`AsIs` *Inhibit Interpretation/Conversion of Objects*

Description

Change the class of an object to indicate that it should be treated "as is".

Usage

```
I(x)
```

Arguments

`x`	an object

Details

Function `I` has two main uses.

- In function `data.frame`. Protecting an object by enclosing it in `I()` in a call to `data.frame` inhibits the conversion of character vectors to factors, and the dropping of names. `I` can also be used to protect objects which are to be added to a data frame, or converted to a data frame *via* `as.data.frame`.
  It achieves this by prepending the class `"AsIs"` to the object's classes. Class `"AsIs"` has a few of its own methods, including for `[`, `as.data.frame`, `print` and `format`.
- In function `formula`. There it is used to inhibit the interpretation of operators such as `"+"`, `"-"`, `"*"` and `"^"` as formula operators, so they are used as arithmetical operators. This is interpreted as a symbol by `terms.formula`.

Value

A copy of the object with class `"AsIs"` prepended to the class(es).

References

Chambers, J. M. (1992) *Linear models.* Chapter 4 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`data.frame`, `formula`

`assign` *Assign a Value to a Name*

Description

Assign a value to a name in an environment.

Usage

```
assign(x, value, pos = -1, envir = as.environment(pos),
       inherits = FALSE, immediate = TRUE)
```

Arguments

`x`	a variable name (given as a quoted string in the function call).
`value`	a value to be assigned to `x`.
`pos`	where to do the assignment. By default, assigns into the current environment. See the details for other possibilities.
`envir`	the `environment` to use. See the details section.
`inherits`	should the enclosing frames of the environment be inspected?
`immediate`	an ignored compatibility feature.

Details

The `pos` argument can specify the environment in which to assign the object in any of several ways: as an integer (the position in the `search` list); as the character string name of an element in the search list; or as an `environment` (including using `sys.frame` to access the currently active function calls). The `envir` argument is an alternative way to specify an environment, but is primarily there for back compatibility.

`assign` does not dispatch assignment methods, so it cannot be used to set elements of vectors, names, attributes, etc.

Note that assignment to an attached list or data frame changes the attached copy and not the original object: see `attach`.

Value

This function is invoked for its side effect, which is assigning `value` to the variable `x`. If no `envir` is specified, then the assignment takes place in the currently active environment.

If `inherits` is `TRUE`, enclosing environments of the supplied environment are searched until the variable `x` is encountered. The value is then assigned in the environment in which the variable is encountered (provided that the binding is not locked: see `lockBinding`: if it is, an error is signaled). If the symbol is not encountered then assignment takes place in the user's workspace (the global environment).

If `inherits` is `FALSE`, assignment takes place in the initial frame of `envir`, unless an existing binding is locked or there is no existing binding and the environment is locked.

Note

Prior to R 2.4.0 the base environment was ignored in the search for a binding by `inherits = TRUE`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`<-`, `get`, `exists`, `environment`.

Examples

```
for(i in 1:6) { #-- Create objects  'r.1', 'r.2', ... 'r.6' --
 nam <- paste("r",i, sep=".")
 assign(nam, 1:i)
}
ls(pat="^r..$")

##-- Global assignment within a function:
myf <- function(x) {
 innerf <- function(x) assign("Global.res", x^2, env = .GlobalEnv)
 innerf(x+1)
}
myf(3)
Global.res # 16

a <- 1:4
assign("a[1]", 2)
a[1] == 2          #FALSE
get("a[1]") == 2   #TRUE
```

`assignOps` *Assignment Operators*

Description

Assign a value to a name.

Usage

```
x <- value
x <<- value
value -> x
value ->> x

x = value
```

Arguments

`x`	a variable name (possibly quoted).
`value`	a value to be assigned to `x`.

Details

There are three different assignment operators: two of them have leftwards and rightwards forms.

The operators <- and = assign into the environment in which they are evaluated. The operator<- can be used anywhere, whereas the operator = is only allowed at the top level (e.g., in the complete expression typed at the command prompt) or as one of the subexpressions in a braced list of expressions.

The operators <<- and ->> cause a search to made through the environment for an existing definition of the variable being assigned. If such a variable is found (and its binding is not locked) then its value is redefined, otherwise assignment takes place in the global environment. Note that their semantics differ from that in the S language, but are useful in conjunction with the scoping rules of R. See 'The R Language Definition' manual for further details and examples.

In all the assignment operator expressions, `x` can be a name or an expression defining a part of an object to be replaced (e.g., `z[[1]]`). A syntactic name does not need to be quoted, though it can be (preferably by backticks).

The leftwards forms of assignment `<- = <<-` group right to left, the other from left to right.

Value

`value`. Thus one can use `a <- b <- c <- 6`.

Note

Prior to R 2.4.0 the base environment was ignored in the search for a binding by <<- and ->>.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Chamber, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer (for =).

See Also

`assign`, `environment`.

---

attach *Attach Set of R Objects to Search Path*

Description

The database is attached to the R search path. This means that the database is searched by R when evaluating a variable, so objects in the database can be accessed by simply giving their names.

Usage

```
attach(what, pos = 2, name = deparse(substitute(what)),
       warn.conflicts = TRUE)
```

Arguments

what	"database". This may currently be a data.frame or a list or a R data file created with save or NULL or an environment. See also Details.
pos	integer specifying position in search() where to attach.
name	alternative way to specify the database to be attached.
warn.conflicts	logical. If TRUE, warnings are printed about conflicts from attaching the database, unless that database contains an object .conflicts.OK. A conflict is a function masking a function, or a non-function masking a non-function.

Details

When evaluating a variable or function name R searches for that name in the databases listed by search. The first name of the appropriate type is used.

By attaching a data frame (or list) to the search path it is possible to refer to the variables in the data frame by their names alone, rather than as components of the data frame (e.g. in the example below, height rather than women$height).

By default the database is attached in position 2 in the search path, immediately after the user's workspace and before all previously loaded packages and previously attached databases. This can be altered to attach later in the search path with the pos option, but you cannot attach at pos=1.

The database is not actually attached. Rather, a new environment is created on the search path and the elements of a list (including columns of a data frame) or objects in a save file or an environment are *copied* into the new environment. If you use <<- or assign to assign to an attached database, you only alter the attached copy, not the original object. (Normal assignment will place a modified version in the user's workspace: see the examples.) For this reason attach can lead to confusion.

One useful 'trick' is to use what = NULL (or equivalently a length-zero list) to create a new environment on the search path into which objects can be assigned by assign or load or sys.source.

There are hooks to attach user-defined table objects of class "UserDefinedDatabase", supported by the Omegahat package **RObjectTables**. See http://www.omegahat.org/RObjectTables/.

Value

The environment is returned invisibly with a "name" attribute.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

library, detach, search, objects, environment, with.

Examples

```
summary(women$height)    # refers to variable 'height' in the data frame
attach(women)
summary(height)          # The same variable now available by name
height <- height*2.54    # Don't do this. It creates a new variable
```

```
                          # in the user's workspace
find("height")
summary(height)           # The new variable in the workspace
rm(height)
summary(height)           # The original variable.
height <<- height*25.4    # Change the copy in the attached environment
find("height")
summary(height)           # The changed copy
detach("women")
summary(women$height)     # unchanged

## Not run:
## create an environment on the search path and populate it
sys.source("myfuns.R", envir=attach(NULL, name="myfuns"))
## End(Not run)
```

`attr` *Object Attributes*

Description

Get or set specific attributes of an object.

Usage

```
attr(x, which)
attr(x, which) <- value
```

Arguments

`x`	an object whose attributes are to be accessed.
`which`	a non-empty character string specifying which attribute is to be accessed.
`value`	an object, the new value of the attribute, or `NULL` to remove the attribute.

Details

These functions provide access to a single attribute of an object. The replacement form causes the named attribute to take the value specified (or create a new attribute with the value given).

The extraction function first looks for an exact match to `which` amongst the attributes of `x`, then a unique partial match. The replacement function only uses exact matches.

Note that some attributes (namely `class`, `comment`, `dim`, `dimnames`, `names`, (from R 2.4.0) `row.names` and `tsp`) are treated specially and have restrictions on the values which can be set.

Specifying `which` as a character `NA` is interpreted as `"NA"` and not as a missing value.

Value

For the first form, the value of the attribute matched, or `NULL` if no exact match is found and no or more than one partial match is found.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

attributes

Examples

```
# create a 2 by 5 matrix
x <- 1:10
attr(x,"dim") <- c(2, 5)
```

---

attributes *Object Attribute Lists*

Description

These functions access an object's attributes. The first form below returns the object's attribute list. The replacement forms uses the list on the right-hand side of the assignment as the object's attributes (if appropriate).

Usage

```
attributes(obj)
attributes(obj) <- value
mostattributes(obj) <- value
```

Arguments

obj	an object
value	an appropriate named list of attributes, or NULL.

Details

Note that some attributes (namely class, comment, dim, dimnames, names, (from R 2.4.0) row.names and tsp) are treated specially and have restrictions on the values which can be set.

Attributes are not stored internally as a list and should be thought of as a set and not a vector. They must have unique names (and NA is taken as "NA", not a missing value).

The mostattributes assignment takes special care for the dim, names and dimnames attributes, and assigns them only when valid whereas an attributes assignment would give an error if any are not.

The names of a pairlist are not stored as attributes, but are reported as if they were (and can be set by the replacement method for attributes).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

attr.

Examples

```
x <- cbind(a=1:3, pi=pi) # simple matrix w/ dimnames
attributes(x)

## strip an object's attributes:
attributes(x) <- NULL
x # now just a vector of length 6

mostattributes(x) <- list(mycomment = "really special", dim = 3:2,
   dimnames = list(LETTERS[1:3], letters[1:5]), names = paste(1:6))
x # dim(), but not {dim}names
```

---

autoload *On-demand Loading of Packages*

Description

`autoload` creates a promise-to-evaluate `autoloader` and stores it with name `name` in `.AutoloadEnv` environment. When R attempts to evaluate `name`, `autoloader` is run, the package is loaded and `name` is re-evaluated in the new package's environment. The result is that R behaves as if `file` was loaded but it does not occupy memory.

`.Autoloaded` contains the "base names" of the packages for which autoloading has been promised.

Usage

```
autoload(name, package, reset = FALSE, ...)
autoloader(name, package, ...)

.AutoloadEnv
.Autoloaded
```

Arguments

`name`	string giving the name of an object.
`package`	string giving the name of a package containing the object.
`reset`	logical: for internal use by `autoloader`.
`...`	other arguments to `library`.

Value

This function is invoked for its side-effect. It has no return value.

See Also

`delayedAssign`, `library`

Examples

```
require(stats)
autoload("interpSpline", "splines")
search()
ls("Autoloads")
.Autoloaded

x <- sort(rnorm(12))
y <- x^2
is <- interpSpline(x,y)
search() ## now has splines
detach("package:splines")
search()
is2 <- interpSpline(x,y+x)
search() ## and again
detach("package:splines")
```

---

backsolve *Solve an Upper or Lower Triangular System*

Description

Solves a system of linear equations where the coefficient matrix is upper or lower triangular.

Usage

```
   backsolve(r, x, k=ncol(r), upper.tri=TRUE, transpose=FALSE)
forwardsolve(l, x, k=ncol(l), upper.tri=FALSE, transpose=FALSE)
```

Arguments

`r,l`	an upper (or lower) triangular matrix giving the coefficients for the system to be solved. Values below (above) the diagonal are ignored.
`x`	a matrix whose columns give "right-hand sides" for the equations.
`k`	The number of columns of `r` and rows of `x` to use.
`upper.tri`	logical; if `TRUE` (default), the *upper tri*angular part of `r` is used. Otherwise, the lower one.
`transpose`	logical; if `TRUE`, solve $r' * y = x$ for y, i.e., `t(r) %*% y == x`.

Value

The solution of the triangular system. The result will be a vector if `x` is a vector and a matrix if `x` is a matrix.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Dongarra, J. J., Bunch,J. R., Moler, C. B. and Stewart, G. W. (1978) *LINPACK Users Guide.* Philadelphia: SIAM Publications.

See Also

chol, qr, solve.

Examples

```
## upper triangular matrix 'r':
r <- rbind(c(1,2,3),
           c(0,1,1),
           c(0,0,2))
( y <- backsolve(r, x <- c(8,4,2)) ) # -1 3 1
r %*% y # == x = (8,4,2)
backsolve(r, x, transpose = TRUE) # 8 -12 -5
```

---

base-deprecated *Deprecated Functions in Base package*

Description

These functions are provided for compatibility with older versions of R only, and may be defunct as soon as the next release.

Usage

```
symbol.C(name)
symbol.For(name)
```

Arguments

name	a character string giving either the name of a C function or Fortran subroutine.

Details

The original help page for these functions is often available at help("oldName-deprecated") (note the quotes). Functions in packages other than the base package are listed in help("pkg-deprecated").

Functions symbol.C and symbol.For are for historical compatibility with S. symbol.C just returns its argument unchanged, whereas symbol.For will map to lower case and append an underscore on those platforms (almost all) which do so for Fortran symbols. These are no longer of much use in R.

See Also

Deprecated, base-defunct

basename *Manipulate File Paths*

Description

basename removes all of the path up to the last path separator (if any).

dirname returns the part of the path up to (but excluding) the last path separator, or "." if there is no path separator.

Usage

```
basename(path)
dirname(path)
```

Arguments

path	character vector, containing path names.

Details

For dirname tilde expansion is done: see the description of path.expand.

Trailing file separators are removed before dissecting the path, and for dirname any trailing file separators are removed from the result.

Value

A character vector of the same length as path. A zero-length input will give a zero-length output with no error.

See Also

file.path, path.expand.

Examples

```
basename(file.path("","p1","p2","p3", c("file1", "file2")))
dirname(file.path("","p1","p2","p3","filename"))
```

Bessel *Bessel Functions*

Description

Bessel Functions of integer and fractional order, of first and second kind, J_ν and Y_ν, and Modified Bessel functions (of first and third kind), I_ν and K_ν.

gammaCody is the (Γ) function as from the Specfun package and originally used in the Bessel code.

Usage

```
besselI(x, nu, expon.scaled = FALSE)
besselK(x, nu, expon.scaled = FALSE)
besselJ(x, nu)
besselY(x, nu)
gammaCody(x)
```

Arguments

`x`	numeric, ≥ 0.
`nu`	numeric; The *order* (maybe fractional!) of the corresponding Bessel function.
`expon.scaled`	logical; if `TRUE`, the results are exponentially scaled in order to avoid overflow (I_ν) or underflow (K_ν), respectively.

Details

The underlying C code stems from *Netlib* (`http://www.netlib.org/specfun/r[ijky]besl`).

If `expon.scaled = TRUE`, $e^{-x} I_\nu(x)$, or $e^x K_\nu(x)$ are returned.

`gammaCody` may be somewhat faster but less precise and/or robust than R's standard `gamma`. It is here for experimental purpose mainly, and *may be defunct very soon*.

For $\nu < 0$, formulae 9.1.2 and 9.6.2 from the reference below are applied (which is probably suboptimal), unless for `besselK` which is symmetric in `nu`.

Value

Numeric vector of the same length of `x` with the (scaled, if `expon.scale=TRUE`) values of the corresponding Bessel function.

Author(s)

Original Fortran code: W. J. Cody, Argonne National Laboratory
Translation to C and adaption to R: Martin Maechler ⟨maechler@stat.math.ethz.ch.⟩

References

Abramowitz, M. and Stegun, I. A. (1972) *Handbook of Mathematical Functions.* Dover, New York; Chapter 9: Bessel Functions of Integer Order.

See Also

Other special mathematical functions, such as `gamma`, $\Gamma(x)$, and `beta`, $B(x)$.

Examples

```
nus <- c(0:5, 10, 20)

x <- seq(0, 4, len = 501)
plot(x, x, ylim = c(0, 6), ylab = "", type = "n",
     main = "Bessel Functions  I_nu(x)")
for(nu in nus) lines(x, besselI(x, nu=nu), col = nu+2)
legend(0, 6, legend = paste("nu=", nus), col = nus+2, lwd = 1)
```

```
x <- seq(0, 40, len=801); yl <- c(-.8, .8)
plot(x, x, ylim = yl, ylab = "", type = "n",
     main = "Bessel Functions  J_nu(x)")
for(nu in nus) lines(x, besselJ(x, nu=nu), col = nu+2)
legend(32,-.18, legend = paste("nu=", nus), col = nus+2, lwd = 1)

## Negative nu's :
xx <- 2:7
nu <- seq(-10, 9, len = 2001)
op <- par(lab = c(16, 5, 7))
matplot(nu, t(outer(xx, nu, besselI)), type = "l", ylim = c(-50, 200),
        main = expression(paste("Bessel ", I[nu](x), " for fixed ", x,
                                ",  as ", f(nu))),
        xlab = expression(nu))
abline(v=0, col = "light gray", lty = 3)
legend(5, 200, legend = paste("x=", xx), col=seq(xx), lty=seq(xx))
par(op)

x0 <- 2^(-20:10)
plot(x0, x0^-8, log="xy", ylab="",type="n",
     main = "Bessel Functions  J_nu(x)  near 0\n log - log  scale")
for(nu in sort(c(nus, nus+.5)))
    lines(x0, besselJ(x0, nu=nu), col = nu+2)
legend(3, 1e50, legend = paste("nu=", paste(nus, nus+.5, sep=",")),
       col = nus + 2, lwd = 1)

plot(x0, x0^-8, log="xy", ylab="", type="n",
     main = "Bessel Functions  K_nu(x)  near 0\n log - log  scale")
for(nu in sort(c(nus, nus+.5)))
    lines(x0, besselK(x0, nu=nu), col = nu+2)
legend(3, 1e50, legend = paste("nu=", paste(nus, nus+.5, sep=",")),
       col = nus + 2, lwd = 1)

x <- x[x > 0]
plot(x, x, ylim=c(1e-18, 1e11), log = "y", ylab = "", type = "n",
     main = "Bessel Functions  K_nu(x)")
for(nu in nus) lines(x, besselK(x, nu=nu), col = nu+2)
legend(0, 1e-5, legend=paste("nu=", nus), col = nus+2, lwd = 1)

yl <- c(-1.6, .6)
plot(x, x, ylim = yl, ylab = "", type = "n",
     main = "Bessel Functions  Y_nu(x)")
for(nu in nus){
    xx <- x[x > .6*nu]
    lines(xx, besselY(xx, nu=nu), col = nu+2)
}
legend(25, -.5, legend = paste("nu=", nus), col = nus+2, lwd = 1)
```

`body` *Access to and Manipulation of the Body of a Function*

Description

Get or set the body of a function.

Usage

```
body(fun = sys.function(sys.parent()))
body(fun, envir = environment(fun)) <- value
```

Arguments

`fun`	a function object, or see Details.
`envir`	environment in which the function should be defined.
`value`	an expression or a list of R expressions.

Details

For the first form, `fun` can be a character string naming the function to be manipulated, which is searched for from the parent environment. If it is not specified, the function calling `body` is used.

Value

`body` returns the body of the function specified.

The replacement form sets the body of a function to the expression/list on the right hand side, and (potentially) resets the environment of the function.

See Also

`alist`, `args`, `function`.

Examples

```
body(body)
f <- function(x) x^5
body(f) <- expression(5^x)
## or equivalently  body(f) <- list(quote(5^x))
f(3) # = 125
body(f)
```

`bquote` *Partial substitution in expressions*

Description

An analogue of the LISP backquote macro. `bquote` quotes its argument except that terms wrapped in `.()` are evaluated in the specified `where` environment.

Usage

```
bquote(expr, where = parent.frame())
```

Arguments

`expr`	A language object.
`where`	An environment.

Value

A language object.

See Also

quote, substitute

Examples

```
a <- 2

bquote(a == a)
quote(a == a)

bquote(a == .(a))
substitute(a == A, list(A = a))

plot(1:10, a*(1:10), main = bquote(a == .(a)))
```

---

browser *Environment Browser*

Description

Interrupt the execution of an expression and allow the inspection of the environment where browser was called from.

Usage

```
browser()
```

Details

A call to browser can be included in the body of a function. When reached, this causes a pause in the execution of the current expression and allows access to the R interpreter.

At the browser prompt the user can enter commands or R expressions. The commands are

c (or just return) exit the browser and continue execution at the next statement.

cont synonym for c.

n enter the step-through debugger. This changes the meaning of c: see the documentation for debug.

where print a stack trace of all active function calls.

Q exit the browser and the current evaluation and return to the top-level prompt.

(Leading and trailing whitespace is ignored, except for return).

Anything else entered at the browser prompt is interpreted as an R expression to be evaluated in the calling environment: in particular typing an object name will cause the object to be printed, and ls() lists the objects in the calling frame. (If you want to look at an object with a name such as n, print it explicitly.)

The number of lines printed for the deparsed call can be limited by setting options(deparse.max.lines).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer.

See Also

`debug`, and `traceback` for the stack on error.

`builtins` *Returns the Names of All Built-in Objects*

Description

Return the names of all the built-in objects. These are fetched directly from the symbol table of the R interpreter.

Usage

```
builtins(internal = FALSE)
```

Arguments

`internal`	a logical indicating whether only "internal" functions (which can be called via `.Internal`) should be returned.

Details

`builtins()` returns an unsorted list of the objects in the symbol table, that is all the objects in the base environment. These are the built-in objects plus any that have been added subsequently when the base package was loaded. It is less confusing to use `ls(baseenv(), all=TRUE)`.

`builtins(TRUE)` returns an unsorted list of the names of internal functions, that is those which can be accessed as `.Internal(foo(args ...))` for *foo* in the list.

Value

A character vector.

`by` *Apply a Function to a Data Frame split by Factors*

Description

Function `by` is an object-oriented wrapper for `tapply` applied to data frames.

Usage

```
by(data, INDICES, FUN, ...)
```

Arguments

`data`	an R object, normally a data frame, possibly a matrix.
`INDICES`	a factor or a list of factors, each of length `nrow(data)`.
`FUN`	a function to be applied to data frame subsets of `data`.
`...`	further arguments to `FUN`.

Details

A data frame is split by row into data frames subsetted by the values of one or more factors, and function `FUN` is applied to each subset in turn.

Object `data` will be coerced to a data frame by default.

Value

A list of class `"by"`, giving the results for each subset.

See Also

`tapply`

Examples

```
require(stats)
attach(warpbreaks)
by(warpbreaks[, 1:2], tension, summary)
by(warpbreaks[, 1], list(wool = wool, tension = tension), summary)
by(warpbreaks, tension, function(x) lm(breaks ~ wool, data = x))

## now suppose we want to extract the coefficients by group
tmp <- by(warpbreaks, tension, function(x) lm(breaks ~ wool, data = x))
sapply(tmp, coef)

detach("warpbreaks")
```

c *Combine Values into a Vector or List*

Description

This is a generic function which combines its arguments.

The default method combines its arguments to form a vector. All arguments are coerced to a common type which is the type of the returned value, and all attributes except names are removed.

Usage

```
c(..., recursive=FALSE)
```

Arguments

`...`	objects to be concatenated.
`recursive`	logical. If `recursive = TRUE`, the function recursively descends through lists (and pairlists) combining all their elements into a vector.

Details

The output type is determined from the highest type of the components in the hierarchy NULL < raw < logical < integer < real < complex < character < list < expression. Pairlists are treated as lists, but non-vector components (such names and calls) are treated as one-element lists which cannot be unlisted even if `recursive = TRUE`.

`c` is sometimes used for its side effect of removing attributes except names, for example to turn an array into a vector. `as.vector` is a more intuitive way to do this, but also drops names.

Value

`NULL` or an expression or a vector of an appropriate mode.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`unlist` and `as.vector` to produce attribute-free vectors.

Examples

```
c(1,7:9)
c(1:5, 10.5, "next")

## uses with a single argument to drop attributes
x <- 1:4
names(x) <- letters[1:4]
x
c(x)           # has names
as.vector(x)   # no names
```

```
dim(x) <- c(2,2)
x
c(x)
as.vector(x)

## append to a list:
ll <- list(A = 1, c="C")
## do *not* use
c(ll, d = 1:3) # which is == c(ll, as.list(c(d=1:3))
## but rather
c(ll, d = list(1:3))# c() combining two lists

c(list(A=c(B=1)), recursive=TRUE)

c(options(), recursive=TRUE)
c(list(A=c(B=1,C=2), B=c(E=7)), recursive=TRUE)
```

`call` *Function Calls*

Description

Create or test for objects of mode `"call"`.

Usage

```
call(name, ...)
is.call(x)
as.call(x)
```

Arguments

`name`	a character string naming the function to be called.
`...`	arguments to be part of the call.
`x`	an arbitrary R object.

Details

`call` returns an unevaluated function call, that is, an unevaluated expression which consists of the named function applied to the given arguments (`name` must be a quoted string which gives the name of a function to be called). Note that although the call is unevaluated, the arguments `...` are evaluated.

`call` is a primitive, so the first argument (named or not) is taken as `name` and the remaining arguments as arguments for the constructed call: `call(x="c", 1,3, name="foo")` is a call to `c` and not to `foo`.

`is.call` is used to determine whether `x` is a call (i.e., of mode `"call"`). It is generic: you can write methods to handle specific classes of objects, see InternalMethods.

Objects of mode `"list"` can be coerced to mode `"call"`. The first element of the list becomes the function part of the call, so should be a function or the name of one (as a symbol; a quoted string will not do).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`do.call` for calling a function by name and argument list; `Recall` for recursive calling of functions; further `is.language`, `expression`, `function`.

Examples

```
is.call(call) #-> FALSE: Functions are NOT calls

## set up a function call to round with argument 10.5
cl <- call("round", 10.5)
is.call(cl)# TRUE
cl
## such a call can also be evaluated.
eval(cl)# [1] 10

A <- 10.5
call("round", A)        # round(10.5)
call("round", quote(A)) # round(A)
f <- "round"
call(f, quote(A))       # round(A)
## if we want to supply a function we need to use as.call or similar
f <- round
## Not run: call(f, quote(A))  # error: first arg must be character
(g <- as.call(list(f, quote(A))))
eval(g)
## alternatively but less transparently
g <- list(f, quote(A))
mode(g) <- "call"
g
eval(g)
## see also the examples in the help for do.call
```

`capabilities` *Report Capabilities of this Build of R*

Description

Report on the optional features which have been compiled into this build of R.

Usage

```
capabilities(what = NULL)
```

Arguments

`what`	character vector or `NULL`, specifying required components. `NULL` implies that all are required.

Value

A named logical vector. Current components are

jpeg	Is the jpeg function operational?
png	Is the png function operational?
tcltk	Is the **tcltk** package operational?
X11	(Unix) Are the X11 graphics device and the X11-based data editor available? This loads the X11 module if not already loaded, and checks that the default display can be contacted unless a X11 device has already been used.
http/ftp	Are url and the internal method for download.file available?
sockets	Are make.socket and related functions available?
libxml	Is there support for integrating libxml with the R event loop?
fifo	are FIFO connections supported?
cledit	Is command-line editing available in the current R session? This is false in non-interactive sessions. It will be true for the command-line interface if readline support has been compiled in and '--no-readline' was *not* invoked.
iconv	is internationalization conversion via iconv supported?
NLS	is there Natural Language Support (for message translations)?

See Also

.Platform

Examples

```
capabilities()

if(!capabilities("http/ftp"))
   warning("internal download.file() is not available")

## See also the examples for 'connections'.
```

---

cat *Concatenate and Print*

Description

Outputs the objects, concatenating the representations. cat performs much less conversion than print.

Usage

```
cat(... , file = "", sep = " ", fill = FALSE, labels = NULL,
    append = FALSE)
```

Arguments

`...`	R objects (see Details for the types of objects allowed).
`file`	A connection, or a character string naming the file to print to. If `""` (the default), `cat` prints to the standard output connection, the console unless redirected by `sink`. If it is `"|cmd"`, the output is piped to the command given by 'cmd', by opening a pipe connection.
`sep`	character string to insert between the objects to print.
`fill`	a logical or (positive) numeric controlling how the output is broken into successive lines. If `FALSE` (default), only newlines created explicitly by '`"\n"`' are printed. Otherwise, the output is broken into lines with print width equal to the option `width` if `fill` is `TRUE`, or the value of `fill` if this is numeric. Non-positive `fill` values are ignored, with a warning.
`labels`	character vector of labels for the lines printed. Ignored if `fill` is `FALSE`.
`append`	logical. Only used if the argument `file` is the name of file (and not a connection or `"|cmd"`). If `TRUE` output will be appended to `file`; otherwise, it will overwrite the contents of `file`.

Details

`cat` is useful for producing output in user-defined functions. It converts its arguments to character strings, concatenates them, separating them by the given `sep=` string, and then outputs them.

No linefeeds are output unless explicitly requested by '`"\n"`' or if generated by filling (if argument `fill` is `TRUE` or numeric.)

Currently only atomic vectors (and so not lists) and names are handled. Character strings are output 'as is' (unlike `print.default` which escapes non-printable characters and backslash — use `encodeString` if you want to output encoded strings using `cat`). Other types of R object should be converted (e.g. by `as.character` or `format`) before being passed to `cat`.

`cat` converts numeric/complex elements in the same way as `print` (and not in the same way as `as.character` which is used by the S equivalent), so `options` `"digits"` and `"scipen"` are relevant. However, it uses the minimum field width necessary for each element, rather than the same field width for all elements.

Value

None (invisible `NULL`).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`print`, `format`, and `paste` which concatenates into a string.

Examples

```
iter <- rpois(1, lambda=10)
## print an informative message
cat("iteration = ", iter <- iter + 1, "\n")
```

```
## 'fill' and label lines:
cat(paste(letters, 100* 1:26), fill = TRUE,
    labels = paste("{",1:10,"}:",sep=""))
```

`cbind` *Combine R Objects by Rows or Columns*

Description

Take a sequence of vector, matrix or data frames arguments and combine by *c*olumns or *r*ows, respectively. These are generic functions with methods for other R classes.

Usage

```
cbind(..., deparse.level = 1)
rbind(..., deparse.level = 1)
```

Arguments

`...`	vectors or matrices. These can be given as named arguments. Other R objects will be coerced as appropriate: see section Details and Value. (For the `"data.frame"` method of `cbind` these can be further arguments to `data.frame` such as `stringsAsFactors` and `check.names`.)
`deparse.level`	integer controlling the construction of labels in the case of "non-matrix-like" arguments (i.e., for the default method): `deparse.level = 0` constructs no labels; the default, `deparse.level = 1 or 2` constructs labels from the argument names, see the 'Value' section below.

Details

The functions `cbind` and `rbind` are S3 generic, with methods for data frames. The data frame method will be used if at least one argument is a data frame and the rest are vectors or matrices. There can be other methods; in particular, there is one for time series objects.

In the default method, all the vectors/matrices must be atomic (see `vector`) or lists. Expressions are not allowed. Language objects (such as formulae and calls) and pairlists will be coerced to lists: other objects (such as names and external pointers) will be included as elements in a list result.

If there are several matrix arguments, they must all have the same number of columns (or rows) and this will be the number of columns (or rows) of the result. If all the arguments are vectors, the number of columns (rows) in the result is equal to the length of the longest vector. Values in shorter arguments are recycled to achieve this length (with a `warning` if they are recycled only *fractionally*).

When the arguments consist of a mix of matrices and vectors the number of columns (rows) of the result is determined by the number of columns (rows) of the matrix arguments. Any vectors have their values recycled or subsetted to achieve this length.

For `cbind` (`rbind`), vectors of zero length (including `NULL`) are ignored unless the result would have zero rows (columns), for S compatibility. (Zero-extent matrices do not occur in S3 and are not ignored in R.)

The `rbind` data frame method takes the classes of the columns from the first data frame, and matches columns by name (rather than by position). Factors have their levels expanded as necessary (in the order of the levels of the levelsets of the factors encountered) and the result is an ordered factor if and only if all the components were ordered factors. (The last point differs from S-PLUS.)

Value

A matrix or data frame combining the `...` arguments column-wise or row-wise. (Exception: if all the inputs are `NULL`, the value is `NULL`.)

For `cbind` (`rbind`) the column (row) names are taken from the `colnames` (`rownames`) of the arguments if these are matrix-like. Otherwise from the names of the arguments or where those are not supplied and `deparse.level > 0`, by deparsing the expressions given, for `deparse.level = 1` only if that gives a sensible name (a 'symbol', see `is.symbol`).

The names will depend on whether data frames are included: see the examples.

If a matrix is created, it will be a list if any of the inputs is not NULL nor an atomic vector. Otherwise its type is determined from the highest type of the inputs in the hierarchy NULL < raw < logical < integer < real < complex < character < list.

Note

The method dispatching is *not* done via `UseMethod()`, but by C-internal dispatching. Therefore, there is no need for, e.g., `rbind.default`.

The dispatch algorithm is described in the source file ('.../src/main/bind.c') as

1. For each argument we get the list of possible class memberships from the class attribute.
2. We inspect each class in turn to see if there is an an applicable method.
3. If we find an applicable method we make sure that it is identical to any method determined for prior arguments. If it is identical, we proceed, otherwise we immediately drop through to the default code.

If you want to combine other objects with data frames, it may be necessary to coerce them to data frames first. (Note that this algorithm can result in calling the data frame method if the arguments are all either data frames or vectors, and this will result in the coercion of character vectors to factors.)

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`c` to combine vectors (and lists) as vectors, `data.frame` to combine vectors and matrices as a data frame.

Examples

```
m <- cbind(1, 1:7) # the '1' (= shorter vector) is recycled
m
m <- cbind(m, 8:14)[, c(1, 3, 2)] # insert a column
m
cbind(1:7, diag(3))# vector is subset -> warning
```

```
cbind(0, rbind(1, 1:3))
cbind(I=0, X=rbind(a=1, b=1:3))  # use some names
xx <- data.frame(I=rep(0,2))
cbind(xx, X=rbind(a=1, b=1:3))   # named differently

cbind(0, matrix(1, nrow=0, ncol=4))#> Warning (making sense)
dim(cbind(0, matrix(1, nrow=2, ncol=0)))#-> 2 x 1

## deparse.level
dd <- 10
rbind(1:4, c=2, "a++" = 10, dd, deparse.level=0)# middle 2 rownames
rbind(1:4, c=2, "a++" = 10, dd, deparse.level=1)# 3 rownames (default)
rbind(1:4, c=2, "a++" = 10, dd, deparse.level=2)# 4 rownames
```

`char.expand` *Expand a String with Respect to a Target Table*

Description

Seeks a unique match of its first argument among the elements of its second. If successful, it returns this element; otherwise, it performs an action specified by the third argument.

Usage

```
char.expand(input, target, nomatch = stop("no match"))
```

Arguments

`input`	a character string to be expanded.
`target`	a character vector with the values to be matched against.
`nomatch`	an R expression to be evaluated in case expansion was not possible.

Details

This function is particularly useful when abbreviations are allowed in function arguments, and need to be uniquely expanded with respect to a target table of possible values.

See Also

`charmatch` and `pmatch` for performing partial string matching.

Examples

```
locPars <- c("mean", "median", "mode")
char.expand("me", locPars, warning("Could not expand!"))
char.expand("mo", locPars)
```

`character` *Character Vectors*

Description

Create or test for objects of type `"character"`.

Usage

```
character(length = 0)
as.character(x, ...)
is.character(x)
```

Arguments

`length`	desired length.
`x`	object to be coerced or tested.
`...`	further arguments passed to or from other methods.

Details

`as.character` and `is.character` are generic: you can write methods to handle specific classes of objects, see InternalMethods. Further, for `as.character` the default method calls `as.vector`, so dispatch is first on methods for `as.character` and then for methods for `as.vector`.

`as.character` represents real and complex numbers to 15 decimal places (technically the compiler's setting of the ISO C constant `DBL_DIG`, which will be 15 on machines supporting IEC60559 arithmetic according to the C99 standard). This ensures that all the digits in the result will be reliable (and not the result of representation error), but does mean that coversion to character and back to numeric may change the number. If you want to convert numbers to character with the maximum possible precision, use `format`.

Value

`character` creates a character vector of the specified length. The elements of the vector are all equal to `""`.

`as.character` attempts to coerce its argument to character type; like `as.vector` it strips attributes including names. For lists it deparses the elements individually, except that it extracts the first element of length-one character vectors.

`is.character` returns `TRUE` or `FALSE` depending on whether its argument is of character type or not. (Methods should always return `FALSE` for classes which are not based on a character vector, but can return `FALSE` even for those which are.)

Note

`as.character` truncates components of language objects to 500 characters (was about 70 before 1.3.1).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`paste`, `substr` and `strsplit` for character concatenation and splitting, `chartr` for character translation and casefolding (e.g., upper to lower case) and `sub`, `grep` etc for string matching and substitutions. Note that `help.search(keyword = "character")` gives even more links. `deparse`, which is normally preferable to `as.character` for language objects.

Examples

```
form <- y ~ a + b + c
as.character(form)  ## length 3
deparse(form)       ## like the input

a0 <- 11/999          # has a repeating decimal representation
(a1 <- as.character(a0))
format(a0, digits=16) # shows one more digit
a2 <- as.numeric(a1)
a2 - a0               # normally around -1e-17
as.character(a2)      # normally different from a1
print(c(a0, a2), digits = 16)
```

`charmatch` *Partial String Matching*

Description

`charmatch` seeks matches for the elements of its first argument among those of its second.

Usage

```
charmatch(x, table, nomatch = NA)
```

Arguments

`x`	the values to be matched: converte to a character vector by `as.character`.
`table`	the values to be matched against: converted to a character vector.
`nomatch`	the value to be returned at non-matching positions.

Details

Exact matches are preferred to partial matches (those where the value to be matched has an exact match to the initial part of the target, but the target is longer).

If there is a single exact match or no exact match and a unique partial match then the index of the matching value is returned; if multiple exact or multiple partial matches are found then `0` is returned and if no match is found then `nomatch` is returned.

`NA` values are treated as the string constant `"NA"`.

Author(s)

This function is based on a C function written by Terry Therneau.

See Also

pmatch, match.

grep or regexpr for more general (regexp) matching of strings.

Examples

```
charmatch("", "")                              # returns 1
charmatch("m",   c("mean", "median", "mode")) # returns 0
charmatch("med", c("mean", "median", "mode")) # returns 2
```

---

chartr *Character Translation and Casefolding*

Description

Translate characters in character vectors, in particular from upper to lower case or vice versa.

Usage

```
chartr(old, new, x)
tolower(x)
toupper(x)
casefold(x, upper = FALSE)
```

Arguments

x	a character vector, or an object that can be coerced to character by as.character.
old	a character string specifying the characters to be translated.
new	a character string specifying the translations.
upper	logical: translate to upper or lower case?.

Details

chartr translates each character in x that is specified in old to the corresponding character specified in new. Ranges are supported in the specifications, but character classes and repeated characters are not. If old contains more characters than new, an error is signaled; if it contains fewer characters, the extra characters at the end of new are ignored.

tolower and toupper convert upper-case characters in a character vector to lower-case, or vice versa. Non-alphabetic characters are left unchanged.

casefold is a wrapper for tolower and toupper provided for compatibility with S-PLUS.

Value

A character vector of the same length and with the same attributes as x (after possible coercion).

See Also

sub and gsub for other substitutions in strings.

Examples

```
x <- "MiXeD cAsE 123"
chartr("iXs", "why", x)
chartr("a-cX", "D-Fw", x)
tolower(x)
toupper(x)

## "Mixed Case" Capitalizing - toupper( every first letter of a word ) :

.simpleCap <- function(x) {
  s <- strsplit(x, " ")[[1]]
  paste(toupper(substring(s, 1,1)), substring(s, 2), sep="", collapse=" ")
}
.simpleCap("the quick red fox jumps over the lazy brown dog")
## ->  [1] "The Quick Red Fox Jumps Over The Lazy Brown Dog"

## and the better, more sophisticated version:
capwords <- function(s, strict = FALSE) {
    cap <- function(s) paste(toupper(substring(s,1,1)),
                  {s <- substring(s,2); if(strict) tolower(s) else s},
                             sep = "", collapse = " " )
    sapply(strsplit(s, split = " "), cap, USE.NAMES = !is.null(names(s)))
}
capwords(c("using AIC for model selection"))
## ->  [1] "Using AIC For Model Selection"
capwords(c("using AIC", "for MODEL selection"), strict=TRUE)
## ->  [1] "Using Aic"  "For Model Selection"
##                 ^^^        ^^^^^
##               'bad'       'good'
```

---

chol *The Choleski Decomposition*

Description

Compute the Choleski factorization of a real symmetric positive-definite square matrix.

Usage

```
chol(x, pivot = FALSE,  LINPACK = pivot)
```

Arguments

x	a real symmetric, positive-definite matrix
pivot	Should pivoting be used?
LINPACK	logical. Should LINPACK be used in the non-pivoting case (for compatibility with R < 1.7.0)?

Details

This is an interface to the LAPACK routine DPOTRF and the LINPACK routines DPOFA and DCHDC.

Note that only the upper triangular part of `x` is used, so that $R'R = x$ when `x` is symmetric.

If `pivot = FALSE` and `x` is not non-negative definite an error occurs. If `x` is positive semi-definite (i.e., some zero eigenvalues) an error will also occur, as a numerical tolerance is used.

If `pivot = TRUE`, then the Choleski decomposition of a positive semi-definite `x` can be computed. The rank of `x` is returned as `attr(Q, "rank")`, subject to numerical errors. The pivot is returned as `attr(Q, "pivot")`. It is no longer the case that `t(Q) %*% Q` equals `x`. However, setting `pivot <- attr(Q, "pivot")` and `oo <- order(pivot)`, it is true that `t(Q[, oo]) %*% Q[, oo]` equals `x`, or, alternatively, `t(Q) %*% Q` equals `x[pivot, pivot]`. See the examples.

Value

The upper triangular factor of the Choleski decomposition, i.e., the matrix R such that $R'R = x$ (see example).

If pivoting is used, then two additional attributes `"pivot"` and `"rank"` are also returned.

Warning

The code does not check for symmetry.

If `pivot = TRUE` and `x` is not non-negative definite then there will be a warning message but a meaningless result will occur. So only use `pivot = TRUE` when `x` is non-negative definite by construction.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Dongarra, J. J., Bunch, J. R., Moler, C. B. and Stewart, G. W. (1978) *LINPACK Users Guide.* Philadelphia: SIAM Publications.

Anderson. E. and ten others (1999) *LAPACK Users' Guide*. Third Edition. SIAM.
Available on-line at http://www.netlib.org/lapack/lug/lapack_lug.html.

See Also

`chol2inv` for its *inverse* (without pivoting), `backsolve` for solving linear systems with upper triangular left sides.

`qr`, `svd` for related matrix factorizations.

Examples

```
( m <- matrix(c(5,1,1,3),2,2) )
( cm <- chol(m) )
t(cm) %*% cm  #-- = 'm'
crossprod(cm)  #-- = 'm'

# now for something positive semi-definite
x <- matrix(c(1:5, (1:5)^2), 5, 2)
x <- cbind(x, x[, 1] + 3*x[, 2])
```

```
m <- crossprod(x)
qr(m)$rank # is 2, as it should be

# chol() may fail, depending on numerical rounding:
# chol() unlike qr() does not use a tolerance.
try(chol(m))

(Q <- chol(m, pivot = TRUE)) # NB wrong rank here ... see Warning section.
## we can use this by
pivot <- attr(Q, "pivot")
oo <- order(pivot)
t(Q[, oo]) %*% Q[, oo] # recover m

## now for a non-positive-definite matrix
( m <- matrix(c(5,-5,-5,3),2,2) )
try(chol(m))  # fails
try(chol(m, LINPACK=TRUE))   # fails
(Q <- chol(m, pivot = TRUE)) # warning
crossprod(Q)  # not equal to m
```

`chol2inv` *Inverse from Choleski Decomposition*

Description

Invert a symmetric, positive definite square matrix from its Choleski decomposition.

Usage

```
chol2inv(x, size = NCOL(x), LINPACK = FALSE)
```

Arguments

`x`	a matrix. The first `size` columns of the upper triangle contain the Choleski decomposition of the matrix to be inverted.
`size`	the number of columns of `x` containing the Choleski decomposition.
`LINPACK`	logical. Should LINPACK be used (for compatibility with R < 1.7.0)?

Details

This is an interface to the LAPACK routine DPOTRI and the LINPACK routine DPODI.

Value

The inverse of the matrix whose Choleski decomposition was given.

References

Dongarra, J. J., Bunch, J. R., Moler, C. B. and Stewart, G. W. (1978) *LINPACK Users Guide*. Philadelphia: SIAM Publications.

Anderson. E. and ten others (1999) *LAPACK Users' Guide*. Third Edition. SIAM. Available on-line at http://www.netlib.org/lapack/lug/lapack_lug.html.

See Also

chol, solve.

Examples

```
cma <- chol(ma  <- cbind(1, 1:3, c(1,3,7)))
ma %*% chol2inv(cma)
```

---

class *Object Classes*

Description

R possesses a simple generic function mechanism which can be used for an object-oriented style of programming. Method dispatch takes place based on the class of the first argument to the generic function.

Usage

```
class(x)
class(x) <- value
unclass(x)
inherits(x, what, which = FALSE)

oldClass(x)
oldClass(x) <- value
```

Arguments

x	a R object
what, value	a character vector naming classes. value can also be NULL.
which	logical affecting return value: see Details.

Details

Many R objects have a class attribute, a character vector giving the names of the classes which the object "inherits" from. If the object does not have a class attribute, it has an implicit class, "matrix", "array" or the result of mode(x). (Functions oldClass and oldClass<- get and set the attribute, which can also be done directly.)

When a generic function fun is applied to an object with class attribute c("first", "second"), the system searches for a function called fun.first and, if it finds it, applies it to the object. If no such function is found, a function called fun.second is tried. If no class name produces a suitable function, the function fun.default is used (if it exists). If there is no class attribute, the implicit class is tried, then the default method.

The function class prints the vector of names of classes an object inherits from. Correspondingly, class<- sets the classes an object inherits from. Assigning a zero-length vector or NULL removes the class attribute.

unclass returns (a copy of) its argument with its class attribute removed. (It is not allowed for objects which cannot be copied, namely environments and external pointers.)

`inherits` indicates whether its first argument inherits from any of the classes specified in the `what` argument. If `which` is `TRUE` then an integer vector of the same length as `what` is returned. Each element indicates the position in the `class(x)` matched by the element of `what`; zero indicates no match. If `which` is `FALSE` then `TRUE` is returned by `inherits` if any of the names in `what` match with any `class`.

Formal classes

An additional mechanism of *formal* classes is available in packages **methods** which is attached by default. For objects which have a formal class, its name is returned by `class` as a character vector of length one.

The replacement version of the function sets the class to the value provided. For classes that have a formal definition, directly replacing the class this way is strongly deprecated. The expression `as(object, value)` is the way to coerce an object to a particular class.

Note

Functions `oldClass` and `oldClass<-` behave in the same way as functions of those names in S-PLUS 5/6, *but* in R `UseMethod` dispatches on the class as returned by `class` (with some interpolated classes: see the link) rather than `oldClass`. *However*, group generics dispatch on the `oldClass` for efficiency.

See Also

`UseMethod`, `NextMethod`, `group generic`.

Examples

```
x <- 10
inherits(x, "a") #FALSE
class(x) <- c("a", "b")
inherits(x,"a") #TRUE
inherits(x, "a", TRUE) # 1
inherits(x, c("a", "b", "c"), TRUE) # 1 2 0
```

---

col *Column Indexes*

Description

Returns a matrix of integers indicating their column number in the matrix.

Usage

```
col(x, as.factor = FALSE)
```

Arguments

`x`	a matrix.
`as.factor`	a logical value indicating whether the value should be returned as a factor rather than as numeric.

Value

An integer matrix with the same dimensions as `x` and whose `ij`-th element is equal to `j`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`row` to get rows.

Examples

```
# extract an off-diagonal of a matrix
ma <- matrix(1:12, 3, 4)
ma[row(ma) == col(ma) + 1]

# create an identity 5-by-5 matrix
x <- matrix(0, nr = 5, nc = 5)
x[row(x) == col(x)] <- 1
```

`Colon` *Colon Operator*

Description

Generate regular sequences.

Usage

```
from:to
   a:b
```

Arguments

`from`	starting value of sequence.
`to`	(maximal) end value of the sequence.
`a, b`	`factor`s of same length.

Details

The binary operator `:` has two meanings: for factors `a:b` is equivalent to `interaction(a, b)` (but the levels are ordered and labelled differently).

For numeric arguments `from:to` is equivalent to `seq(from, to)`, and generates a sequence from `from` to `to` in steps of `1` or `1-`. Value `to` will be included if it differs from `from` by an integer up to a numeric fuzz of about `1e-7`.

Value

For numeric arguments, a numeric vector. This will be of type `integer` if `from` and `to` are both integers and representable in the integer type, otherwise of type `numeric`.

For factors, an unordered factor with levels labelled as `la:lb` and ordered lexicographically (that is, `lb` varies fastest).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.
(for numeric arguments: S does not have `:` for factors.)

See Also

`seq`.

As an alternative to using `:` for factors, `interaction`.

For `:` used in the formal representation of an interaction, see `formula`.

Examples

```
1:4
pi:6 # real
6:pi # integer

f1 <- gl(2,3); f1
f2 <- gl(3,2); f2
f1:f2 # a factor, the "cross"  f1 x f2
```

`colSums` *Form Row and Column Sums and Means*

Description

Form row and column sums and means for numeric arrays.

Usage

```
colSums (x, na.rm = FALSE, dims = 1)
rowSums (x, na.rm = FALSE, dims = 1)
colMeans(x, na.rm = FALSE, dims = 1)
rowMeans(x, na.rm = FALSE, dims = 1)
```

Arguments

`x`	an array of two or more dimensions, containing numeric, complex, integer or logical values, or a numeric data frame.
`na.rm`	logical. Should missing values (including `NaN`) be omitted from the calculations?
`dims`	Which dimensions are regarded as "rows" or "columns" to sum over. For `row*`, the sum or mean is over dimensions `dims+1, ...`; for `col*` it is over dimensions `1:dims`.

Details

These functions are equivalent to use of `apply` with `FUN = mean` or `FUN = sum` with appropriate margins, but are a lot faster. As they are written for speed, they blur over some of the subtleties of `NaN` and `NA`. If `na.rm = FALSE` and either `NaN` or `NA` appears in a sum, the result will be one of `NaN` or `NA`, but which might be platform-dependent.

Value

A numeric or complex array of suitable size, or a vector if the result is one-dimensional. The `dimnames` (or `names` for a vector result) are taken from the original array.

If there are no values in a range to be summed over (after removing missing values with `na.rm = TRUE`), that component of the output is set to `0` (`*Sums`) or `NA` (`*Means`), consistent with `sum` and `mean`.

See Also

`apply`, `rowsum`

Examples

```
## Compute row and column sums for a matrix:
x <- cbind(x1 = 3, x2 = c(4:1, 2:5))
rowSums(x); colSums(x)
dimnames(x)[[1]] <- letters[1:8]
rowSums(x); colSums(x); rowMeans(x); colMeans(x)
x[] <- as.integer(x)
rowSums(x); colSums(x)
x[] <- x < 3
rowSums(x); colSums(x)
x <- cbind(x1 = 3, x2 = c(4:1, 2:5))
x[3, ] <- NA; x[4, 2] <- NA
rowSums(x); colSums(x); rowMeans(x); colMeans(x)
rowSums(x, na.rm = TRUE); colSums(x, na.rm = TRUE)
rowMeans(x, na.rm = TRUE); colMeans(x, na.rm = TRUE)

## an array
dim(UCBAdmissions)
rowSums(UCBAdmissions); rowSums(UCBAdmissions, dims = 2)
colSums(UCBAdmissions); colSums(UCBAdmissions, dims = 2)

## complex case
x <- cbind(x1 = 3 + 2i, x2 = c(4:1, 2:5) - 5i)
x[3, ] <- NA; x[4, 2] <- NA
rowSums(x); colSums(x); rowMeans(x); colMeans(x)
rowSums(x, na.rm = TRUE); colSums(x, na.rm = TRUE)
rowMeans(x, na.rm = TRUE); colMeans(x, na.rm = TRUE)
```

`commandArgs` *Extract Command Line Arguments*

Description

Provides access to a copy of the command line arguments supplied when this R session was invoked.

Usage

```
commandArgs()
```

Details

These arguments are captured before the standard R command line processing takes place. This means that they are the unmodified values. If it were useful, we could provide support an argument which indicated whether we want the unprocessed or processed values.

This is especially useful with the `--args` command-line flag to R, as all of the command line after than flag is skipped.

Value

A character vector containing the name of the executable and the user-supplied command line arguments. The first element is the name of the executable by which R was invoked. As far as I am aware, the exact form of this element is platform dependent. It may be the fully qualified name, or simply the last component (or basename) of the application.

See Also

`Startup` `BATCH`

Examples

```
commandArgs()
## Spawn a copy of this application as it was invoked.
## system(paste(commandArgs(), collapse=" "))
```

---

comment *Query or Set a 'Comment' Attribute*

Description

These functions set and query a *comment* attribute for any R objects. This is typically useful for `data.frame`s or model fits.

Contrary to other `attributes`, the `comment` is not printed (by `print` or `print.default`).

Assigning `NULL` or a zero-length character vector removes the comment.

Usage

```
comment(x)
comment(x) <- value
```

Arguments

`x`	any R object
`value`	a `character` vector, or `NULL`.

See Also

`attributes` and `attr` for "normal" attributes.

Examples

```
x <- matrix(1:12, 3,4)
comment(x) <- c("This is my very important data from experiment #0234",
                "Jun 5, 1998")
x
comment(x)
```

`Comparison` *Relational Operators*

Description

Binary operators which allow the comparison of values in atomic vectors.

Usage

```
x < y
x > y
x <= y
x >= y
x == y
x != y
```

Arguments

`x, y`	atomic vectors, or other objects for which methods have been written.

Details

The binary comparison operators are generic functions: methods can be written for them individually or via the `Ops`) group generic function. (See `Ops` for how dispatch is computed.)

Comparison of strings in character vectors is lexicographic within the strings using the collating sequence of the locale in use: see `locales`. The collating sequence of locales such as '`en_US`' is normally different from '`C`' (which should use ASCII) and can be surprising.

At least one of `x` and `y` must be an atomic vector, but if the other is a list R attempts to coerce it to the type of the atomic vector: this will succeed if the list is made up of elements of length one that can be coerced to the correct type.

If the two arguments are atomic vectors of different types, one is coerced to the type of the other, the (decreasing) order of precedence being character, complex, numeric, integer, logical and raw.

When comparisons are made between character strings, parts of the strings after embedded `nul` characters are ignored. (This is necessary as the position of `nul` in the collation sequence is undefined, and we want one of <, == and > to be true for any comparison.)

Missing values (`NA`) and `NaN` values are regarded as non-comparable even to themselves, so comparisons involving them will always result in `NA`. Missing values can also result when character strings are compared and one is not valid in the current collation locale.

Value

A vector of logicals indicating the result of the element by element comparison. The elements of shorter vectors are recycled as necessary.

Objects such as arrays or time-series can be compared this way provided they are conformable.

Note

Do not use == and != for tests, such as in if expressions, where you must get a single TRUE or FALSE. Unless you are absolutely sure that nothing unusual can happen, you should use the identical function instead.

For numerical and complex values, remember == and != do not allow for the finite representation of fractions, nor for rounding error. Using all.equal with identical is almost always preferable. See the examples.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

factor for the behaviour with factor arguments.

Syntax for operator precedence.

Examples

```
x <- rnorm(20)
x < 1
x[x > 0]

x1 <- 0.5 - 0.3
x2 <- 0.3 - 0.1
x1 == x2                              # FALSE on most machines
identical(all.equal(x1, x2), TRUE) # TRUE everywhere
```

---

complex *Complex Vectors*

Description

Basic functions which support complex arithmetic in R.

Usage

```
complex(length.out = 0, real = numeric(), imaginary = numeric(),
        modulus = 1, argument = 0)
as.complex(x, ...)
is.complex(x)

Re(x)
Im(x)
Mod(x)
Arg(x)
Conj(x)
```

Arguments

`length.out`	numeric. Desired length of the output vector, inputs being recycled as needed.
`real`	numeric vector.
`imaginary`	numeric vector.
`modulus`	numeric vector.
`argument`	numeric vector.
`x`	an object, probably of mode `complex`.
`...`	further arguments passed to or from other methods.

Details

Complex vectors can be created with `complex`. The vector can be specified either by giving its length, its real and imaginary parts, or modulus and argument. (Giving just the length generates a vector of complex zeroes.)

`as.complex` attempts to coerce its argument to be of complex type: like `as.vector` it strips attributes including names.

Note that `is.complex` and `is.numeric` are never both `TRUE`.

The functions `Re`, `Im`, `Mod`, `Arg` and `Conj` have their usual interpretation as returning the real part, imaginary part, modulus, argument and complex conjugate for complex values. Modulus and argument are also called the *polar coordinates*. If $z = x + iy$ with real x and y, for $r = \texttt{Mod}(z) = \sqrt{x^2 + y^2}$, and $\phi = \texttt{Arg}(z)$, $x = r * \cos(\phi)$ and $y = r * \sin(\phi)$. They are all generic functions: methods can be defined for them individually or via the `Complex` group generic.

In addition, the elementary trigonometric, logarithmic and exponential functions are available for complex values.

`is.complex` is generic: you can write methods to handle specific classes of objects, see Internal-Methods.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
0i ^ (-3:3)

matrix(1i^ (-6:5), nr=4)#- all columns are the same
0 ^ 1i # a complex NaN

## create a complex normal vector
z <- complex(real = rnorm(100), imag = rnorm(100))
## or also (less efficiently):
z2 <- 1:2 + 1i*(8:9)

## The Arg(.) is an angle:
zz <- (rep(1:4,len=9) + 1i*(9:1))/10
zz.shift <- complex(modulus = Mod(zz), argument= Arg(zz) + pi)
plot(zz, xlim=c(-1,1), ylim=c(-1,1), col="red", asp = 1,
     main = expression(paste("Rotation by ","  ", pi == 180^o)))
abline(h=0,v=0, col="blue", lty=3)
points(zz.shift, col="orange")
```

`conditions` *Condition Handling and Recovery*

Description

These functions provide a mechanism for handling unusual conditions, including errors and warnings.

Usage

```
tryCatch(expr, ..., finally)
withCallingHandlers(expr, ...)

signalCondition(cond)

simpleCondition(message, call = NULL)
simpleError    (message, call = NULL)
simpleWarning  (message, call = NULL)
simpleMessage  (message, call = NULL)

## S3 method for class 'condition':
as.character(x, ...)
## S3 method for class 'error':
as.character(x, ...)
## S3 method for class 'condition':
print(x, ...)
## S3 method for class 'restart':
print(x, ...)

conditionCall(c)
## S3 method for class 'condition':
conditionCall(c)
conditionMessage(c)
## S3 method for class 'condition':
conditionMessage(c)

withRestarts(expr, ...)

computeRestarts(cond = NULL)
findRestart(name, cond = NULL)
invokeRestart(r, ...)
invokeRestartInteractively(r)

isRestart(x)
restartDescription(r)
restartFormals(r)

.signalSimpleWarning(msg, call)
.handleSimpleError(h, msg, call)
```

Arguments

`c`	a condition object.
`call`	call expression.
`cond`	a condition object.
`expr`	expression to be evaluated.
`finally`	expression to be evaluated before returning or exiting.
`h`	function.
`message`	character string.
`msg`	character string.
`name`	character string naming a restart.
`r`	restart object.
`x`	object.
`...`	additional arguments; see details below.

Details

The condition system provides a mechanism for signaling and handling unusual conditions, including errors and warnings. Conditions are represented as objects that contain information about the condition that occurred, such as a message and the call in which the condition occurred. Currently conditions are S3-style objects, though this may eventually change.

Conditions are objects inheriting from the abstract class `condition`. Errors and warnings are objects inheriting from the abstract subclasses `error` and `warning`. The class `simpleError` is the class used by `stop` and all internal error signals. Similarly, `simpleWarning` is used by `warning`, and `simpleMessage` is used by `message`. The constructors by the same names take a string describing the condition as argument and an optional call. The functions `conditionMessage` and `conditionCall` are generic functions that return the message and call of a condition.

Conditions are signaled by `signalCondition`. In addition, the `stop` and `warning` functions have been modified to also accept condition arguments.

The function `tryCatch` evaluates its expression argument in a context where the handlers provided in the `...` argument are available. The `finally` expression is then evaluated in the context in which `tryCatch` was called; that is, the handlers supplied to the current `tryCatch` call are not active when the `finally` expression is evaluated.

Handlers provided in the `...` argument to `tryCatch` are established for the duration of the evaluation of `expr`. If no condition is signaled when evaluating `expr` then `tryCatch` returns the value of the expression.

If a condition is signaled while evaluating `expr` then established handlers are checked, starting with the most recently established ones, for one matching the class of the condition. When several handlers are supplied in a single `tryCatch` then the first one is considered more recent than the second. If a handler is found then control is transferred to the `tryCatch` call that established the handler, the handler found and all more recent handlers are disestablished, the handler is called with the condition as its argument, and the result returned by the handler is returned as the value of the `tryCatch` call.

Calling handlers are established by `withCallingHandlers`. If a condition is signaled and the applicable handler is a calling handler, then the handler is called by `signalCondition` in the context where the condition was signaled but with the available handlers restricted to those below

the handler called in the handler stack. If the handler returns, then the next handler is tried; once the last handler has been tried, `signalCondition` returns `NULL`.

User interrupts signal a condition of class `interrupt` that inherits directly from class `condition` before executing the default interrupt action.

Restarts are used for establishing recovery protocols. They can be established using `withRestarts`. One pre-established restart is an `abort` restart that represents a jump to top level.

`findRestart` and `computeRestarts` find the available restarts. `findRestart` returns the most recently established restart of the specified name. `computeRestarts` returns a list of all restarts. Both can be given a condition argument and will then ignore restarts that do not apply to the condition.

`invokeRestart` transfers control to the point where the specified restart was established and calls the restart's handler with the arguments, if any, given as additional arguments to `invokeRestart`. The restart argument to `invokeRestart` can be a character string, in which case `findRestart` is used to find the restart.

New restarts for `withRestarts` can be specified in several ways. The simplest is in `name=function` form where the function is the handler to call when the restart is invoked. Another simple variant is as `name=string` where the string is stored in the `description` field of the restart object returned by `findRestart`; in this case the handler ignores its arguments and returns `NULL`. The most flexible form of a restart specification is as a list that can include several fields, including `handler`, `description`, and `test`. The `test` field should contain a function of one argument, a condition, that returns `TRUE` if the restart applies to the condition and `FALSE` if it does not; the default function returns `TRUE` for all conditions.

One additional field that can be specified for a restart is `interactive`. This should be a function of no arguments that returns a list of arguments to pass to the restart handler. The list could be obtained by interacting with the user if necessary. The function `invokeRestartInteractively` calls this function to obtain the arguments to use when invoking the restart. The default `interactive` method queries the user for values for the formal arguments of the handler function.

`.signalSimpleWarning` and `.handleSimpleError` are used internally and should not be called directly.

References

The `tryCatch` mechanism is similar to Java error handling. Calling handlers are based on Common Lisp and Dylan. Restarts are based on the Common Lisp restart mechanism.

See Also

`stop` and `warning` signal conditions, and `try` is essentially a simplified version of `tryCatch`.

Examples

```
tryCatch(1, finally=print("Hello"))
e <- simpleError("test error")
## Not run:
 stop(e)
 tryCatch(stop(e), finally=print("Hello"))
 tryCatch(stop("fred"), finally=print("Hello"))
## End(Not run)
tryCatch(stop(e), error = function(e) e, finally=print("Hello"))
tryCatch(stop("fred"),  error = function(e) e, finally=print("Hello"))
```

```
withCallingHandlers({ warning("A"); 1+2 }, warning = function(w) {})
{ try(invokeRestart("tryRestart")); 1}
## Not run:
 withRestarts(stop("A"), abort = function() {}); 1
## End(Not run)
withRestarts(invokeRestart("foo", 1, 2), foo = function(x, y) {x + y})
```

`conflicts` *Search for Masked Objects on the Search Path*

Description

`conflicts` reports on objects that exist with the same name in two or more places on the `search` path, usually because an object in the user's workspace or a package is masking a system object of the same name. This helps discover unintentional masking.

Usage

```
conflicts(where = search(), detail = FALSE)
```

Arguments

`where`	A subset of the search path, by default the whole search path.
`detail`	If `TRUE`, give the masked or masking functions for all members of the search path.

Value

If `detail=FALSE`, a character vector of masked objects. If `detail=TRUE`, a list of character vectors giving the masked or masking objects in that member of the search path. Empty vectors are omitted.

Examples

```
lm <- 1:3
conflicts(, TRUE)
## gives something like
# $.GlobalEnv
# [1] "lm"
#
# $package:base
# [1] "lm"

## Remove things from your "workspace" that mask others:
remove(list = conflicts(detail=TRUE)$.GlobalEnv)
```

connections *Functions to Manipulate Connections*

Description

Functions to create, open and close connections.

Usage

```
file(description = "", open = "", blocking = TRUE,
     encoding = getOption("encoding"))

pipe(description, open = "", encoding = getOption("encoding"))

fifo(description = "", open = "", blocking = FALSE,
     encoding = getOption("encoding"))

gzfile(description, open = "", encoding = getOption("encoding"),
       compression = 6)

unz(description, filename, open = "",
    encoding = getOption("encoding"))

bzfile(description, open = "", encoding = getOption("encoding"))

url(description, open = "", blocking = TRUE,
    encoding = getOption("encoding"))

socketConnection(host = "localhost", port, server = FALSE,
                 blocking = FALSE, open = "a+",
                 encoding = getOption("encoding"))

open(con, ...)
## S3 method for class 'connection':
open(con, open = "r", blocking = TRUE, ...)

close(con, ...)
## S3 method for class 'connection':
close(con, type = "rw", ...)

flush(con)

isOpen(con, rw = "")
isIncomplete(con)
```

Arguments

description	character. A description of the connection. For `file` and `pipe` this is a path to the file to be opened. For `url` it is a complete URL, including schemes (`http://`, `ftp://` or `file://` – see Details). `file` also accepts complete URLs.

filename	a filename within a zip file.
con	a connection.
host	character. Host name for port.
port	integer. The TCP port number.
server	logical. Should the socket be a client or a server?
open	character. A description of how to open the connection (if at all). See Details for possible values.
blocking	logical. See the 'Blocking' section below.
encoding	The name of the encoding to be used. See the 'Encoding' section below.
compression	integer in 0–9. The amount of compression to be applied when writing, from none to maximal. The default is a good space/time compromise.
type	character. Currently ignored.
rw	character. Empty or "read" or "write", partial matches allowed.
...	arguments passed to or from other methods.

Details

The first eight functions create connections. By default the connection is not opened (except for socketConnection), but may be opened by setting a non-empty value of argument open.

gzfile applies to files compressed by 'gzip', and bzfile to those compressed by 'bzip2'. gzfile can also open uncompressed files. In each case the actual file is opened in binary mode and so no translations are done if the original file was a text file.

unz reads (only) single files within zip files, in binary mode. The description is the full path, with '.zip' extension if required.

All platforms support file, gzfile, bzfile, unz and url("file://") connections. The other types may be partially implemented or not implemented at all. (They do work on most Unix platforms, and all but fifo on Windows.)

Proxies can be specified for url connections: see download.file.

open, close and seek are generic functions: the following applies to the methods relevant to connections.

open opens a connection. In general functions using connections will open them if they are not open, but then close them again, so to leave a connection open call open explicitly.

Possible values for the mode open to open a connection are

"r" or "rt" Open for reading in text mode.

"w" or "wt" Open for writing in text mode.

"a" or "at" Open for appending in text mode.

"rb" Open for reading in binary mode.

"wb" Open for writing in binary mode.

"ab" Open for appending in binary mode.

"r+", "r+b" Open for reading and writing.

"w+", "w+b" Open for reading and writing, truncating file initially.

"a+", "a+b" Open for reading and appending.

Not all modes are applicable to all connections: for example URLs can only be opened for reading. Only file and socket connections can be opened for reading and writing/appending. For many connections there is little or no difference between text and binary modes, but there is for file-like connections on Windows, and `pushBack` is text-oriented and is only allowed on connections open for reading in text mode.

`close` closes and destroys a connection.

`flush` flushes the output stream of a connection open for write/append (where implemented).

If for a `file` connection the description is `""`, the file is immediately opened (in `"w+"` mode unless `open="w+b"` is specified) and unlinked from the file system. This provides a temporary file to write to and then read from.

A note on `file://` URLs. The most general form (from RFC1738) is `file://host/path/to/file`, but R only accepts the form with an empty `host` field referring to the local machine. This is then `file:///path/to/file`, where `path/to/file` is relative to `/`. So although the third slash is strictly part of the specification not part of the path, this can be regarded as a way to specify the file '/path/to/file'. It is not possible to specify a relative path using a file URL. Also, no attempt is made to decode an encoded URL: call `URLdecode` if necessary.

Value

`file`, `pipe`, `fifo`, `url`, `gzfile`, `bzfile`, `unz` and `socketConnection` return a connection object which inherits from class `"connection"` and has a first more specific class.

`isOpen` returns a logical value, whether the connection is currently open.

`isIncomplete` returns a logical value, whether last read attempt was blocked, or for an output text connection whether there is unflushed output.

Encoding

The encoding of the input/output stream of a connection in *text* mode can be specified by name, in the same way as it would be given to `iconv`: see that help page for how to find out what names are recognized on your platform. Additionally, `""` and `"native.enc"` both mean the 'native' encoding, that is the internal encoding of the current locale and hence no translation is done. Not all builds of R support this, and if yours does not, specifying a non-default encoding will give an error when the connection is opened.

Re-encoding only works for connections in text mode.

The encoding `"UCS-2LE"` is treated specially, as it is the appropriate value for Windows 'Unicode' text files. If the first two bytes are the Byte Order Mark `0xFFFE` then these are removed as most implementations of `iconv` do not accept BOMs. Note that some implementations will handle BOMs using encoding `"UCS2"` but many will not.

Exactly what happens when the requested translation cannot be done is in general undocumented. Requesting a conversion that is not supported is an error, reported when the connection is opened. On output the result is likely to be that up to the error, with a warning. On input, it will most likely be all or some of the input up to the error.

The encoding for `stdin` when redirected from a file can be set by the command-line flag `--encoding`.

Blocking

The default condition for all but fifo and socket connections is to be in blocking mode. In that mode, functions do not return to the R evaluator until they are complete. In non-blocking mode,

operations return as soon as possible, so on input they will return with whatever input is available (possibly none) and for output they will return whether or not the write succeeded.

The function `readLines` behaves differently in respect of incomplete last lines in the two modes: see its help page.

Even when a connection is in blocking mode, attempts are made to ensure that it does not block the event loop and hence the operation of GUI parts of R. These do not always succeed, and the whole process will be blocked during a DNS lookup on Unix, for example.

Most blocking operations on URLs and sockets are subject to the timeout set by `options("timeout")`. Note that this is a timeout for no response at all, not for the whole operation.

Fifos

Fifos default to non-blocking. That follows Svr4 and it probably most natural, but it does have some implications. In particular, opening a non-blocking fifo connection for writing (only) will fail unless some other process is reading on the fifo.

Opening a fifo for both reading and writing (in any mode: one can only append to fifos) connects both sides of the fifo to the R process, and provides an similar facility to `file()`.

Clipboard

`file` can also be used with `description = "clipboard"` in mode `"r"` only. This reads the X11 primary selection (see `http://standards.freedesktop.org/clipboards-spec/clipboards-latest.txt`), which can also be specified as `"X11_primary"` and the secondary selection as `"X11_secondary"`.

When a clipboard is opened for reading, the contents are immediately copied to internal storage in the connection.

Unix users wishing to *write* to one of the selections may be able to do so via `xclip` (`http://people.debian.org/~kims/xclip/`), for example by `pipe("xclip -i", "w")` for the primary selection.

MacOS X users can use `pipe("pbpaste")` and `pipe("pbcopy", "w")` to read from and write to that system's clipboard.

Note

R's connections are modelled on those in S version 4 (see Chambers, 1998). However R goes well beyond the Svr4 model, for example in output text connections and URL, `gzfile`, `bzfile` and socket connections.

The default mode in R is `"r"` except for socket connections. This differs from Svr4, where it is the equivalent of `"r+"`, known as `"*"`.

On platforms where `vsnprintf` does not return the needed length of output there is a 100,000 character output limit on the length of line for `fifo`, `gzfile` and `bzfile` connections: longer lines will be truncated with a warning.

References

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language.* Springer.

See Also

textConnection, seek, readLines, readBin, writeLines, writeBin, showConnections, pushBack.

capabilities to see if url, fifo and socketConnection are supported by this build of R.

Examples

```
zz <- file("ex.data", "w")  # open an output file connection
cat("TITLE extra line", "2 3 5 7", "", "11 13 17", file = zz, sep = "\n")
cat("One more line\n", file = zz)
close(zz)
readLines("ex.data")
unlink("ex.data")

zz <- gzfile("ex.gz", "w")  # compressed file
cat("TITLE extra line", "2 3 5 7", "", "11 13 17", file = zz, sep = "\n")
close(zz)
readLines(zz <- gzfile("ex.gz"))
close(zz)
unlink("ex.gz")

zz <- bzfile("ex.bz2", "w")  # bzip2-ed file
cat("TITLE extra line", "2 3 5 7", "", "11 13 17", file = zz, sep = "\n")
close(zz)
print(readLines(zz <- bzfile("ex.bz2")))
close(zz)
unlink("ex.bz2")

## An example of a file open for reading and writing
Tfile <- file("test1", "w+")
c(isOpen(Tfile, "r"), isOpen(Tfile, "w")) # both TRUE
cat("abc\ndef\n", file=Tfile)
readLines(Tfile)
seek(Tfile, 0, rw="r") # reset to beginning
readLines(Tfile)
cat("ghi\n", file=Tfile)
readLines(Tfile)
close(Tfile)
unlink("test1")

## We can do the same thing with an anonymous file.
Tfile <- file()
cat("abc\ndef\n", file=Tfile)
readLines(Tfile)
close(Tfile)

if(capabilities("fifo")) {
  zz <- fifo("foo", "w+")
  writeLines("abc", zz)
  print(readLines(zz))
  close(zz)
  unlink("foo")
}

## Not run: ## Unix examples of use of pipes
```

```
# read listing of current directory
readLines(pipe("ls -1"))

# remove trailing commas. Suppose
% cat data2
450, 390, 467, 654,  30, 542, 334, 432, 421,
357, 497, 493, 550, 549, 467, 575, 578, 342,
446, 547, 534, 495, 979, 479
# Then read this by
scan(pipe("sed -e s/,$// data2"), sep=",")

# convert decimal point to comma in output
# both R strings and (probably) the shell need \ doubled
zz <- pipe(paste("sed s/\\\\./,/ >", "outfile"), "w")
cat(format(round(rnorm(100), 4)), sep = "\n", file = zz)
close(zz)
file.show("outfile", delete.file=TRUE)## End(Not run)

## Not run: ## example for Unix machine running a finger daemon

con <- socketConnection(port = 79, blocking = TRUE)
writeLines(paste(system("whoami", intern=TRUE), "\r", sep=""), con)
gsub(" *$", "", readLines(con))
close(con)## End(Not run)

## Not run: ## two R processes communicating via non-blocking sockets
# R process 1
con1 <- socketConnection(port = 6011, server=TRUE)
writeLines(LETTERS, con1)
close(con1)

# R process 2
con2 <- socketConnection(Sys.info()["nodename"], port = 6011)
# as non-blocking, may need to loop for input
readLines(con2)
while(isIncomplete(con2)) {Sys.sleep(1); readLines(con2)}
close(con2)
## End(Not run)

## Not run:
 ## examples of use of encodings
cat(x, file = file("foo", "w", encoding="UTF-8"))
# read a 'Windows Unicode' file including names
A <- read.table(file("students", encoding="UCS-2LE"))
## End(Not run)
```

`Constants` *Built-in Constants*

Description

Constants built into R.

Usage

```
LETTERS
letters
month.abb
month.name
pi
```

Details

R has a limited number of built-in constants (there is also a rather larger library of data sets which can be loaded with the function `data`).

The following constants are available:

- `LETTERS`: the 26 upper-case letters of the Roman alphabet;
- `letters`: the 26 lower-case letters of the Roman alphabet;
- `month.abb`: the three-letter abbreviations for the English month names;
- `month.name`: the English names for the months of the year;
- `pi`: the ratio of the circumference of a circle to its diameter.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`data`, `DateTimeClasses`.

`Quotes` for the parsing of character constants, `NumericConstants` for numeric constants.

Examples

```
# John Machin (1705) computed 100 decimals of pi :
pi - 4*(4*atan(1/5) - atan(1/239))

## months in English
month.name
## months in your current locale
format(ISOdate(2000, 1:12, 1), "%B")
format(ISOdate(2000, 1:12, 1), "%b")
```

`contributors` *R Project Contributors*

Description

The R Who-is-who, describing who made significant contributions to the development of R.

Usage

```
contributors()
```

`Control` *Control Flow*

Description

These are the basic control-flow constructs of the R language. They function in much the same way as control statements in any Algol-like language.

Usage

```
if(cond) expr
if(cond) cons.expr  else  alt.expr

for(var in seq) expr
while(cond) expr
repeat expr
break
next
```

Arguments

`cond`	A length-one logical vector that is not `NA`. Conditions of length greater than one are accepted with a warning, but only the first element is used.
`var`	A syntactical name for a variable.
`seq`	An expression evaluating to a vector (including a list and an expression) or to a pairlist or `NULL`.
`expr, cons.expr, alt.expr`	An *expression* in a formal sense. This is either a simple expression or a so called *compound expression*, usually of the form `{ expr1 ; expr2 }`.

Details

`break` breaks out of a `for`, `while` or `repeat` loop; control is transferred to the first statement outside the inner-most loop. `next` halts the processing of the current iteration and advances the looping index. Both `break` and `next` apply only to the innermost of nested loops.

Note that it is a common mistake to forget to put braces (`{ .. }`) around your statements, e.g., after `if(..)` or `for(....)`. In particular, you should not have a newline between `}` and `else` to avoid a syntax error in entering a `if ... else` construct at the keyboard or via `source`. For that reason, one (somewhat extreme) attitude of defensive programming is to always use braces, e.g., for `if` clauses.

The index `seq` in a `for` loop is evaluated at the start of the loop; changing it subsequently does not affect the loop. The variable `var` has the same type as `seq`, and is read-only: assigning to it does not alter `seq`. If `seq` is a factor (which is not strictly allowed) then its internal codes are used: the effect is that of `as.integer` not `as.vector`.

Value

`if` returns the value of the expression evaluated, or `NULL` if none was (which may happen if there is no `else`).

`for`, `while` and `repeat` return the value of the last expression evaluated (or `NULL` if none was), invisibly. `for` sets `var` to the last used element of `seq`, or to `NULL` if it was of length zero.

`break` and `next` have value `NULL`, although it would be strange to look for a return value.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`Syntax` for the basic R syntax and operators, `Paren` for parentheses and braces; further, `ifelse`, `switch`.

Examples

```
for(i in 1:5) print(1:i)
for(n in c(2,5,10,20,50)) {
   x <- rnorm(n)
   cat(n,":", sum(x^2),"\n")
}
```

---

copyright *Copyrights of Files Used to Build R*

Description

R is released under the 'GNU Public License': see `license` for details. The license describes your right to use R. Copyright is concerned with ownership of intellectual rights, and some of the software used has conditions that the copyright must be explicitly stated: see the Details section. We are grateful to these people and other contributors (see `contributors`) for the ability to use their work.

Details

The file '$R_HOME/COPYRIGHTS' lists the copyrights in full detail.

---

count.fields *Count the Number of Fields per Line*

Description

`count.fields` counts the number of fields, as separated by `sep`, in each of the lines of `file` read.

Usage

```
count.fields(file, sep = "", quote = "\"'", skip = 0,
             blank.lines.skip = TRUE, comment.char = "#")
```

Arguments

file	a character string naming an ASCII data file, or a connection, which will be opened if necessary, and if so closed at the end of the function call.
sep	the field separator character. Values on each line of the file are separated by this character. By default, arbitrary amounts of whitespace can separate fields.
quote	the set of quoting characters
skip	the number of lines of the data file to skip before beginning to read data.
blank.lines.skip	logical: if TRUE blank lines in the input are ignored.
comment.char	character: a character vector of length one containing a single character or an empty string.

Details

This used to be used by read.table and can still be useful in discovering problems in reading a file by that function.

For the handling of comments, see scan.

Value

A vector with the numbers of fields found.

See Also

read.table

Examples

```
cat("NAME", "1:John", "2:Paul", file = "foo", sep = "\n")
count.fields("foo", sep = ":")
unlink("foo")
```

---

crossprod *Matrix Crossproduct*

Description

Given matrices x and y as arguments, return a matrix cross-product. This is formally equivalent to (but faster than) the call t(x) %*% y (crossprod) or x %*% t(y) (tcrossprod).

Usage

```
crossprod(x, y = NULL)

tcrossprod(x, y = NULL)
```

Arguments

x, y	matrices: y = NULL is taken to be the same matrix as x. Vectors are promoted to single-column matrices.

Note

When `x` or `y` are not matrices, they are treated as column or row matrices, but their `names` are usually **not** promoted to `dimnames`. Hence, currently, the last example has empty dimnames.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`%*%` and outer product `%o%`.

Examples

```
(z <- crossprod(1:4))     # = sum(1 + 2^2 + 3^2 + 4^2)
drop(z)                   # scalar
x <- 1:4; names(x) <- letters[1:4]; x
tcrossprod(as.matrix(x)) # is
identical(tcrossprod(as.matrix(x)),
          crossprod(t(x)))
tcrossprod(x)             # no dimnames
```

Cstack_info *Report Information on C Stack Size and Usage*

Description

Report information on the C stack size and usage (if available).

Usage

```
Cstack_info()
```

Details

On most platforms, C stack information is recorded when R is initialized and used for stack-checking. If this information is unavailable, the `size` will be returned as `NA`, and stack-checking is not performed.

The information on the stack base address is thought to be accurate on Windows, Linux and FreeBSD (including MacOS X), but a heuristic is used on other platforms. Because this might be slightly inaccurate, the current usage could be estimated as negative. (The heuristic is not used on embedded uses of R on platforms where the stack base is not thought to be accurate.)

Value

An integer vector. This has named elements

`size`	The size of the stack (in bytes), or `NA` if unknown.
`current`	The estimated current usage (in bytes), possibly `NA`.
`direction`	`1` (stack grows down, the usual case) or `-1` (stack grows up).
`eval_depth`	The current evaluation depth (including two calls for the call to `Cstack_info`).

Examples

```
Cstack_info()
```

`cumsum` *Cumulative Sums, Products, and Extremes*

Description

Returns a vector whose elements are the cumulative sums, products, minima or maxima of the elements of the argument.

Usage

```
cumsum(x)
cumprod(x)
cummax(x)
cummin(x)
```

Arguments

`x`	a numeric or complex (not `cummin` or `cummax`) object, or an object that can be coerced to one of these.

Details

These are generic functions: methods can be defined for them individually or via the `Math` group generic.

Value

A vector of the same length and type as `x` (after coercion), except that `cumprod` returns a numeric vector for integer input (for consistency with `*`). Names are preserved.

An `NA` value in `x` causes the corresponding and following elements of the return value to be `NA`, as does integer overflow in `cumsum` (with a warning).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (`cumsum` only.)

Examples

```
cumsum(1:10)
cumprod(1:10)
cummin(c(3:1, 2:0, 4:2))
cummax(c(3:1, 2:0, 4:2))
```

`cut` *Convert Numeric to Factor*

Description

`cut` divides the range of `x` into intervals and codes the values in `x` according to which interval they fall. The leftmost interval corresponds to level one, the next leftmost to level two and so on.

Usage

```
cut(x, ...)

## Default S3 method:
cut(x, breaks, labels = NULL,
    include.lowest = FALSE, right = TRUE, dig.lab = 3, ...)
```

Arguments

`x`	a numeric vector which is to be converted to a factor by cutting.
`breaks`	either a numeric vector of cut points or number giving the number of intervals which `x` is to be cut into.
`labels`	labels for the levels of the resulting category. By default, labels are constructed using `"(a,b]"` interval notation. If `labels = FALSE`, simple integer codes are returned instead of a factor.
`include.lowest`	logical, indicating if an 'x[i]' equal to the lowest (or highest, for `right = FALSE`) 'breaks' value should be included.
`right`	logical, indicating if the intervals should be closed on the right (and open on the left) or vice versa.
`dig.lab`	integer which is used when labels are not given. It determines the number of digits used in formatting the break numbers.
`...`	further arguments passed to or from other methods.

Details

If a `labels` parameter is specified, its values are used to name the factor levels. If none is specified, the factor level labels are constructed as `"(b1, b2]"`, `"(b2, b3]"` etc. for `right = TRUE` and as `"[b1, b2)"`, ...if `right = FALSE`. In this case, `dig.lab` indicates the minimum number of digits should be used in formatting the numbers `b1`, `b2`, ... A larger value (up to 12) will be used if needed to distinguish between any pair of endpoints: if this fails labels such as `"Range3"` will be used.

Value

A `factor` is returned, unless `labels = FALSE` which results in the mere integer level codes.

Note

Instead of `table(cut(x, br))`, `hist(x, br, plot = FALSE)` is more efficient and less memory hungry. Instead of `cut(*, labels = FALSE)`, `findInterval()` is more efficient.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`split` for splitting a variable according to a group factor; `factor`, `tabulate`, `table`, `findInterval()`.

Examples

```
Z <- rnorm(10000)
table(cut(Z, br = -6:6))
sum(table(cut(Z, br = -6:6, labels=FALSE)))
sum(   hist  (Z, br = -6:6, plot=FALSE)$counts)

cut(rep(1,5),4)#-- dummy
tx0 <- c(9, 4, 6, 5, 3, 10, 5, 3, 5)
x <- rep(0:8, tx0)
stopifnot(table(x) == tx0)

table( cut(x, b = 8))
table( cut(x, br = 3*(-2:5)))
table( cut(x, br = 3*(-2:5), right = FALSE))

##--- some values OUTSIDE the breaks :
table(cx  <- cut(x, br = 2*(0:4)))
table(cxl <- cut(x, br = 2*(0:4), right = FALSE))
which(is.na(cx));  x[is.na(cx)]  #-- the first 9  values  0
which(is.na(cxl)); x[is.na(cxl)] #-- the last  5  values  8

## Label construction:
y <- rnorm(100)
table(cut(y, breaks = pi/3*(-3:3)))
table(cut(y, breaks = pi/3*(-3:3), dig.lab=4))

table(cut(y, breaks =  1*(-3:3), dig.lab=4))
# extra digits don't "harm" here
table(cut(y, breaks =  1*(-3:3), right = FALSE))
#- the same, since no exact INT!

## sometimes the default dig.lab is not enough to be avoid confusion:
aaa <- c(1,2,3,4,5,2,3,4,5,6,7)
cut(aaa, 3)
cut(aaa, 3, dig.lab=4)
```

`cut.POSIXt` *Convert a Date or Date-Time Object to a Factor*

Description

Method for `cut` applied to date-time objects.

Usage

```
## S3 method for class 'POSIXt':
cut(x, breaks, labels = NULL, start.on.monday = TRUE,
    right = FALSE, ...)

## S3 method for class 'Date':
cut(x, breaks, labels = NULL, start.on.monday = TRUE,
    right = FALSE, ...)
```

Arguments

`x`	an object inheriting from class `"POSIXt"` or `"Date"`.
`breaks`	a vector of cut points *or* number giving the number of intervals which `x` is to be cut into *or* an interval specification, one of `"sec"`, `"min"`, `"hour"`, `"day"`, `"DSTday"`, `"week"`, `"month"` or `"year"`, optionally preceded by an integer and a space, or followed by `"s"`. For `"Date"` objects only `"day"`, `"week"`, `"month"` and `"year"` are allowed.
`labels`	labels for the levels of the resulting category. By default, labels are constructed from the left-hand end of the intervals (which are include for the default value of `right`). If `labels = FALSE`, simple integer codes are returned instead of a factor.
`start.on.monday`	logical. If `breaks = "weeks"`, should the week start on Mondays or Sundays?
`right, ...`	arguments to be passed to or from other methods.

Details

Using both `right = TRUE` and `include.lowest = TRUE` will include both ends of the range of dates.

Value

A factor is returned, unless `labels = FALSE` which returns the integer level codes.

See Also

`seq.POSIXt`, `seq.Date`, `cut`

Examples

```
## random dates in a 10-week period
cut(ISOdate(2001, 1, 1) + 70*86400*runif(100), "weeks")
cut(as.Date("2001/1/1") + 70*runif(100), "weeks")
```

`data.class` *Object Classes*

Description

Determine the class of an arbitrary R object.

Usage

```
data.class(x)
```

Arguments

`x`	an R object.

Value

character string giving the "class" of `x`.

The "class" is the (first element) of the `class` attribute if this is non-`NULL`, or inferred from the object's `dim` attribute if this is non-`NULL`, or `mode(x)`.

Simply speaking, `data.class(x)` returns what is typically useful for method dispatching. (Or, what the basic creator functions already and maybe eventually all will attach as a class attribute.)

Note

For compatibility reasons, there is one exception to the rule above: When `x` is `integer`, the result of `data.class(x)` is `"numeric"` even when `x` is classed.

See Also

`class`

Examples

```
x <- LETTERS
data.class(factor(x))                 # has a class attribute
data.class(matrix(x, nc = 13))        # has a dim attribute
data.class(list(x))                   # the same as mode(x)
data.class(x)                         # the same as mode(x)

stopifnot(data.class(1:2) == "numeric")# compatibility "rule"
```

`data.frame` *Data Frames*

Description

This function creates data frames, tightly coupled collections of variables which share many of the properties of matrices and of lists, used as the fundamental data structure by most of R's modeling software.

Usage

```
data.frame(..., row.names = NULL, check.rows = FALSE,
           check.names = TRUE,
           stringsAsFactors = default.stringsAsFactors())

default.stringsAsFactors()
```

Arguments

`...`	these arguments are of either the form `value` or `tag = value`. Component names are created based on the tag (if present) or the deparsed argument itself.
`row.names`	`NULL` or a single integer or character string specifying a column to be used as row names, or a character or integer vector giving the row names for the data frame.
`check.rows`	if `TRUE` then the rows are checked for consistency of length and names.
`check.names`	logical. If `TRUE` then the names of the variables in the data frame are checked to ensure that they are syntactically valid variable names and are not duplicated. If necessary they are adjusted (by `make.names`) so that they are.
`stringsAsFactors`	logical: should character vectors be converted to factors?

Details

A data frame is a list of variables of the same length with unique row names, given class `"data.frame"`.

`data.frame` converts each of its arguments to a data frame by calling `as.data.frame(optional=TRUE)`. As that is a generic function, methods can be written to change the behaviour of arguments according to their classes: R comes with many such methods. Character variables passed to `data.frame` are converted to factor columns unless protected by `I`. If a list or data frame or matrix is passed to `data.frame` it is as if each component or column had been passed as a separate argument.

Objects passed to `data.frame` should have the same number of rows, but atomic vectors, factors and character vectors protected by `I` will be recycled a whole number of times if necessary.

If row names are not supplied in the call to `data.frame`, the row names are taken from the first component that has suitable names, for example a named vector or a matrix with rownames or a data frame. (If that component is subsequently recycled, the names are discarded with a warning.) If `row.names` was supplied as `NULL` or no suitable component was found the row names are the integer sequence starting at one.

If row names are supplied of length one and the data frame has a single row, the `row.names` is taken to specify the row names and not a column (by name or number).

Names are removed from vector inputs not protected by `I`.

`default.stringsAsFactors` is a utility that takes `getOption("stringsAsFactors")` and ensures the result is `TRUE` or `FALSE`.

Value

A data frame, a matrix-like structure whose columns may be of differing types (numeric, logical, factor and character and so on).

Note

In versions of R prior to 2.4.0 `row.names` had to be character: to ensure compatibility with earlier versions of R, supply a character vector as the `row.names` argument.

References

Chambers, J. M. (1992) *Data for models.* Chapter 3 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`I`, `plot.data.frame`, `print.data.frame`, `row.names`, `names` (for the column names), `[.data.frame` for subsetting methods, `Math.data.frame` etc, about *Group* methods for `data.frames`; `read.table`, `make.names`.

Examples

```
L3 <- LETTERS[1:3]
(d <- data.frame(cbind(x=1, y=1:10), fac=sample(L3, 10, repl=TRUE)))

## The same with automatic column names:
data.frame(cbind(  1,   1:10),     sample(L3, 10, repl=TRUE))

is.data.frame(d)

## do not convert to factor, using I() :
(dd <- cbind(d, char = I(letters[1:10])))
rbind(class=sapply(dd, class), mode=sapply(dd, mode))

stopifnot(1:10 == row.names(d))# {coercion}

(d0  <- d[, FALSE]) # NULL data frame with 10 rows
(d.0 <- d[FALSE, ]) # <0 rows> data frame  (3 cols)
(d00 <- d0[FALSE,])  # NULL data frame with 0 rows
```

`data.matrix` *Data Frame to Numeric Matrix*

Description

Return the matrix obtained by converting all the variables in a data frame to numeric mode and then binding them together as the columns of a matrix. Factors and ordered factors are replaced by their internal codes.

Usage

```
data.matrix(frame)
```

Arguments

`frame`	a data frame whose components are logical vectors, factors or numeric vectors.

Details

Supplying a data frame with columns which are not numeric, factor or logical is an error. A warning is given if any non-factor column has a class, as then information can be lost.

Value

If `frame` is a data frame, a numeric matrix of the same dimensions as `frame`, with dimnames taken from the row.names and names.

Otherwise, the result of `as.matrix`.

References

Chambers, J. M. (1992) *Data for models.* Chapter 3 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`as.matrix`, `data.frame`, `matrix`.

Examples

```
DF <- data.frame(a=1:3, b=letters[10:12],
                 c=seq(as.Date("2004-01-01"), by = "week", len = 3),
                 stringsAsFactors = TRUE)
data.matrix(DF[1:2])
data.matrix(DF) # gives a warning and quotes dates as #days since 1970.
```

`date` *System Date and Time*

Description

Returns a character string of the current system date and time.

Usage

```
date()
```

Value

The string has the form `"Fri Aug 20 11:11:00 1999"`, i.e., length 24, since it relies on POSIX's `ctime` ensuring the above fixed format. Timezone and Daylight Saving Time are taken account of, but *not* indicated in the result.

The day and month abbreviations are always in English, irrespective of locale.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`Sys.time`; `Date` and `DateTimeClasses` for objects representing date and time.

Examples

```
(d <- date())
nchar(d) == 24

## something similar in the current locale
format(Sys.time(), "%a %b %d %H:%M:%S %Y")
```

`Dates` *Date Class*

Description

Description of the class `"Date"` representing calendar dates.

Usage

```
## S3 method for class 'Date':
summary(object, digits = 12, ...)
```

Arguments

object	An object summarized.
digits	Number of significant digits for the computations.
...	Further arguments to be passed from or to other methods.

Details

Dates are represented as the number of days since 1970-01-01, with negative values for earlier dates. They are always printed following the rules of the current Gregorian calendar, even though that calendar was not in use long ago (it was adopted in 1752 in Great Britain and its colonies).

It is intended that the date should be an integer, but this is not enforced in the internal representation. Fractional days will be ignored when printing. It is possible to produce fractional days via the mean method or by adding or subtracting (see Ops.Date).

See Also

Sys.Date for the current date.

Ops.Date for operators on "Date" objects.

format.Date for conversion to and from character strings.

plot.Date and hist.Date for plotting.

weekdays for convenience extraction functions.

seq.Date, cut.Date, round.Date for utility operations.

DateTimeClasses for date-time classes.

Examples

```
(today <- Sys.Date())
format(today, "%d %b %Y")  # with month as a word
(tenweeks <- seq(today, len=10, by="1 week")) # next ten weeks
weekdays(today)
months(tenweeks)
as.Date(.leap.seconds)
```

DateTimeClasses *Date-Time Classes*

Description

Description of the classes "POSIXlt" and "POSIXct" representing calendar dates and times (to the nearest second).

Usage

```
## S3 method for class 'POSIXct':
print(x, ...)

## S3 method for class 'POSIXct':
summary(object, digits = 15, ...)
```

```
time + z
time - z
time1 lop time2
```

Arguments

x, object	An object to be printed or summarized from one of the date-time classes.
digits	Number of significant digits for the computations: should be high enough to represent the least important time unit exactly.
...	Further arguments to be passed from or to other methods.
time	date-time objects
time1, time2	date-time objects or character vectors. (Character vectors are converted by as.POSIXct.
z	a numeric vector (in seconds) *or* an object of class "difftime".
lop	One of ==, !=, <, <=, > or >=.

Details

There are two basic classes of date/times. Class "POSIXct" represents the (signed) number of seconds since the beginning of 1970 as a numeric vector. Class "POSIXlt" is a named list of vectors representing

sec 0–61: seconds

min 0–59: minutes

hour 0–23: hours

mday 1–31: day of the month

mon 0–11: months after the first of the year.

year Years since 1900.

wday 0–6 day of the week, starting on Sunday.

yday 0–365: day of the year.

isdst Daylight savings time flag. Positive if in force, zero if not, negative if unknown.

The classes correspond to the ANSI C constructs of "calendar time" (the time_t data type) and "local time" (or broken-down time, the struct tm data type), from which they also inherit their names.

"POSIXct" is more convenient for including in data frames, and "POSIXlt" is closer to human-readable forms. A virtual class "POSIXt" inherits from both of the classes: it is used to allow operations such as subtraction to mix the two classes.

Logical comparisons and limited arithmetic are available for both classes. One can add or subtract a number of seconds or a difftime object from a date-time object, but not add two date-time objects. Subtraction of two date-time objects is equivalent to using difftime. Be aware that "POSIXlt" objects will be interpreted as being in the current timezone for these operations, unless a timezone has been specified.

"POSIXlt" objects will often have an attribute "tzone", a character vector of length 3 giving the timezone name from the TZ environment variable and the names of the base timezone and the alternate (daylight-saving) timezone. Sometimes this may just be of length one, giving the timezone name.

"POSIXct" objects may also have an attribute "tzone", a character vector of length one. If set, it will determine how the object is converted to class "POSIXlt" and in particular how it is printed. This is usually desirable, but if you want to specify an object in a particular timezone but to be printed in the current timezone you may want to remove the "tzone" attribute (e.g. by c(x)).

Unfortunately, the conversion is complicated by the operation of time zones and leap seconds (23 days have been 86401 seconds long so far: the times of the extra seconds are in the object .leap.seconds). The details of this are entrusted to the OS services where possible. This will usually cover the period 1970–2037, and on Unix machines back to 1902 (when time zones were in their infancy). Outside those ranges we use our own C code. This uses the offset from GMT in use in the timezone in one of 1970 to 1978 (chosen so that the likely DST transition days are Sundays), and uses the alternate (daylight-saving) timezone only if isdst is positive.

It seems that some systems use leap seconds but most do not. This is detected and corrected for at build time, so all "POSIXct" times used by R do not include leap seconds. (Conceivably this could be wrong if the system has changed since build time, just possibly by changing locales.)

Using c on "POSIXlt" objects converts them to the current time zone.

Sub-section Accuracy

Classes "POSIXct" and "POSIXlt" are able to express fractions of a second. (Conversion of fractions between the two forms may not be exact, but will have better than microsecond accuracy.)

Fractional seconds are printed only if options("digits.secs") is set: see strftime.

Warning

Some Unix-like systems (especially Linux ones) do not have "TZ" set, yet have internal code that expects it (as does POSIX). We have tried to work around this, but if you get unexpected results try setting "TZ". See as.POSIXlt for valid settings.

See Also

Dates for dates without times.

as.POSIXct and as.POSIXlt for conversion between the classes.

strptime for conversion to and from character representations.

Sys.time for clock time as a "POSIXct" object.

difftime for time intervals.

cut.POSIXt, seq.POSIXt, round.POSIXt and trunc.POSIXt for methods for these classes.

weekdays.POSIXt for convenience extraction functions.

Examples

```
(z <- Sys.time())             # the current date, as class "POSIXct"

Sys.time() - 3600             # an hour ago

as.POSIXlt(Sys.time(), "GMT") # the current time in GMT
format(.leap.seconds)         # all 23 leapseconds in your timezone
print(.leap.seconds, tz="PST8PDT")  # and in Seattle's
```

`dcf` *Read and Write Data in DCF Format*

Description

Reads or writes an R object from/to a file in Debian Control File format.

Usage

```
read.dcf(file, fields=NULL)
write.dcf(x, file = "", append = FALSE,
          indent = 0.1 * getOption("width"),
          width = 0.9 * getOption("width"))
```

Arguments

`file`	either a character string naming a file or a connection. `""` indicates output to the console. For `read.dcf` this can name a `gzip`-compressed file.
`fields`	Fields to read from the DCF file. Default is to read all fields.
`x`	the object to be written, typically a data frame. If not, it is attempted to coerce `x` to a data frame.
`append`	logical. If `TRUE`, the output is appended to the file. If `FALSE`, any existing file of the name is destroyed.
`indent`	a positive integer specifying the indentation for continuation lines in output entries.
`width`	a positive integer giving the target column for wrapping lines in the output.

Details

DCF is a simple format for storing databases in plain text files that can easily be directly read and written by humans. DCF is used in various places to store R system information, like descriptions and contents of packages.

The DCF rules as implemented in R are:

1. A database consists of one or more records, each with one or more named fields. Not every record must contain each field, a field may appear only once in a record.
2. Regular lines start with a non-whitespace character.
3. Regular lines are of form `tag:value`, i.e., have a name tag and a value for the field, separated by `:` (only the first `:` counts). The value can be empty (=whitespace only).
4. Lines starting with whitespace are continuation lines (to the preceding field) if at least one character in the line is non-whitespace.
5. Records are separated by one or more empty (=whitespace only) lines.

`read.dcf` returns a character matrix with one line per record and one column per field. Leading and trailing whitespace of field values is ignored. If a tag name is specified, but the corresponding value is empty, then an empty string of length 0 is returned. If the tag name of a fields is never used in a record, then `NA` is returned. If there are multiple records with the same tag name, the last one encountered is returned. Malformed lines are ignored (with a warning).

See Also

write.table.

Examples

```
## Create a reduced version of the 'CONTENTS' file in package 'splines'
x <- read.dcf(file = system.file("CONTENTS", package = "splines"),
              fields = c("Entry", "Description"))
write.dcf(x)
```

---

debug *Debug a Function*

Description

Set or unset the debugging flag on a function.

Usage

```
debug(fun)
undebug(fun)
```

Arguments

fun	any interpreted R function.

Details

When a function flagged for debugging is entered, normal execution is suspended and the body of function is executed one statement at a time. A new browser context is initiated for each step (and the previous one destroyed).

At the debug prompt the user can enter commands or R expressions. The commands are

n (or just return). Advance to the next step.

c continue to the end of the current context: e.g. to the end of the loop if within a loop or to the end of the function.

cont synonym for c.

where print a stack trace of all active function calls.

Q exit the browser and the current evaluation and return to the top-level prompt.

(Leading and trailing whitespace is ignored, except for return).

Anything else entered at the debug prompt is interpreted as an R expression to be evaluated in the calling environment: in particular typing an object name will cause the object to be printed, and ls() lists the objects in the calling frame. (If you want to look at an object with a name such as n, print it explicitly.)

If a function is defined inside a function, single-step though to the end of its definition, and then call debug on its name.

In order to debug S4 methods (see Methods), you need to use trace, typically calling browser, e.g., as

```
trace("plot", browser, exit=browser, signature = c("track",
"missing"))
```

See Also

`browser`, `trace`; `traceback` to see the stack after an `Error: ... ` message; `recover` for another debugging approach.

`Defunct` *Marking Objects as Defunct*

Description

When an object is removed from R it should be replaced by a call to `.Defunct`.

Usage

```
.Defunct(new, package = NULL)
```

Arguments

`new`	character string: A suggestion for a replacement function.
`package`	character string: The package to be used when suggesting where the defunct function might be listed.

Details

`.Defunct` is called from defunct functions. Functions should be listed in `help("pkg-defunct")` for an appropriate `pkg`, including `base`.

See Also

`Deprecated`.

`base-defunct` and so on which list the defunct functions in the packages.

`delayedAssign` *Delay Evaluation*

Description

`delayedAssign` creates a *promise* to evaluate the given expression if its value is requested. This provides direct access to the *lazy evaluation* mechanism used by R for the evaluation of (interpreted) functions.

Usage

```
delayedAssign(x, value, eval.env = parent.frame(1),
              assign.env = parent.frame(1))
```

Arguments

x	a variable name (given as a quoted string in the function call)
value	an expression to be assigned to x
eval.env	an environment in which to evaluate value
assign.env	an environment in which to assign x

Details

Both eval.env and assign.env default to the currently active environment.

The expression assigned to a promise by delayedAssign will not be evaluated until it is eventually "forced". This happens when the variable is first accessed.

When the promise is eventually forced, it is evaluated within the environment specified by eval.env (whose contents may have changed in the meantime). After that, the value is fixed and the expression will not be evaluated again.

This function is meant to replace the delay() function, to make it more difficult for R code to see "naked" promises.

Value

This function is invoked for its side effect, which is assigning a promise to evaluate value to the variable x.

See Also

substitute, to see the expression associated with a promise.

Examples

```
msg <- "old"
delayedAssign("x", msg)
msg <- "new!"
x #- new!
substitute(x) #- msg

delayedAssign("x", {
    for(i in 1:3)
        cat("yippee!\n")
    10
})

x^2 #- yippee
x^2 #- simple number

e <- (function(x, y = 1, z) environment())(1+2, "y", {cat(" HO! "); pi+2})
(le <- as.list(e)) # evaluates the promises
```

`deparse` *Expression Deparsing*

Description

Turn unevaluated expressions into character strings.

Usage

```
deparse(expr, width.cutoff = 60,
        backtick = mode(expr) %in% c("call", "expression", "("),
        control = "showAttributes")
```

Arguments

`expr`	any R expression.
`width.cutoff`	integer in $[20, 500]$ determining the cutoff at which line-breaking is tried.
`backtick`	logical indicating whether symbolic names should be enclosed in backticks if they do not follow the standard syntax.
`control`	character vector of deparsing options. See `.deparseOpts`.

Details

This function turns unevaluated expressions (where "expression" is taken in a wider sense than the strict concept of a vector of mode `"expression"` used in `expression`) into character strings (a kind of inverse `parse`).

A typical use of this is to create informative labels for data sets and plots. The example shows a simple use of this facility. It uses the functions `deparse` and `substitute` to create labels for a plot which are character string versions of the actual arguments to the function `myplot`.

The default for the `backtick` option is not to quote single symbols but only composite expressions. This is a compromise to avoid breaking existing code.

Using `control = "all"` comes closest to making `deparse()` an inverse of `parse()`. However, not all objects are deparseable even with this option and a warning will be issued if the function recognizes that it is being asked to do the impossible.

Note

To avoid the risk of a source attribute out of sync with the actual function definition, the source attribute of a function will never be deparsed as an attribute.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`substitute`, `parse`, `expression`.

`Quotes` for quoting conventions, including backticks.

Examples

```
require(stats)
deparse(args(lm))
deparse(args(lm), width = 500)
myplot <-
function(x, y) {
    plot(x, y, xlab=deparse(substitute(x)),
        ylab=deparse(substitute(y)))
}
e <- quote(`foo bar`)
deparse(e)
deparse(e, backtick=TRUE)
e <- quote(`foo bar`+1)
deparse(e)
deparse(e, control = "all")
```

---

deparseOpts *Options for Expression Deparsing*

Description

Process the deparsing options for `deparse`, `dput` and `dump`.

Usage

```
.deparseOpts(control)
```

Arguments

`control`	character vector of deparsing options.

Details

This is called by `deparse`, `dput` and `dump` to process their `control` argument.

The `control` argument is a vector containing zero or more of the following strings. Partial string matching is used.

`keepInteger` Surround integer vectors by `as.integer()`, so they are not converted to floating point when re-parsed.

`quoteExpressions` Surround expressions with `quote()`, so they are not evaluated when re-parsed.

`showAttributes` If the object has attributes (other than a `source` attribute), use `structure()` to display them as well as the object value. This is the default for `deparse` and `dput`.

`useSource` If the object has a `source` attribute, display that instead of deparsing the object. Currently only applies to function definitions.

`warnIncomplete` Some exotic objects such as environments, external pointers, etc. can not be deparsed properly. This option causes a warning to be issued if any of those may give problems.

`all` An abbreviated way to specify all of the options listed above. May not be used with other options. This is the default for `dump`.

delayPromises Deparse promises in the form <promise: expression> rather than evaluating them. The value and the environment of the promise will not be shown and the deparsed code cannot be sourced.

For the most readable (but perhaps incomplete) display, use `control = NULL`. This displays the object's value, but not its attributes. The default is to display the attributes as well, but not to use any of the other options to make the result parseable.

Using `control = "all"` comes closest to making `deparse()` an inverse of `parse()`. However, not all objects are deparseable even with this option. A warning will be issued if the function recognizes that it is being asked to do the impossible.

Value

A numerical value corresponding to the options selected.

`Deprecated` *Marking Objects as Deprecated*

Description

When an object is about removed from R it is first deprecated and should include a call to `.Deprecated`.

Usage

```
.Deprecated(new, package=NULL)
```

Arguments

`new`	character string: A suggestion for a replacement function.
`package`	character string: The package to be used when suggesting where the deprecated function might be listed.

Details

`.Deprecated("<new name>")` is called from deprecated functions. The original help page for these functions is often available at `help("oldName-deprecated")` (note the quotes). Functions should be listed in `help("pkg-deprecated")` for an appropriate `pkg`, including `base`.

See Also

`Defunct`

`base-deprecated` and so on which list the deprecated functions in the packages.

det *Calculate the Determinant of a Matrix*

Description

det calculates the determinant of a matrix. determinant is a generic function that returns separately the modulus of the determinant, optionally on the logarithm scale, and the sign of the determinant.

Usage

```
det(x, ...)
determinant(x, logarithm = TRUE, ...)
```

Arguments

x	numeric matrix.
logarithm	logical; if TRUE (default) return the logarithm of the modulus of the determinant.
...	Optional arguments. At present none are used. Previous versions of det allowed an optional method argument. This argument will be ignored but will not produce an error.

Details

The determinant function uses an LU decomposition and the det function is simply a wrapper around a call to determinant.

Often, computing the determinant is *not* what you should be doing to solve a given problem.

Value

For det, the determinant of x. For determinant, a list with components

modulus	a numeric value. The modulus (absolute value) of the determinant if logarithm is FALSE; otherwise the logarithm of the modulus.
sign	integer; either $+1$ or -1 according to whether the determinant is positive or negative.

Examples

```
(x <- matrix(1:4, ncol=2))
unlist(determinant(x))
det(x)

det(print(cbind(1,1:3,c(2,0,1))))
```

detach *Detach Objects from the Search Path*

Description

Detach a database, i.e., remove it from the search() path of available R objects. Usually, this is either a data.frame which has been attached or a package which was required previously.

Usage

```
detach(name, pos = 2, version)
```

Arguments

name	The object to detach. Defaults to search()[pos]. This can be an unquoted name or a character string but *not* a character vector. If a number is supplied this is taken as pos.
pos	Index position in search() of database to detach. When name is a number, pos = name is used.
version	A character string denoting a version number of the package to be removed. This should be used only with a versioned installation of the package: see library.

Details

This most commonly used with a single number argument referring to a position on the search list, and can also be used with a unquoted or quoted name of an item on the search list such as package:tools.

When a package have been loaded with an explicit version number it can be detached using the name shown by search or by supplying name and version: see the examples.

Value

The attached database is returned invisibly, either as data.frame or as list.

Note

You cannot detach either the workspace (position 1) or the **base** package (the last item in the search list).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

attach, library, search, objects.

Examples

```
require(splines)#package
detach(package:splines)
## could equally well use detach("package:splines")
## but NOT  pkg <- "package:splines"; detach(pkg)
## Instead, use
library(splines)
pkg <- "package:splines"
detach(pos = match(pkg, search()))

## careful: do not do this unless 'splines' is not already loaded.
library(splines)
detach(2)# 'pos' used for 'name'

## an example of the name argument to attach
## and of detaching a database named by a character vector
attach_and_detach <- function(db, pos=2)
{
   name <- deparse(substitute(db))
   attach(db, pos=pos, name=name)
   print(search()[pos])
   eval(substitute(detach(n), list(n=name)))
}
attach_and_detach(women, pos=3)

## Not run:
## Using a versioned install
library(ash, version="1.0-9")  # or perhaps just library(ash)
# then one of
detach("package:ash", version="1.0-9")
# or
detach("package:ash_1.0-9")
## End(Not run)
```

---

diag *Matrix Diagonals*

Description

Extract or replace the diagonal of a matrix, or construct a diagonal matrix.

Usage

```
diag(x = 1, nrow, ncol = n)
diag(x) <- value
```

Arguments

`x`	a matrix, vector or 1D array.
`nrow, ncol`	Optional dimensions for the result.
`value`	either a single value or a vector of length equal to that of the current diagonal. Should be of a mode which can be coerced to that of `x`.

Value

If `x` is a matrix then `diag(x)` returns the diagonal of `x`. The resulting vector will have `names` if the matrix `x` has matching column and row names.

If `x` is a vector (or 1D array) of length two or more, then `diag(x)` returns a diagonal matrix whose diagonal is `x`.

If `x` is a vector of length one then `diag(x)` returns an identity matrix of order the nearest integer to `x`. The dimension of the returned matrix can be specified by `nrow` and `ncol` (the default is square).

The replacement form sets the diagonal of the matrix `x` to the given value(s).

Note

Using `diag(x)` can have unexpected effects if `x` is a vector that could be of length one. Use `diag(x, nrow = length(x))` for consistent behaviour.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`upper.tri`, `lower.tri`, `matrix`.

Examples

```
require(stats)
dim(diag(3))
diag(10,3,4) # guess what?
all(diag(1:3) == {m <- matrix(0,3,3); diag(m) <- 1:3; m})

diag(var(M <- cbind(X=1:5, Y=rnorm(5))))#-> vector with names "X" and "Y"
rownames(M) <- c(colnames(M),rep("",3));
M; diag(M) #  named as well
```

`diff` *Lagged Differences*

Description

Returns suitably lagged and iterated differences.

Usage

```
diff(x, ...)

## Default S3 method:
diff(x, lag = 1, differences = 1, ...)

## S3 method for class 'POSIXt':
diff(x, lag = 1, differences = 1, ...)
```

```
## S3 method for class 'Date':
diff(x, lag = 1, differences = 1, ...)
```

Arguments

x	a numeric vector or matrix containing the values to be differenced.
lag	an integer indicating which lag to use.
differences	an integer indicating the order of the difference.
...	further arguments to be passed to or from methods.

Details

diff is a generic function with a default method and ones for classes "ts", "POSIXt" and "Date".

NA's propagate.

Value

If x is a vector of length n and differences=1, then the computed result is equal to the successive differences x[(1+lag):n] - x[1:(n-lag)].

If difference is larger than one this algorithm is applied recursively to x. Note that the returned value is a vector which is shorter than x.

If x is a matrix then the difference operations are carried out on each column separately.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

diff.ts, diffinv.

Examples

```
diff(1:10, 2)
diff(1:10, 2, 2)
x <- cumsum(cumsum(1:10))
diff(x, lag = 2)
diff(x, differences = 2)

diff(.leap.seconds)
```

`difftime` *Time Intervals*

Description

Create, print and round time intervals.

Usage

```
time1 - time2

difftime(time1, time2, tz = "",
         units = c("auto", "secs", "mins", "hours", "days", "weeks"))

as.difftime(tim, format = "%X")

## S3 method for class 'difftime':
round(x, digits = 0)
```

Arguments

`time1, time2`	date-time or date objects.
`tz`	a timezone specification to be used for the conversion. System-specific, but `""` is the current time zone, and `"GMT"` is UTC.
`units`	character. Units in which the results are desired. Can be abbreviated.
`tim`	character string specifying a time interval.
`format`	character specifying the format of `tim`: see `strptime`. The default is a locale-specific time format.
`x`	an object inheriting from class `"difftime"`.
`digits`	integer. Number of significant digits to retain.

Details

Function `difftime` takes a difference of two date/time objects and returns an object of class `"difftime"` with an attribute indicating the units. There is a `round` method for objects of this class, as well as methods for the group-generic (see `Ops`) logical and arithmetic operations.

If `units = "auto"`, a suitable set of units is chosen, the largest possible (excluding `"weeks"`) in which all the absolute differences are greater than one.

Subtraction of date-time objects gives an object of this class, by calling `difftime` with `units="auto"`. Alternatively, `as.difftime()` works on character-coded time intervals.

Limited arithmetic is available on `"difftime"` objects: they can be added or subtracted, and multiplied or divided by a numeric vector. In addition, adding or subtracting a numeric vector implicitly converts the numeric vector to a `"difftime"` object with the same units as the `"difftime"` object.

See Also

`DateTimeClasses`.

Examples

```
(z <- Sys.time() - 3600)
Sys.time() - z                # just over 3600 seconds.

## time interval between releases of 1.2.2 and 1.2.3.
ISOdate(2001, 4, 26) - ISOdate(2001, 2, 26)

as.difftime(c("0:3:20", "11:23:15"))
as.difftime(c("3:20", "23:15", "2:"), format= "%H:%M")# 3rd gives NA
```

---

dim *Dimensions of an Object*

Description

Retrieve or set the dimension of an object.

Usage

```
dim(x)
dim(x) <- value
```

Arguments

x	an R object, for example a matrix, array or data frame.
value	For the default method, either NULL or a numeric vector which coerced to integer (by truncation).

Details

The functions dim and dim<- are generic.

dim has a method for data.frames, which returns the length of the row.names attribute of x and the length of x (the numbers of "rows" and "columns").

Value

For an array (and hence in particular, for a matrix) dim retrieves the dim attribute of the object. It is NULL or a vector of mode integer.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

ncol, nrow and dimnames.

Examples

```
x <- 1:12 ; dim(x) <- c(3,4)
x

# simple versions of nrow and ncol could be defined as follows
nrow0 <- function(x) dim(x)[1]
ncol0 <- function(x) dim(x)[2]
```

---

dimnames *Dimnames of an Object*

Description

Retrieve or set the dimnames of an object.

Usage

```
dimnames(x)
dimnames(x) <- value
```

Arguments

x	an R object, for example a matrix, array or data frame.
value	a possible value for dimnames(x): see "Value".

Details

The functions dimnames and dimnames<- are generic.

For an array (and hence in particular, for a matrix), they retrieve or set the dimnames attribute (see attributes) of the object. A list value can have names, and these will be used to label the dimensions of the array where appropriate.

The replacement method for arrays/matrices coerces vector and factor elements of value to character, but does not dispatch methods for as.character. It coerces zero-length elements to NULL.

Both have methods for data frames. The dimnames of a data frame are its row.names and its names. For the replacement method each component of value will be coerced by as.character.

For a 1D matrix the names are the same thing as the (only) component of the dimnames.

Value

The dimnames of a matrix or array can be NULL or a list of the same length as dim(x). If a list, its components are either NULL or a character vector with positive length of the appropriate dimension of x.

For the "data.frame" method both dimnames are character vectors, and the rownames must contain no duplicates nor missing values.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

rownames, colnames; array, matrix, data.frame.

Examples

```
## simple versions of rownames and colnames
## could be defined as follows
rownames0 <- function(x) dimnames(x)[[1]]
colnames0 <- function(x) dimnames(x)[[2]]
```

---

do.call *Execute a Function Call*

Description

do.call constructs and executes a function call from a name or a function and a list of arguments to be passed to it.

Usage

```
do.call(what, args, quote = FALSE, envir = parent.frame())
```

Arguments

what	either a function or a character string naming the function to be called.
args	a *list* of arguments to the function call. The names attribute of args gives the argument names.
quote	a logical value indicating whether to quote the arguments.
envir	an environment within which to evaluate the call. This will be most useful if what is a character string and the arguments are symbols or quoted expressions.

Details

If quote is FALSE, the default, then the arguments are evaluated (in the calling environment, not envir.). If quote is TRUE then each argument is quoted (see quote) so that the effect of argument evaluation is to remove the quote – leaving the original argument unevaluated when the call is constructed.

The behavior of some functions, such as substitute, will not be the same for functions evaluated using do.call as if they were evaluated from the interpreter. The precise semantics are currently undefined and subject to change.

Value

The result of the (evaluated) function call.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`call` which creates an unevaluated call.

Examples

```
do.call("complex", list(imag = 1:3))

## if we already have a list (e.g. a data frame)
## we need c() to add further arguments
tmp <- expand.grid(letters[1:2], 1:3, c("+", "-"))
do.call("paste", c(tmp, sep=""))

do.call(paste, list(as.name("A"), as.name("B")), quote=TRUE)

## examples of where objects will be found.
A <- 2
f <- function(x) print(x^2)
env <- new.env()
assign("A", 10, envir = env)
assign("f", f, envir = env)
f <- function(x) print(x)
f(A)                                    # 2
do.call("f", list(A))                   # 2
do.call("f", list(A), envir=env)        # 4
do.call(f, list(A), envir=env)          # 2
do.call("f", list(quote(A)), envir=env) # 100
do.call(f, list(quote(A)), envir=env)   # 10
do.call("f", list(as.name("A")), envir=env) # 100

eval(call("f", A))                      # 2
eval(call("f", quote(A)))               # 2
eval(call("f", A), envir=env)           # 4
eval(call("f", quote(A)), envir=env)    # 100
```

`double` *Double Precision Vectors*

Description

Create, coerce to or test for a double-precision vector.

Usage

```
double(length = 0)
as.double(x, ...)
is.double(x)

single(length = 0)
as.single(x, ...)
```

Arguments

`length`	desired length.
`x`	object to be coerced or tested.
`...`	further arguments passed to or from other methods.

Value

`double` creates a double precision vector of the specified length. The elements of the vector are all equal to `0`.

`as.double` attempts to coerce its argument to be of double type: like `as.vector` it strips attributes including names. Character strings containing either a decimal representation or a heaxadecimal representation (starting with `0x` or `0X`) can be converted.

`is.double` returns `TRUE` or `FALSE` depending on whether its argument is of double type or not. It is generic: you can write methods to handle specific classes of objects, see InternalMethods.

Note

R *has no single precision data type. All real numbers are stored in double precision format.* The functions `as.single` and `single` are identical to `as.double` and `double` except they set the attribute `Csingle` that is used in the `.C` and `.Fortran` interface, and they are intended only to be used in that context.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`integer`, `numeric`.

Examples

```
is.double(1)
all(double(3) == 0)
```

---

dput *Write an Internal Object to a File*

Description

Writes an ASCII text representation of an R object to a file or connection, or uses one to recreate the object.

Usage

```
dput(x, file = "", control = "showAttributes")
dget(file)
```

Arguments

x	an object.
file	either a character string naming a file or a connection. "" indicates output to the console.
control	character vector indicating deparsing options. See .deparseOpts for their description.

Details

dput opens file and deparses the object x into that file. The object name is not written (contrary to dump). If x is a function the associated environment is stripped. Hence scoping information can be lost.

Deparsing an object is difficult, and not always possible. With the default control = c("showAttributes"), dput() attempts to deparse in a way that is readable, but for more complex or unusual objects, not likely to be parsed as identical to the original. Use control = "all" for the most complete deparsing; use control = NULL for the simplest deparsing, not even including attributes.

dput will warn if fewer characters were written to a file than expected, which may indicate a full or corrupt file system.

To display saved source rather than deparsing the internal representation include "useSource" in control. R currently saves source only for function definitions.

Note

To avoid the risk of a source attribute out of sync with the actual function definition, the source attribute of a function will never be written as an attribute.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

deparse, dump, write.

Examples

```
## Write an ASCII version of mean to the file "foo"
dput(mean, "foo")
## And read it back into 'bar'
bar <- dget("foo")
unlink("foo")
## Create a function with comments
baz <- function(x) {
  # Subtract from one
  1-x
}
## and display it
dput(baz)
## and now display the saved source
dput(baz, control = "useSource")
```

drop *Drop Redundant Extent Information*

Description

Delete the dimensions of an array which have only one level.

Usage

```
drop(x)
```

Arguments

x	an array (including a matrix).

Value

If x is an object with a dim attribute (e.g., a matrix or array), then drop returns an object like x, but with any extents of length one removed. Any accompanying dimnames attribute is adjusted and returned with x.

Array subsetting ([) performs this reduction unless used with drop = FALSE, but sometimes it is useful to invoke drop directly.

See Also

drop1 which is used for dropping terms in models.

Examples

```
dim(drop(array(1:12, dim=c(1,3,1,1,2,1,2))))# = 3 2 2
drop(1:3 %*% 2:4)# scalar product
```

dump *Text Representations of R Objects*

Description

This function takes a vector of names of R objects and produces text representations of the objects on a file or connection. A dump file can usually be sourced into another R (or S) session.

Usage

```
dump(list, file = "dumpdata.R", append = FALSE,
     control = "all", envir = parent.frame(), evaluate = TRUE)
```

Arguments

list	character. The names of one or more R objects to be dumped.
file	either a character string naming a file or a connection. "" indicates output to the console.
append	if TRUE, output will be appended to file; otherwise, it will overwrite the contents of file.
control	character vector indicating deparsing options. See .deparseOpts for their description.
envir	the environment to search for objects.
evaluate	logical. Should promises be evaluated?

Details

If some of the objects named do not exist (in scope), they are omitted, with a warning. If file is a file and no objects exist then no file is created.

At present sourceing may not produce an identical copy of dumped objects. A warning is issued if it is likely that problems will arise, for example when dumping exotic objects such as environments and external pointers.

dump will also warn if fewer characters were written to a file than expected, which may indicate a full or corrupt file system.

A dump file can be sourced into another R (or perhaps S) session, but the function save is designed to be used for transporting R data, and will work with R objects that dump does not handle.

To produce a more readable representation of an object, use control = NULL. This will skip attributes, and will make other simplifications that make source less likely to produce an identical copy. See deparse for details.

To deparse the internal function representation rather than displaying the saved source, use control = c("keepInteger", "quoteExpressions", "showAttributes", "warnIncomplete"). This will lose all formatting and comments, but may be useful in those cases where the saved source is no longer correct.

Promises will normally only be encountered by users as a result of lazy-loading (when the default evaluate = TRUE is essential) and after the use of delayedAssign, when evaluate = FALSE might be intended.

Value

An invisible character vector containing the names of the objects which were dumped.

Note

As dump is defined in the base namespace, the **base** package will be searched *before* the global environment unless dump is called from the top level or the envir argument is given explicitly.

To avoid the risk of a source attribute out of sync with the actual function definition, the source attribute of a function will never be dumped as an attribute.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

dput, dget, write.
save for a more reliable way to save R objects.

Examples

```
x <- 1; y <- 1:10
dump(ls(patt='^[xyz]'), "xyz.Rdmped")
print(.Last.value)
unlink("xyz.Rdmped")
```

---

duplicated *Determine Duplicate Elements*

Description

Determines which elements of a vector or data frame are duplicates of elements with smaller subscripts, and returns a logical vector indicating which elements (rows) are duplicates.

Usage

```
duplicated(x, incomparables = FALSE, ...)

## S3 method for class 'array':
duplicated(x, incomparables = FALSE, MARGIN = 1, ...)
```

Arguments

x	a vector or a data frame or an array or NULL.
incomparables	a vector of values that cannot be compared. Currently, FALSE is the only possible value, meaning that all values can be compared.
...	arguments for particular methods.
MARGIN	the array margin to be held fixed: see apply.

Details

This is a generic function with methods for vectors (including lists), data frames and arrays (including matrices).

The data frame method works by pasting together a character representation of the rows separated by \r, so may be imperfect if the data frame has characters with embedded carriage returns or columns which do not reliably map to characters.

The array method calculates for each element of the sub-array specified by MARGIN if the remaining dimensions are identical to those for an earlier element (in row-major order). This would most commonly be used to find duplicated rows (the default) or columns (with MARGIN = 2).

Warning

Using this for lists is potentially slow, especially if the elements are not atomic vectors (see vector) or differ only in their attributes. In the worst case it is $O(n^2)$.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`unique`.

Examples

```
x <- c(9:20, 1:5, 3:7, 0:8)
## extract unique elements
(xu <- x[!duplicated(x)])
## xu == unique(x) but unique(x) is more efficient

duplicated(iris)[140:143]

duplicated(iris3, MARGIN = c(1, 3))
```

`dyn.load` *Foreign Function Interface*

Description

Load or unload shared libraries, and test whether a C function or Fortran subroutine is available.

Usage

```
dyn.load(x, local = TRUE, now = TRUE)
dyn.unload(x)

is.loaded(symbol, PACKAGE = "", type = "")
```

Arguments

`x`	a character string giving the pathname to a shared library or DLL.
`local`	a logical value controlling whether the symbols in the shared library are stored in their own local table and not shared across shared libraries, or added to the global symbol table. Whether this has any effect is system-dependent.
`now`	a logical controlling whether all symbols are resolved (and relocated) immediately the library is loaded or deferred until they are used. This control is useful for developers testing whether a library is complete and has all the necessary symbols, and for users to ignore missing symbols. Whether this has any effect is system-dependent.
`symbol`	a character string giving a symbol name.
`PACKAGE`	if supplied, confine the search for the `name` to the DLL given by this argument (plus the conventional extension, `.so`, `.sl`, `.dll`, ...). This is intended to add safety for packages, which can ensure by using this argument that no other package can override their external symbols. Use `PACKAGE="base"` for symbols linked in to R. This is used in the same way as in `.C`, `.Call`, `.Fortran` and `.External` functions

`type` The type of symbol to look for: can be any ("", the default), "Fortran", "Call" or "External".

Details

See 'See Also' and the *Writing R Extensions* and *R Installation and Administration* manuals for how to create and install a suitable shared library. Note that unlike some versions of S-PLUS, `dyn.load` does not load an object ('.o') file but a shared library or DLL.

Unfortunately a very few platforms (Compaq Tru64) do not handle the `PACKAGE` argument correctly, and may incorrectly find symbols linked into R.

The additional arguments to `dyn.load` mirror the different aspects of the mode argument to the dlopen() routine on UNIX systems. They are available so that users can exercise greater control over the loading process for an individual library. In general, the defaults values are appropriate and you should override them only if there is good reason and you understand the implications.

The `local` argument allows one to control whether the symbols in the DLL being attached are visible to other DLLs. While maintaining the symbols in their own namespace is good practice, the ability to share symbols across related "chapters" is useful in many cases. Additionally, on certain platforms and versions of an operating system, certain libraries must have their symbols loaded globally to successfully resolve all symbols.

One should be careful of the potential side-effect of using lazy loading via the `now` argument as `FALSE`. If a routine is called that has a missing symbol, the process will terminate immediately and unsaved session variables will be lost. The intended use is for library developers to call specify a value `TRUE` to check that all symbols are actually resolved and for regular users to all with `FALSE` so that missing symbols can be ignored and the available ones can be called.

The initial motivation for adding these was to avoid such termination in the `_init()` routines of the Java virtual machine library. However, symbols loaded locally may not be (read probably) available to other DLLs. Those added to the global table are available to all other elements of the application and so can be shared across two different DLLs.

Some systems do not provide (explicit) support for local/global and lazy/eager symbol resolution. This can be the source of subtle bugs. One can arrange to have warning messages emitted when unsupported options are used. This is done by setting either of the options `verbose` or `warn` to be non-zero via the `options` function. Currently, we know of only 2 platforms that do not provide a value for local load (RTLD_LOCAL). These are IRIX6.4 and unpatched versions of Solaris 2.5.1.

There is a short discussion of these additional arguments with some example code available at http://cm.bell-labs.com/stat/duncan/R/dynload.

Value

The function `dyn.load` is used for its side effect which links the specified shared library to the executing R image. Calls to `.C`, `.Call`, `.Fortran` and `.External` can then be used to execute compiled C functions or Fortran subroutines contained in the library. The return value of `dyn.load` is an object of class `DLLInfo`. See `getLoadedDLLs` for information about this class.

The function `dyn.unload` unlinks the shared library.

`is.loaded` checks if the symbol name is loaded and hence available for use in `.C` or `.Fortran` or `.Call` or `.Exernal`: it requires the name you would give to `.C` etc and **not** (as in S) that remapped by deprecated functions `symbol.C` or `symbol.For`. It will succeed if any one of the four calling functions would succeed in using the entry point unless `type` is specified: for Fortran entry points `is.loaded(symbol.For("symbol"))` will also succeed *unless* the registration mechanism has been used.

Warning

Do not use `dyn.unload` on a shared object loaded by `library.dynam`: use `library.dynam.unload`.

Note

The creation of shared libraries and the runtime linking of them into executing programs is very platform dependent. In recent years there has been some simplification in the process because the C subroutine call `dlopen` has become the standard for doing this under UNIX. Under UNIX `dyn.load` uses the `dlopen` mechanism and should work on all platforms which support it. On Windows it uses the standard mechanisms for loading 32-bit DLLs.

The original code for loading DLLs in UNIX was provided by Heiner Schwarte. The compatibility code for HP-UX was provided by Luke Tierney.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`library.dynam` to be used inside a package's `.First.lib` initialization.

`SHLIB` for how to create suitable shared objects.

`.C`, `.Fortran`, `.External`, `.Call`.

Examples

```
is.loaded("hcass2") #-> probably TRUE, as stats is loaded
is.loaded("supsmu") # Fortran entry point in stats
is.loaded("supsmu", "stats", "Fortran")
is.loaded("PDF", type = "External")
```

`eapply` *Apply a Function over values in an environment*

Description

`eapply` applies `FUN` to the named values from an environment and returns the results as a list. The user can request that all named objects are used (normally names that begin with a dot are not). The output is not sorted and no parent environments are searched.

Usage

```
eapply(env, FUN, ..., all.names = FALSE)
```

Arguments

`env`	environment to be used.
`FUN`	the function to be applied, found *via* `match.fun`. In the case of functions like `+`, `%*%`, etc., the function name must be backquoted or quoted.
`...`	optional arguments to `FUN`.
`all.names`	a logical indicating whether to apply the function to all values

See Also

lapply.

Examples

```
env <- new.env()
env$a <- 1:10
env$beta <- exp(-3:3)
env$logic <- c(TRUE,FALSE,FALSE,TRUE)
# compute the list mean for each list element
eapply(env,mean)
# median and quartiles for each list element
eapply(env, quantile, probs = 1:3/4)
eapply(env, quantile)
```

---

eigen *Spectral Decomposition of a Matrix*

Description

Computes eigenvalues and eigenvectors of real or complex matrices.

Usage

```
eigen(x, symmetric, only.values = FALSE, EISPACK = FALSE)
```

Arguments

x	a matrix whose spectral decomposition is to be computed.
symmetric	if TRUE, the matrix is assumed to be symmetric (or Hermitian if complex) and only its lower triangle is used. If symmetric is not specified, the matrix is inspected for symmetry.
only.values	if TRUE, only the eigenvalues are computed and returned, otherwise both eigenvalues and eigenvectors are returned.
EISPACK	logical. Should EISPACK be used (for compatibility with R < 1.7.0)?

Details

By default eigen uses the LAPACK routines DSYEVR, DGEEV, ZHEEV and ZGEEV whereas eigen(EISPACK=TRUE) provides an interface to the EISPACK routines RS, RG, CH and CG.

If symmetric is unspecified, the code attempts to determine if the matrix is symmetric up to plausible numerical inaccuracies. It is faster and surer to set the value yourself.

eigen is preferred to eigen(EISPACK = TRUE) for new projects, but its eigenvectors may differ in sign and (in the asymmetric case) in normalization. (They may also differ between methods and between platforms.)

Computing the eigenvectors is the slow part for large matrices.

Computing the eigendecomposition of a matrix is subject to errors on a real-world computer: the definitive analysis is Wilkinson (1965). All you can hope for is a solution to a problem suitably close to x. So even though a real asymmetric x may have an algebraic solution with repeated real eigenvalues, the computed solution may be of a similar matrix with complex conjugate pairs of eigenvalues.

Value

The spectral decomposition of `x` is returned as components of a list with components

`values` a vector containing the p eigenvalues of `x`, sorted in *decreasing* order, according to `Mod(values)` in the asymmetric case when they might be complex (even for real matrices). For real asymmetric matrices the vector will be complex only if complex conjugate pairs of eigenvalues are detected.

`vectors` either a $p \times p$ matrix whose columns contain the eigenvectors of `x`, or `NULL` if `only.values` is `TRUE`.

For `eigen(, symmetric = FALSE, EISPACK =TRUE)` the choice of length of the eigenvectors is not defined by EISPACK. In all other cases the vectors are normalized to unit length.

Recall that the eigenvectors are only defined up to a constant: even when the length is specified they are still only defined up to a scalar of modulus one (the sign for real matrices).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Smith, B. T, Boyle, J. M., Dongarra, J. J., Garbow, B. S., Ikebe,Y., Klema, V., and Moler, C. B. (1976). *Matrix Eigensystems Routines – EISPACK Guide*. Springer-Verlag Lecture Notes in Computer Science.

Anderson. E. and ten others (1999) *LAPACK Users' Guide*. Third Edition. SIAM.
Available on-line at http://www.netlib.org/lapack/lug/lapack_lug.html.

Wilkinson, J. H. (1965) *The Algebraic Eigenvalue Problem.* Clarendon Press, Oxford.

See Also

`svd`, a generalization of `eigen`; `qr`, and `chol` for related decompositions.

To compute the determinant of a matrix, the `qr` decomposition is much more efficient: `det`.

`capabilities` to test for IEEE 754 arithmetic.

Examples

```
eigen(cbind(c(1,-1),c(-1,1)))
eigen(cbind(c(1,-1),c(-1,1)), symmetric = FALSE)# same (different algorithm).

eigen(cbind(1,c(1,-1)), only.values = TRUE)
eigen(cbind(-1,2:1)) # complex values
eigen(print(cbind(c(0,1i), c(-1i,0))))# Hermite ==> real Eigen values
## 3 x 3:
eigen(cbind( 1,3:1,1:3))
eigen(cbind(-1,c(1:2,0),0:2)) # complex values
```

encodeString *Encode Character Vector as for Printing*

Description

encodeString escapes the strings in a character vector in the same way print.default does, and optionally fits the encoded strings within a field width.

Usage

```
encodeString(x, width = 0, quote = "", na.encode = TRUE,
             justify = c("left", "right", "centre", "none"))
```

Arguments

x	A character vector, or an object that can be coerced to one by as.character.
width	integer: the minimum field width. If NULL or NA, this is taken to be the largest field width needed for any element of x.
quote	character: quoting character, if any.
na.encode	logical: should NA strings be encoded?
justify	character: partial matches are allowed. If padding to the minimum field width is needed, how should spaces be inserted? justify == "none" is equivalent to width = 0, for consistency with format.default.

Details

This escapes backslash and the control characters
a (bell),
b (backspace),
f (formfeed),
n (line feed),
r (carriage return),
t (tab),
v (vertical tab) and
0 (nul) as well as any non-printable characters in a single-byte locale, which are printed in octal notation (
xyz with leading zeroes). (Which characters are non-printable depends on the current locale.) See print.default for how non-printable characters are handled in multi-byte locales.

If quote is a single or double quote any embedded quote of the same type is escaped. Note that justification is of the quoted string, hence spaces are added outside the quotes.

Value

A character vector of the same length as x, with the same attributes (including names and dimensions) but with no class set.

Note

The default for width is different from format.default, which does similar things for character vectors but without encoding using escapes.

See Also

print.default

Examples

```
x <- "ab\bc\ndef"
print(x)
cat(x) # interprets escapes
cat(encodeString(x), "\n", sep="") # similar to print()

factor(x) # makes use of this to print the levels

x <- c("a", "ab", "abcde")
encodeString(x, w = NA) # left justification
encodeString(x, w = NA, justify = "c")
encodeString(x, w = NA, justify = "r")
encodeString(x, w = NA, quote = "'", justify = "r")
```

---

environment *Environment Access*

Description

Get, set, test for and create environments.

Usage

```
environment(fun = NULL)
environment(fun) <- value

is.environment(obj)

.GlobalEnv
globalenv()
.BaseNamespaceEnv

emptyenv()
baseenv()

new.env(hash = FALSE, parent = parent.frame())

parent.env(env)
parent.env(env) <- value
```

Arguments

fun	a function, a formula, or NULL, which is the default.
value	an environment to associate with the function
obj	an arbitrary R object.
hash	a logical, if TRUE the environment will be hashed
parent	an environment to be used as the enclosure of the environment created.
env	an environment

Details

Environments consist of a *frame*, or collection of named objects, and a pointer to an *enclosing environment*. The most common example is the frame of variables local to a function call; its enclosure is the environment where the function was defined. The enclosing environment is distinguished from the *parent frame*: the latter (returned by `parent.frame`) refers to the environment of the caller of a function.

When `get` or `exists` search an environment with the default `inherits = TRUE`, they look for the variable in the frame, then in the enclosing frame, and so on.

The global environment `.GlobalEnv`, more often known as the user's workspace, is the first item on the search path. It can also be accessed by `globalenv()`. On the search path, each item's enclosure is the next item.

The object `.BaseNamespaceEnv` is the namespace environment for the base package. The environment of the base package itself is available as `baseenv()`. The ultimate enclosure of any environment is the empty environment `emptyenv()`, to which nothing may be assigned. If one follows the `parent.env()` chain of enclosures back far enough from any environment, eventually one reaches the empty environment.

The replacement function `parent.env<-` is extremely dangerous as it can be used to destructively change environments in ways that violate assumptions made by the internal C code. It may be removed in the near future.

`is.environment` is generic: you can write methods to handle specific classes of objects, see InternalMethods.

Value

If `fun` is a function or a formula then `environment(fun)` returns the environment associated with that function or formula. If `fun` is `NULL` then the current evaluation environment is returned.

The replacement form sets the environment of the function or formula `fun` to the `value` given.

`is.environment(obj)` returns `TRUE` iff `obj` is an `environment`.

`new.env` returns a new (empty) environment enclosed in the parent's environment, by default.

`parent.env` returns the parent environment of its argument.

`parent.env<-` sets the enclosing environment of its first argument.

See Also

The `envir` argument of `eval`, `get`, and `exists`.

`ls` may be used to view the objects in an environment.

Examples

```
f <- function() "top level function"

##-- all three give the same:
environment()
environment(f)
.GlobalEnv

ls(envir=environment(approxfun(1:2,1:2, method="const")))

is.environment(.GlobalEnv) # TRUE
```

```
e1 <- new.env(parent = baseenv())  # this one has enclosure package:base.
e2 <- new.env(parent = e1)
assign("a", 3, env=e1)
ls(e1)
ls(e2)
exists("a", env=e2)     # this succeeds by inheritance
exists("a", env=e2, inherits = FALSE)
exists("+", env=e2)     # this succeeds by inheritance
```

`eval` *Evaluate an (Unevaluated) Expression*

Description

Evaluate an R expression in a specified environment.

Usage

```
eval(expr, envir = parent.frame(),
           enclos = if(is.list(envir) || is.pairlist(envir))
                       parent.frame() else baseenv())
evalq(expr, envir, enclos)
eval.parent(expr, n = 1)
local(expr, envir = new.env())
```

Arguments

`expr`	an object to be evaluated. See Details.
`envir`	the `environment` in which `expr` is to be evaluated. May also be `NULL`, a list, a data frame, a pairlist or an integer as specified to `sys.call`.
`enclos`	Relevant when `envir` is a (pair)list or a data frame. Specifies the enclosure, i.e., where R looks for objects not found in `envir`. This can be `NULL` (interpreted as the base package environment) or an environment.
`n`	number of parent generations to go back

Details

`eval` evaluates the `expr` argument in the environment specified by `envir` and returns the computed value. If `envir` is not specified, then the default is `parent.frame()` (the environment where the call to `eval` was made).

Objects to be evaluated can be of types `call` or `expression` or symbol (when the symbol is looked up in the current scope and its binding is evaluated), a promise or any of the basic types such as vectors, functions and environments (which are returned unchanged).

The `evalq` form is equivalent to `eval(quote(expr), ...)`. `eval` evaluates its first argument in the current scope before passing it to the evaluator: `evalq` avoids this.

`eval.parent(expr, n)` is a shorthand for `eval(expr, parent.frame(n))`.

If `envir` is a list (such as a data frame) or pairlist, it is copied into a temporary environment (with enclosure `enclos`), and the temporary environment is used for evaluation. So if `expr` changes any of the components named in the (pair)list, the changes are lost.

If `envir` is `NULL` it is interpreted as an empty list so no values could be found in `envir` and look-up goes directly to `enclos`.

`local` evaluates an expression in a local environment. It is equivalent to `evalq` except that its default argument creates a new, empty environment. This is useful to create anonymous recursive functions and as a kind of limited namespace feature since variables defined in the environment are not visible from the outside.

Note

Due to the difference in scoping rules, there are some differences between R and S in this area. In particular, the default enclosure in S is the global environment.

When evaluating expressions in data frames that has been passed as argument to a function, the relevant enclosure is often the caller's environment, i.e., one needs `eval(x, data, parent.frame())`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (`eval` only.)

See Also

`expression`, `quote`, `sys.frame`, `parent.frame`, `environment`.

Further, `force` to *force* evaluation, typically of function arguments.

Examples

```
eval(2 ^ 2 ^ 3)
mEx <- expression(2^2^3); mEx; 1 + eval(mEx)
eval({ xx <- pi; xx^2}) ; xx

a <- 3 ; aa <- 4 ; evalq(evalq(a+b+aa, list(a=1)), list(b=5)) # == 10
a <- 3 ; aa <- 4 ; evalq(evalq(a+b+aa, -1), list(b=5))        # == 12

ev <- function() {
   e1 <- parent.frame()
   ## Evaluate a in e1
   aa <- eval(expression(a),e1)
   ## evaluate the expression bound to a in e1
   a <- expression(x+y)
   list(aa = aa, eval = eval(a, e1))
}
tst.ev <- function(a = 7) { x <- pi; y <- 1; ev() }
tst.ev()#-> aa : 7,  eval : 4.14

a <- list(a=3, b=4)
with(a, a <- 5) # alters the copy of a from the list, discarded.

##
## Example of evalq()
##

N <- 3
env <- new.env()
assign("N", 27, envir=env)
```

```
## this version changes the visible copy of N only, since the argument
## passed to eval is '4'.
eval(N <- 4, env)
N
get("N", envir=env)
## this version does the assignment in env, and changes N only there.
evalq(N <- 5, env)
N
get("N", envir=env)

##
## Uses of local()
##

# Mutual recursives.
# gg gets value of last assignment, an anonymous version of f.

gg <- local({
    k <- function(y)f(y)
    f <- function(x) if(x) x*k(x-1) else 1
})
gg(10)
sapply(1:5, gg)

# Nesting locals. a is private storage accessible to k
gg <- local({
    k <- local({
        a <- 1
        function(y){print(a <<- a+1);f(y)}
    })
    f <- function(x) if(x) x*k(x-1) else 1
})
sapply(1:5, gg)

ls(envir=environment(gg))
ls(envir=environment(get("k", envir=environment(gg))))
```

`exists` *Is an Object Defined?*

Description

Look for an R object of the given name.

Usage

```
exists(x, where = -1, envir = , frame, mode = "any",
       inherits = TRUE)
```

Arguments

`x`	a variable name (given as a character string).
`where`	where to look for the object (see the details section); if omitted, the function will search as if the name of the object appeared unquoted in an expression.

`envir`	an alternative way to specify an environment to look in, but it is usually simpler to just use the `where` argument.
`frame`	a frame in the calling list. Equivalent to giving `where` as `sys.frame(frame)`.
`mode`	the mode or type of object sought: see the Details section.
`inherits`	should the enclosing frames of the environment be searched?

Details

The `where` argument can specify the environment in which to look for the object in any of several ways: as an integer (the position in the `search` list); as the character string name of an element in the search list; or as an `environment` (including using `sys.frame` to access the currently active function calls). The `envir` argument is an alternative way to specify an environment, but is primarily there for back compatibility.

This function looks to see if the name `x` has a value bound to it in the specified environment. If `inherits` is `TRUE` and a value is not found for `x` in the specified environment, the enclosing frames of the environment are searched until the name `x` is encountered. See `environment` and the 'R Language Definition' manual for details about the structure of environments and their enclosures.

Warning: `inherits = TRUE` is the default behaviour for R but not for S.

If `mode` is specified then only objects of that type are sought. The `mode` may specify one of the collections `"numeric"` and `"function"` (see `mode`): any member of the collection will suffice. (This is true even if a member of a collection is specified, so for example `mode="special"` will seek any type of function.)

Value

Logical, true if and only if an object of the correct name and mode is found.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`get`.

Examples

```
##  Define a substitute function if necessary:
if(!exists("some.fun", mode="function"))
  some.fun <- function(x) { cat("some.fun(x)\n"); x }
search()
exists("ls", 2) # true even though ls is in pos=3
exists("ls", 2, inherits = FALSE) # false
```

`expand.grid` *Create a Data Frame from All Combinations of Factors*

Description

Create a data frame from all combinations of the supplied vectors or factors. See the description of the return value for precise details of the way this is done.

Usage

```
expand.grid(..., KEEP.OUT.ATTRS = TRUE)
```

Arguments

`...`	vectors, factors or a list containing these.
`KEEP.OUT.ATTRS`	a logical indicating the `"out.attrs"` attribute (see below) should be computed and returned.

Value

A data frame containing one row for each combination of the supplied factors. The first factors vary fastest. The columns are labelled by the factors if these are supplied as named arguments or named components of a list.

Attribute `"out.attrs"` is a list which gives the dimension and dimnames for use by `predict` methods.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S.* Wadsworth & Brooks/Cole.

See Also

`combn` (package `utils`) for the generation of all combinations of n elements, taken m at a time.

Examples

```
expand.grid(height = seq(60, 80, 5), weight = seq(100, 300, 50),
            sex = c("Male","Female"))

x <- seq(0,10, length=100)
y <- seq(-1,1, length=20)
d1 <- expand.grid(x=x, y=y)
d2 <- expand.grid(x=x, y=y, KEEP.OUT.ATTRS = FALSE)
object.size(d1) - object.size(d2)
##-> 5992 or 8832 (on 32- / 64-bit platform)
```

`expression` *Unevaluated Expressions*

Description

Creates or tests for objects of mode `"expression"`.

Usage

```
expression(...)

is.expression(x)
as.expression(x, ...)
```

Arguments

`...`	valid R calls, symbols or constants.
`x`	an arbitrary R object.

Details

'Expression' here is not being used in its colloquial sense, that of mathematical expressions. Those are calls (see `call`) in R, and an R expression vector is a list of calls etc, typically as returned by `parse`.

As an object of mode `"expression"` is a list, it can be subsetted by both `[` and by `[[`, the latter extracting individual calls.

Value

`expression` returns a vector of type `"expression"` containing its arguments (unevaluated).

`is.expression` returns `TRUE` if `expr` is an expression object and `FALSE` otherwise.

`as.expression` attempts to coerce its argument into an expression object.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`call`, `eval`, `function`. Further, `text` and `legend` for plotting mathematical expressions.

Examples

```
length(ex1 <- expression(1+ 0:9))# 1
ex1
eval(ex1)# 1:10

length(ex3 <- expression(u,v, 1+ 0:9))# 3
mode(ex3 [3]) # expression
mode(ex3[[3]])# call
rm(ex3)
```

`Extract` *Extract or Replace Parts of an Object*

Description

Operators acting on vectors, matrices, arrays and lists to extract or replace parts.

Usage

```
x[i]
x[i, j, ... , drop = TRUE]
x[[i]]
x[[i, j, ...]]
x$name
```

Arguments

`x`	object from which to extract element(s) or in which to replace element(s).
`i, j, ..., name`	indices specifying elements to extract or replace. `i, j` are `numeric` or `character` vectors or empty (missing) or `NULL` whereas `name` must be a character string or an (unquoted or backtick quoted) name. Numeric values are coerced to integer as by `as.integer`. For extraction with `[[` and `$` character strings are normally (see under Environments) partially matched to the `names` of the object if exact matching does not succeed. For `[`-indexing only: `i, j, ...` can be logical vectors, indicating elements/slices to select. Such vectors are recycled if necessary to match the corresponding extent. `i, j, ...` can also be negative integers, indicating elements/slices to leave out of the selection. When indexing arrays by `[` a single argument `i` can be a matrix with as many columns as there are dimensions of `x`; the result is then a vector with elements corresponding to the sets of indices in each row of `i`. An index value of `NULL` is treated as if it were `integer(0)`.
`drop`	For matrices and arrays. If `TRUE` the result is coerced to the lowest possible dimension (see the examples). This only works for extracting elements, not for the replacement.

Details

These operators are generic. You can write methods to handle indexing of specific classes of objects, see InternalMethods as well as `[.data.frame` and `[.factor`. The descriptions here apply only to the default methods. Note that separate methods are required for the replacement functions `[<-`, `[[<-` and `$<-` for use when indexing occurs on the assignment side of an expression.

The most important distinction between `[`, `[[` and `$` is that the `[` can select more than one element whereas the other two select a single element.

The default methods work somewhat differently for atomic vectors, matrices/arrays and for recursive (list-like, see `is.recursive`) objects. `$` returns `NULL` except for recursive objects, and is only discussed in the section below on recursive objects.

Subsetting (except by an empty index) will drop all attributes except `names`, `dim` and `dimnames`.

Indexing can occur on the right-hand-side of an expression for extraction, or on the left-hand-side for replacement. When an index expression appears on the left side of an assignment (known as *subassignment*) then that part of `x` is set to the value of the right hand side of the assignment. In this case no partial matching of indices is done, and the left-hand-side is coerced as needed to accept the values. Attributes are preserved (although `names`, `dim` and `dimnames` will be adjusted suitably).

Atomic vectors

The usual form of indexing is `"["`. `"[["` can be used to select a single element, but `"["` can also do so (but will not partially match a character index).

The index object `i` can be numeric, logical, character or empty. Indexing by factors is allowed and is equivalent to indexing by the numeric codes (see `factor`) and not by the character values which are printed (for which use `[as.character(i)]`).

An empty index selects all values: this is most often used to replace all the entries but keep the `attributes`.

Matrices and arrays

Matrices and arrays are vectors with a dimension attribute and so all the vector forms of indexing can be used with a single index. The result will be an unnamed vector unless `x` is one-dimensional when it will be a one-dimensional array.

The most common form of indexing a k-dimensional array is to specify k indices to `[`. As for vector indexing, the indices can be numeric, logical, character, empty or even factor. An empty index (a comma separated blank) indicates that all entries in that dimension are selected. The argument `drop` applies to this form of indexing.

A third form of indexing is via a numeric matrix with the one column for each dimension: each row of the index matrix then selects a single element of the array, and the result is a vector. Negative indices are not allowed in the index matrix. `NA` and zero values are allowed: rows of an index matrix containing a zero are ignored, whereas rows containing an `NA` produce an `NA` in the result.

A vector obtained by matrix indexing will be unnamed unless `x` is one-dimensional when the row names (if any) will be indexed to provide names for the result.

Recursive (list-like) objects

Indexing by `[` is similar to atomic vectors and selects a list of the specified element(s).

Both `[[` and `$` select a single element of the list. The main difference is that `$` does not allow computed indices, whereas `[[` does. `x$name` is equivalent to `x[["name"]]`.

`[` and `[[` are sometimes applied to other recursive objects such as calls and expressions. Pairlists are coerced to lists for extraction by `[`, but all three operators can be used for replacement.

`[[` can be applied recursively to lists, so that if the single index `i` is a vector of length `p`, `alist[[i]]` is equivalent to `alist[[i1]]...[[ip]]` providing all but the final indexing results in a list.

When `$<-` is applied to a `NULL` `x`, it first coerces `x` to `list()`. This is what also happens with `[[<-` if the replacement value `value` is of length greater than one: if `value` has length 1 or 0, `x` is first coerced to a zero-length vector of the type of `value`.

Environments

Both `$` and `[[` can be applied to environments. Only character arguments are allowed and no partial matching is done. The semantics of these operations are those of `get(i, env=x,`

`inherits=FALSE)`. If no match is found then `NULL` is returned. The assignment versions, `$<-` and `[[<-`, can also be used. Again, only character arguments are allowed. The semantics in this case are those of `assign(i, value, env=x, inherits=FALSE)`. Such an assignment will either create a new binding or change the existing binding in `x`.

NAs in indexing

When extracting, a numerical, logical or character `NA` index picks an unknown element and so returns `NA` in the corresponding element of a logical, integer, numeric, complex or character result, and `NULL` for a list. (It returns `00` for a raw result.]

When replacing (that is using indexing on the lhs of an assignment) `NA` does not select any element to be replaced. As there is ambiguity as to whether an element of the rhs should be used or not, this is only allowed if the rhs value is of length one (so the two interpretations would have the same outcome).

Argument matching

Note that these operations do not match their index arguments in the standard way: argument names are ignored and positional matching only is used. So `m[j=2,i=1]` is equivalent to `m[2,1]` and **not** to `m[1,2]`.

This may not be true for methods defined for them; for example it is not true for the `data.frame` methods described in `[.data.frame`.

To avoid confusion, do not name index arguments (but `drop` must be named).

Note

S uses partial matching when extracting by `[` (Becker *et al* p. 358) whereas R does not.

The documented behaviour of S is that an `NA` replacement index 'goes nowhere' but uses up an element of `value` (Becker *et al* p. 359). However, that is not the current behaviour of S-PLUS.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`list`, `array`, `matrix`.

`[.data.frame` and `[.factor` for the behaviour when applied to data.frame and factors.

`Syntax` for operator precedence, and the *R Language* reference manual about indexing details.

Examples

```
x <- 1:12; m <- matrix(1:6,nr=2); li <- list(pi=pi, e = exp(1))
x[10]                 # the tenth element of x
x <- x[-1]            # delete the 1st element of x
m[1,]                 # the first row of matrix m
m[1, , drop = FALSE]  # is a 1-row matrix
m[,c(TRUE,FALSE,TRUE)]# logical indexing
m[cbind(c(1,2,1),3:1)]# matrix index
m <- m[,-1]           # delete the first column of m
li[[1]]               # the first element of list li
y <- list(1,2,a=4,5)
```

```
y[c(3,4)]             # a list containing elements 3 and 4 of y
y$a                   # the element of y named a

## non-integer indices are truncated:
(i <- 3.999999999) # "4" is printed
(1:5)[i]  # 3

## recursive indexing into lists
z <- list( a=list( b=9, c='hello'), d=1:5)
unlist(z)
z[[c(1, 2)]]
z[[c(1, 2, 1)]]  # both "hello"
z[[c("a", "b")]] <- "new"
unlist(z)

## check $ and [[ for environments
e1 <- new.env()
e1$a <- 10
e1[["a"]]
e1[["b"]] <- 20
e1$b
ls(e1)
```

`Extract.data.frame` *Extract or Replace Parts of a Data Frame*

Description

Extract or replace subsets of data frames.

Usage

```
x[i]
x[i] <- value
x[i, j, drop]
x[i, j] <- value

x[[i]]
x[[i]] <- value
x[[i, j]]
x[[i, j]] <- value

x$name
x$name <- value
```

Arguments

`x`	data frame.
`i, j`	elements to extract or replace. `i, j` are `numeric` or `character` or, for `[` only, empty. Numeric values are coerced to integer as if by `as.integer`. For replacement by `[`, a logical matrix is allowed.

`drop`	logical. If `TRUE` the result is coerced to the lowest possible dimension. The default is to drop if only one column is left, but **not** to drop if only one row is left.
`value`	A suitable replacement value: it will be repeated a whole number of times if necessary and it may be coerced: see the Coercion section. If `NULL`, deletes the column if a single column is selected.
`name`	name or literal character string.

Details

Data frames can be indexed in several modes. When `[` and `[[` are used with a single index, they index the data frame as if it were a list. In this usage a `drop` argument is ignored, with a warning. Using `$` is equivalent to using `[[` with a single index.

When `[` and `[[` are used with two indices they act like indexing a matrix: `[[` can only be used to select one element.

If `[` returns a data frame it will have unique (and non-missing) row names, if necessary transforming the row names using `make.unique`. Similarly, column names will be transformed (if columns are selected more than once).

When `drop =TRUE`, this is applied to the subsetting of any matrices contained in the data frame as well as to the data frame itself.

The replacement methods can be used to add whole column(s) by specifying non-existent column(s), in which case the column(s) are added at the right-hand edge of the data frame and numerical indices must be contiguous to existing indices. On the other hand, rows can be added at any row after the current last row, and the columns will be in-filled with missing values. Missing values in the indices are not allowed for replacement.

For `[` the replacement value can be a list: each element of the list is used to replace (part of) one column, recycling the list as necessary. If columns specified by number are created, the names (if any) of the corresponding list elements are used to name the columns. If the replacement is not selecting rows, list values can contain `NULL` elements which will cause the corresponding columns to be deleted. (See the Examples.)

Matrixing indexing using `[` is not recommended, and barely supported. For extraction, `x` is first coerced to a matrix. For replacement a logical matrix (only) can be used to select the elements to be replaced in the same way as for a matrix.

Value

For `[` a data frame, list or a single column (the latter two only when dimensions have been dropped). If matrix indexing is used for extraction a matrix results.

For `[[` a column of the data frame (extraction with one index) or a length-one vector (extraction with two indices).

For `[<-`, `[[<-` and `$<-`, a data frame.

Coercion

The story over when replacement values are coerced is a complicated one, and one that has changed during R's development. This section is a guide only.

When `[` and `[[` are used to add or replace a whole column, no coercion takes place but `value` will be replicated (by calling the generic function `rep`) to the right length if an exact number of repeats can be used.

When `[` is used with a logical matrix, each value is coerced to the type of the column in which it is to be placed.

When `[` and `[[` are used with two indices, the column will be coerced as necessary to accommodate the value.

Note that when the replacement value is an array (including a matrix) it is *not* treated as a series of columns (as `data.frame` and `as.data.frame` do) but inserted as a single column.

Warning

The default behaviour when only one *row* is left is equivalent to specifying `drop = FALSE`. To drop from a data frame to a list, `drop = TRUE` has to be specified explicitly.

See Also

`subset` which is often easier for extraction, `data.frame`, `Extract`.

Examples

```
sw <- swiss[1:5, 1:4]  # select a manageable subset

sw[1:3]      # select columns
sw[, 1:3]    # same
sw[4:5, 1:3] # select rows and columns
sw[1]        # a one-column data frame
sw[, 1, drop = FALSE]  # the same
sw[, 1]      # a (unnamed) vector
sw[[1]]      # the same

sw[1,]       # a one-row data frame
sw[1,, drop=TRUE]  # a list

swiss[ c(1, 1:2), ]   # duplicate row, unique row names are created

sw[sw <= 6] <- 6  # logical matrix indexing
sw

## adding a column
sw["new1"] <- LETTERS[1:5]   # adds a character column
sw[["new2"]] <- letters[1:5] # ditto
sw[, "new3"] <- LETTERS[1:5] # ditto
sw$new4 <- 1:5
sapply(sw, class)
sw$new4 <- NULL              # delete the column
sw
sw[6:8] <- list(letters[10:14], NULL, aa=1:5) # delete col7, update 6, append
sw

## matrices in a data frame
A <- data.frame(x=1:3, y=I(matrix(4:6)), z=I(matrix(letters[1:9],3,3)))
A[1:3, "y"] # a matrix
A[1:3, "z"] # a matrix
A[, "y"]    # a matrix
```

`Extract.factor` *Extract or Replace Parts of a Factor*

Description

Extract or replace subsets of factors.

Usage

```
x[..., drop = FALSE]

x[[i]]

x[...] <- value
```

Arguments

`x`	a factor
`..., i`	a specification of indices – see `Extract`.
`drop`	logical. If true, unused levels are dropped.
`value`	character: a set of levels. Factor values are coerced to character.

Details

When unused levels are dropped the ordering of the remaining levels is preserved.

If `value` is not in `levels(x)`, a missing value is assigned with a warning.

Any `contrasts` assigned to the factor are preserved unless `drop=TRUE`.

Value

A factor with the same set of levels as `x` unless `drop=TRUE`. (Prior to R 2.4.0, `[[` returned the integer code.)

See Also

`factor`, `Extract`.

Examples

```
## following example(factor)
(ff <- factor(substring("statistics", 1:10, 1:10), levels=letters))
ff[, drop=TRUE]
factor(letters[7:10])[2:3, drop = TRUE]
```

`Extremes` *Maxima and Minima*

Description

Returns the (parallel) maxima and minima of the input values.

Usage

```
max(..., na.rm=FALSE)
min(..., na.rm=FALSE)

pmax(..., na.rm=FALSE)
pmin(..., na.rm=FALSE)
```

Arguments

`...`	numeric arguments (see Note).
`na.rm`	a logical indicating whether missing values should be removed.

Value

`max` and `min` return the maximum or minimum of *all* the values present in their arguments, as `integer` if all are `integer`, or as `double` otherwise.

The minimum and maximum of an empty set are `+Inf` and `-Inf` (in this order!) which ensures *transitivity*, e.g., `min(x1, min(x2)) == min(x1, x2)`. From R version 1.5.0, `max(x) == -Inf` and `min(x) == +Inf` whenever `length(x) == 0` (after removing missing values if requested).

If `na.rm` is `FALSE` an `NA` value in any of the arguments will cause a value of `NA` to be returned, otherwise `NA` values are ignored.

`pmax` and `pmin` take several vectors (or matrices) as arguments and return a single vector giving the "parallel" maxima (or minima) of the vectors. The first element of the result is the maximum (minimum) of the first elements of all the arguments, the second element of the result is the maximum (minimum) of the second elements of all the arguments and so on. Shorter vectors are recycled if necessary. If `na.rm` is `FALSE`, `NA` values in the input vectors will produce `NA` values in the output. If `na.rm` is `TRUE`, `NA` values are ignored. `attributes` (such as `names` or `dim`) are transferred from the first argument (if applicable).

`max` and `min` are generic functions: methods can be defined for them individually or via the `Summary` group generic. For this to work properly, the arguments `...` should be unnamed, and dispatch is on the first argument.

By definition the min/max of any vector containing an `NaN` is `NaN`, except that the min/max of any vector containing an `NA` is `NA` even if it also contains an `NaN`. Note that `max(NA, Inf) == NA` even though the maximum would be `inf` whatever the missing value actually is.

Note

'Numeric' arguments are vectors of type integer and numeric, and logical (coerced to integer). For historical reasons, `NULL` is accepted as equivalent to `integer(0)`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`range` (*both* min and max) and `which.min` (`which.max`) for the *arg min*, i.e., the location where an extreme value occurs.

Examples

```
require(stats)
 min(5:1, pi) #-> one number
pmin(5:1, pi) #->  5  numbers

x <- sort(rnorm(100));  cH <- 1.35
pmin(cH, quantile(x)) # no names
pmin(quantile(x), cH) # has names
plot(x, pmin(cH, pmax(-cH, x)), type='b', main= "Huber's function")
```

`factor` *Factors*

Description

The function `factor` is used to encode a vector as a factor (the terms 'category' and 'enumerated type' are also used for factors). If `ordered` is `TRUE`, the factor levels are assumed to be ordered. For compatibility with S there is also a function `ordered`.

`is.factor`, `is.ordered`, `as.factor` and `as.ordered` are the membership and coercion functions for these classes.

Usage

```
factor(x = character(), levels = sort(unique.default(x), na.last = TRUE),
       labels = levels, exclude = NA, ordered = is.ordered(x))
ordered(x, ...)

is.factor(x)
is.ordered(x)

as.factor(x)
as.ordered(x)
```

Arguments

`x`	a vector of data, usually taking a small number of distinct values.
`levels`	an optional vector of the values that `x` might have taken. The default is the set of values taken by `x`, sorted into increasing order.
`labels`	*either* an optional vector of labels for the levels (in the same order as `levels` after removing those in `exclude`), *or* a character string of length 1.

exclude	a vector of values to be excluded when forming the set of levels. This should be of the same type as x, and will be coerced if necessary.
ordered	logical flag to determine if the levels should be regarded as ordered (in the order given).
...	(in ordered(.)): any of the above, apart from ordered itself.

Details

The type of the vector x is not restricted.

Ordered factors differ from factors only in their class, but methods and the model-fitting functions treat the two classes quite differently.

The encoding of the vector happens as follows. First all the values in exclude are removed from levels. If x[i] equals levels[j], then the i-th element of the result is j. If no match is found for x[i] in levels, then the i-th element of the result is set to NA.

Normally the 'levels' used as an attribute of the result are the reduced set of levels after removing those in exclude, but this can be altered by supplying labels. This should either be a set of new labels for the levels, or a character string, in which case the levels are that character string with a sequence number appended.

factor(x, exclude=NULL) applied to a factor is a no-operation unless there are unused levels: in that case, a factor with the reduced level set is returned. If exclude is used it should also be a factor with the same level set as x or a set of codes for the levels to be excluded.

The codes of a factor may contain NA. For a numeric x, set exclude=NULL to make NA an extra level ("NA"), by default the last level.

If "NA" is a level, the way to set a code to be missing is to use is.na on the left-hand-side of an assignment. Under those circumstances missing values are printed as <NA>.

is.factor is generic: you can write methods to handle specific classes of objects, see Internal-Methods.

Value

factor returns an object of class "factor" which has a set of integer codes the length of x with a "levels" attribute of mode character. If ordered is true (or ordered is used) the result has class c("ordered", "factor").

Applying factor to an ordered or unordered factor returns a factor (of the same type) with just the levels which occur: see also [.factor for a more transparent way to achieve this.

is.factor returns TRUE or FALSE depending on whether its argument is of type factor or not. Correspondingly, is.ordered returns TRUE when its argument is ordered and FALSE otherwise.

as.factor coerces its argument to a factor. It is an abbreviated form of factor.

as.ordered(x) returns x if this is ordered, and ordered(x) otherwise.

Warning

The interpretation of a factor depends on both the codes and the "levels" attribute. Be careful only to compare factors with the same set of levels (in the same order). In particular, as.numeric applied to a factor is meaningless, and may happen by implicit coercion. To "revert" a factor f to its original numeric values, as.numeric(levels(f))[f] is recommended and slightly more efficient than as.numeric(as.character(f)).

The levels of a factor are by default sorted, but the sort order may well depend on the locale at the time of creation, and should not be assumed to be ASCII.

Comparison operators and group generic methods

There are `"factor"` and `"ordered"` methods for the group generic `Ops`, which provide methods for the Comparison operators. (The rest of the group and the `Math` and `Summary` groups generate an error as they are not meaningful for factors.)

Only == and != can be used for factors: a factor can only be compared to another factor with an identical set of levels (not necessarily in the same ordering) or to a character vector. Ordered factors are compared in the same way, but the general dispatch mechanism precludes comparing ordered and unordered factors.

All the comparison operators are available for ordered factors. Sorting is done by the levels of the operands: if both operands are ordered factors they must have the same level set.

Note

Storing character data as a factor is more efficient storage if there is even a small proportion of repeats. On a 32-bit machine storing a string of n bytes takes $28 + 8\lceil (n+1)/8 \rceil$ bytes whereas storing a factor code takes 4 bytes. (On a 64-bit machine 28 is replaced by 56 or more.) Only if they were computed from the same values (or in some cases read from a file: see `scan`) will identical strings share storage.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

See Also

`[.factor` for subsetting of factors.

`gl` for construction of "balanced" factors and `C` for factors with specified contrasts. `levels` and `nlevels` for accessing the levels, and `unclass` to get integer codes.

Examples

```
(ff <- factor(substring("statistics", 1:10, 1:10), levels=letters))
as.integer(ff)  # the internal codes
factor(ff)      # drops the levels that do not occur
ff[, drop=TRUE] # the same, more transparently

factor(letters[1:20], label="letter")

class(ordered(4:1)) # "ordered", inheriting from "factor"

## suppose you want "NA" as a level, and to allowing missing values.
(x <- factor(c(1, 2, "NA"), exclude = ""))
is.na(x)[2] <- TRUE
x  # [1] 1    <NA> NA, <NA> used because NA is a level.
is.na(x)
# [1] FALSE  TRUE FALSE
factor()
```

`file.access` *Ascertain File Accessibility*

Description

Utility function to access information about files on the user's file systems.

Usage

```
file.access(names, mode = 0)
```

Arguments

`names`	character vector containing file names.
`mode`	integer specifying access mode required.

Details

The `mode` value can be the exclusive or of the following values

0 test for existence.

1 test for execute permission.

2 test for write permission.

4 test for read permission.

Permission will be computed for real user ID and real group ID (rather than the effective IDs).

Please note that it is not good to use this function to test before trying to open a file. On a multitasking system, it is possible that the accessibility of a file will change between the time you call `file.access()` and the time you try to open the file, and in recent Windows versions the underlying function in `msvcrt.dll` sometimes returns inaccurate values. It is better to wrap file open attempts in `try` instead.

Value

An integer vector with values `0` for success and `-1` for failure.

Note

This is intended as a replacement for the S-PLUS function `access`, a wrapper for the C function of the same name, which explains the return value encoding. Note that the return value is **false** for **success**.

See Also

`file.info`, `try`

Examples

```
fa <- file.access(dir("."))
table(fa) # count successes & failures
```

file.choose *Choose a File Interactively*

Description

Choose a file interactively.

Usage

```
file.choose(new = FALSE)
```

Arguments

new	Logical: choose the style of dialog box presented to the user: at present only new = FALSE is used.

Value

A character vector of length one giving the file path.

See Also

list.files for non-interactive selection.

file.info *Extract File Information*

Description

Utility function to extract information about files on the user's file systems.

Usage

```
file.info(...)
```

Arguments

...	character vectors containing file names.

Details

What is meant by "file access" and hence the last access time is system-dependent.

On most systems symbolic links are followed, so information is given about the file to which the link points rather than about the link.

Value

A data frame with row names the file names and columns

`size`	double: File size in bytes.
`isdir`	logical: Is the file a directory?
`mode`	integer of class `"octmode"`. The file permissions, printed in octal, for example `644`.
`mtime, ctime, atime`	integer of class `"POSIXct"`: file modification, creation and last access times.
`uid`	integer: the user ID of the file's owner.
`gid`	integer: the group ID of the file's group.
`uname`	character: `uid` interpreted as a user name.
`grname`	character: `gid` interpreted as a group name.

Unknown user and group names will be `NA`.

Entries for non-existent or non-readable files will be `NA`. The `uid`, `gid`, `uname` and `grname` columns may not be supplied on a non-POSIX Unix system.

Note

This function will only be operational on systems with the `stat` system call, but that seems very widely available.

Some (broken) systems allow files of more than 2Gb to be created but not accessed by the `stat` system call. Such files will show up as non-readable (and very likely not be readable by any of R's input functions).

See Also

`files`, `file.access`, `list.files`, and `DateTimeClasses` for the date formats.

Examples

```
ncol(finf <- file.info(dir()))# at least six
## Not run: finf # the whole list
## Those that are more than 100 days old :
finf[difftime(Sys.time(), finf[,"mtime"], units="days") > 100 , 1:4]

file.info("no-such-file-exists")
```

---

file.path *Construct Path to File*

Description

Construct the path to a file from components in a platform-independent way.

Usage

```
file.path(..., fsep = .Platform$file.sep)
```

Arguments

...	character vectors.
fsep	the path separator to use.

Value

A character vector of the arguments concatenated term-by-term and separated by fsep if all arguments have positive length; otherwise, an empty character vector.

---

file.show *Display One or More Files*

Description

Display one or more files.

Usage

```
file.show(..., header = rep("",nfiles), title = "R Information",
          delete.file=FALSE, pager=getOption("pager"))
```

Arguments

...	one or more character vectors containing the names of the files to be displayed.
header	character vector (of the same length as the number of files specified in ...) giving a header for each file being displayed. Defaults to empty strings.
title	an overall title for the display. If a single separate window is used for the display, title will be used as the window title. If multiple windows are used, their titles should combine the title and the file-specific header.
delete.file	should the files be deleted after display? Used for temporary files.
pager	the pager to be used.

Details

This function provides the core of the R help system, but it can be used for other purposes as well.

Note

How the pager is implemented is highly system dependent.

The basic Unix version concatenates the files (using the headers) to a temporary file, and displays it in the pager selected by the pager argument, which is a character vector specifying a system command to run on the set of files.

Most GUI systems will use a separate pager window for each file, and let the user leave it up while R continues running. The selection of such pagers could either be done using "magic" pager names being intercepted by lower-level code (such as "internal" and "console" on Windows), or by letting pager be an R function which will be called with the same arguments as file.show and take care of interfacing to the GUI.

Not all implementations will honour delete.file.

Author(s)

Ross Ihaka, Brian Ripley.

See Also

files, list.files, help.

Examples

```
file.show(file.path(R.home("doc"), "COPYRIGHTS"))
```

---

files *File and Directory Manipulation*

Description

These functions provide a low-level interface to the computer's file system.

Usage

```
file.create(...)
file.exists(...)
file.remove(...)
file.rename(from, to)
file.append(file1, file2)
file.copy(from, to, overwrite = FALSE)
file.symlink(from, to)
dir.create(path, showWarnings = TRUE, recursive = FALSE)
```

Arguments

..., file1, file2, from, to	character vectors, containing file names.
path	a character vector containing a single path name.
overwrite	logical; should the destination files be overwritten?
showWarnings	logical; should the warnings on failure be shown?
recursive	logical: should elements of the path other than the last be created? If true, like Unix's `mkdir -p`.

Details

The ... arguments are concatenated to form one character string: you can specify the files separately or as one vector. All of these functions expand path names: see path.expand.

file.create creates files with the given names if they do not already exist and truncates them if they do.

file.exists returns a logical vector indicating whether the files named by its argument exist.

file.remove attempts to remove the files named in its argument.

file.rename attempts to rename a single file.

`file.append` attempts to append the files named by its second argument to those named by its first. The R subscript recycling rule is used to align names given in vectors of different lengths.

`file.copy` works in a similar way to `file.append` but with the arguments in the natural order for copying. Copying to existing destination files is skipped unless `overwrite = TRUE`. The `to` argument can specify a single existing directory.

`file.symlink` makes symbolic links on those Unix-like platforms which support them. The `to` argument can specify a single existing directory.

`dir.create` creates the last element of the path, unless `recursive = TRUE`. As from R 2.2.1 trailing path separators are ignored.

Value

`dir.create` and `file.rename` return a logical, true for success.

The remaining functions return a logical vector indicating which operation succeeded for each of the files attempted.

`dir.create` will return failure if the directory already exists.

Author(s)

Ross Ihaka, Brian Ripley

See Also

`file.info`, `file.access`, `file.path`, `file.show`, `list.files`, `unlink`, `basename`, `path.expand`.

Examples

```
cat("file A\n", file="A")
cat("file B\n", file="B")
file.append("A", "B")
file.create("A")
file.append("A", rep("B", 10))
if(interactive()) file.show("A")
file.copy("A", "C")
dir.create("tmp")
file.copy(c("A", "B"), "tmp")
list.files("tmp")
setwd("tmp")
file.remove("B")
file.symlink(file.path("..", c("A", "B")), ".")
setwd("..")
unlink("tmp", recursive=TRUE)
file.remove("A", "B", "C")
```

`findInterval` *Find Interval Numbers or Indices*

Description

Find the indices of `x` in `vec`, where `vec` must be sorted (non-decreasingly); i.e., if `i <- findInterval(x,v)`, we have $v_{i_j} \le x_j < v_{i_j+1}$ where $v_0 := -\infty$, $v_{N+1} := +\infty$, and `N <- length(vec)`. At the two boundaries, the returned index may differ by 1, depending on the optional arguments `rightmost.closed` and `all.inside`.

Usage

```
findInterval(x, vec, rightmost.closed = FALSE, all.inside = FALSE)
```

Arguments

`x`	numeric.
`vec`	numeric, sorted (weakly) increasingly, of length `N`, say.
`rightmost.closed`	logical; if true, the rightmost interval, `vec[N-1] .. vec[N]` is treated as *closed*, see below.
`all.inside`	logical; if true, the returned indices are coerced into $\{1, \dots, N-1\}$, i.e., 0 is mapped to 1 and N to $N-1$.

Details

The function `findInterval` finds the index of one vector `x` in another, `vec`, where the latter must be non-decreasing. Where this is trivial, equivalent to `apply( outer(x, vec, ">="), 1, sum)`, as a matter of fact, the internal algorithm uses interval search ensuring $O(n \log N)$ complexity where `n <- length(x)` (and `N <- length(vec)`). For (almost) sorted `x`, it will be even faster, basically $O(n)$.

This is the same computation as for the empirical distribution function, and indeed, `findInterval(t, sort(X))` is *identical* to $nF_n(t; X_1, \dots, X_n)$ where F_n is the empirical distribution function of $X_1, \dots, X_n$.

When `rightmost.closed = TRUE`, the result for `x[j] = vec[N]` ($= \max(vec)$), is `N - 1` as for all other values in the last interval.

Value

vector of length `length(x)` with values in `0:N` (and `NA`) where `N <- length(vec)`, or values coerced to `1:(N-1)` iff `all.inside = TRUE` (equivalently coercing all x values *inside* the intervals). Note that `NA`s are propagated from `x`, and `Inf` values are allowed in both `x` and `vec`.

Author(s)

Martin Maechler

See Also

`approx(*, method = "constant")` which is a generalization of `findInterval()`, `ecdf` for computing the empirical distribution function which is (up to a factor of n) also basically the same as findInterval(.).

Examples

```
N <- 100
X <- sort(round(rt(N, df=2), 2))
tt <- c(-100, seq(-2,2, len=201), +100)
it <- findInterval(tt, X)
tt[it < 1 | it >= N] # only first and last are outside range(X)
```

---

force *Force Evaluation of an Argument*

Description

Forces the evaluation of a function argument.

Usage

```
force(x)
```

Arguments

x	a formal argument of the enclosing function.

Details

`force` forces the evaluation of a formal argument. This can be useful if the argument will be captured in a closure by the lexical scoping rules and will later be altered by an explicit assignment or an implicit assignment in a loop or an apply function.

Note

This is semantic sugar: just evaluating the symbol will do the same thing (see the examples).

`force` does not force the evaluation of other promises. (It works by forcing the promise that is created when the actual arguments of a call are matched to the formal arguments of a closure, the mechanism which implememnts *lazy evaluation*.)

Examples

```
f <- function(y) function() y
lf <- vector("list", 5)
for (i in seq(along = lf)) lf[[i]] <- f(i)
lf[[1]]()  # returns 5

g <- function(y) { force(y); function() y }
lg <- vector("list", 5)
for (i in seq(along = lg)) lg[[i]] <- g(i)
lg[[1]]()  # returns 1

## This is identical to
g <- function(y) { y; function() y }
```

Foreign *Foreign Function Interface*

Description

Functions to make calls to compiled code that has been loaded into R.

Usage

```
                  .C(name, ..., NAOK = FALSE, DUP = TRUE, PACKAGE)
            .Fortran(name, ..., NAOK = FALSE, DUP = TRUE, PACKAGE)
           .External(name, ..., PACKAGE)
               .Call(name, ..., PACKAGE)
.External.graphics(name, ..., PACKAGE)
    .Call.graphics(name, ..., PACKAGE)
```

Arguments

`name`	a character string giving the name of a C function or Fortran subroutine, or an object of class `"NativeSymbolInfo"` or `"NativeSymbol"` referring to such a name.
`...`	arguments to be passed to the foreign function.
`NAOK`	if `TRUE` then any `NA` or `NaN` or `Inf` values in the arguments are passed on to the foreign function. If `FALSE`, the presence of `NA` or `NaN` or `Inf` values is regarded as an error.
`DUP`	if `TRUE` then arguments are "duplicated" before their address is passed to C or Fortran.
`PACKAGE`	if supplied, confine the search for the `name` to the DLL given by this argument (plus the conventional extension, `.so`, `.sl`, `.dll`, ...). This is intended to add safety for packages, which can ensure by using this argument that no other package can override their external symbols. Use `PACKAGE="base"` for symbols linked in to R.

Details

The functions `.C` and `.Fortran` can be used to make calls to C and Fortran code.

`.External` and `.External.graphics` can be used to call compiled code that uses R objects in the same way as internal R functions.

`.Call` and `.Call.graphics` can be used call compiled code which makes use of internal R objects. The arguments are passed to the C code as a sequence of R objects. It is included to provide compatibility with S version 4.

For details about how to write code to use with `.Call` and `.External`, see the chapter on "System and foreign language interfaces" in "Writing R Extensions" in the 'doc/manual' subdirectory of the R source tree.

Value

The functions `.C` and `.Fortran` return a list similar to the `...` list of arguments passed in, but reflecting any changes made by the C or Fortran code.

`.External`, `.Call`, `.External.graphics`, and `.Call.graphics` return an R object.

These calls are typically made in conjunction with `dyn.load` which links DLLs to R.

The `.graphics` versions of `.Call` and `.External` are used when calling code which makes low-level graphics calls. They take additional steps to ensure that the device driver display lists are updated correctly.

Argument types

The mapping of the types of R arguments to C or Fortran arguments in `.C` or `.Fortran` is

R	C	Fortran
integer	int *	integer
numeric	double *	double precision
– or –	float *	real
complex	Rcomplex *	double complex
logical	int *	integer
character	char **	[see below]
raw	unsigned char *	not allowed
list	SEXP *	not allowed
other	SEXP	not allowed

Numeric vectors in R will be passed as type `double *` to C (and as `double precision` to Fortran) unless (i) `.C` or `.Fortran` is used, (ii) `DUP` is false and (iii) the argument has attribute `Csingle` set to `TRUE` (use `as.single` or `single`). This mechanism is only intended to be used to facilitate the interfacing of existing C and Fortran code.

The C type `Rcomplex` is defined in 'Complex.h' as a `typedef struct {double r; double i;}`. Fortran type `double complex` is an extension to the Fortran standard, and the availability of a mapping of `complex` to Fortran may be compiler dependent.

Note: The C types corresponding to `integer` and `logical` are `int`, not `long` as in S. This difference matters on 64-bit platforms.

The first character string of a character vector is passed as a C character array to Fortran: that string may be usable as `character*255` if its true length is passed separately. Only up to 255 characters of the string are passed back. (How well this works, or even if it works at all, depends on the C and Fortran compilers and the platform.)

Missing (`NA`) string values are passed to `.C` as the string "NA". As the C `char` type can represent all possible bit patterns there appears to be no way to distinguish missing strings from the string `"NA"`. If this distinction is important use `.Call`.

Functions, expressions, environments and other language elements are passed as the internal R pointer type `SEXP`. This type is defined in 'Rinternals.h' or the arguments can be declared as generic pointers, `void *`. Lists are passed as C arrays of `SEXP` and can be declared as `void *` or `SEXP *`. Note that you cannot assign values to the elements of the list within the C routine. Assigning values to elements of the array corresponding to the list bypasses R's memory management/garbage collection and will cause problems. Essentially, the array corresponding to the list is read-only. If you need to return S objects created within the C routine, use the `.Call` interface.

R functions can be invoked using `call_S` or `call_R` and can be passed lists or the simple types as arguments.

Warning

DUP=FALSE is dangerous.

There are two dangers with using DUP=FALSE.

The first is that if you pass a local variable to .C/.Fortran with DUP=FALSE, your compiled code can alter the local variable and not just the copy in the return list. Worse, if you pass a local variable that is a formal parameter of the calling function, you may be able to change not only the local variable but the variable one level up. This will be very hard to trace.

The second is that lists are passed as a single R SEXP with DUP=FALSE, not as an array of SEXP. This means the accessor macros in 'Rinternals.h' are needed to get at the list elements and the lists cannot be passed to call_S/call_R. New code using R objects should be written using .Call or .External, so this is now only a minor issue.

In addition, character vectors and lists cannot be used with DUP=FALSE.

It is safe and useful to set DUP=FALSE if you do not change any of the variables that might be affected, e.g.,

```
.C("Cfunction", input=x, output=numeric(10)).
```

In this case the output variable did not exist before the call so it cannot cause trouble. If the input variable is not changed in the C code of Cfunction you are safe.

Neither .Call nor .External copy their arguments. You should treat arguments you receive through these interfaces as read-only.

Fortran symbol names

All compilers that can be used with R map symbol names to lower case, and so does .Fortran.

Symbol names containing underscores are not valid Fortran 77 (although they are valid in Fortran 9x). Many Fortran 77 compilers (including g77) will allow them but translate them in a different way to names not containing underscores. As from R 2.4.0 such names will work with .Fortran, but portable code should not use Fortran names containing underscores.

Use .Fortran with care for compiled Fortran 9x code: it may not work if the Fortran 9x compiler used differs from the Fortran compiler used when configuring R, especially if the subroutine name is not lower-case or includes an underscore.

Header files for external code

Writing code for use with .External and .Call will need to use internal R structures. If possible use just those defined in 'Rinternals.h' and/or the macros in 'Rdefines.h', as other header files are not installed and are even more likely to be changed.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (.C and .Fortran.)

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer. (.Call.)

See Also

dyn.load.

`formals` *Access to and Manipulation of the Formal Arguments*

Description

Get or set the formal arguments of a function.

Usage

```
formals(fun = sys.function(sys.parent()))
formals(fun, envir = environment(fun)) <- value
```

Arguments

`fun`	a function object, or see Details.
`envir`	environment in which the function should be defined.
`value`	a list (or pairlist) of R expressions.

Details

For the first form, `fun` can also be a character string naming the function to be manipulated, which is searched for from the parent environment. If it is not specified, the function calling `formals` is used.

Value

`formals` returns the formal argument list of the function specified, as a `pairlist`.

The replacment form sets the formals of a function to the list/pairlist on the right hand side, and (potentially) resets the environment of the function.

See Also

`args` for a "human-readable" version, `alist`, `body`, `function`.

Examples

```
length(formals(lm))       # the number of formal arguments
names(formals(boxplot))   # formal arguments names

f <- function(x) a+b
formals(f) <- alist(a=,b=3) # function(a,b=3)a+b
f(2) # result = 5
```

format *Encode in a Common Format*

Description

Format an R object for pretty printing.

Usage

```
format(x, ...)

## Default S3 method:
format(x, trim = FALSE, digits = NULL, nsmall = 0,
       justify = c("left", "right", "centre", "none"),
       width = NULL, na.encode = TRUE, scientific = NA,
       big.mark = "",   big.interval = 3,
     small.mark = "", small.interval = 5,
   decimal.mark = ".", zero.print = NULL, ...)

## S3 method for class 'data.frame':
format(x, ..., justify = "none")

## S3 method for class 'factor':
format(x, ...)

## S3 method for class 'AsIs':
format(x, width = 12, ...)
```

Arguments

x	any R object (conceptually); typically numeric.
trim	logical; if FALSE, logical, numeric and complex values are right-justified to a common width: if TRUE the leading blanks for justification are suppressed.
digits	how many significant digits are to be used for numeric and complex x. The default, NULL, uses getOption(digits). This is a suggestion: enough decimal places will be used so that the smallest (in magnitude) number has this many significant digits, and also to satisfy nsmall. (For the intepretation for complex numbers see signif.)
nsmall	the minimum number of digits to the right of the decimal point in formatting real/complex numbers in non-scientific formats. Allowed values are 0 <= nsmall <= 20.
justify	should a *character* vector be left-justified (the default), right-justified, centred or left alone.
width	default method: the *minimum* field width or NULL or 0 for no restriction. AsIs method: the *maximum* field width for non-character objects. NULL corresponds to the default 12.
na.encode	logical: should NA strings be encoded?

`scientific` Either a logical specifiying whether elements of a real or complex vector should be encoded in scientific format, or an integer penalty (see `options("scipen")`. Missing values correspond to the current default penalty.

`...` further arguments passed to or from other methods.

`big.mark, big.interval, small.mark, small.interval, decimal.mark, zero.print` used for prettying longer decimal sequences, passed to `prettyNum`: that help page explains the details.

Details

`format` is a generic function. Apart from the methods described here there are methods for dates (see `format.Date`), date-times (see `format.POSIXct`)) and for other classes such as `format.octmode` and `format.dist`.

`format.data.frame` formats the data frame column by column, applying the appropriate method of `format` for each column. Methods for columns are often similar to `as.character` but offer more control. Matrix and data-frame columns will be converted to separate columns in the result, and character columns (normally all) will be given class `"AsIs"`.

`format.factor` converts the factor to a character vector and then calls the default method (and so `justify` applies).

`format.AsIs` deals with columns of complicated objects that have been extracted from a data frame. Character objects are passed to the default method (and so `width` does not apply). Otherwise it calls `toString` to convert the object to character (if a vector or list, element by element) and then right-justifies the result.

Justification for character vectors (and objects converted to character vectors by their methods) is done on display width (see `nchar`), taking double-width characters and the rendering of special characters (as escape sequences, including escaping backslash: see `print.default`) into account. Character strings are padded with blanks to the display width of the widest. (If `na.encode = FALSE` missing character strings are not included in the width computations and are not encoded.)

Numeric vectors are encoded with the minimum number of decimal places needed to display all the elements to at least the `digit` significant digits. However, if all the elements then have trailing zeroes, the number of decimal places is reduced until at least one element has a non-zero final digit.

Raw vectors are converted to their 2-digit hexadecimal representation by `as.character`.

Value

An object of similar structure to `x` containing character representations of the elements of the first argument `x` in a common format.

For numeric or complex `x`, dims and dimnames are preserved on matrices/arrays and names on vectors: no other attributes are copied.

If `x` is a list, the result is a character vector obtained by applying `format.default(x, ...)` to each element of the list (after `unlist`ing elements which are themselves lists), and then collapsing the result for each element with `paste(collapse = ", ")`. The defaults in this case are `trim = TRUE, justify = "none"` since one does not usually want alignment in the collapsed strings.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

format.info indicates how an atomic vector would be formatted.

formatC, paste, as.character, sprintf, print, toString, encodeString.

Examples

```
format(1:10)
format(1:10, trim = TRUE)

zz <- data.frame("(row names)"= c("aaaaa", "b"), check.names=FALSE)
format(zz)
format(zz, justify = "left")

## use of nsmall
format(13.7)
format(13.7, nsmall = 3)
format(c(6.0, 13.1), digits = 2)
format(c(6.0, 13.1), digits = 2, nsmall = 1)

## use of scientific
format(2^31-1)
format(2^31-1, sci = TRUE)

## a list
z <- list(a=letters[1:3], b=(-pi+0i)^((-2:2)/2), c=c(1,10,100,1000),
          d=c("a", "longer", "character", "string"))
format(z, digits = 2)
format(z, digits = 2, justify = "left", trim = FALSE)
```

---

format.Date *Date Conversion Functions to and from Character*

Description

Functions to convert between character representations and objects of class "Date" representing calendar dates.

Usage

```
as.Date(x, ...)
## S3 method for class 'character':
as.Date(x, format = "", ...)

## S3 method for class 'Date':
format(x, ...)

## S3 method for class 'Date':
as.character(x, ...)
```

Arguments

x	An object to be converted.
format	A character string. The default is "%Y-%m-%d". For details see strftime.
...	Further arguments to be passed from or to other methods, including format for as.character and as.Date methods.

Details

The usual vector re-cycling rules are applied to x and format so the answer will be of length that of the longer of the vectors.

Locale-specific conversions to and from character strings are used where appropriate and available. This affects the names of the days and months.

The as.Date methods accept character strings, factors, logical NA and objects of classes "POSIXlt" and "POSIXct". (The last are converted to days by ignoring the time after midnight in the representation of the time in UTC.) Also objects of class "date" (from package **date** or **survival**) and "dates" (from package **chron**).

See the examples for how to convert a day given as the number of days since an epoch.

The format and as.character methods ignore any fractional part of the date.

Value

The format and as.character methods return a character vector representing the date.

The as.Date methods return an object of class "Date".

Note

The default formats follow the rules of the ISO 8601 international standard which expresses a day as "2001-02-03".

If the date string does not specify the date completely, the returned answer may be system-specific. The most common behaviour is to assume that a missing year, month or day is the current one. If it specifies a date incorrectly, reliable implementations will give an error and the date is reported as NA. Unfortunately some common implementations (such as 'glibc') are unreliable and guess at the intended meaning.

Years before 1CE (aka 1AD) will probably not be handled correctly.

References

International Organization for Standardization (2004, 1988, 1997, ...) *ISO 8601. Data elements and interchange formats – Information interchange – Representation of dates and times.* For links to versions available on-line see (at the time of writing) http://www.qsl.net/g1smd/isopdf.htm; for information on the current official version, see http://www.iso.org/iso/en/prods-services/popstds/datesandtime.html.

See Also

Date for details of the date class; locales to query or set a locale.

Your system's help pages on strftime and strptime to see how to specify their formats.

Examples

```
## locale-specific version of the date
format(Sys.Date(), "%a %b %d")

## read in date info in format 'ddmmmyyyy'
## This will give NA(s) in some locales; setting the C locale
## as in the commented lines will overcome this on most systems.
## lct <- Sys.getlocale("LC_TIME"); Sys.setlocale("LC_TIME", "C")
x <- c("1jan1960", "2jan1960", "31mar1960", "30jul1960")
z <- as.Date(x, "%d%b%Y")
## Sys.setlocale("LC_TIME", lct)
z

## read in date/time info in format 'm/d/y'
dates <- c("02/27/92", "02/27/92", "01/14/92", "02/28/92", "02/01/92")
as.Date(dates, "%m/%d/%y")

## date given as number of days since 1900-01-01 (a date in 1989)
as.Date("1900-01-01") + 32768
```

---

format.info *format(.) Information*

Description

Information is returned on how `format(x, digits, nsmall)` would be formatted.

Usage

```
format.info(x, digits = NULL, nsmall = 0)
```

Arguments

`x`	an atomic vector; a potential argument of `format(x, ...)`.
`digits`	how many significant digits are to be used for numeric and complex `x`. The default, `NULL`, uses `getOption(digits)`.
`nsmall`	(see `format(..., nsmall)`).

Value

An `integer vector` of length 1, 3 or 6, say `r`.

For logical, integer and character vectors a single element, the width which would be used by `format` if `width = NULL`.

For numeric vectors:

`r[1]`	width (in characters) used by `format(x)`
`r[2]`	number of digits after decimal point.
`r[3]`	in `0:2`; if ≥ 1, *exponential* representation would be used, with exponent length of `r[3]+1`.

For a complex vector the first three elements refer to the real parts, and there are three further elements corresponding to the imaginary parts.

See Also

format, formatC.

Examples

```
dd <- options("digits") ; options(digits = 7) #-- for the following
format.info(123)    # 3 0 0
format.info(pi)     # 8 6 0
format.info(1e8)    # 5 0 1 - exponential "1e+08"
format.info(1e222) # 6 0 2 - exponential "1e+222"

x <- pi*10^c(-10,-2,0:2,8,20)
names(x) <- formatC(x,w=1,dig=3,format="g")
cbind(sapply(x,format))
t(sapply(x, format.info))

## using at least 8 digits right of "."
t(sapply(x, format.info, nsmall = 8))

# Reset old options:
options(dd)
```

---

format.pval *Format P Values*

Description

format.pval is intended for formatting p-values.

Usage

```
format.pval(pv, digits = max(1, getOption("digits") - 2),
            eps = .Machine$double.eps, na.form = "NA")
```

Arguments

pv	a numeric vector.
digits	how many significant digits are to be used.
eps	a numerical tolerance: see Details.
na.form	character representation of NAs.

Details

format.pval is mainly an auxiliary function for print.summary.lm etc., and does separate formatting for fixed, floating point and very small values; those less than eps are formatted as "< [eps]" (where "[eps]" stands for format(eps, digits)).

Value

A character vector.

Examples

```
format.pval(c(runif(5), pi^-100, NA))
format.pval(c(0.1, 0.0001, 1e-27))
```

`formatC` *Formatting Using C-style Formats*

Description

Formatting numbers individually and flexibly, using `C` style format specifications.

Usage

```
formatC(x, digits = NULL, width = NULL,
        format = NULL, flag = "", mode = NULL,
        big.mark = "", big.interval = 3,
      small.mark = "", small.interval = 5,
    decimal.mark = ".", preserve.width = "individual")

prettyNum(x, big.mark = "",   big.interval = 3,
          small.mark = "", small.interval = 5,
          decimal.mark = ".",
          preserve.width = c("common", "individual", "none"),
          zero.print = NULL, ...)
```

Arguments

`x`	an atomic numerical or character object, typically a vector of real numbers.
`digits`	the desired number of digits after the decimal point (`format = "f"`) or *significant* digits (`format = "g"`, `= "e"` or `= "fg"`). Default: 2 for integer, 4 for real numbers. If less than 0, the C default of 6 digits is used.
`width`	the total field width; if both `digits` and `width` are unspecified, `width` defaults to 1, otherwise to `digits + 1`. `width = 0` will use `width = digits`, `width < 0` means left justify the number in this field (equivalent to `flag ="-"`). If necessary, the result will have more characters than `width`.
`format`	equal to `"d"` (for integers), `"f"`, `"e"`, `"E"`, `"g"`, `"G"`, `"fg"` (for reals), or `"s"` (for strings). Default is `"d"` for integers, `"g"` for reals. `"f"` gives numbers in the usual `xxx.xxx` format; `"e"` and `"E"` give `n.ddde+nn` or `n.dddE+nn` (scientific format); `"g"` and `"G"` put `x[i]` into scientific format only if it saves space to do so. `"fg"` uses fixed format as `"f"`, but `digits` as the minimum number of *significant* digits. That this can lead to quite long result strings, see examples below. Note that unlike `signif` this prints large numbers with more significant digits than `digits`.
`flag`	For `formatC`, a character string giving a format modifier as in Kernighan and Ritchie (1988, page 243). `"0"` pads leading zeros; `"-"` does left adjustment, others are `"+"`, `" "`, and `"#"`. There can be more than one of these, in any order.

mode	"double" (or "real"), "integer" or "character". Default: Determined from the storage mode of x.
big.mark	character; if not empty used as mark between every big.interval decimals *before* (hence big) the decimal point.
big.interval	see big.mark above; defaults to 3.
small.mark	character; if not empty used as mark between every small.interval decimals *after* (hence small) the decimal point.
small.interval	see small.mark above; defaults to 5.
decimal.mark	the character to be used to indicate the numeric decimal point.
preserve.width	string specifying if the string widths should be preserved where possible in those cases where marks (big.mark or small.mark) are added. "common", the default, corresponds to format-like behavior whereas "individual" is the default in formatC().
zero.print	logical, character string or NULL specifying if and how *zeros* should be formatted specially. Useful for pretty printing "sparse" objects.
...	arguments passed to format.

Details

If you set format it overrides the setting of mode, so formatC(123.45, mode="double", format="d") gives 123.

The rendering of scientific format is platform-dependent: some systems use n.ddde+nnn or n.dddenn rather than n.ddde+nn.

formatC does not necessarily align the numbers on the decimal point, so formatC(c(6.11, 13.1), digits=2, format="fg") gives c("6.1", " 13"). If you want common formatting for several numbers, use format.

prettyNum is the utility function for prettifying x. If x is not a character, format(x[i], ...) is applied to each element, and then it is left unchanged if all the other arguments are at their defaults. Note that prettyNum(x) may behave unexpectedly if x is a character vector not resulting from something like format(<number>): in particular it assumes that a period is a decimal mark.

Value

A character object of same size and attributes as x. Unlike format, each number is formatted individually. Looping over each element of x, the C function sprintf(...) is called (inside the C function str_signif).

formatC: for character x, do simple (left or right) padding with white space.

Author(s)

formatC was originally written by Bill Dunlap, later much improved by Martin Maechler. It was first adapted for R by Friedrich Leisch.

References

Kernighan, B. W. and Ritchie, D. M. (1988) *The C Programming Language.* Second edition. Prentice Hall.

See Also

format.

sprintf for more general C like formatting.

Examples

```
xx  <- pi * 10^(-5:4)
cbind(format(xx, digits=4), formatC(xx))
cbind(formatC(xx, wid = 9, flag = "-"))
cbind(formatC(xx, dig = 5, wid = 8, format = "f", flag = "0"))
cbind(format(xx, digits=4), formatC(xx, dig = 4, format = "fg"))

formatC(    c("a", "Abc", "no way"), wid = -7)  # <=> flag = "-"
formatC(c((-1:1)/0,c(1,100)*pi), wid=8, dig=1)

xx <- c(1e-12,-3.98765e-10,1.45645e-69,1e-70,pi*1e37,3.44e4)
##       1        2             3        4      5       6
formatC(xx)
formatC(xx, format="fg")       # special "fixed" format.
formatC(xx, format="f", dig=80)#>> also long strings

r <- c("76491283764.97430", "29.12345678901", "-7.1234", "-100.1","1123")
## American:
prettyNum(r, big.mark = ",")
## Some Europeans:
prettyNum(r, big.mark = "'", decimal.mark = ",")

(dd <- sapply(1:10, function(i)paste((9:0)[1:i],collapse="")))
prettyNum(dd, big.mark="'")

## examples of 'small.mark'
pN <- stats::pnorm(1:7, lower=FALSE)
cbind(format (pN, small.mark = " ", digits = 15))
cbind(formatC(pN, small.mark = " ", digits = 17, format = "f"))

cbind(ff <- format(1.2345 + 10^(0:5), width = 11, big.mark = "'"))
## all with same width (one more than the specified minimum)

## individual formatting to common width:
fc <- formatC(1.234 + 10^(0:8), format="fg", width=11, big.mark = "'")
cbind(fc)
```

---

formatDL *Format Description Lists*

Description

Format vectors of items and their descriptions as 2-column tables or LaTeX-style description lists.

Usage

```
formatDL(x, y, style = c("table", "list"),
         width = 0.9 * getOption("width"), indent = NULL)
```

Arguments

x	a vector giving the items to be described, or a list of length 2 or a matrix with 2 columns giving both items and descriptions.
y	a vector of the same length as x with the corresponding descriptions. Only used if x does not already give the descriptions.
style	a character string specifying the rendering style of the description information. If "table", a two-column table with items and descriptions as columns is produced (similar to Texinfo's @table environment. If "list", a LaTeX-style tagged description list is obtained.
width	a positive integer giving the target column for wrapping lines in the output.
indent	a positive integer specifying the indentation of the second column in table style, and the indentation of continuation lines in list style. Must not be greater than width/2, and defaults to width/3 for table style and width/9 for list style.

Details

After extracting the vectors of items and corresponding descriptions from the arguments, both are coerced to character vectors.

In table style, items with more than indent - 3 characters are displayed on a line of their own.

Value

a character vector with the formatted entries.

Examples

```
## Use R to create the 'INDEX' for package 'splines' from its 'CONTENTS'
x <- read.dcf(file = system.file("CONTENTS", package = "splines"),
              fields = c("Entry", "Description"))
x <- as.data.frame(x)
writeLines(formatDL(x$Entry, x$Description))
## or equivalently: writeLines(formatDL(x))
## Same information in tagged description list style:
writeLines(formatDL(x$Entry, x$Description, style = "list"))
## or equivalently: writeLines(formatDL(x, style = "list"))
```

---

function *Function Definition*

Description

These functions provide the base mechanisms for defining new functions in the R language.

Usage

```
function( arglist ) expr
return(value)
```

Arguments

arglist	Empty or one or more name or name=expression terms.
value	An expression.

Details

In R (unlike S) the names in an argument list cannot be quoted non-standard names.

If value is missing, NULL is returned. If it is a single expression, the value of the evaluated expression is returned.

If the end of a function is reached without calling return, the value of the last evaluated expression is returned.

Warning

Prior to R 1.8.0, value could be a series of non-empty expressions separated by commas. In that case the value returned is a list of the evaluated expressions, with names set to the expressions where these are the names of R objects. That is, a=foo() names the list component a and gives it value the result of evaluating foo().

This has been deprecated (and a warning is given), as it was never documented in S, and whether or not the list is named differs by S versions.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

args and body for accessing the arguments and body of a function.

debug for debugging; using invisible inside return(.) for returning *invisibly*.

Examples

```
norm <- function(x) sqrt(x%*%x)
norm(1:4)

## An anonymous function:
(function(x,y){ z <- x^2 + y^2; x+y+z })(0:7, 1)
```

---

gc *Garbage Collection*

Description

A call of gc causes a garbage collection to take place. gcinfo sets a flag so that automatic collection is either silent (verbose=FALSE) or prints memory usage statistics (verbose=TRUE).

Usage

```
gc(verbose = getOption("verbose"), reset=FALSE)
gcinfo(verbose)
```

Arguments

verbose	logical; if TRUE, the garbage collection prints statistics about cons cells and the space allocated for vectors.
reset	logical; if TRUE the values for maximum space used are reset to the current values.

Details

A call of gc causes a garbage collection to take place. This will also take place automatically without user intervention, and the primary purpose of calling gc is for the report on memory usage.

However, it can be useful to call gc after a large object has been removed, as this may prompt R to return memory to the operating system.

R allocates space for vectors in multiples of 8 bytes: hence the report of "Vcells", a relict of an earlier allocator (that used a vector heap).

Value

gc returns a matrix with rows "Ncells" (*cons cells*), usually 28 bytes each on 32-bit systems and 56 bytes on 64-bit systems, and "Vcells" (*vector cells*, 8 bytes each), and columns "used" and "gc trigger", each also interpreted in megabytes (rounded up to the next 0.1Mb).

If maxima have been set for either "Ncells" or "Vcells", a fifth column is printed giving the current limits in Mb (with NA denoting no limit).

The final two columns show the maximum space used since the last call to gc(reset=TRUE) (or since R started).

gcinfo returns the previous value of the flag.

See Also

Memory on R's memory management, and gctorture if you are an R hacker.

reg.finalizer for actions to happen at garbage collection.

Examples

```
gc() #- do it now
gcinfo(TRUE) #-- in the future, show when R does it
x <- integer(100000); for(i in 1:18) x <- c(x,i)
gcinfo(verbose = FALSE)#-- don't show it anymore

gc(TRUE)

gc(reset=TRUE)
```

gc.time *Report Time Spent in Garbage Collection*

Description

This function reports the time spent in garbage collection so far in the R session while GC timing was enabled..

Usage

```
gc.time(on = TRUE)
```

Arguments

on	logical; if TRUE, GC timing is enabled.

Value

A numerical vector of length 5 giving the user CPU time, the system CPU time, the elapsed time and children's user and system CPU times (normally both zero).

Warnings

This is experimental functionality, likely to be removed as soon as the next release.

The timings are rounded up by the sampling interval for timing processes, and so are likely to be over-estimates.

See Also

gc, proc.time for the timings for the session.

Examples

```
gc.time()
```

gctorture *Torture Garbage Collector*

Description

Provokes garbage collection on (nearly) every memory allocation. Intended to ferret out memory protection bugs. Also makes R run *very* slowly, unfortunately.

Usage

```
gctorture(on = TRUE)
```

Arguments

on	logical; turning it on/off.

Value

Previous value.

Author(s)

Peter Dalgaard

`get` *Return the Value of a Named Object*

Description

Search for an R object with a given name and return it.

Usage

```
get(x, pos = -1, envir = as.environment(pos), mode = "any",
    inherits = TRUE)

mget(x, envir, mode = "any",
     ifnotfound = list(function(x) stop(paste("value for '",
     x, "' not found", sep = "")), call. = FALSE)), inherits = FALSE)
```

Arguments

`x`	a variable name (given as a character string).
`pos`	where to look for the object (see the details section); if omitted, the function will search as if the name of the object appeared unquoted in an expression.
`envir`	an alternative way to specify an environment to look in; see the Details section.
`mode`	the mode or type of object sought: see the Details section.
`inherits`	should the enclosing frames of the environment be searched?
`ifnotfound`	A `list` of values to be used if the item is not found: it will be coerced to list if necessary.

Details

The `pos` argument can specify the environment in which to look for the object in any of several ways: as an integer (the position in the `search` list); as the character string name of an element in the search list; or as an `environment` (including using `sys.frame` to access the currently active function calls). The `envir` argument is an alternative way to specify an environment, but is primarily there for back compatibility.

This function looks to see if the name `x` has a value bound to it in the specified environment. If `inherits` is `TRUE` and a value is not found for `x` in the specified environment, the enclosing frames of the environment are searched until the name `x` is encountered. See `environment` and the 'R Language Definition' manual for details about the structure of environments and their enclosures.

Warning: `inherits = TRUE` is the default behaviour for R but not for S.

If `mode` is specified then only objects of that type are sought. The `mode` may specify one of the collections `"numeric"` and `"function"` (see `mode`): any member of the collection will suffice.

Using a NULL environment is equivalent to using the current environment.

For mget multiple values are returned in a named list. This is true even if only one value is requested. The value in mode and ifnotfound can be either the same length as the number of requested items or of length 1. The argument ifnotfound must be a list containing either the value to use if the requested item is not found or a function of one argument which will be called if the item is not found, with argument the name of the item being requested. The default value for inherits is FALSE, in contrast to the default behavior for get.

mode here is a mixture of the meanings of typeof and mode: "function" covers primitive functions and operators, "numeric", "integer", "real" and "double" all refer to any numeric type, "symbol" and "name" are equivalent *but* "language" must be used.

Value

The object found. (If no object is found an error results.)

Note

The reverse of a <- get(nam) is assign(nam, a).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

exists, assign.

Examples

```
get("%o%")

##test mget
e1 <- new.env()
mget(letters, e1, ifnotfound = as.list(LETTERS))
```

---

getCallingDLL *Compute DLL for native interface call*

Description

This is an internal function that is called from R's C code to determine the enclosing namespace of a .C/.Call/.Fortran/.External call which has no PACKAGE argument. If the call has been made from a function within a namespace, then we can find the DLL associated with that namespace. The purpose of this is to avoid having to use the PACKAGE argument in these native calls and so better support versions of packages.

This is an internal function that may be migrated to internal C code in the future and so should not be used by R programmers.

Usage

```
getCallingDLL(f = sys.function(-1), doStop = FALSE)
getCallingDLLe(e)
```

Arguments

`f`	the function whose namespace and DLL are to be found. By default, this is the current function being called which is the one in which the native routine is being invoked.
`doStop`	a logical value indicating whether failure to find a namespace and/or DLL is an error (`TRUE`) or not (`FALSE`). The default is `FALSE` so that when this is called because there is no PACKAGE argument in a `.C`, `.Call`, `.Fortran`, `.External` call, no error occurs and the regular lookup is performed by searching all DLLs in order.
`e`	an environment.

See Also

`.C`, `.Call`, `.Fortran`, `.External`

Examples

```
if(exists("ansari.test"))
   getCallingDLL(ansari.test)
```

`getDLLRegisteredRoutines`

Reflectance Information for C/Fortran routines in a DLL

Description

This function allows us to query the set of routines in a DLL that are registered with R to enhance dynamic lookup, error handling when calling native routines, and potentially security in the future. This function provides a description of each of the registered routines in the DLL for the different interfaces, i.e. `.C`, `.Call`, `.Fortran` and `.External`.

Usage

```
getDLLRegisteredRoutines(dll, addNames = TRUE)
```

Arguments

`dll`	a character string or `DLLInfo` object (as returned by `getLoadedDLLs`).
`addNames`	a logical value. If this is `TRUE`, the elements of the returned lists are named using the names of the routines (as seen by R via registration or raw name). If `FALSE`, these names are not computed and assigned to the lists. As a result, the call should be quicker. The name information is also available in the `NativeSymbolInfo` objects in the lists.

Details

This takes the registration information after it has been registered and processed by the R internals. In other words, it uses the extended information

Value

A list with four elements corresponding to the routines registered for the .C, .Call, .Fortran and .External interfaces. Each element is a list with as many elements as there were routines registered for that interface. Each element identifies a routine and is an object of class `NativeSymbolInfo`. An object of this class has the following fields:

`name`	the registered name of the routine (not necessarily the name in the C code).
`address`	the memory address of the routine as resolved in the loaded DLL. This may be `NULL` if the symbol has not yet been resolved.
`dll`	an object of class `DLLInfo` describing the DLL. This is same for all elements returned.
`numParameters`	the number of arguments the native routine is to be called with. In the future, we will provide information about the types of the parameters also.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

References

"Writing R Extensions Manual" for symbol registration. R News, Volume 1/3, September 2001. "In search of C/C++ & Fortran Symbols"

See Also

`getLoadedDLLs`

Examples

```
dlls <- getLoadedDLLs()
getDLLRegisteredRoutines(dlls[["base"]])

getDLLRegisteredRoutines("stats")
```

`getLoadedDLLs` *Get DLLs Loaded in Current Session*

Description

This function provides a way to get a list of all the Dynamically Loadable Libraries (DLLs) that are currently loaded in the current R session.

Usage

```
getLoadedDLLs()
```

Details

This queries the internal table that manages the DLLs.

Value

An object of class `"DLLInfoList"` which is a list with an element corresponding to each DLL that is currently loaded in the session. Each element is an object of class `"DLLInfo"` which has the following entries.

`name`	the abbreviated name.
`path`	the fully qualified name of the file which was dynamically loaded.
`dynamicLookup`	a logical value indicating whether R uses only the registration information to resolve symbols or whether it searches the entire symbol table of the DLL.
`handle`	a reference to the C-level data structure that provides access to the contents of the DLL. This is an object of class `"DLLHandle"`.

Note that the class `DLLInfo` has an overloaded method for `$` which can be used to resolve native symbols within that DLL. Therefore, one must access the R-level elements described above using `[[`, e.g. `x[["name"]]` or `x[["handle"]]`.

Note

We are starting to use the `handle` elements in the DLL object to resolve symbols more directly in R.

Author(s)

Duncan Temple Lang ⟨duncan@wald.ucdavis.edu⟩.

See Also

`getDLLRegisteredRoutines`, `getNativeSymbolInfo`

Examples

```
getLoadedDLLs()
```

---

getNativeSymbolInfo

Obtain a Description of one or more Native (C/Fortran) Symbols

---

Description

This finds and returns as comprehensive a description of one or more dynamically loaded or "exported" built-in native symbols. For each name, it returns information about the name of the symbol, the library in which it is located and, if available, the number of arguments it expects and by which interface it should be called (i.e `.Call`, `.C`, `.Fortran`, or `.External`). Additionally, it returns the address of the symbol and this can be passed to other C routines which can invoke. Specifically,

this provides a way to explicitly share symbols between different dynamically loaded package libraries. Also, it provides a way to query where symbols were resolved, and aids diagnosing strange behavior associated with dynamic resolution.

This is now vectorized in the name argument so can process multiple symbols in a single call. The result is a list that can be indexed by the given symbol names.

Usage

```
getNativeSymbolInfo(name, PACKAGE, unlist = TRUE,
                    withRegistrationInfo = FALSE)
```

Arguments

name	the name(s) of the native symbol(s) as used in a call to is.loaded, etc. Note that Fortran symbols should be supplied as-is, not wrapped in symbol.For.
PACKAGE	an optional argument that specifies to which dynamically loaded library we restrict the search for this symbol. If this is "base", we search in the R executable itself.
unlist	a logical value which controls how the result is returned if the function is called with the name of a single symbol. If unlist is TRUE and the number of symbol names in name is one, then the NativeSymbolInfo object is returned. If it is FALSE, then a list of NativeSymbolInfo objects is returned. This is ignored if the number of symbols passed in name is more than one. To be compatible with earlier versions of this function, this defaults to TRUE.
withRegistrationInfo	a logical value indicating whether, if TRUE, to return information that was registered with R about the symbol and its parameter types if such information is available, or if FALSE to return the address of the symbol.

Details

This uses the same mechanism for resolving symbols as is used in all the native interfaces (.Call, etc.). If the symbol has been explicitly registered by the shared library in which it is contained, information about the number of arguments and the interface by which it should be called will be returned. Otherwise, a generic native symbol object is returned.

Value

Generally, a list of NativeSymbolInfo elements whose elements can be indexed by the elements of name in the call. Each NativeSymbolInfo object is a list containing the following elements:

name	the name of the symbol, as given by the name argument.
address	if withRegistrationInfo is FALSE, this is the native memory address of the symbol which can be used to invoke the routine, and also to compare with other symbol addresses. This is an external pointer object and of class NativeSymbol. If withRegistrationInfo is TRUE and registration information is available for the symbol, then this is an object of class RegisteredNativeSymbol and is a reference to an internal data type that has access to the routine pointer and registration information. This too can be used in calls to .Call, .C, .Fortran and .External.
package	a list containing 3 elements:

name the short form of the library name which can be used as the value of the `PACKAGE` argument in the different native interface functions.

path the fully qualified name of the shared library file.

dynamicLookup a logical value indicating whether dynamic resolution is used when looking for symbols in this library, or only registered routines can be located.

`numParameters`
: the number of arguments that should be passed in a call to this routine.

Additionally, the list will have an additional class, being `CRoutine`, `CallRoutine`, `FortranRoutine` or `ExternalRoutine` corresponding to the R interface by which it should be invoked.

If any of the symbols is not found, an error is immediately raised.

If `name` contains only one symbol name and `unlist` is `TRUE`, then the single `NativeSymbolInfo` is returned rather than the list containing that one element.

Note

One motivation for accessing this reflectance information is to be able to pass native routines to C routines as "function pointers" in C. This allows us to treat native routines and R functions in a similar manner, such as when passing an R function to C code that makes callbacks to that function at different points in its computation (e.g., `nls`). Additionally, we can resolve the symbol just once and avoid resolving it repeatedly or using the internal cache. In the future, one may be able to treat `NativeSymbol` objects directly as callback objects.

Author(s)

Duncan Temple Lang

References

For information about registering native routines, see "In Search of C/C++ & FORTRAN Routines", R News, volume 1, number 3, 2001, p20–23 (http://CRAN.R-project.org/doc/Rnews/).

See Also

`getDLLRegisteredRoutines`, `is.loaded`, `.C`, `.Fortran`, `.External`, `.Call`, `dyn.load`.

Examples

```
library(stats) # normally loaded
getNativeSymbolInfo("dansari")

getNativeSymbolInfo("hcass2")  # a Fortran symbol
```

`getNumCConverters` *Management of .C argument conversion list*

Description

These functions provide facilities to manage the extensible list of converters used to translate R objects to C pointers for use in `.C` calls. The number and a description of each element in the list can be retrieved. One can also query and set the activity status of individual elements, temporarily ignoring them. And one can remove individual elements.

Usage

```
getNumCConverters()
getCConverterDescriptions()
getCConverterStatus()
setCConverterStatus(id, status)
removeCConverter(id)
```

Arguments

`id`	either a number or a string identifying the element of interest in the converter list. A string is matched against the description strings for each element to identify the element. Integers are specified starting at 1 (rather than 0).
`status`	a logical value specifying whether the element is to be considered active (`TRUE`) or not (`FALSE`).

Details

The internal list of converters is potentially used when converting individual arguments in a `.C` call. If an argument has a non-trivial class attribute, we iterate over the list of converters looking for the first that "matches". If we find a matching converter, we have it create the C-level pointer corresponding to the R object. When the call to the C routine is complete, we use the same converter for that argument to reverse the conversion and create an R object from the current value in the C pointer. This is done separately for all the arguments.

The functions documented here provide R user-level capabilities for investigating and managing the list of converters. There is currently no mechanism for adding an element to the converter list within the R language. This must be done in C code using the routine `R_addToCConverter()`.

Value

`getNumCConverters` returns an integer giving the number of elements in the list, both active and inactive.

`getCConverterDescriptions` returns a character vector containing the description string of each element of the converter list.

`getCConverterStatus` returns a logical vector with a value for each element in the converter list. Each value indicates whether that converter is active (`TRUE`) or inactive (`FALSE`). The names of the elements are the description strings returned by `getCConverterDescriptions`.

`setCConverterStatus` returns the logical value indicating the activity status of the specified element before the call to change it took effect. This is `TRUE` for active and `FALSE` for inactive.

`removeCConverter` returns `TRUE` if an element in the converter list was identified and removed. In the case that no such element was found, an error occurs.

Author(s)

Duncan Temple Lang

References

http://developer.R-project.org/CObjectConversion.pdf

See Also

.C

Examples

```
getNumCConverters()
getCConverterDescriptions()
getCConverterStatus()
## Not run:
old <- setCConverterStatus(1, FALSE)

setCConverterStatus(1, old)
## End(Not run)
## Not run:
removeCConverter(1)
removeCConverter(getCConverterDescriptions()[1])
## End(Not run)
```

---

getpid *Get the Process ID of the R Session*

Description

Get the process ID of the R Session. It is guaranteed by the operating system that two R sessions running simultaneously will have different IDs, but it is possible that R sessions running at different times will have the same ID.

Usage

```
Sys.getpid()
```

Value

An integer, usually a small integer between 0 and 32767 under Unix-alikes and a much small integer under Windows.

Examples

```
Sys.getpid()
```

gettext *Translate Text Messages*

Description

If Native Language Support was enabled in this build of R, attempt to translate character vectors or set where the translations are to be found.

Usage

```
gettext(..., domain = NULL)

ngettext(n, msg1, msg2, domain = NULL)

bindtextdomain(domain, dirname = NULL)
```

Arguments

`...`	One or more character vectors.
`domain`	The 'domain' for the translation.
`n`	a non-negative integer.
`msg1`	the message to be used in English for `n = 1`.
`msg2`	the message to be used in English for `n = 0, 2, 3, ...`.
`dirname`	The directory in which to find translated message catalogs for the domain.

Details

If `domain` is `NULL` or `""`, a domain is searched for based on the namespace which contains the function calling `gettext` or `ngettext`. If a suitable domain can be found, each character string is offered for translation, and replaced by its translation into the current language if one is found.

Conventionally the domain for R warning/error messages in package **pkg** is `"R-pkg"`, and that for C-level messages is `"pkg"`.

For `gettext`, leading and trailing whitespace is ignored when looking for the translation.

`ngettext` is used where the message needs to vary by a single integer. Translating such messages is subject to very specific rules for different languages: see the GNU Gettext Manual. The string will often contain a single instance of `%d` to be used in `sprintf`. If English is used, `msg1` is returned if `n == 1` and `msg2` in all other cases.

Value

For `gettext`, a character vector, one element per string in `...`. If translation is not enabled or no domain is found or no translation is found in that domain, the original strings are returned.

For `ngettext`, a character string.

For `bindtextdomain`, a character string giving the current base directory, or `NULL` if setting it failed.

See Also

`stop` and `warning` make use of `gettext` to translate messages.

`xgettext` for extracting translatable strings from R source files.

Examples

```
bindtextdomain("R")  # non-null iff NLS is enabled

for(n in 0:3)
    print(sprintf(ngettext(n, "%d variable has missing values",
                              "%d variables have missing values"),
                  n))

## Not run:
## for translation, those strings should appear in R-pkg.pot as
msgid        "%d variable has missing values"
msgid_plural "%d variables have missing values"
msgstr[0] ""
msgstr[1] ""
## End(Not run)

miss <- c("one", "or", "another")
cat(ngettext(length(miss), "variable", "variables"),
    paste(sQuote(miss), collapse=", "),
    ngettext(length(miss), "contains", "contain"), "missing values\n")

## better for translators would be to use
cat(sprintf(ngettext(length(miss),
                     "variable %s contains missing values\n",
                     "variables %s contain missing values\n"),
            paste(sQuote(miss), collapse=", ")))
```

`getwd` *Get or Set Working Directory*

Description

`getwd` returns an absolute filename representing the current working directory of the R process; `setwd(dir)` is used to set the working directory to `dir`.

Usage

```
getwd()
setwd(dir)
```

Arguments

`dir`	A character string.

Value

`getwd` returns a character vector, or `NULL` if the working directory is not available.

`setwd` returns the currrent directory before the change, invisibly. It will give an error if it does not succeed.

Note

These functions are not implemented on all platforms.

See Also

`list.files` for the *contents* of a directory.

Examples

```
(WD <- getwd())
if (!is.null(WD)) setwd(WD)
```

---

gl *Generate Factor Levels*

Description

Generate factors by specifying the pattern of their levels.

Usage

```
gl(n, k, length = n*k, labels = 1:n, ordered = FALSE)
```

Arguments

`n`	an integer giving the number of levels.
`k`	an integer giving the number of replications.
`length`	an integer giving the length of the result.
`labels`	an optional vector of labels for the resulting factor levels.
`ordered`	a logical indicating whether the result should be ordered or not.

Value

The result has levels from `1` to `n` with each value replicated in groups of length `k` out to a total length of `length`.

`gl` is modelled on the *GLIM* function of the same name.

See Also

The underlying `factor()`.

Examples

```
## First control, then treatment:
gl(2, 8, label = c("Control", "Treat"))
## 20 alternating 1s and 2s
gl(2, 1, 20)
## alternating pairs of 1s and 2s
gl(2, 2, 20)
```

`grep` *Pattern Matching and Replacement*

Description

`grep` searches for matches to `pattern` (its first argument) within the character vector `x` (second argument). `regexpr` and `gregexpr` do too, but return more detail in a different format.

`sub` and `gsub` perform replacement of matches determined by regular expression matching.

Usage

```
grep(pattern, x, ignore.case = FALSE, extended = TRUE,
     perl = FALSE, value = FALSE, fixed = FALSE, useBytes = FALSE)

sub(pattern, replacement, x,
    ignore.case = FALSE, extended = TRUE, perl = FALSE,
    fixed = FALSE, useBytes = FALSE)

gsub(pattern, replacement, x,
     ignore.case = FALSE, extended = TRUE, perl = FALSE,
     fixed = FALSE, useBytes = FALSE)

regexpr(pattern, text, extended = TRUE, perl = FALSE,
        fixed = FALSE, useBytes = FALSE)

gregexpr(pattern, text, extended = TRUE, perl = FALSE,
         fixed = FALSE, useBytes = FALSE)
```

Arguments

`pattern`	character string containing a regular expression (or character string for `fixed = TRUE`) to be matched in the given character vector. Coerced by `as.character` to a character string if possible.
`x, text`	a character vector where matches are sought, or an object which can be coerced by `as.character` to a character vector.
`ignore.case`	if `FALSE`, the pattern matching is *case sensitive* and if `TRUE`, case is ignored during matching.
`extended`	if `TRUE`, extended regular expression matching is used, and if `FALSE` basic regular expressions are used.
`perl`	logical. Should perl-compatible regexps be used? Has priority over `extended`.
`value`	if `FALSE`, a vector containing the (`integer`) indices of the matches determined by `grep` is returned, and if `TRUE`, a vector containing the matching elements themselves is returned.
`fixed`	logical. If `TRUE`, `pattern` is a string to be matched as is. Overrides all conflicting arguments.
`useBytes`	logical. If `TRUE` the matching is done byte-by-byte rather than character-by-character. See Details.

`replacement` a replacement for matched pattern in `sub` and `gsub`. Coerced to character if possible. For `fixed = FALSE` this can include backreferences `"\1"` to `"\9"` to parenthesized subexpressions of `pattern`. For `perl = TRUE` only, it can also contain `"\U"` or `"\L"` to convert the rest of the replacement to upper or lower case.

Details

Arguments which should be character strings or character vectors are coerced to character if possible.

The two `*sub` functions differ only in that `sub` replaces only the first occurrence of a `pattern` whereas `gsub` replaces all occurrences.

For `regexpr` it is an error for `pattern` to be `NA`, otherwise `NA` is permitted and matches only itself.

The regular expressions used are those specified by POSIX 1003.2, either extended or basic, depending on the value of the `extended` argument, unless `perl = TRUE` when they are those of PCRE, http://www.pcre.org/. (The exact set of patterns supported may depend on the version of PCRE installed on the system in use, if R was configured to use the system PCRE. R's internal copy used PCRE 6.7.)

`useBytes` is only used if `fixed = TRUE` or `perl = TRUE`. For `grep` its main effect is to avoid errors/warnings about invalid inputs, but for `regexpr` it changes the interpretation of the output.

Value

For `grep` a vector giving either the indices of the elements of `x` that yielded a match or, if `value` is `TRUE`, the matched elements of `x` (after coercion, preserving names but no other attributes).

For `sub` and `gsub` a character vector of the same length and with the same attributes as `x` (after possible coercion).

For `regexpr` an integer vector of the same length as `text` giving the starting position of the first match, or -1 if there is none, with attribute `"match.length"` giving the length of the matched text (or -1 for no match). In a multi-byte locale these quantities are in characters rather than bytes unless `useBytes = TRUE` is used with `fixed = TRUE` or `perl = TRUE`.

For `gregexpr` a list of the same length as `text` each element of which is an integer vector as in `regexpr`, except that the starting positions of every match are given.

If in a multi-byte locale the pattern or replacement is not a valid sequence of bytes, an error is thrown. An invalid string in `x` or `text` is a non-match with a warning for `grep` or `regexpr`, but an error for `sub` or `gsub`.

Warning

The standard regular-expression code has been reported to be very slow when applied to extremely long character strings (tens of thousands of characters or more): the code used when `perl = TRUE` seems much faster and more reliable for such usages.

The standard version of `gsub` does not substitute correctly repeated word-boundaries (e.g. `pattern = "\b"`). Use `perl = TRUE` for such matches.

The `perl = TRUE` option is only implemented for single-byte and UTF-8 encodings, and will warn if used in a non-UTF-8 multi-byte locale (unless `useBytes = TRUE`).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole (grep)

See Also

regular expression (aka regexp) for the details of the pattern specification.

glob2rx to turn wildcard matches into regular expressions.

agrep for approximate matching.

tolower, toupper and chartr for character translations. charmatch, pmatch, match. apropos uses regexps and has nice examples.

Examples

```
grep("[a-z]", letters)

txt <- c("arm","foot","lefroo", "bafoobar")
if(any(i <- grep("foo",txt)))
   cat("'foo' appears at least once in\n\t",txt,"\n")
i # 2 and 4
txt[i]

## Double all 'a' or 'b's;  "\" must be escaped, i.e., 'doubled'
gsub("([ab])", "\\1_\\1_", "abc and ABC")

txt <- c("The", "licenses", "for", "most", "software", "are",
  "designed", "to", "take", "away", "your", "freedom",
  "to", "share", "and", "change", "it.",
   "", "By", "contrast,", "the", "GNU", "General", "Public", "License",
   "is", "intended", "to", "guarantee", "your", "freedom", "to",
   "share", "and", "change", "free", "software", "--",
   "to", "make", "sure", "the", "software", "is",
   "free", "for", "all", "its", "users")
( i <- grep("[gu]", txt) ) # indices
stopifnot( txt[i] == grep("[gu]", txt, value = TRUE) )

## Note that in locales such as en_US this includes B as the
## collation order is aAbBcCdEe ...
(ot <- sub("[b-e]",".", txt))
txt[ot != gsub("[b-e]",".", txt)]#- gsub does "global" substitution

txt[gsub("g","#", txt) !=
    gsub("g","#", txt, ignore.case = TRUE)] # the "G" words

regexpr("en", txt)

gregexpr("e", txt)

## trim trailing white space
str = 'Now is the time      '
sub(' +$', '', str)  ## spaces only
sub('[[:space:]]+$', '', str) ## white space, POSIX-style
sub('\\s+$', '', str, perl = TRUE) ## Perl-style white space

## capitalizing
```

```
gsub("(\\w)(\\w*)", "\\U\\1\\L\\2", "a test of capitalizing", perl=TRUE)
gsub("\\b(\\w)", "\\U\\1", "a test of capitalizing", perl=TRUE)
```

`groupGeneric` *S3 Group Generic Functions*

Description

Group generic methods can be defined for four pre-specified groups of functions, `Math`, `Ops`, `Summary` and `Complex`. (There are no objects of these names in base R, but there are in the **methods** package.)

A method defined for an individual member of the group takes precedence over a method defined for the group as a whole.

Usage

```
## S3 methods for group generics have prototypes:
Math(x, ...)
Ops(e1, e2)
Complex(z)
Summary(..., na.rm = FALSE)
```

Arguments

`x, z, e1, e2`	objects.
`...`	further arguments passed to methods.
`na.rm`	logical: should missing values be removed?

Details

There are four *groups* for which S3 methods can be written, namely the `"Math"`, `"Ops"`, `"Summary"` and `"Complex"` groups. These are not R objects, but methods can be supplied for them and base R contains `factor`, `data.frame` and `difftime` methods for the first three groups. (There is also a `ordered` method for `Ops`, `POSIXt` and `Date` methods for `Math` and `Ops`, `package_version` methods for `Ops` and `Summary`, as well as a `ts` method for `Ops` in package **stats**.)

1. Group `"Math"`:
   - `abs`, `sign`, `sqrt`,
     `floor`, `ceiling`, `trunc`,
     `round`, `signif`
   - `exp`, `log`,
     `cos`, `sin`, `tan`,
     `acos`, `asin`, `atan`
     `cosh`, `sinh`, `tanh`,
     `acosh`, `asinh`, `atanh`
   - `lgamma`, `gamma`, `gammaCody`,
     `digamma`, `trigamma`
   - `cumsum`, `cumprod`, `cummax`, `cummin`

   Members of this group dispatch on `x`.

2. Group "Ops":
   - "+", "-", "*", "/", "^", "%%", "%/%"
   - "&", "|", "!"
   - "==", "!=", "<", "<=", ">=", ">"

   This group contains both binary and unary operators (+, - and !): when a unary operator is encountered the Ops method is called with one argument and e2 is missing.

   The classes of both arguments are considered in dispatching any member of this group. For each argument its vector of classes is examined to see if there is a matching specific (preferred) or Ops method. If a method is found for just one argument or the same method is found for both, it is used. If different methods are found, there is a warning about 'incompatible methods': in that case or if no method is found for either argument the internal method is used.
3. Group "Summary":
   - all, any
   - sum, prod
   - min, max
   - range

   Members of this group dispatch on the first argument supplied.
4. Group "Complex":
   - Arg, Conj, Im, Mod, Re

   Members of this group dispatch on z.

Note that a method will used for either one of these groups or one of its members *only* if it corresponds to a "class" attribute, as the internal code dispatches on oldClass and not on class. This is for efficiency: having to dispatch on, say, Ops.integer would be too slow.

The number of arguments supplied for "Math" group generic methods is not checked prior to dispatch. (Most have default methods expecting one argument, but log, round and signif expect two.)

Technical Details

The details of method dispatch and variables such as .Generic are discussed in the help for UseMethod. There are a few small differences:

- For the operators of group Ops, the object .Method is a length-two character vector with elements the methods selected for the left and right arguments respectively. (If no method was selected, the corresponding element is "".)
- Object .Group records the group used for dispatch (if a specific method is used this is "").

References

Appendix A, *Classes and Methods* of
Chambers, J. M. and Hastie, T. J. eds (1992) *Statistical Models in S.* Wadsworth & Brooks/Cole.

See Also

methods for methods of non-Internal generic functions.

S4groupGeneric for group generics for S4 methods.

Examples

```
d.fr <- data.frame(x=1:9, y=rnorm(9))
class(1 + d.fr) == "data.frame" ##-- add to d.f. ...

methods("Math")
methods("Ops")
methods("Summary")
methods("Complex")  # none in base R
```

---

gzcon *(De)compress I/O Through Connections*

Description

`gzcon` provides a modified connection that wraps an existing connection, and decompresses reads or compresses writes through that connection. Standard `gzip` headers are assumed.

Usage

```
gzcon(con, level = 6, allowNonCompressed = TRUE)
```

Arguments

`con`	a connection.
`level`	integer between 0 and 9, the compression level when writing.
`allowNonCompressed`	logical. When reading, should non-compressed input be allowed?

Details

If `con` is open then the modified connection is opened. Closing the wrapper connection will also close the underlying connection.

Reading from a connection which does not supply a `gzip` magic header is equivalent to reading from the original connection if `allowNonCompressed` is true, otherwise an error.

The original connection becomes unusable: any object pointing to it will now refer to the modified connection.

When the connection is opened for reading, the input is expected to start with the `gzip` magic header. If it does not and if `allowNonCompressed = TRUE` (the default) the input is read as-is.

Value

An object inheriting from class `"connection"`. This is the same connection *number* as supplied, but with a modified internal structure.

See Also

`gzfile`

Examples

```
## Not run:
## This example is no longer available
## print the value to see what objects were created.
con <- url("http://hesweb1.med.virginia.edu/biostat/s/data/sav/kprats.sav")
print(load(con))
close(con)
## End(Not run)

## gzfile and gzcon can inter-work.
## Of course here one would used gzfile, but file() can be replaced by
## any other connection generator.
zz <- gzfile("ex.gz", "w")
cat("TITLE extra line", "2 3 5 7", "", "11 13 17", file = zz, sep = "\n")
close(zz)
readLines(zz <- gzcon(file("ex.gz", "rb")))
close(zz)
unlink("ex.gz")

zz <- gzcon(file("ex.gz", "wb"))
cat("TITLE extra line", "2 3 5 7", "", "11 13 17", file = zz, sep = "\n")
close(zz)
readLines(zz <- gzfile("ex.gz"))
close(zz)
unlink("ex.gz")
```

---

hexmode *Display Numbers in Hexadecimal*

Description

Convert or print integers in hexadecimal format, with as many digits as are needed to display the largest, using leading zeroes as necessary.

Usage

```
## S3 method for class 'hexmode':
as.character(x, ...)

## S3 method for class 'hexmode':
format(x, ...)

## S3 method for class 'hexmode':
print(x, ...)
```

Arguments

x	An object inheriting from class "hexmode".
...	further arguments passed to or from other methods.

Details

Class "hexmode" consists of integer vectors with that class attribute, used merely to ensure that they are printed in hex.

See Also

octmode

Hyperbolic *Hyperbolic Functions*

Description

These functions give the obvious hyperbolic functions. They respectively compute the hyperbolic cosine, sine, tangent, and their inverses, arc-cosine, arc-sine, arc-tangent (or "*area cosine*", etc).

Usage

```
cosh(x)
sinh(x)
tanh(x)
acosh(x)
asinh(x)
atanh(x)
```

Arguments

x	a numeric or complex vector

Details

These are generic functions: methods can be defined for them individually or via the Math group generic.

Branch cuts are consistent with the inverse trigonometric functions asin() et seq, and agree with those defined in Abramowitz and Stegun, figure 4.7, page 86.

References

Abramowitz, M. and Stegun, I. A. (1972) *Handbook of Mathematical Functions.* New York: Dover. Chapter 4. Elementary Transcendental Functions: Logarithmic, Exponential, Circular and Hyperbolic Functions

See Also

The trigonometric functions, cos, sin, tan, and their inverses acos, asin, atan.

The logistic distribution function plogis is a shifted version of tanh() for numeric x.

iconv *Convert Character Vector between Encodings*

Description

This uses system facilities to convert a character vector between encodings: the 'i' stands for 'internationalization'.

Usage

```
iconv(x, from, to, sub=NA)

iconvlist()
```

Arguments

x	A character vector, or an object to be converted to a character vector by as.character.
from	A character string describing the current encoding.
to	A character string describing the target encoding.
sub	character string. If not NA it is used to replace any non-convertible bytes in the input. (This would normally be a single character, but can be more. If "byte", the indication is "<xx>" with the hex code of the byte.

Details

The names of encodings and which ones are available (and indeed, if any are) is platform-dependent. On systems that support R's iconv you can use "" for the encoding of the current locale, as well as "latin1" and "UTF-8".

On many platforms iconvlist provides an alphabetical list of the supported encodings. On others, the information is on the man page for iconv(5) or elsewhere in the man pages (and beware that the system command iconv may not support the same set of encodings as the C functions R calls). Unfortunately, the names are rarely common across platforms.

Elements of x which cannot be converted (perhaps because they are invalid or because they cannot be represented in the target encoding) will be returned as NA unless sub is specified.

Some versions of iconv will allow transliteration by appending //TRANSLIT to the to encoding: see the examples.

Value

A character vector of the same length and the same attributes as x (after conversion).

Note

Not all platforms support these functions. See also capabilities("iconv").

See Also

localeToCharset, file.

Examples

```
## Not run:
iconvlist()

## convert from Latin-2 to UTF-8: two of the glibc iconv variants.
iconv(x, "ISO_8859-2", "UTF-8")
iconv(x, "LATIN2", "UTF-8")

## Both x below are in latin1 and will only display correctly in a
## latin1 locale.
(x <- "fa\xE7ile")
charToRaw(xx <- iconv(x, "latin1", "UTF-8"))
## in a UTF-8 locale, print(xx)

iconv(x, "latin1", "ASCII")           #   NA
iconv(x, "latin1", "ASCII", "?")      # "fa?ile"
iconv(x, "latin1", "ASCII", "")       # "faile"
iconv(x, "latin1", "ASCII", "byte")   # "fa<e7>ile"

# Extracts from R help files
(x <- c("Ekstr\xf8m", "J\xf6reskog", "bi\xdfchen Z\xfcrcher"))
iconv(x, "latin1", "ASCII//TRANSLIT")
iconv(x, "latin1", "ASCII", sub="byte")
## End(Not run)
```

---

identical *Test Objects for Exact Equality*

Description

The safe and reliable way to test two objects for being *exactly* equal. It returns `TRUE` in this case, `FALSE` in every other case.

Usage

```
identical(x, y)
```

Arguments

`x, y`	any R objects.

Details

A call to `identical` is the way to test exact equality in `if` and `while` statements, as well as in logical expressions that use `&&` or `||`. In all these applications you need to be assured of getting a single logical value.

Users often use the comparison operators, such as `==` or `!=`, in these situations. It looks natural, but it is not what these operators are designed to do in R. They return an object like the arguments. If you expected `x` and `y` to be of length 1, but it happened that one of them wasn't, you will *not* get a single `FALSE`. Similarly, if one of the arguments is `NA`, the result is also `NA`. In either case, the expression `if(x == y)....` won't work as expected.

The function `all.equal` is also sometimes used to test equality this way, but was intended for something different: It allows for "reasonable" differences in numeric results.

The computations in `identical` are also reliable and usually fast. There should never be an error. The only known way to kill `identical` is by having an invalid pointer at the C level, generating a memory fault. It will usually find inequality quickly. Checking equality for two large, complicated objects can take longer if the objects are identical or nearly so, but represent completely independent copies. For most applications, however, the computational cost should be negligible.

`identical` sees `NaN` as different from `as.double(NA)`, but all `NaN`s are equal (and all `NA` of the same type are equal).

Comparison of character strings allows for embedded `nul` characters. Comparison of attributes view them as a set (and not a vector, so order is not tested).

Value

A single logical value, `TRUE` or `FALSE`, never `NA` and never anything other than a single value.

Author(s)

John Chambers

References

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer.

See Also

`all.equal` for descriptions of how two objects differ; Comparison for operators that generate elementwise comparisons. `isTRUE` is a simple wrapper based on `identical`.

Examples

```
identical(1, NULL) ## FALSE -- don't try this with ==
identical(1, 1.)   ## TRUE in R (both are stored as doubles)
identical(1, as.integer(1)) ## FALSE, stored as different types

x <- 1.0; y <- 0.99999999999
## how to test for object equality allowing for numeric fuzz :
(E <- all.equal(x,y))
isTRUE(E) # which is simply defined to just use
identical(TRUE, E)
## If all.equal thinks the objects are different, it returns a
## character string, and the above expression evaluates to FALSE

# even for unusual R objects :
identical(.GlobalEnv, environment())
```

`ifelse` *Conditional Element Selection*

Description

`ifelse` returns a value with the same shape as `test` which is filled with elements selected from either `yes` or `no` depending on whether the element of `test` is `TRUE` or `FALSE`.

Usage

```
ifelse(test, yes, no)
```

Arguments

`test`	an object which can be coerced to logical mode.
`yes`	return values for true elements of `test`.
`no`	return values for false elements of `test`.

Details

If `yes` or `no` are too short, their elements are recycled. `yes` will be evaluated if and only if any element of `test` is true, and analogously for `no`.

Missing values in `test` give missing values in the result.

Value

A vector of the same length and attributes (including class) as `test` and data values from the values of `yes` or `no`. The mode of the answer will be coerced from logical to accommodate first any values taken from `yes` and then any values taken from `no`.

Warning

The mode of the result may depend on the value of `test`, and the class attribute of the result is taken from `test` and may be inappropriate for the values selected from `yes` and `no`.

Sometimes it is better to use a construction such as `(tmp <- yes; tmp[!test] <- no[!test]; tmp)`, possibly extended to handle missing values in `test`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`if`.

Examples

```
x <- c(6:-4)
sqrt(x)#- gives warning
sqrt(ifelse(x >= 0, x, NA))# no warning

## Note: the following also gives the warning !
ifelse(x >= 0, sqrt(x), NA)

## example of different return modes:
yes <- 1:3
no <- pi^(0:3)
typeof(ifelse(NA, yes, no))   # logical
typeof(ifelse(TRUE, yes, no)) # integer
typeof(ifelse(FALSE, yes, no))# double
```

---

integer *Integer Vectors*

Description

Creates or tests for objects of type `"integer"`.

Usage

```
integer(length = 0)
as.integer(x, ...)
is.integer(x)
```

Arguments

`length`	desired length.
`x`	object to be coerced or tested.
`...`	further arguments passed to or from other methods.

Details

Integer vectors exist so that data can be passed to C or Fortran code which expects them, and so that small integer data can be represented exactly and compactly.

Note that on almost all implementations of R the range of representable integers is restricted to about $\pm 2 \times 10^9$: `double`s can hold much larger integers exactly.

Value

`integer` creates a integer vector of the specified length. Each element of the vector is equal to `0`.

`as.integer` attempts to coerce its argument to be of integer type. The answer will be `NA` unless the coercion succeeds. Real values larger in modulus than the largest integer are coerced to `NA` (unlike S which gives the most extreme integer of the same sign). Non-integral numeric values are truncated towards zero (i.e., `as.integer(x)` equals `trunc(x)` there), and imaginary parts of complex numbers are discarded (with a warning). Character strings containing either a decimal representation or a heaxadecimal representation (starting with `0x` or `0X`) can be converted, as well

as any allowed by the platform for real numbers. Like `as.vector` it strips attributes including names.

`is.integer` returns `TRUE` or `FALSE` depending on whether its argument is of integer type or not. `is.integer` is generic: you can write methods to handle specific classes of objects, see InternalMethods. There is a method for factors which returns `FALSE`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`round` (and `ceiling` and `floor` on that help page) to convert to integral values.

Examples

```
## as.integer() truncates:
x <- pi * c(-1:1,10)
as.integer(x)
```

---

interaction *Compute Factor Interactions*

Description

`interaction` computes a factor which represents the interaction of the given factors. The result of `interaction` is always unordered.

Usage

```
interaction(..., drop = FALSE, sep = ".")
```

Arguments

`...`	the factors for which interaction is to be computed, or a single list giving those factors.
`drop`	if `drop` is `TRUE`, unused factor levels are dropped from the result. The default is to retain all factor levels.
`sep`	string to construct the new level labels by joining the constituent ones.

Value

A factor which represents the interaction of the given factors. The levels are labelled as the levels of the individual factors joined by `sep`, by `.` by default.

The levels are ordered so the level of the first factor varies fastest, then the second and so on. This is the reverse of lexicographic ordering, and differs from `:`. (It is done this way for compatibility with S.)

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

See Also

factor; : where f:g is similar to interaction(f, g, sep=":") when f and g are factors.

Examples

```
a <- gl(2, 4, 8)
b <- gl(2, 2, 8, label = c("ctrl", "treat"))
s <- gl(2, 1, 8, label = c("M", "F"))
interaction(a, b)
interaction(a, b, s, sep = ":")
```

---

interactive *Is R Running Interactively?*

Description

Return TRUE when R is being used interactively and FALSE otherwise.

Usage

```
interactive()
```

See Also

source, .First

Examples

```
.First <- function() if(interactive()) x11()
```

Internal *Call an Internal Function*

Description

.Internal performs a call to an internal code which is built in to the R interpreter.

Only true R wizards should even consider using this function, and only R developers can add to the list of internal functions.

Usage

```
.Internal(call)
```

Arguments

call	a call expression

See Also

.Primitive, .External (the nearest equivalent available to users).

InternalMethods *Internal Generic Functions*

Description

Many R-internal functions are *generic* and allow methods to be written for.

Details

The following primitive and internal functions are *generic*, i.e., you can write `methods` for them:

`[`, `[[`, `$`, `[<-`, `[[<-`, `$<-`,

`length`, `length<-`,

`dimnames<-`, `dimnames`, `dim<-`, `dim`,

`c`, `unlist`,

`as.character`, `as.vector`, `is.array`, `is.atomic`, `is.call`, `is.character`, `is.complex`, `is.double`, `is.environment`, `is.function`, `is.integer`, `is.language`, `is.logical`, `is.list`, `is.matrix`, `is.na`, `is.nan`, `is.null`, `is.numeric`, `is.object`, `is.pairlist`, `is.recursive`, `is.single`, `is.symbol`, `rep` and `seq.int`.

Note that most of the group generic functions are also internal/primitive and allow methods to be written for them.

See Also

`methods` for the methods of non-Internal generic functions.

invisible *Change the Print Mode to Invisible*

Description

Return a (temporarily) invisible copy of an object.

Usage

```
invisible(x)
```

Arguments

`x`	an arbitrary R object.

Details

This function can be useful when it is desired to have functions return values which can be assigned, but which do not print when they are not assigned.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`return`, `function`.

Examples

```
# These functions both return their argument
f1 <- function(x) x
f2 <- function(x) invisible(x)
f1(1)# prints
f2(1)# does not
```

`is.finite` *Finite, Infinite and NaN Numbers*

Description

`is.finite` and `is.infinite` return a vector of the same length as `x`, indicating which elements are finite (not infinite and not missing).

`Inf` and `-Inf` are positive and negative "infinity" whereas `NaN` means "Not a Number". (These apply to numeric values and real and imaginary parts of complex values but not to values of integer vectors.)

Usage

```
is.finite(x)
is.infinite(x)
Inf
NaN
is.nan(x)
```

Arguments

`x`	(numerical) object to be tested.

Details

`is.finite` returns a vector of the same length as `x` the jth element of which is `TRUE` if `x[j]` is finite (i.e., it is not one of the values `NA`, `NaN`, `Inf` or `-Inf`). All elements of types other than logical, integer, numeric and complex vectors are false. Complex numbers are finite if both the real and imaginary parts are.

`is.infinite` returns a vector of the same length as `x` the jth element of which is `TRUE` if `x[j]` is infinite (i.e., equal to one of `Inf` or `-Inf`). This will be false unless `x` is numeric or complex. Complex numbers are infinite if either the real and imaginary part is.

`is.nan` tests if a numeric value is `NaN`. Do not test equality to `NaN`, or even use `identical`, since systems typically have many different NaN values. One of these is used for the numeric missing value `NA`, and `is.nan` is false for that value. It is generic: you can write methods to handle

specific classes of objects, see InternalMethods. The default method handles real and complex vectors: a complex number is regarded as `NaN` if either the real or imaginary part is `NaN` but not `NA`.

Note

In R, basically all mathematical functions (including basic `Arithmetic`), are supposed to work properly with `+/- Inf` and `NaN` as input or output.

The basic rule should be that calls and relations with `Inf`s really are statements with a proper mathematical *limit*.

References

The IEC 60559 standard, also know as the ANSI/IEEE 754 Floating-Point Standard.

D. Goldberg (1991) *What Every Computer Scientist Should Know about Floating-Point Arithmetic* ACM Computing Surveys, **23(1)**.
Postscript version available at http://www.validlab.com/goldberg/paper.ps Extended PDF version at http://www.validlab.com/goldberg/paper.pdf

http://grouper.ieee.org/groups/754/ for accessible information.

The C99 function `isfinite` is used for `is.finite` if available.

See Also

`NA`, '*Not Available*' which is not a number as well, however usually used for missing values and applies to many modes, not just numeric.

Examples

```
pi / 0 ## = Inf a non-zero number divided by zero creates infinity
0 / 0  ## =  NaN

1/0 + 1/0# Inf
1/0 - 1/0# NaN

stopifnot(
    1/0 == Inf,
    1/Inf == 0
)
sin(Inf)
cos(Inf)
tan(Inf)
```

---

is.function *Is an Object of Type (Primitive) Function?*

Description

Checks whether its argument is a (primitive) function.

Usage

```
is.function(x)
is.primitive(x)
```

Arguments

`x`	an R object.

Details

`is.function` is generic: you can write methods to handle specific classes of objects, see InternalMethods.

`is.primitive(x)` tests if `x` is a primitive function (either a `"builtin"` or `"special"` as from `typeof`)?

Value

`TRUE` if `x` is a (primitive) function, and `FALSE` otherwise.

Examples

```
is.function(1) # FALSE
is.function(is.primitive)  # TRUE: it is a function, but ..
is.primitive(is.primitive) # FALSE:it's not a primitive one, whereas
is.primitive(is.function)  # TRUE: that one *is*
```

`is.language` *Is an Object a Language Object?*

Description

`is.language` returns `TRUE` if `x` is a variable `name`, a `call`, or an `expression`.

Usage

```
is.language(x)
```

Arguments

`x`	object to be tested.

Details

`is.language` is generic: you can write methods to handle specific classes of objects, see InternalMethods.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
ll <- list(a = expression(x^2 - 2*x + 1), b = as.name("Jim"),
           c = as.expression(exp(1)), d = call("sin", pi))
sapply(ll, typeof)
sapply(ll, mode)
stopifnot(sapply(ll, is.language))
```

---

is.object *Is an Object "internally classed"?*

Description

A function rather for internal use. It returns `TRUE` if the object `x` has the R internal `OBJECT` attribute set, and `FALSE` otherwise.

Usage

```
is.object(x)
```

Arguments

`x`	object to be tested.

Details

`is.object` is generic: you can write methods to handle specific classes of objects, see Internal-Methods.

See Also

`class`, and `methods`.

Examples

```
is.object(1) # FALSE
is.object(as.factor(1:3)) # TRUE
```

---

is.R *Are we using R, rather than S?*

Description

Test if running under R.

Usage

```
is.R()
```

Details

The function has been written such as to correctly run in all versions of R, S and S-PLUS. In order for code to be runnable in both R and S dialects previous to S-PLUS 8.0, your code must either define `is.R` or use it as

```
if (exists("is.R") && is.function(is.R) && is.R()) {
## R-specific code
} else {
## S-version of code
}
```

Value

`is.R` returns `TRUE` if we are using R and `FALSE` otherwise.

See Also

`R.version`, `system`.

Examples

```
x <- runif(20); small <- x < 0.4
## In the early years of R, 'which()' only existed in R:
if(is.R()) which(small) else seq(along=small)[small]
```

`is.recursive` *Is an Object Atomic or Recursive?*

Description

`is.atomic` returns `TRUE` if `x` is an atomic vector (or `NULL`) and `FALSE` otherwise.

`is.recursive` returns `TRUE` if `x` has a recursive (list-like) structure and `FALSE` otherwise.

Usage

```
is.atomic(x)
is.recursive(x)
```

Arguments

`x`	object to be tested.

Details

These are generic: you can write methods to handle specific classes of objects, see InternalMethods. The description here applies only to the default method.

`is.atomic` is true for the atomic vector types (`"logical"`, `"integer"`, `"numeric"`, `"complex"`, `"character"` and `"raw"`) and `NULL`.

Most types of language objects are regarded as recursive: those which are not are the atomic vector types, `NULL` and symbols (as given by `as.name`).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`is.list`, `is.language`, etc, and the `demo("is.things")`.

Examples

```
is.a.r <- function(x) c(is.atomic(x), is.recursive(x))

is.a.r(c(a=1,b=3))      # TRUE FALSE
is.a.r(list())          # FALSE TRUE ??
is.a.r(list(2))         # FALSE TRUE
is.a.r(lm)              # FALSE TRUE
is.a.r(y ~ x)           # FALSE TRUE
is.a.r(expression(x+1)) # FALSE TRUE (not in 0.62.3!)
```

---

is.single *Is an Object of Single Precision Type?*

Description

`is.single` reports an error. There are no single precision values in R.

Usage

```
is.single(x)
```

Arguments

`x`	object to be tested.

Details

`is.single` is generic: you can write methods to handle specific classes of objects, see Internal-Methods.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

isS4 *Test for an S4 object*

Description

Tests whether the object was created from a formally defined class.

Usage

```
isS4(object)

asS4(object, value = TRUE)
```

Arguments

object	Any R object.
value	A single logical value; not NA.

Details

Note that this function does not rely on the methods package, so in particular it can be used to detect the need to require() that package.

You should not set the flag directly unless you really know why. In the future, S4 methods may be restricted S4 objects, for primitive functions; then asS4 would allow method dispatch of S4 methods for S3 classes. For all other purposes, an object will satisfy isS4(x) if and only if it should.

Value

isS4 always returns TRUE or FALSE according to whether the internal flag marking an S4 object has been turned on for this object.

asS4 will turn this flag on or off. But see the details.

Examples

```
isS4(pi) # FALSE
isS4(getClass("MethodDefinition")) # TRUE
```

isSymmetric *Test if a Matrix or other Object is Symmetric*

Description

Generic function to test if object is symmetric or not. Currently only a matrix method is implemented.

Usage

```
isSymmetric(object, ...)
## S3 method for class 'matrix':
isSymmetric(object, tol = 100 * .Machine$double.eps, ...)
```

Arguments

`object`	any R object; a `matrix` for the matrix method.
`tol`	numeric scalar >= 0. Smaller differences are not considered, see `all.equal.numeric`.
`...`	further arguments passed to methods; the matrix method passes these to `all.equal`.

Details

The `matrix` method is used inside `eigen` by default to test symmetry of matrices "up to rounding error", using `all.equal`. It might not be appropriate in all situations.

Value

logical indicating if `object` is symmetric or not.

See Also

`eigen` which calls `isSymmetric` when its `symmetric` argument is missing, as per default.

Examples

```
isSymmetric(D3 <- diag(3)) # -> TRUE

D3[2,1] <- 1e-100
D3
isSymmetric(D3) # TRUE
isSymmetric(D3, tol = 0) # FALSE for zero-tolerance
```

`jitter` *Add 'Jitter' (Noise) to Numbers*

Description

Add a small amount of noise to a numeric vector.

Usage

```
jitter(x, factor=1, amount = NULL)
```

Arguments

`x`	numeric vector to which *jitter* should be added.
`factor`	numeric
`amount`	numeric; if positive, used as *amount* (see below), otherwise, if = `0` the default is `factor * z/50`. Default (`NULL`): `factor * d/5` where `d` is about the smallest difference between `x` values.

Details

The result, say `r`, is `r <- x + runif(n, -a, a)` where `n <- length(x)` and `a` is the `amount` argument (if specified).

Let `z <- max(x) - min(x)` (assuming the usual case). The amount `a` to be added is either provided as *positive* argument `amount` or otherwise computed from `z`, as follows:

If `amount == 0`, we set `a <- factor * z/50` (same as S).

If `amount` is `NULL` (*default*), we set `a <- factor * d/5` where *d* is the smallest difference between adjacent unique (apart from fuzz) `x` values.

Value

`jitter(x,...)` returns a numeric of the same length as `x`, but with an `amount` of noise added in order to break ties.

Author(s)

Werner Stahel and Martin Maechler, ETH Zurich

References

Chambers, J. M., Cleveland, W. S., Kleiner, B. and Tukey, P.A. (1983) *Graphical Methods for Data Analysis.* Wadsworth; figures 2.8, 4.22, 5.4.

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S.* Wadsworth & Brooks/Cole.

See Also

`rug` which you may want to combine with `jitter`.

Examples

```
round(jitter(c(rep(1,3),  rep(1.2, 4), rep(3,3))), 3)
## These two 'fail' with S-plus 3.x:
jitter(rep(0, 7))
jitter(rep(10000,5))
```

`kappa` *Estimate the Condition Number*

Description

An estimate of the condition number of a matrix or of the R matrix of a QR decomposition, perhaps of a linear fit. The condition number is defined as the ratio of the largest to the smallest *non-zero* singular value of the matrix.

Usage

```
kappa(z, ...)
## S3 method for class 'lm':
kappa(z, ...)
## Default S3 method:
kappa(z, exact = FALSE, ...)
## S3 method for class 'qr':
kappa(z, ...)

kappa.tri(z, exact = FALSE, ...)
```

Arguments

`z`	A matrix or a the result of `qr` or a fit from a class inheriting from `"lm"`.
`exact`	logical. Should the result be exact?
`...`	further arguments passed to or from other methods.

Details

If `exact = FALSE` (the default) the condition number is estimated by a cheap approximation. Following S, this uses the LINPACK routine 'dtrco.f'. However, in R (or S) the exact calculation is also likely to be quick enough.

`kappa.tri` is an internal function called by `kappa.qr`.

Value

The condition number, *kappa*, or an approximation if `exact = FALSE`.

Author(s)

The design was inspired by (but differs considerably from) the S function of the same name described in Chambers (1992).

References

Chambers, J. M. (1992) *Linear models.* Chapter 4 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`svd` for the singular value decomposition and `qr` for the QR one.

Examples

```
kappa(x1 <- cbind(1,1:10))# 15.71
kappa(x1, exact = TRUE)        # 13.68
kappa(x2 <- cbind(x1,2:11))# high! [x2 is singular!]

hilbert <- function(n) { i <- 1:n; 1 / outer(i - 1, i, "+") }
sv9 <- svd(h9 <- hilbert(9))$ d
kappa(h9)# pretty high!
kappa(h9, exact = TRUE) == max(sv9) / min(sv9)
kappa(h9, exact = TRUE) / kappa(h9) # .677 (i.e., rel.error = 32%)
```

---

kronecker *Kronecker products on arrays*

Description

Computes the generalised kronecker product of two arrays, `X` and `Y`. `kronecker(X, Y)` returns an array `A` with dimensions `dim(X) * dim(Y)`.

Usage

```
kronecker(X, Y, FUN = "*", make.dimnames = FALSE, ...)
X %x% Y
```

Arguments

`X`	A vector or array.
`Y`	A vector or array.
`FUN`	a function; it may be a quoted string.
`make.dimnames`	Provide dimnames that are the product of the dimnames of `X` and `Y`.
`...`	optional arguments to be passed to `FUN`.

Details

If `X` and `Y` do not have the same number of dimensions, the smaller array is padded with dimensions of size one. The returned array comprises submatrices constructed by taking `X` one term at a time and expanding that term as `FUN(x, Y, ...)`.

`%x%` is an alias for `kronecker` (where `FUN` is hardwired to `"*"`).

Author(s)

Jonathan Rougier, ⟨J.C.Rougier@durham.ac.uk⟩

References

Shayle R. Searle (1982) *Matrix Algebra Useful for Statistics.* John Wiley and Sons.

See Also

`outer`, on which `kronecker` is built and `%*%` for usual matrix multiplication.

Examples

```
# simple scalar multiplication
( M <- matrix(1:6, ncol=2) )
kronecker(4, M)
# Block diagonal matrix:
kronecker(diag(1, 3), M)

# ask for dimnames

fred <- matrix(1:12, 3, 4, dimnames=list(LETTERS[1:3], LETTERS[4:7]))
bill <- c("happy" = 100, "sad" = 1000)
kronecker(fred, bill, make.dimnames = TRUE)

bill <- outer(bill, c("cat"=3, "dog"=4))
kronecker(fred, bill, make.dimnames = TRUE)
```

---

l10n_info *Localization Information*

Description

Report on localization information.

Usage

```
l10n_info()
```

Value

A list with two logical components:

MBCS	If a multi-byte character set in use?
UTF-8	Is this a UTF-8 locale?

See Also

Sys.getlocale, localeconv

Examples

```
l10n_info()
```

labels *Find Labels from Object*

Description

Find a suitable set of labels from an object for use in printing or plotting, for example. A generic function.

Usage

```
labels(object, ...)
```

Arguments

`object`	Any R object: the function is generic.
`...`	further arguments passed to or from other methods.

Value

A character vector or list of such vectors. For a vector the results is the names or `seq(along=x)` and for a data frame or array it is the dimnames (with `NULL` expanded to `seq(len=d[i])`.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S.* Wadsworth & Brooks/Cole.

lapply *Apply a Function over a List or Vector*

Description

`lapply` returns a list of the same length as `X`, each element of which is the result of applying `FUN` to the corresponding element of `X`.

`sapply` is a "user-friendly" version of `lapply` by default returning a vector or matrix if appropriate.

`replicate` is a wrapper for the common use of `sapply` for repeated evaluation of an expression (which will usually involve random number generation).

Usage

```
lapply(X, FUN, ...)

sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)

replicate(n, expr, simplify = TRUE)
```

Arguments

X	a vector (atomic or list) or an expressions vector. Other objects (including classed objects) will be coerced by as.list.
FUN	the function to be applied to each element of X: see Details. In the case of functions like +, %*%, etc., the function name must be backquoted or quoted.
...	optional arguments to FUN.
simplify	logical; should the result be simplified to a vector or matrix if possible?
USE.NAMES	logical; if TRUE and if X is character, use X as names for the result unless it had names already.
n	number of replications.
expr	expression (language object, usually a call) to evaluate repeatedly.

Details

FUN is found by a call to match.fun and typically is specified as a function or a symbol (e.g. a backquoted name) or a character string specifying a function to be searched for from the environment of the call to lapply.

Function FUN must be able to accept as input any of the elements of X. If the latter is an atomic vector, FUN will always be passed a length-one vector of the same type as X.

Simplification in sapply is only attempted if X has length greater than zero and if the return values from all elements of X are all of the same (positive) length. If the common length is one the result is a vector, and if greater than one is a matrix with a column corresponding to each element of X.

The mode of the simplified answer is chosen to accommodate the modes of all the values returned by the calls to FUN: see unlist.

if X has length 0, the return value of sapply is always a 0-length list.

Note

sapply(*, simplify = FALSE, USE.NAMES = FALSE) is equivalent to lapply(*).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

apply, tapply, mapply for applying a function to **m**ultiple arguments, and rapply for a **r**ecursive version of lapply().

Examples

```
x <- list(a = 1:10, beta = exp(-3:3), logic = c(TRUE,FALSE,FALSE,TRUE))
# compute the list mean for each list element
lapply(x,mean)
# median and quartiles for each list element
lapply(x, quantile, probs = 1:3/4)
sapply(x, quantile)
i39 <- sapply(3:9, seq) # list of vectors
sapply(i39, fivenum)
```

```
hist(replicate(100, mean(rexp(10))))
```

`Last.value` *Value of Last Evaluated Expression*

Description

The value of the internal evaluation of a top-level R expression is always assigned to `.Last.value` (in `package:base`) before further processing (e.g., printing).

Usage

```
.Last.value
```

Details

The value of a top-level assignment *is* put in `.Last.value`, unlike S.

Do not assign to `.Last.value` in the workspace, because this will always mask the object of the same name in `package:base`.

See Also

`eval`

Examples

```
## These will not work correctly from example(),
## but they will in make check or if pasted in,
## as example() does not run them at the top level
gamma(1:15)          # think of some intensive calculation...
fac14 <- .Last.value # keep them

library("splines") # returns invisibly
.Last.value    # shows what library(.) above returned
```

`length` *Length of an Object*

Description

Get or set the length of vectors (including lists) and factors, and of any other R object for which a method has been defined.

Usage

```
length(x)
length(x) <- value
```

Arguments

x	an R object. For replacement, a vector or factor.
value	an integer.

Details

Both functions are generic: you can write methods to handle specific classes of objects, see InternalMethods. length<- has a "factor" method.

The replacement form can be used to reset the length of a vector. If a vector is shortened, extra values are discarded and when a vector is lengthened, it is padded out to its new length with NAs (nul for raw vectors).

Value

The default method currently returns an integer of length 1. Since this may change in the future and may differ for other methods, programmers should not rely on it. (Should the length exceed the maximum representable integer, it is returned as NA.)

For vectors (including lists) and factors the length is the number of elements. For an environment it is the number of objects in the environment, and NULL has length 0. For expressions and pairlists (including language objects and dotlists) it is the length of the pairlist chain. All other objects (including functions) have length one: note that for functions this differs from S.

The replacement form removes all the attributes of x except its names.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

nchar for counting the number of characters in character vectors.

Examples

```
length(diag(4))# = 16 (4 x 4)
length(options())# 12 or more
length(y ~ x1 + x2 + x3)# 3
length(expression(x, {y <- x^2; y+2}, x^y)) # 3

## from example(warpbreaks)
fm1 <- lm(breaks ~ wool * tension, data = warpbreaks)
length(fm1$call) # 3, lm() and two arguments.
length(formula(fm1)) # 3, ~ lhs rhs
```

`levels` *Levels Attributes*

Description

`levels` provides access to the levels attribute of a variable. The first form returns the value of the levels of its argument and the second sets the attribute.

The replacement form (`"levels<-"`) of `levels` is a generic function and new methods can be written for it. The most important method is that for `factor`s:

Usage

```
levels(x)
levels(x) <- value
```

Arguments

`x`	an object, for example a factor.
`value`	A valid value for `levels(x)`. For the default method, `NULL` or a character vector. For the `factor` method, a vector of character strings with length at least the number of levels of `x`, or a named list specifying how to rename the levels.

Details

For the factor replacement method, a `NA` in `value` causes that level to be removed from the levels and the elements formerly with that level to be replaced by `NA`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`nlevels`, `relevel`, `reorder`.

Examples

```
## assign individual levels
x <- gl(2, 4, 8)
levels(x)[1] <- "low"
levels(x)[2] <- "high"
x

## or as a group
y <- gl(2, 4, 8)
levels(y) <- c("low", "high")
y

## combine some levels
z <- gl(3, 2, 12)
```

```
levels(z) <- c("A", "B", "A")
z

## same, using a named list
z <- gl(3, 2, 12)
levels(z) <- list(A=c(1,3), B=2)
z

## we can add levels this way:
f <- factor(c("a","b"))
levels(f) <- c("c", "a", "b")
f

f <- factor(c("a","b"))
levels(f) <- list(C="C", A="a", B="b")
f
```

`libPaths` *Search Paths for Packages*

Description

`.libPaths` gets/sets the library trees within which packages are looked for.

Usage

```
.libPaths(new)

.Library
```

Arguments

`new`	a character vector with the locations of R library trees.

Details

`.Library` is a character string giving the location of the default library, the 'library' subdirectory of `R_HOME`.

`.libPaths` is used for getting or setting the library trees that R knows about (and hence uses when looking for packages). If called with argument `new`, the library search path is set to the existing files in `unique(c(new, .Library))` and this is returned. If given no argument, a character vector with the currently known library trees is returned.

The library search path is initialized at startup from the environment variable `R_LIBS` (which should be a colon-separated list of directories at which R library trees are rooted) by calling `.libPaths` with the directories specified in `R_LIBS`.

Value

A character vector of file paths.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`library`

Examples

```
.libPaths()                 # all library trees R knows about
```

`library` *Loading and Listing of Packages*

Description

`library` and `require` load add-on packages.

`.First.lib` is called when a package is loaded; `.Last.lib` is called when a package is detached.

Usage

```
library(package, help, pos = 2, lib.loc = NULL,
        character.only = FALSE, logical.return = FALSE,
        warn.conflicts = TRUE,
        keep.source = getOption("keep.source.pkgs"),
        verbose = getOption("verbose"),
        version)

require(package, lib.loc = NULL, quietly = FALSE, warn.conflicts = TRUE,
        keep.source = getOption("keep.source.pkgs"),
        character.only = FALSE, version, save = TRUE)

.First.lib(libname, pkgname)
.Last.lib(libpath)
```

Arguments

`package, help`
: the name of a package, given as a name or literal character string, or a character string, depending on whether `character.only` is `FALSE` (default) or `TRUE`).

`pos`
: the position on the search list at which to attach the loaded package. Note that `.First.lib` may attach other packages, and `pos` is computed *after* `.First.lib` has been run. Can also be the name of a position on the current search list as given by `search()`.

`lib.loc`
: a character vector describing the location of R library trees to search through, or `NULL`. The default value of `NULL` corresponds to all libraries currently known. Non-existent library trees are silently ignored.

`character.only`
: a logical indicating whether `package` or `help` can be assumed to be character strings.

`version`
: A character string denoting a version number of the package to be loaded, for use with *versioned installs*: see the section later in this document.

`logical.return`
: logical. If it is `TRUE`, `FALSE` or `TRUE` is returned to indicate success.

`warn.conflicts`
: logical. If `TRUE`, warnings are printed about `conflicts` from attaching the new package, unless that package contains an object `.conflicts.OK`. A conflict is a function masking a function, or a non-function masking a non-function.

`keep.source`
: logical. If `TRUE`, functions "keep their source" including comments, see argument `keep.source` to `options`. This applies only to the named package, and not to any packages or namespaces which might be loaded to satisfy dependencies or imports.

  This argument does not apply to packages using lazy-loading or saved images. Whether they have kept source is determined when they are installed (and is most likely false).

`verbose`
: a logical. If `TRUE`, additional diagnostics are printed.

`quietly`
: a logical. If `TRUE`, no message confirming package loading is printed.

`save`
: logical or environment. If `TRUE`, a call to `require` from the source for a package will save the name of the required package in the variable `".required"`, allowing function `detach` to warn if a required package is detached. See section 'Packages that require other packages' below.

`libname`
: a character string giving the library directory where the package was found.

`pkgname`
: a character string giving the name of the package, including the version number if the package was installed with `--with-package-versions`.

`libpath`
: a character string giving the complete path to the package.

Details

`library(package)` and `require(package)` both load the package with name `package`. `require` is designed for use inside other functions; it returns `FALSE` and gives a warning (rather than an error as `library()` does by default) if the package does not exist. Both functions check and update the list of currently loaded packages and do not reload a package which is already loaded.

If `library` is called with no `package` or `help` argument, it lists all available packages in the libraries specified by `lib.loc`, and returns the corresponding information in an object of class `"libraryIQR"`. The structure of this class may change in future versions. In earlier versions of R, only the names of all available packages were returned; use `.packages(all = TRUE)` for obtaining these. Note that `installed.packages()` returns even more information.

`library(help = somename)` computes basic information about the package `somename`, and returns this in an object of class `"packageInfo"`. The structure of this class may change in future versions. When used with the default value (`NULL`) for `lib.loc`, the loaded packages are searched before the libraries.

`.First.lib` is called when a package without a namespace is loaded by `library`. (For packages with namespaces see `.onLoad`.) It is called with two arguments, the name of the library directory where the package was found (i.e., the corresponding element of `lib.loc`), and the name of the package (in that order, and with the package name including the version for a versioned install, e.g. `tree_1.0-16`). It is a good place to put calls to

`library.dynam` which are needed when loading a package into this function (don't call `library.dynam` directly, as this will not work if the package is not installed in a "standard" location). `.First.lib` is invoked after the search path interrogated by `search()` has been updated, so `as.environment(match("package:name", search()))` will return the environment in which the package is stored. If calling `.First.lib` gives an error the loading of the package is abandoned, and the package will be unavailable. Similarly, if the option `".First.lib"` has a list element with the package's name, this element is called in the same manner as `.First.lib` when the package is loaded. This mechanism allows the user to set package "load hooks" in addition to startup code as provided by the package maintainers, but `setHook` is preferred.

`.Last.lib` is called when a package is detached. Beware that it might be called if `.First.lib` has failed, so it should be written defensively. (It is called within `try`, so errors will not stop the package being detached.)

Value

`library` returns the list of loaded (or available) packages (or `TRUE` if `logical.return` is `TRUE`). `require` returns a logical indicating whether the required package is available.

Packages that require other packages

NB: This mechanism has been almost entirely superseded by using the `Depends:` field in the 'DESCRIPTION' file of a package.

The source code for a package that requires one or more other packages should have a call to `require`, preferably near the beginning of the source, and of course before any code that uses functions, classes or methods from the other package. The default for argument `save` will save the names of all required packages in the environment of the new package. The saved package names are used by `detach` when a package is detached to warn if other packages still require the to-be-detached package. Also, if a package is installed with saved image (see INSTALL), the saved package names are used to require these packages when the new package is attached.

Formal methods

`library` takes some further actions when package **methods** is attached (as it is by default). Packages may define formal generic functions as well as re-defining functions in other packages (notably **base**) to be generic, and this information is cached whenever such a package is loaded after **methods** and re-defined functions are excluded from the list of conflicts. The check requires looking for a pattern of objects; the pattern search may be avoided by defining an object `.noGenerics` (with any value) in the package. Naturally, if the package *does* have any such methods, this will prevent them from being used.

Versioned installs

Packages can be installed with version information by `R CMD INSTALL --with-package-versions` or `install.packages(installWithVers = TRUE)`. This allows more than one version of a package to be installed in a library directory, using package directory names like `foo_1.5-1`. When such packages are loaded, it is this *versioned* name that `search()` returns. Some utility functions require the versioned name and some the unversioned name (here `foo`).

If `version` *is not* specified, `library` looks first for a packages of that name, and then for versioned installs of the package, selecting the one with the latest version number. If `version` *is* specified, a versioned install with an exactly matching version is looked for.

If `version` is not specified, `require` will accept any version that is already loaded, whereas `library` will look for an unversioned install even if a versioned install is already loaded.

Loading more than one version of a package into an R session is not currently supported. Support for versioned installs is patchy.

Note

`library` and `require` can only load an *installed* package, and this is detected by having a 'DESCRIPTION' file containing a `Built:` field.

Under Unix-alikes, the code checks that the package was installed under a similar operating system as given by `R.version$platform` (the canonical name of the platform under which R was compiled), provided it contains compiled code. Packages which do not contain compiled code can be shared between Unix-alikes, but not to other OSes because of potential problems with line endings and OS-specific help files.

The package name given to `library` and `require` must match the name given in the package's 'DESCRIPTION' file exactly, even on case-insensitive file systems such as MS Windows.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`.libPaths`, `.packages`.

`attach`, `detach`, `search`, `objects`, `autoload`, `library.dynam`, `data`, `install.packages` and `installed.packages`; `INSTALL`, `REMOVE`.

Examples

```
library()                    # list all available packages
library(lib = .Library)      # list all packages in the default library
library(help = splines)      # documentation on package 'splines'
library(splines)             # load package 'splines'
require(splines)             # the same
search()                     # "splines", too
detach("package:splines")

# if the package name is in a character vector, use
pkg <- "splines"
library(pkg, character.only = TRUE)
detach(pos = match(paste("package", pkg, sep=":"), search()))

require(pkg, character.only = TRUE)
detach(pos = match(paste("package", pkg, sep=":"), search()))

require(nonexistent)         # FALSE
## Not run:
## Suppose a package needs to call a shared library named 'fooEXT',
## where 'EXT' is the system-specific extension.  Then you should use
.First.lib <- function(lib, pkg) {
  library.dynam("foo", pkg, lib)
}
```

```
## if you want to mask as little as possible, use
library(mypkg, pos = "package:base")
## End(Not run)
```

`library.dynam` *Loading Shared Libraries*

Description

Load the specified file of compiled code if it has not been loaded already, or unloads it.

Usage

```
library.dynam(chname, package = NULL, lib.loc = NULL,
              verbose = getOption("verbose"),
              file.ext = .Platform$dynlib.ext, ...)

library.dynam.unload(chname, libpath,
              verbose = getOption("verbose"),
              file.ext = .Platform$dynlib.ext)

.dynLibs(new)
```

Arguments

`chname`	a character string naming a shared library to load.
`package`	a character vector with the names of packages to search through, or `NULL`. By default, all packages in the search path are used.
`lib.loc`	a character vector describing the location of R library trees to search through, or `NULL`. The default value of `NULL` corresponds to all libraries currently known.
`libpath`	the path to the loaded package whose shared library is to be unloaded.
`verbose`	a logical value indicating whether an announcement is printed on the console before loading the shared library. The default value is taken from the verbose entry in the system options.
`file.ext`	the extension to append to the file name to specify the library to be loaded. This defaults to the appropriate value for the operating system.
`...`	additional arguments needed by some libraries that are passed to the call to `dyn.load` to control how the library is loaded.
`new`	a list of `"DLLInfo"` objects corresponding to the shared libraries loaded by packages. Can be missing.

Details

`library.dynam` is designed to be used inside a package rather than at the command line, and should really only be used inside `.First.lib` or `.onLoad`. The system-specific extension for shared libraries (e.g., '.so' or '.sl' on Unix systems) should not be added. Note that to allow for versioned installs, the `chname` argument should not be set to the `pkgname` argument of `.First.lib` or `.onLoad`.

`library.dynam.unload` is designed for use in `.Last.lib` or `.onUnload`: it unloads the shared object and updates the value of `.dynLibs()`

`.dynLibs` is used for getting (with no argument) or setting the shared libraries which are currently loaded by packages (using `library.dynam`).

Value

If `chname` is not specified, `library.dynam` returns an object of class `"DLLInfoList"` corresponding to the shared libraries loaded by packages.

If `chname` is specifed, an object of class `"DLLInfo"` that identifies the DLL and can be used in future calls is returned invisibly. For packages that have namespaces, a list of these objects is stored in the namespace's environment for use at run-time.

Note that the class `DLLInfo` has an overloaded method for `$` which can be used to resolve native symbols within that DLL.

`library.dynam.unload` invisibly returns an object of class `"DLLInfo"` identifying the DLL successfully unloaded.

`.dynLibs` returns an object of class `"DLLInfoList"` corresponding corresponding to its current value.

Warning

Do not use `dyn.unload` on a shared object loaded by `library.dynam`: use `library.dynam.unload` to ensure that `.dynLibs` gets updated. Otherwise a subsequent call to `library.dynam` will be told the object is already loaded.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`getLoadedDLLs` for information on `"DLLInfo"` and `"DLLInfoList"` objects.

`.First.lib`, `library`, `dyn.load`, `.packages`, `.libPaths`

`SHLIB` for how to create suitable shared libraries.

Examples

```
## Which DLLs were "dynamically loaded" by packages?
library.dynam()
```

`license` *The R License Terms*

Description

The license terms under which R is distributed.

Usage

```
license()
licence()
```

Details

R is distributed under the terms of the GNU GENERAL PUBLIC LICENSE Version 2, June 1991. A copy of this license is in '$R_HOME/COPYING'.

A small number of files (the API header files and import library) are distributed under the LESSER GNU GENERAL PUBLIC LICENSE version 2.1. A copy of this license is in '$R_HOME/COPYING.LIB'.

`list` *Lists – Generic and Dotted Pairs*

Description

Functions to construct, coerce and check for both kinds of R lists.

Usage

```
list(...)
pairlist(...)

as.list(x, ...)
## S3 method for class 'environment':
as.list(x, all.names = FALSE, ...)
as.pairlist(x)

is.list(x)
is.pairlist(x)

alist(...)
```

Arguments

`...`	objects, possibly named.
`x`	object to be coerced or tested.
`all.names`	a logical indicating whether to copy all values or (default) only those whose names do not begin with a dot.

Details

Most lists in R internally are *Generic Vectors*, whereas traditional *dotted pair* lists (as in LISP) are available but rarely seen by users (except as `formals` of functions).

The arguments to `list` or `pairlist` are of the form `value` or `tag=value`. The functions return a list or dotted pair list composed of its arguments with each value either tagged or untagged, depending on how the argument was specified.

`alist` handles its arguments as if they described function arguments. So the values are not evaluated, and tagged arguments with no value are allowed whereas `list` simply ignores them. `alist` is most often used in conjunction with `formals`.

`as.list` attempts to coerce its argument to a list. For functions, this returns the concatenation of the list of formal arguments and the function body. For expressions, the list of constituent calls is returned. `as.list` is generic, and as the default method calls `as.vector(mode="list")` methods for `as.vector` may be invoked. As from R 2.4.0, `as.list` turns a factor into a list of one-element factors. All attributes will be dropped unless the argument already is a list. (This is inconsistent with functions such as `as.character`, and is for efficiency since lists can be expensive to copy.)

`is.list` returns `TRUE` iff its argument is a `list` *or* a `pairlist` of `length` > 0. `is.pairlist` returns `TRUE` iff the argument is a pairlist or `NULL` (see below).

`is.list` and `is.pairlist` are generic: you can write methods to handle specific classes of objects, see InternalMethods.

The `"environment"` method for `as.list` copies the name-value pairs (for names not beginning with a dot) from an environment to a named list. The user can request that all named objects are copied. The list is in no particular order (the order depends on the order of creation of objects and whether the environmnent is hashed). No parent environments are searched. (Objects copied are duplicated so this can be an expensive operation.)

An empty pairlist, `pairlist()` is the same as `NULL`. This is different from `list()`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`vector("list", length)` for creation of a list with empty components; `c`, for concatenation; `formals`.

Examples

```
# create a plotting structure
pts <- list(x=cars[,1], y=cars[,2])
plot(pts)

is.pairlist(.Options)  # a user-level pairlist

## "pre-allocate" an empty list of length 5
vector("list", 5)

# Argument lists
f <- function() x
# Note the specification of a "..." argument:
formals(f) <- al <- alist(x=, y=2+3, ...=)
f
al

## environment->list coercion

e1 <- new.env()
e1$a <- 10
```

```
e1$b <- 20
as.list(e1)
```

`list.files` *List the Files in a Directory/Folder*

Description

These functions produce a character vector of the names of files in the named directory.

Usage

```
list.files(path = ".", pattern = NULL, all.files = FALSE,
           full.names = FALSE, recursive = FALSE)
       dir(path = ".", pattern = NULL, all.files = FALSE,
           full.names = FALSE, recursive = FALSE)
```

Arguments

`path`	a character vector of full path names; the default corresponds to the working directory `getwd()`.
`pattern`	an optional regular expression. Only file names which match the regular expression will be returned.
`all.files`	a logical value. If `FALSE`, only the names of visible files are returned. If `TRUE`, all file names will be returned.
`full.names`	a logical value. If `TRUE`, the directory path is prepended to the file names. If `FALSE`, only the file names are returned.
`recursive`	logical. Should the listing recurse into directories?

Value

A character vector containing the names of the files in the specified directories, or `""` if there were no files. If a path does not exist or is not a directory or is unreadable it is skipped, with a warning.

The files are sorted in alphabetical order, on the full path if `full.names = TRUE`.

Note

File naming conventions are very platform dependent.

`recursive = TRUE` is not supported on all platforms, and may be ignored, with a warning.

Author(s)

Ross Ihaka, Brian Ripley

See Also

`file.info`, `file.access` and `files` for many more file handling functions and `file.choose` for interactive selection.

`glob2rx` to convert wildcards (as used by system file commands and shells) to regular expressions.

Examples

```
list.files(R.home())
## Only files starting with a-l or r (*including* uppercase):
dir("../..", pattern = "^[a-lr]",full.names=TRUE)
```

---

load *Reload Saved Datasets*

Description

Reload datasets written with the function `save`.

Usage

```
load(file, envir = parent.frame())
```

Arguments

`file`	a (readable binary) connection or a character string giving the name of the file to load.
`envir`	the environment where the data should be loaded.

Details

`load` can load R objects saved in the current or any earlier format. It can read a compressed file (see `save`) directly from a file or from a suitable connection (including a call to `url`).

Only R objects saved in the current format (used since R 1.4.0) can be read from a connection. If no input is available on a connection a warning will be given, but any input not in the current format will result in a error.

Loading from an earlier version will give a warning about the 'magic number': magic numbers `1971:1977` are from R < 0.99.0, and `R[ADX]1` from R 0.99.0 to R 1.3.1.

Value

A character vector of the names of objects created, invisibly.

Warning

Saved R objects are binary files, even those saved with `ascii = TRUE`, so ensure that they are transferred without conversion of end of line markers. `load` tries to detect this case and give an informative error message.

Note

Prior to R 2.4.0 an object saved as a `pairlist` was coerced to a `list`, silently (and undocumented).

See Also

`save`, `download.file`.

Examples

```
## save all data
xx <- pi # to ensure there is some data
save(list = ls(all=TRUE), file= "all.Rdata")
rm(xx)

## restore the saved values to the current environment
local({
   load("all.Rdata")
   ls()
})
## restore the saved values to the user's workspace
load("all.Rdata", .GlobalEnv)

unlink("all.Rdata")

## Not run:
con <- url("http://some.where.net/R/data/kprats.rda")
## print the value to see what objects were created.
print(load(con))
close(con)
## End(Not run)
```

---

localeconv *Find Details of the Numerical and Monetary Representations in the Current Locale*

Description

Get details of the numerical and monetary representations in the current locale.

Usage

```
Sys.localeconv()
```

Details

These settings are usually controlled by the environment variables `LC_NUMERIC` and `LC_MONETARY` and if not set the values of `LC_ALL` or `LANG`.

Normally R is run without looking at the value of `LC_NUMERIC`, so the decimal point remains '`.`'. So the first three of these values will not be useful unless you have set `LC_NUMERIC` in the current R session.

Value

A character vector with 18 named components. See your ISO C documentation for details of the meaning.

It is possible to compile R without support for locales, in which case the value will be `NULL`.

See Also

`Sys.setlocale` for ways to set locales.

Examples

```
Sys.localeconv()
## The results in the C locale are
##    decimal_point     thousands_sep          grouping   int_curr_symbol
##              "."                ""                ""                ""
##  currency_symbol mon_decimal_point mon_thousands_sep      mon_grouping
##                ""                ""                ""                ""
##    positive_sign     negative_sign   int_frac_digits       frac_digits
##                ""                ""             "127"             "127"
##    p_cs_precedes    p_sep_by_space     n_cs_precedes    n_sep_by_space
##             "127"             "127"             "127"             "127"
##      p_sign_posn       n_sign_posn
##             "127"             "127"

## Now try your default locale (which might be "C").
## Not run:
old <- Sys.getlocale()
Sys.setlocale(locale = "")
Sys.localeconv()
Sys.setlocale(locale = old)
## End(Not run)

## Not run: read.table("foo", dec=Sys.localeconv()["decimal_point"])
```

`locales` *Query or Set Aspects of the Locale*

Description

Get details of or set aspects of the locale for the R process.

Usage

```
Sys.getlocale(category = "LC_ALL")
Sys.setlocale(category = "LC_ALL", locale = "")
```

Arguments

`category`	character string. The following categories should always be supported: `"LC_ALL"`, `"LC_COLLATE"`, `"LC_CTYPE"`, `"LC_MONETARY"`, `"LC_NUMERIC"` and `"LC_TIME"`. Some systems will also support `"LC_MESSAGES"`, `"LC_PAPER"` and `"LC_MEASUREMENT"`.
`locale`	character string. A valid locale name on the system in use. Normally `""` (the default) will pick up the default locale for the system.

Details

The locale describes aspects of the internationalization of a program. Initially most aspects of the locale of R are set to `"C"` (which is the default for the C language and reflects North-American usage). R sets `"LC_CTYPE"` and `"LC_COLLATE"`, which allow the use of a different character set and alphabetic comparisons in that character set (including the use of `sort`), `"LC_MONETARY"` (for use by `Sys.localeconv`) and `"LC_TIME"` may affect the behaviour of `as.POSIXlt` and `strptime` and functions which use them (but not `date`).

R can be built with no support for locales, but it is normally available on Unix and is available on Windows.

The first seven categories described here are those specified by POSIX. `"LC_MESSAGES"` will be `"C"` on systems that do not support message translation, and is not supported on Windows. Trying to use an unsupported category is an error for `Sys.setlocale`.

Note that setting `"LC_ALL"` sets only `"LC_COLLATE"`, `"LC_CTYPE"`, `"LC_MONETARY"` and `"LC_TIME"`.

Value

A character string of length one describing the locale in use (after setting for `Sys.setlocale`), or an empty character string if the locale is invalid (with a warning) or `NULL` if locale information is unavailable.

For `category = "LC_ALL"` the details of the string are system-specific: it might be a single locale or a set of locales separated by `"/"` (Solaris) or `";"` (Windows, Linux). For portability, it is best to query categories individually. It is guaranteed that the result of `foo <- Sys.getlocale()` can be used in `Sys.setlocale("LC_ALL", locale = foo)` on the same machine.

Warning

Setting `"LC_NUMERIC"` may cause R to function anomalously, so gives a warning. (The known problems are with input conversion in locales with `,` as the decimal point.) Setting it temporarily to produce graphical or text output may work well enough, but `options(OutDec)` is often preferable.

See Also

`strptime` for uses of `category = "LC_TIME"`. `Sys.localeconv` for details of numerical and monetary representations.

Examples

```
Sys.getlocale()
Sys.getlocale("LC_TIME")
## Not run:
Sys.setlocale("LC_TIME", "de")      # Solaris: details are OS-dependent
Sys.setlocale("LC_TIME", "German") # Windows
## End(Not run)
Sys.getlocale("LC_PAPER")           # may or may not be set

Sys.setlocale("LC_COLLATE", "C")    # turn off locale-specific sorting
```

`log` *Logarithms and Exponentials*

Description

log computes natural logarithms, log10 computes common (i.e., base 10) logarithms, and log2 computes binary (i.e., base 2) logarithms. The general form logb(x, base) computes logarithms with base base.

log1p(x) computes $\log(1 + x)$ accurately also for $|x| \ll 1$ (and less accurately when $x \approx -1$).

exp computes the exponential function.

expm1(x) computes $\exp(x) - 1$ accurately also for $|x| \ll 1$.

Usage

```
log(x, base = exp(1))
logb(x, base = exp(1))
log10(x)
log2(x)

exp(x)
expm1(x)

log1p(x)
```

Arguments

x	a numeric or complex vector.
base	positive number. The base with respect to which logarithms are computed. Defaults to e=exp(1).

Details

exp and log are generic functions: methods can be defined for them individually or via the Math group generic.

log10 and log2 are only special cases, but will be computed more efficiently and accurately where supported by the OS.

Value

A vector of the same length as x containing the transformed values. log(0) gives -Inf (when available).

Note

log and logb are the same thing in R, but logb is preferred if base is specified, for S-PLUS compatibility.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (for log, log10 and exp.)

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer. (for logb.)

See Also

`Trig`, `sqrt`, `Arithmetic`.

Examples

```
log(exp(3))
log10(1e7)# = 7

x <- 10^-(1+2*1:9)
cbind(x, log(1+x), log1p(x), exp(x)-1, expm1(x))
```

`Logic` *Logical Operators*

Description

These operators act on logical vectors.

Usage

```
! x
x & y
x && y
x | y
x || y
xor(x, y)

isTRUE(x)
```

Arguments

`x, y`	logical vectors, or objects which can be coerced to such or for which methods have been written.

Details

`!` indicates logical negation (NOT).

`&` and `&&` indicate logical AND and `|` and `||` indicate logical OR. The shorter form performs elementwise comparisons in much the same way as arithmetic operators. The longer form evaluates left to right examining only the first element of each vector. Evaluation proceeds only until the result is determined. The longer form is appropriate for programming control-flow and typically preferred in `if` clauses.

`xor` indicates elementwise exclusive OR.

`isTRUE(x)` is an abbreviation of `identical(TRUE, x)`, and so is true if and only if `x` is a length-one logical vector with no attributes (not even names).

Numeric and complex vectors will be coerced to logical values, with zero being false and all non-zero values being true. Raw vectors are handled without any coercion for `!`, `&` and `|`, with these operators being applied bitwise (so `!` is the 1-complement).

The operators `!`, `&` and `|` are generic functions: methods can be written for them individually or via the `Ops`) group generic function. (See `Ops` for how dispatch is computed.)

NA is a valid logical object. Where a component of x or y is NA, the result will be NA if the outcome is ambiguous. In other words NA & TRUE evaluates to NA, but NA & FALSE evaluates to FALSE. See the examples below.

See Syntax for the precedence of these operators: unlike many other languages (including S) the AND and OR operators do not have the same precedence (the AND operators are higher than the OR operators).

Value

For !, a logical or raw vector of the same length as x.

For |, & and xor a logical or raw vector. The elements of shorter vectors are recycled as necessary (with a warning when they are recycled only *fractionally*). The rules for determining the attributes of the result are rather complicated. Most attributes are taken from the longer argument, the first if they are of the same length. Names will be copied from the first if it is the same length as the answer, otherwise from the second if that is. For time series, these operations are allowed only if the series are compatible, when the class and tsp attribute of whichever is a time series (the same, if both are) are used. For arrays (and an array result) the dimensions and dimnames are taken from first argument if it is an array, otherwise the second.

For ||, && and isTRUE, a length-one logical vector.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

TRUE or logical.

any and all for OR and AND on many scalar arguments.

Syntax for operator precedence.

Examples

```
y <- 1 + (x <- rpois(50, lambda=1.5) / 4 - 1)
x[(x > 0) & (x < 1)]    # all x values between 0 and 1
if (any(x == 0) || any(y == 0)) "zero encountered"

## construct truth tables :

x <- c(NA, FALSE, TRUE)
names(x) <- as.character(x)
outer(x, x, "&")## AND table
outer(x, x, "|")## OR  table
```

---

logical *Logical Vectors*

Description

Create or test for objects of type "logical", and the basic logical "constants".

Usage

```
TRUE
FALSE
T; F

logical(length = 0)
as.logical(x, ...)
is.logical(x)
```

Arguments

`length`	desired length.
`x`	object to be coerced or tested.
`...`	further arguments passed to or from other methods.

Details

`TRUE` and `FALSE` are part of the R language, where `T` and `F` are global variables set to these. All four are `logical(1)` vectors.

`is.logical` is generic: you can write methods to handle specific classes of objects, see Internal-Methods.

Value

`logical` creates a logical vector of the specified length. Each element of the vector is equal to `FALSE`.

`as.logical` attempts to coerce its argument to be of logical type. For `factor`s, this uses the `levels` (labels). Like `as.vector` it strips attributes including names.

`is.logical` returns `TRUE` or `FALSE` depending on whether its argument is of logical type or not.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

`lower.tri` *Lower and Upper Triangular Part of a Matrix*

Description

Returns a matrix of logicals the same size of a given matrix with entries `TRUE` in the lower or upper triangle.

Usage

```
lower.tri(x, diag = FALSE)
upper.tri(x, diag = FALSE)
```

Arguments

x	a matrix.
diag	logical. Should the diagonal be included?

See Also

diag, matrix.

Examples

```
(m2 <- matrix(1:20, 4, 5))
lower.tri(m2)
m2[lower.tri(m2)] <- NA
m2
```

---

ls *List Objects*

Description

ls and objects return a vector of character strings giving the names of the objects in the specified environment. When invoked with no argument at the top level prompt, ls shows what data sets and functions a user has defined. When invoked with no argument inside a function, ls returns the names of the functions local variables. This is useful in conjunction with browser.

Usage

```
ls(name, pos = -1, envir = as.environment(pos),
   all.names = FALSE, pattern)
objects(name, pos= -1, envir = as.environment(pos),
        all.names = FALSE, pattern)
```

Arguments

name	which environment to use in listing the available objects. Defaults to the *current* environment. Although called name for back compatibility, in fact this argument can specify the environment in any form; see the details section.
pos	An alternative argument to name for specifying the environment as a position in the search list. Mostly there for back compatibility.
envir	an alternative argument to name for specifying the environment evaluation environment. Mostly there for back compatibility.
all.names	a logical value. If TRUE, all object names are returned. If FALSE, names which begin with a '.' are omitted.
pattern	an optional regular expression. Only names matching pattern are returned. glob2rx can be used to convert wildcard patterns to regular expressions.

Details

The name argument can specify the environment from which object names are taken in one of several forms: as an integer (the position in the search list); as the character string name of an element in the search list; or as an explicit environment (including using sys.frame to access the currently active function calls). By default, the environment of the call to ls or objects is used. The pos and envir arguments are an alternative way to specify an environment, but are primarily there for back compatibility.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

glob2rx for converting wildcard patterns to regular expressions.

ls.str for a long listing based on str. apropos (or find) for finding objects in the whole search path; grep for more details on "regular expressions"; class, methods, etc., for object-oriented programming.

Examples

```
.Ob <- 1
ls(pat="O")
ls(pat="O", all = TRUE)    # also shows ".[foo]"

# shows an empty list because inside myfunc no variables are defined
myfunc <- function() {ls()}
myfunc()

# define a local variable inside myfunc
myfunc <- function() {y <- 1; ls()}
myfunc()                # shows "y"
```

---

make.names *Make Syntactically Valid Names*

Description

Make syntactically valid names out of character vectors.

Usage

```
make.names(names, unique = FALSE, allow_ = TRUE)
```

Arguments

names	character vector to be coerced to syntactically valid names. This is coerced to character if necessary.
unique	logical; if TRUE, the resulting elements are unique. This may be desired for, e.g., column names.
allow_	logical. For compatibility with R prior to 1.9.0.

Details

A syntactically valid name consists of letters, numbers and the dot or underline characters and starts with a letter or the dot not followed by a number. Names such as ".2way" are not valid, and neither are the reserved words.

The character "X" is prepended if necessary. All invalid characters are translated to ".". A missing value is translated to "NA". Names which match R keywords have a dot appended to them. Duplicated values are altered by make.unique.

Value

A character vector of same length as names with each changed to a syntactically valid name.

Note

Prior to R version 1.9.0, underscores were not valid in variable names, and code that relies on them being converted to dots will no longer work. Use allow_ = FALSE for back-compatibility.

allow_ = FALSE is also useful when creating names for export to applications which do not allow underline in names (for example, S-PLUS and some DBMSs).

See Also

make.unique, names, character, data.frame.

Examples

```
make.names(c("a and b", "a-and-b"), unique=TRUE)
# "a.and.b"  "a.and.b.1"
make.names(c("a and b", "a_and_b"), unique=TRUE)
# "a.and.b"  "a_and_b"
make.names(c("a and b", "a_and_b"), unique=TRUE, allow_=FALSE)
# "a.and.b"  "a.and.b.1"

state.name[make.names(state.name) != state.name] # those 10 with a space
```

---

make.unique *Make Character Strings Unique*

Description

Makes the elements of a character vector unique by appending sequence numbers to duplicates.

Usage

```
make.unique(names, sep = ".")
```

Arguments

names	a character vector
sep	a character string used to separate a duplicate name from its sequence number.

Details

The algorithm used by `make.unique` has the property that `make.unique(c(A, B)) == make.unique(c(make.unique(A), B))`.

In other words, you can append one string at a time to a vector, making it unique each time, and get the same result as applying `make.unique` to all of the strings at once.

If character vector `A` is already unique, then `make.unique(c(A, B))` preserves `A`.

Value

A character vector of same length as `names` with duplicates changed.

Author(s)

Thomas P Minka

See Also

`make.names`

Examples

```
make.unique(c("a", "a", "a"))
make.unique(c(make.unique(c("a", "a")), "a"))

make.unique(c("a", "a", "a.2", "a"))
make.unique(c(make.unique(c("a", "a")), "a.2", "a"))

rbind(data.frame(x=1), data.frame(x=2), data.frame(x=3))
rbind(rbind(data.frame(x=1), data.frame(x=2)), data.frame(x=3))
```

`manglePackageName` *Mangle the Package Name*

Description

This function takes the package name and the package version number and pastes them together with a separating underscore.

Usage

```
manglePackageName(pkgName, pkgVersion)
```

Arguments

`pkgName`	The package name, as a character string.
`pkgVersion`	The package version, as a character string.

Value

A character string with the two inputs pasted together.

Examples

```
manglePackageName("foo", "1.2.3")
```

`mapply` *Apply a function to multiple list or vector arguments*

Description

`mapply` is a multivariate version of `sapply`. `mapply` applies `FUN` to the first elements of each ... argument, the second elements, the third elements, and so on. Arguments are recycled if necessary.

`Vectorize` returns a new function that acts as if `mapply` was called.

Usage

```
mapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE,
       USE.NAMES = TRUE)

Vectorize(FUN, vectorize.args = arg.names, SIMPLIFY = TRUE,
          USE.NAMES = TRUE)
```

Arguments

`FUN`	function to apply, found via `match.fun`.
`...`	arguments to vectorise over (list or vector).
`MoreArgs`	a list of other arguments to `FUN`.
`SIMPLIFY`	logical; attempt to reduce the result to a vector or matrix?
`USE.NAMES`	logical; use names if the first ... argument has names, or if it is a character vector, use that character vector as the names.
`vectorize.args`	a character vector of arguments which should be vectorized. Defaults to all arguments to `FUN`.

Details

The arguments named in the `vectorize.args` argument to `Vectorize` correspond to the arguments passed in the `...` list to `mapply`. However, only those that are actually passed will be vectorized; default values will not. See the example below.

`Vectorize` cannot be used with primitive functions as they have no formal list.

Value

`mapply` returns a list, vector, or matrix.

`Vectorize` returns a function with the same arguments as `FUN`, but wrapping a call to `mapply`.

See Also

`sapply`, `outer`

Examples

```
mapply(rep, 1:4, 4:1)

mapply(rep, times=1:4, x=4:1)

mapply(rep, times=1:4, MoreArgs=list(x=42))

# Repeat the same using Vectorize: use rep.int as rep is primitive
vrep <- Vectorize(rep.int)
vrep(1:4, 4:1)
vrep(times=1:4, x=4:1)

vrep <- Vectorize(rep.int, "times")
vrep(times=1:4, x=42)

mapply(function(x,y) seq_len(x) + y,
       c(a= 1, b=2, c= 3),  # names from first
       c(A=10, B=0, C=-10))

word <- function(C,k) paste(rep.int(C,k), collapse='')
str(mapply(word, LETTERS[1:6], 6:1, SIMPLIFY = FALSE))

f <- function(x=1:3, y) c(x,y)
vf <- Vectorize(f, SIMPLIFY = FALSE)
f(1:3,1:3)
vf(1:3,1:3)
vf(y=1:3) # Only vectorizes y, not x

# Nonlinear regression contour plot, based on nls() example

SS <- function(Vm, K, resp, conc) {
    pred <- (Vm * conc)/(K + conc)
    sum((resp - pred)^2 / pred)
}
vSS <- Vectorize(SS, c("Vm", "K"))
Treated <- subset(Puromycin, state == "treated")

Vm <- seq(140, 310, len=50)
K <- seq(0, 0.15, len=40)
SSvals <- outer(Vm, K, vSS, Treated$rate, Treated$conc)
contour(Vm, K, SSvals, levels=(1:10)^2, xlab="Vm", ylab="K")
```

---

margin.table *Compute table margin*

Description

For a contingency table in array form, compute the sum of table entries for a given index.

Usage

```
margin.table(x, margin=NULL)
```

Arguments

x	an array
margin	index number (1 for rows, etc.)

Details

This is really just `apply(x, margin, sum)` packaged up for newbies, except that if `margin` has length zero you get `sum(x)`.

Value

The relevant marginal table. The class of `x` is copied to the output table, except in the summation case.

Author(s)

Peter Dalgaard

See Also

`prop.table` and `addmargins`.

Examples

```
m <- matrix(1:4,2)
margin.table(m,1)
margin.table(m,2)
```

---

mat.or.vec *Create a Matrix or a Vector*

Description

`mat.or.vec` creates an `nr` by `nc` zero matrix if `nc` is greater than 1, and a zero vector of length `nr` if `nc` equals 1.

Usage

```
mat.or.vec(nr, nc)
```

Arguments

nr, nc	numbers of rows and columns.

Examples

```
mat.or.vec(3, 1)
mat.or.vec(3, 2)
```

match *Value Matching*

Description

match returns a vector of the positions of (first) matches of its first argument in its second.

%in% is a more intuitive interface as a binary operator, which returns a logical vector indicating if there is a match or not for its left operand.

Usage

```
match(x, table, nomatch = NA, incomparables = FALSE)

x %in% table
```

Arguments

x	vector or NULL: the values to be matched.
table	vector or NULL: the values to be matched against.
nomatch	the value to be returned in the case when no match is found. Note that it is coerced to integer.
incomparables	a vector of values that cannot be matched. Any value in x matching a value in this vector is assigned the nomatch value. Currently, FALSE is the only possible value, meaning that all values can be matched.

Details

%in% is currently defined as
`"%in%" <- function(x, table) match(x, table, nomatch = 0) > 0`

Factors, raw vectors and lists are converted to character vectors, and then x and table are coerced to a common type (the later of the two types in R's ordering, logical < integer < numeric < complex < character) before matching.

Matching for lists is potentially very slow and best avoided except in simple cases.

Exactly what matches what is to some extent a matter of definition. For all types, NA matches NA and no other value. For real and complex values, NaN values are regarded as matching any other NaN value, but not matching NA.

Value

A vector of the same length as x.

match: An integer vector giving the position in table of the first match if there is a match, otherwise nomatch.

If x[i] is found to equal table[j] then the value returned in the i-th position of the return value is j, for the smallest possible j. If no match is found, the value is nomatch.

%in%: A logical vector, indicating if a match was located for each element of x.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`pmatch` and `charmatch` for (*partial*) string matching, `match.arg`, etc for function argument matching. `findInterval` similarly returns a vector of positions, but finds numbers within intervals, rather than exact matches.

`is.element` for an S-compatible equivalent of `%in%`.

Examples

```
## The intersection of two sets :
intersect <- function(x, y) y[match(x, y, nomatch = 0)]
intersect(1:10,7:20)

1:10 %in% c(1,3,5,9)
sstr <- c("c","ab","B","bba","c","@","bla","a","Ba","%")
sstr[sstr %in% c(letters,LETTERS)]

"%w/o%" <- function(x,y) x[!x %in% y] #--  x without y
(1:10) %w/o% c(3,7,12)
```

`match.arg` *Argument Verification Using Partial Matching*

Description

`match.arg` matches `arg` against a table of candidate values as specified by `choices`.

Usage

```
match.arg(arg, choices, several.ok = FALSE)
```

Arguments

`arg`	a character string
`choices`	a character vector of candidate values
`several.ok`	logical specifying if `arg` should be allowed to have more than one element.

Details

In the one-argument form `match.arg(arg)`, the choices are obtained from a default setting for the formal argument `arg` of the function from which `match.arg` was called.

Matching is done using `pmatch`, so `arg` may be abbreviated.

Value

The unabbreviated version of the unique partial match if there is one; otherwise, an error is signalled if `several.ok` is false, as per default. When `several.ok` is true and there is more than one match, all unabbreviated versions of matches are returned.

See Also

pmatch, match.fun, match.call.

Examples

```
require(stats)
## Extends the example for 'switch'
center <- function(x, type = c("mean", "median", "trimmed")) {
  type <- match.arg(type)
  switch(type,
         mean = mean(x),
         median = median(x),
         trimmed = mean(x, trim = .1))
}
x <- rcauchy(10)
center(x, "t")       # Works
center(x, "med")     # Works
try(center(x, "m"))  # Error

## Allowing more than one match:
match.arg(c("gauss", "rect", "ep"),
          c("gaussian", "epanechnikov", "rectangular", "triangular"),
          several.ok = TRUE)
```

---

match.call *Argument Matching*

Description

match.call returns a call in which all of the specified arguments are specified by their full names.

Usage

```
match.call(definition = NULL, call = sys.call(sys.parent()),
           expand.dots = TRUE)
```

Arguments

definition	a function, by default the function from which match.call is called.
call	an unevaluated call to the function specified by definition, as generated by call.
expand.dots	logical. Should arguments matching ... in the call be included or left as a ... argument?

Details

match.call is most commonly used in two circumstamces:

- To record the call for later re-use: for example most model-fitting functions record the call as element call of the list they return. Here the default expand.dots = TRUE is appropriate.

- To pass most of the call to another function, often `model.frame`. Here the common idiom is that `expand.dots = FALSE` is used, and the `...` elememt of the matched call is removed. An alternative is to explicitly select the arguments to be passed on, as is done in `lm`.

Value

An object of class `call`.

References

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer.

See Also

`call`, `pmatch`, `match.arg`, `match.fun`.

Examples

```
match.call(get, call("get", "abc", i = FALSE, p = 3))
## -> get(x = "abc", pos = 3, inherits = FALSE)
fun <- function(x, lower = 0, upper = 1) {
  structure((x - lower) / (upper - lower), CALL = match.call())
}
fun(4 * atan(1), u = pi)
```

---

match.fun *Function Verification for "Function Variables"*

Description

When called inside functions that take a function as argument, extract the desired function object while avoiding undesired matching to objects of other types.

Usage

```
match.fun(FUN, descend = TRUE)
```

Arguments

`FUN`	item to match as function: a function, symbol or character string. See Details.
`descend`	logical; control whether to search past non-function objects.

Details

`match.fun` is not intended to be used at the top level since it will perform matching in the *parent* of the caller.

If `FUN` is a function, it is returned. If it is a symbol (for example, enclosed in backquotes) or a character vector of length one, it will be looked up using `get` in the environment of the parent of the caller. If it is of any other mode, it is attempted first to get the argument to the caller as a symbol (using `substitute` twice), and if that fails, an error is declared.

If `descend = TRUE`, `match.fun` will look past non-function objects with the given name; otherwise if `FUN` points to a non-function object then an error is generated.

This is used in base functions such as `apply`, `lapply`, `outer`, and `sweep`.

Value

A function matching FUN or an error is generated.

Bugs

The descend argument is a bit of misnomer and probably not actually needed by anything. It may go away in the future.

It is impossible to fully foolproof this. If one attaches a list or data frame containing a length-one character vector with the same name as a function, it may be used (although namespaces will help).

Author(s)

Peter Dalgaard and Robert Gentleman, based on an earlier version by Jonathan Rougier.

See Also

match.arg, get

Examples

```
# Same as get("*"):
match.fun("*")
# Overwrite outer with a vector
outer <- 1:5
## Not run:
match.fun(outer, descend = FALSE) #-> Error:  not a function
## End(Not run)
match.fun(outer) # finds it anyway
is.function(match.fun("outer")) # as well
```

---

matmult *Matrix Multiplication*

Description

Multiplies two matrices, if they are conformable. If one argument is a vector, it will be promoted to either a row or column matrix to make the two arguments conformable. If both are vectors it will return the inner product.

Usage

```
a %*% b
```

Arguments

a, b	numeric or complex matrices or vectors.

Details

When a vector is promoted to a matrix, its names are not promoted to row or column names, unlike as.matrix.

Value

The matrix product. Use `drop` to get rid of dimensions which have only one level.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`matrix`, `Arithmetic`, `diag`.

Examples

```
x <- 1:4
(z <- x %*% x)     # scalar ("inner") product (1 x 1 matrix)
drop(z)              # as scalar

y <- diag(x)
z <- matrix(1:12, ncol = 3, nrow = 4)
y %*% z
y %*% x
x %*% z
```

`matrix` *Matrices*

Description

`matrix` creates a matrix from the given set of values.

`as.matrix` attempts to turn its argument into a matrix.

`is.matrix` tests if its argument is a (strict) matrix. It is generic: you can write methods to handle specific classes of objects, see InternalMethods.

Usage

```
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE,
       dimnames = NULL)

as.matrix(x)

is.matrix(x)
```

Arguments

`data`	an optional data vector.
`nrow`	the desired number of rows
`ncol`	the desired number of columns
`byrow`	logical. If `FALSE` (the default) the matrix is filled by columns, otherwise the matrix is filled by rows.

`dimnames` A `dimnames` attribute for the matrix: a `list` of length 2 giving the row and column names respectively.

`x` an R object.

Details

If either of `nrow` or `ncol` is not given, an attempt is made to infer it from the length of `data` and the other parameter.

If there are too few elements in `data` to fill the array, then the elements in `data` are recycled. If `data` has length zero, `NA` of an appropriate type is used for atomic vectors (`0` for raw vectors) and `NULL` for lists.

`is.matrix` returns `TRUE` if `x` is a matrix (i.e., it is *not* a `data.frame` and has a `dim` attribute of length 2) and `FALSE` otherwise.

`as.matrix` is a generic function. The method for data frames will convert any non-numeric/complex column into a character vector using `format` and so return a character matrix, except that all-logical data frames will be coerced to a logical matrix. When coercing a vector, it produces a one-column matrix, and promotes the names (if any) of the vector to the rownames of the matrix.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`data.matrix`, which attempts to convert to a numeric matrix.

Examples

```
is.matrix(as.matrix(1:10))
!is.matrix(warpbreaks)# data.frame, NOT matrix!
warpbreaks[1:10,]
as.matrix(warpbreaks[1:10,]) #using as.matrix.data.frame(.) method

# Example of setting row and column names
mdat <- matrix(c(1,2,3, 11,12,13), nrow = 2, ncol=3, byrow=TRUE,
               dimnames = list(c("row1", "row2"), c("C.1", "C.2", "C.3")))
mdat
```

`maxCol` *Find Maximum Position in Matrix*

Description

Find the maximum position for each row of a matrix, breaking ties at random.

Usage

```
max.col(m, ties.method=c("random", "first", "last"))
```

Arguments

m	numerical matrix
ties.method	a character string specifying how "ties" are handled, "random" by default; can be abbreviated; see *Details*.

Details

When ties.method = "random", as per default, ties are broken at random. In this case, the determination of "tie" assumes that the entries are probabilities: there is a relative tolerance of 10^{-5}, relative to the largest entry in the row.

If ties.method = "first", max.col returns the column number of the *first* of several maxima in every row, the same as unname(unname(m, 1, unname)).
Correspondingly, ties.method = "last" returns the *last* of possibly several indices.

Value

index of a maximal value for each row, an integer vector of length nrow(m).

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* New York: Springer (4th ed).

See Also

which.max for vectors.

Examples

```
table(mc <- max.col(swiss))# mostly "1" and "5", 5 x "2" and once "4"
swiss[unique(print(mr <- max.col(t(swiss)))) , ] # 3 33 45 45 33 6

set.seed(1)# reproducible example:
(mm <- rbind(x = round(2*runif(12)),
             y = round(5*runif(12)),
             z = round(8*runif(12))))
## Not run:
  [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
x    1    1    1    2    0    2    2    1    1     0     0     0
y    3    2    4    2    4    5    2    4    5     1     3     1
z    2    3    0    3    7    3    4    5    4     1     7     5
## End(Not run)
## column indices of all row maxima :
str(lapply(1:3, function(i) which(mm[i,] == max(mm[i,]))))
max.col(mm) ; max.col(mm) # "random"
max.col(mm, "first")# -> 4 6 5
max.col(mm, "last") # -> 7 9 11
```

mean *Arithmetic Mean*

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

Arguments

x	An R object. Currently there are methods for numeric data frames, numeric vectors and dates. A complex vector is allowed for `trim = 0`, only.
trim	the fraction (0 to 0.5) of observations to be trimmed from each end of `x` before the mean is computed.
na.rm	a logical value indicating whether `NA` values should be stripped before the computation proceeds.
...	further arguments passed to or from other methods.

Value

For a data frame, a named vector with the appropriate method being applied column by column.

If `trim` is zero (the default), the arithmetic mean of the values in `x` is computed, as a numeric or complex vector of length one. If any argument is not logical (coerced to numeric), integer, numeric or complex, `NA` is returned, with a warning.

If `trim` is non-zero, a symmetrically trimmed mean is computed with a fraction of `trim` observations deleted from each end before the mean is computed.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`weighted.mean`, `mean.POSIXct`

Examples

```
x <- c(0:10, 50)
xm <- mean(x)
c(xm, mean(x, trim = 0.10))

mean(USArrests, trim = 0.2)
```

`Memory` *Memory Available for Data Storage*

Description

Use command line options to control the memory available for R.

Usage

```
R --min-vsize=vl --max-vsize=vu --min-nsize=nl --max-nsize=nu --max-ppsize=N

mem.limits(nsize = NA, vsize = NA)
```

Arguments

`vl, vu, vsize`	Heap memory in bytes.
`nl, nu, nsize`	Number of cons cells.
`N`	Number of nested `PROTECT` calls.

.

Details

R has a variable-sized workspace. There is much less need to set memory options than prior to R 1.2.0, and most users will never need to set these. They are provided both as a way to control the overall memory usage (which can also be done by operating-system facilities such as `limit` on Unix), and since setting larger values of the minima will make R slightly more efficient on large tasks.

To understand the options, one needs to know that R maintains separate areas for fixed and variable sized objects. The first of these is allocated as an array of "*cons cells*" (Lisp programmers will know what they are, others may think of them as the building blocks of the language itself, parse trees, etc.), and the second are thrown on a "*heap*" of "Vcells" of 8 bytes each. Effectively, the inputs `vl` and `vu` are rounded up to the next multiple of 8.

Each cons cell occupies 28 bytes on a 32-bit machine, (usually) 56 bytes on a 64-bit machine.

The '`--*-nsize`' options can be used to specify the number of cons cells and the '`--*-vsize`' options specify the size of the vector heap in bytes. Both options must be integers or integers followed by `G`, `M`, `K`, or `k` meaning `Giga` ($2^{30} = 1073741824$) *Mega* ($2^{20} = 1048576$), (computer) *Kilo* ($2^{10} = 1024$), or regular *kilo* (1000).

The '`--min-*`' options set the minimal sizes for the number of cons cells and for the vector heap. These values are also the initial values, but thereafter R will grow or shrink the areas depending on usage, but never exceeding the limits set by the '`--max-*`' options nor decreasing below the initial values.

The default values are currently minima of 350k cons cells, 6Mb of vector heap and no maxima (other than machine resources). The maxima can be changed during an R session by calling `mem.limits`. (If this is called with the default values, it reports the current settings.)

You can find out the current memory consumption (the heap and cons cells used as numbers and megabytes) by typing `gc()` at the R prompt. Note that following `gcinfo(TRUE)`, automatic

garbage collection always prints memory use statistics. Maxima will never be reduced below the current values for triggering garbage collection, and attempts to do so will be silently ignored.

The option '`--max-ppsize`' controls the maximum size of the pointer protection stack. This defaults to 50000, but can be increased to allow deep recursion or large and complicated calculations to be done. *Note* that parts of the garbage collection process goes through the full reserved pointer protection stack and hence becomes slower when the size is increased. Currently the maximum value accepted is 500000.

Value

`mem.limits()` returns an integer vector giving the current settings of the maxima, possibly `NA`.

See Also

An Introduction to R for more command-line options

`Memory-limits` for the design limitations.

`gc` for information on the garbage collector and total memory usage, `object.size(a)` for the (approximate) size of R object `a`. `memory.profile` for profiling the usage of cons cells.

Examples

```
# Start R with 10MB of heap memory and 500k cons cells, limit to
# 100Mb and 1M cells
## Not run:
## Unix
R --min-vsize=10M --max-vsize=100M --min-nsize=500k --max-nsize=1M
## End(Not run)
```

`Memory-limits` *Memory Limits in R*

Description

R holds objects it is using in memory. This help file documents the current design limitations on large objects: these differ between 32-bit and 64-bit builds of R.

Details

R holds all objects in memory, and there are limits based on the amount of memory that can be used by all objects:

- There may be limits on the size of the heap and the number of cons cells allowed – see `Memory` – but these are usually not imposed.
- There is a limit on the address space of a single process such as the R executable. This is system-specific, but 32-bit OSes imposes a limit of no more than 4Gb: it is often 3Gb or less.
- The environment may impose limitations on the resources available to a single process – see the OS/shell's help on commands such as `limit` or `ulimit`.

Error messages beginning `cannot allocate vector of size` indicate a failure to obtain memory, either because the size exceeded the address-space limit for a process or, more likely, because the system was unable to provide the memory. Note that on a 32-bit OS there may well be enough free memory available, but not a large enough contiguous block of address space into which to map it.

There are also limits on individual objects. On all versions of R, the maximum length (number of elements) of a vector is $2^{31} - 1 \approx 2\,10^9$, as lengths are stored as signed integers. In addition, the storage space cannot exceed the address limit, and if you try to exceed that limit, the error message begins `cannot allocate vector of length`. The number of characters in a character string is in theory only limited by the address space.

See Also

`object.size(a)` for the (approximate) size of R object `a`.

`memory.profile` *Profile the Usage of Cons Cells*

Description

Lists the usage of the cons cells by `SEXPREC` type.

Usage

```
memory.profile()
```

Details

The current types and their uses are listed in the include file 'Rinternals.h'.

Value

A vector of counts, named by the types. See `typeof` for an explanation of types.

See Also

`gc` for the overall usage of cons cells.

Examples

```
memory.profile()
```

`merge` *Merge Two Data Frames*

Description

Merge two data frames by common columns or row names, or do other versions of database "join" operations.

Usage

```
merge(x, y, ...)

## Default S3 method:
merge(x, y, ...)

## S3 method for class 'data.frame':
merge(x, y, by = intersect(names(x), names(y)),
      by.x = by, by.y = by, all = FALSE, all.x = all, all.y = all,
      sort = TRUE, suffixes = c(".x",".y"), ...)
```

Arguments

`x, y`	data frames, or objects to be coerced to one.
`by, by.x, by.y`	specifications of the common columns. See Details.
`all`	logical; `all = L` is shorthand for `all.x = L` and `all.y = L`.
`all.x`	logical; if `TRUE`, then extra rows will be added to the output, one for each row in `x` that has no matching row in `y`. These rows will have `NA`s in those columns that are usually filled with values from `y`. The default is `FALSE`, so that only rows with data from both `x` and `y` are included in the output.
`all.y`	logical; analogous to `all.x` above.
`sort`	logical. Should the results be sorted on the `by` columns?
`suffixes`	character(2) specifying the suffixes to be used for making non-`by` `names()` unique.
`...`	arguments to be passed to or from methods.

Details

By default the data frames are merged on the columns with names they both have, but separate specifications of the columns can be given by `by.x` and `by.y`. Columns can be specified by name, number or by a logical vector: the name `"row.names"` or the number `0` specifies the row names. The rows in the two data frames that match on the specified columns are extracted, and joined together. If there is more than one match, all possible matches contribute one row each.

If `by` or both `by.x` and `by.y` are of length 0 (a length zero vector or `NULL`), the result, `r`, is the "Cartesian product" of `x` and `y`, i.e., `dim(r) = c(nrow(x)*nrow(y), ncol(x) + ncol(y))`.

If `all.x` is true, all the non matching cases of `x` are appended to the result as well, with `NA` filled in the corresponding columns of `y`; analogously for `all.y`.

If the remaining columns in the data frames have any common names, these have `suffixes` (`".x"` and `".y"` by default) appended to make the names of the result unique.

Prior to R 2.4.0 the complexity was proportional to the product of the numbers of rows of `x` and `y`. As from R 2.4.0 it is proportional to the length of the answer, which can be much less if there are few matches (for example, only 1–1 matches).

Value

A data frame. The rows are by default lexicographically sorted on the common columns, but for `sort=FALSE` are in an unspecified order. The columns are the common columns followed by the remaining columns in `x` and then those in `y`. If the matching involved row names, an extra character column called `Row.names` is added at the left, and in all cases the result has no special row names.

See Also

`data.frame`, `by`, `cbind`

Examples

```
## use character columns of names to get sensible sort order
authors <- data.frame(
    surname = I(c("Tukey", "Venables", "Tierney", "Ripley", "McNeil")),
    nationality = c("US", "Australia", "US", "UK", "Australia"),
    deceased = c("yes", rep("no", 4)))
books <- data.frame(
    name = I(c("Tukey", "Venables", "Tierney",
               "Ripley", "Ripley", "McNeil", "R Core")),
    title = c("Exploratory Data Analysis",
              "Modern Applied Statistics ...",
              "LISP-STAT",
              "Spatial Statistics", "Stochastic Simulation",
              "Interactive Data Analysis",
              "An Introduction to R"),
    other.author = c(NA, "Ripley", NA, NA, NA, NA,
                     "Venables & Smith"))

(m1 <- merge(authors, books, by.x = "surname", by.y = "name"))
(m2 <- merge(books, authors, by.x = "name", by.y = "surname"))
stopifnot(as.character(m1[,1]) == as.character(m2[,1]),
          all.equal(m1[, -1], m2[, -1][ names(m1)[-1] ]),
          dim(merge(m1, m2, by = integer(0))) == c(36, 10))

## "R core" is missing from authors and appears only here :
merge(authors, books, by.x = "surname", by.y = "name", all = TRUE)
```

---

message *Diagnostic Messages*

Description

Generate a diagnostic message from its arguments.

Usage

```
message(..., domain = NULL)
suppressMessages(expr)
```

Arguments

`...`	character vectors (which are pasted together with no separator), a condition object, or `NULL`.
`domain`	see `gettext`. If `NA`, messages will not be translated.
`expr`	expression to evaluate.

Details

`message` is used for generating "simple" diagnostic messages which are neither warnings nor errors, but nevertheless represented as conditions.

While the message is being processed, a `muffleMessage` restart is available.

`suppressMessages` evaluates its expression in a context that ignores all "simple" diagnostic messages.

See Also

`warning` and `stop` for generating warnings and errors; `conditions` for condition handling and recovery.

`gettext` for the mechanisms for the automated translation of text.

Examples

```
message("ABC", "DEF")
suppressMessages(message("ABC"))
```

`missing` *Does a Formal Argument have a Value?*

Description

`missing` can be used to test whether a value was specified as an argument to a function.

Usage

```
missing(x)
```

Arguments

`x`	a formal argument.

Details

`missing(x)` is only reliable if `x` has not been altered since entering the function: in particular it will *always* be false after `x <- match.arg(x)`.

The example shows how a plotting function can be written to work with either a pair of vectors giving x and y coordinates of points to be plotted or a single vector giving y values to be plotted against their indexes.

Currently `missing` can only be used in the immediate body of the function that defines the argument, not in the body of a nested function or a `local` call. This may change in the future.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer.

See Also

`substitute` for argument expression; `NA` for "missing values" in data.

Examples

```
myplot <- function(x,y) {
                if(missing(y)) {
                        y <- x
                        x <- 1:length(y)
                }
                plot(x,y)
        }
```

---

mode *The (Storage) Mode of an Object*

Description

Get or set the type or storage mode of an object.

Usage

```
mode(x)
mode(x) <- value
storage.mode(x)
storage.mode(x) <- value
```

Arguments

`x`	any R object.
`value`	a character string giving the desired (storage) mode of the object.

Details

Both `mode` and `storage.mode` return a character string giving the (storage) mode of the object — often the same — both relying on the output of `typeof(x)`, see the example below.

The two replacement versions are currently identical. Both `mode(x) <- "newmode"` and `storage.mode(x) <- "newmode"` change the `mode` or `storage.mode` of object `x` to `newmode`. This is only supported if there is an appropriate `as.newmode` function, for example `"logical"`, `"integer"`, `"double"`, `"complex"`, `"raw"`, `"character"`, `"list"`, `"expression"`, `"name"`, `"symbol"` and `"function"`.

As storage mode `"single"` is only a pseudo-mode in R, it will not be reported by `mode` or `storage.mode`: use `attr(object, "Csingle")` to examine this. However, the replacement versions can be used to set the mode to `"single"`, which sets the real mode to `"double"` and the `"Csingle"` attribute to `TRUE`. Setting any other mode will remove this attribute.

Note (in the examples below) that some `call`s have mode `"("` which is S compatible.

Mode names

Modes have the same set of names as types (see `typeof`) except that

- types `"integer"` and `"double"` are returned as `"numeric"`.
- types `"special"` and `"builtin"` are returned as `"function"`.
- type `"symbol"` is called mode `"name"`.
- type `"language"` is returned as `"("` or `"call"`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`typeof` for the R-internal "mode", `attributes`.

Examples

```
sapply(options(),mode)

cex3 <- c("NULL","1","1:1","1i","list(1)","data.frame(x=1)", "pairlist(pi)",
  "c", "lm", "formals(lm)[[1]]",  "formals(lm)[[2]]",
  "y~x","expression((1))[[1]]", "(y~x)[[1]]", "expression(x <- pi)[[1]][[1]]")
lex3 <- sapply(cex3, function(x) eval(parse(text=x)))
mex3 <- t(sapply(lex3, function(x) c(typeof(x), storage.mode(x), mode(x))))
dimnames(mex3) <- list(cex3, c("typeof(.)","storage.mode(.)","mode(.)"))
mex3

## This also makes a local copy of 'pi':
storage.mode(pi) <- "complex"
storage.mode(pi)
rm(pi)
```

NA *Not Available / "Missing" Values*

Description

NA is a logical constant of length 1 which contains a missing value indicator. NA can be freely coerced to any other vector type except raw.

The generic function is.na indicates which elements are missing.

The generic function is.na<- sets elements to NA.

Usage

```
NA
is.na(x)
## S3 method for class 'data.frame':
is.na(x)

is.na(x) <- value
```

Arguments

x	an R object to be tested.
value	a suitable index vector for use with x.

Details

The NA of character type is distinct from the string "NA". Programmers who need to specify an explicit string NA should use as.character(NA) rather than "NA", or set elements to NA using is.na<-.

is.na(x) works elementwise when x is a list. It is generic: you can write methods to handle specific classes of objects, see InternalMethods.

Function is.na<- may provide a safer way to set missingness. It behaves differently for factors, for example.

Value

The default method for is.na returns a logical vector of the same "form" as its argument x, containing TRUE for those elements marked NA or NaN (!) and FALSE otherwise. dim, dimnames and names attributes are preserved.

The method is.na.data.frame returns a logical matrix with the same dimensions as the data frame, and with dimnames taken from the row and column names of the data frame.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer.

See Also

NaN, is.nan, etc., and the utility function complete.cases.

na.action, na.omit, na.fail on how methods can be tuned to deal with missing values.

Examples

```
is.na(c(1, NA))         #> FALSE  TRUE
is.na(paste(c(1, NA))) #> FALSE FALSE

(xx <- c(0:4))
is.na(xx) <- c(2, 4)
xx                      #> 0 NA  2 NA  4
```

---

name *Names and Symbols*

Description

as.symbol coerces its argument to be a *symbol*, or equivalently, a *name*. The argument must be of mode "character". as.name is an alias for as.symbol.

is.symbol (and is.name equivalently) returns TRUE or FALSE depending on whether its argument is a symbol (i.e., name) or not.

Usage

```
as.symbol(x)
is.symbol(y)

as.name(x)
is.name(y)
```

Arguments

x, y	objects to be coerced or tested.

Details

is.symbol is generic: you can write methods to handle specific classes of objects, see Internal-Methods.

Note

The term "symbol" is from the LISP background of R, whereas "name" has been the standard S term for this.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

call, is.language. For the internal object mode, typeof.

Examples

```
an <- as.name("arrg")
is.name(an) # TRUE
mode(an)    # name
typeof(an) # symbol
```

---

names *The Names of an Object*

Description

Functions to get or set the names of an object.

Usage

```
names(x)
names(x) <- value
```

Arguments

x	an R object.
value	a character vector of up to the same length as x, or NULL.

Details

names is a generic accessor function, and names<- is a generic replacement function. The default methods get and set the "names" attribute of a vector (including a list) or pairlist.

If value is shorter than x, it is extended by character NAs to the length of x.

It is possible to update just part of the names attribute via the general rules: see the examples. This works because the expression there is evaluated as z <- "names<-"(z, "[<-"(names(z), 3, "c2")).

The name "" is special: it is used to indicate that there is no name associated with an element of a (atomic or generic) vector. Subscripting by "" will match nothing (not even elements which have no name).

A name can be character NA, but such a name will never be matched and is likely to lead to confusion.

Value

For names, NULL or a character vector of the same length as x. (NULL is given if the object has no names, including for objects of types which cannot have names.)

For names<-, the updated object. (Note that the value of names(x) <- value is that of the assignment, value, not the return value from the left-hand side.)

Note

For vectors, the names are one of the attributes with restrictions on the possible values. For pairlists, the names are the tags and converted to and from a character vector.

For a one-dimensional array the `names` attribute really is `dimnames[[1]]`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
# print the names attribute of the islands data set
names(islands)

# remove the names attribute
names(islands) <- NULL
islands
rm(islands) # remove the copy made

z <- list(a=1, b="c", c=1:3)
names(z)
# change just the name of the third element.
names(z)[3] <- "c2"
z

z <- 1:3
names(z)
## assign just one name
names(z)[2] <- "b"
z
```

`nargs` *The Number of Arguments to a Function*

Description

When used inside a function body, `nargs` returns the number of arguments supplied to that function, *including* positional arguments left blank.

Usage

```
nargs()
```

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`args`, `formals` and `sys.call`.

Examples

```
tst <- function(a, b = 3, ...) {nargs()}
tst() # 0
tst(clicketyclack) # 1 (even non-existing)
tst(c1, a2, rr3) # 3

foo <- function(x, y, z, w) {
   cat("call was", deparse(match.call()), "\n")
   nargs()
}
foo()    # 0
foo(,,3) # 3
foo(z=3) # 1, even though this is the same call

nargs()# not really meaningful
```

---

nchar *Count the Number of Characters (Bytes)*

Description

`nchar` takes a character vector as an argument and returns a vector whose elements contain the sizes of the corresponding elements of `x`.

Usage

```
nchar(x, type = "bytes")
```

Arguments

`x`	character vector, or a vector to be coerced to a character vector.
`type`	character string: partial matching to one of `c("bytes", "chars", "width")`. See Details.

Details

The 'size' of a character string can be measured in one of three ways

`bytes` The number of bytes needed to store the string (plus in C a final terminator which is not counted).

`chars` The number of human-readable characters.

`width` The number of columns `cat` will use to print the string in a monospaced font. The same as `chars` if this cannot be calculated.

These will often be the same, and almost always will be in single-byte locales. There will be differences between the first two with multibyte character sequences, e.g. in UTF-8 locales. If the byte stream contains embedded `nul` bytes, `type = "bytes"` looks at all the bytes whereas the other two types look only at the string as printed by `cat`, up to the first `nul` byte.

The internal equivalent of the default method of `as.character` is performed on `x` (so there is no method dispatch). If you want to operate on non-vector objects passing them through `deparse` first will be required.

Value

An integer vector giving the sizes of each element, currently always 2 for missing values (for NA).

If an element is invalid in a multi-byte character set such as UTF-8, its number of characters and the width will be NA. Otherwise the number of characters will be non-negative, so `!is.na(nchar(x, "chars"))` is a test of validity.

Names, dims and dimnames are copied from the input.

Note

This does **not** by default give the number of characters that will be used to `print()` the string. Use `encodeString` to find the characters used to print the string.

Embedded `nul` bytes are included in the byte count (but not the final `nul`).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`strwidth` giving width of strings for plotting; `paste`, `substr`, `strsplit`

Examples

```
x <- c("asfef","qwerty","yuiop[","b","stuff.blah.yech")
nchar(x)
# 5  6  6  1 15

nchar(deparse(mean))
# 18 17
```

---

nlevels *The Number of Levels of a Factor*

Description

Return the number of levels which its argument has.

Usage

```
nlevels(x)
```

Arguments

x	an object, usually a factor.

Details

If the argument is not a `factor`, NA is returned.

The actual factor levels (if they exist) can be obtained with the `levels` function.

Examples

```
nlevels(gl(3,7)) # = 3
```

`noquote` *Class for "no quote" Printing of Character Strings*

Description

Print character strings without quotes.

Usage

```
noquote(obj)

## S3 method for class 'noquote':
print(x, ...)

## S3 method for class 'noquote':
c(..., recursive = FALSE)
```

Arguments

`obj`	any R object, typically a vector of `character` strings.
`x`	an object of class `"noquote"`.
`...`	further options passed to next methods, such as `print`.
`recursive`	for compatibility with the generic `c` function.

Details

`noquote` returns its argument as an object of class `"noquote"`. There is a method for `c()` and subscript method (`"[.noquote"`) which ensures that the class is not lost by subsetting. The print method (`print.noquote`) prints character strings *without* quotes (`"..."`).

These functions exist both as utilities and as an example of using (S3) `class` and object orientation.

Author(s)

Martin Maechler ⟨maechler@stat.math.ethz.ch⟩

See Also

`methods`, `class`, `print`.

Examples

```
letters
nql <- noquote(letters)
nql
nql[1:4] <- "oh"
nql[1:12]

cmp.logical <- function(log.v)
```

```
{
  ## Purpose: compact printing of logicals
  log.v <- as.logical(log.v)
  noquote(if(length(log.v)==0)"()" else c(".","|")[1+log.v])
}
cmp.logical(runif(20) > 0.8)
```

NotYet *Not Yet Implemented Functions and Unused Arguments*

Description

In order to pinpoint missing functionality, the R core team uses these functions for missing R functions and not yet used arguments of existing R functions (which are typically there for compatibility purposes).

You are very welcome to contribute your code ...

Usage

```
.NotYetImplemented()
.NotYetUsed(arg, error = TRUE)
```

Arguments

arg	an argument of a function that is not yet used.
error	a logical. If TRUE, an error is signalled; if FALSE; only a warning is given.

See Also

the contrary, Deprecated and Defunct for outdated code.

Examples

```
require(graphics)
require(stats)
plot.mlm         # to see how the "NotYetImplemented"
                 # reference is made automagically
try(plot.mlm())

barplot(1:5, inside = TRUE) # 'inside' is not yet used
```

nrow *The Number of Rows/Columns of an Array*

Description

nrow and ncol return the number of rows or columns present in x. NCOL and NROW do the same treating a vector as 1-column matrix.

Usage

```
nrow(x)
ncol(x)
NCOL(x)
NROW(x)
```

Arguments

x	a vector, array or data frame

Value

an integer of length 1 or NULL.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole (ncol and nrow.)

See Also

dim which returns *all* dimensions; array, matrix.

Examples

```
ma <- matrix(1:12, 3, 4)
nrow(ma)   # 3
ncol(ma)   # 4

ncol(array(1:24, dim = 2:4)) # 3, the second dimension
NCOL(1:12) # 1
NROW(1:12) # 12
```

ns-dblcolon *Double Colon and Triple Colon Operators*

Description

Accessing exported and internal variables in a name space.

Usage

```
pkg::name
pkg:::name
```

Arguments

pkg	package name symbol or literal character string.
name	variable name symbol or literal character string.

Details

The expression `pkg::name` returns the value of the exported variable `name` in package `pkg` if the package has a name space. The expression `pkg:::name` returns the value of the internal variable `name` in package `pkg` if the package has a name space. The package will be loaded if it was not loaded already before the call. Assignment into name spaces is not supported.

If the package `pkg` does not have a name space but is on the search path then `pkg::name` returns the value of `name` in the package environent.

See Also

`get` to access an object masked by another of the same name.

Examples

```
base::log
base::"+"
```

ns-hooks *Hooks for Name Space events*

Description

Packages with name spaces can supply functions to be called when loaded, attached or unloaded.

Usage

```
.onLoad(libname, pkgname)
.onAttach(libname, pkgname)
.onUnload(libpath)
```

Arguments

libname	a character string giving the library directory where the package defining the namespace was found.
pkgname	a character string giving the name of the package, including the version number if the package was installed with --with-package-versions.
libpath	a character string giving the complete path to the package.

Details

These functions apply only to packages with name spaces.

After loading, loadNamespace looks for a hook function named .onLoad and runs it before sealing the namespace and processing exports.

If a name space is unloaded (via unloadNamespace), a hook function .onUnload is run before final unloading.

Note that the code in .onLoad and .onUnload is run without the package being on the search path, but (unless circumvented) lexical scope will make objects in the namespace and its imports visible. (Do not use the double colon operator in .onLoad as exports have not been processed at the point it is run.)

When the package is attached (via library), the hook function .onAttach is looked for and if found is called after the exported functions are attached and before the package environment is sealed. This is less likely to be useful than .onLoad, which should be seen as the analogue of .First.lib (which is only used for packages without a name space).

.onLoad, .onUnload and .onAttach are looked for as internal variables in the name space and should not be exported.

If a function .Last.lib is visible in the package, it will be called when the package is detached: this does need to be exported.

Anything needed for the functioning of the name space should be handled at load/unload times by the .onLoad and .onUnload hooks. For example, shared libraries can be loaded (unless done by a useDynLib directive in the 'NAMESPACE' file) and initialized in .onLoad and unloaded in .onUnload. Use .onAttach only for actions that are needed only when the package becomes visible to the user, for example a start-up message.

If a package was installed with --with-package-versions, the pkgname supplied will be something like tree_1.0-16.

See Also

setHook shows how users can set hooks on the same events.

---

ns-load *Loading and Unloading Name Spaces*

Description

Functions to load and unload namespaces.

Usage

```
attachNamespace(ns, pos = 2, dataPath = NULL)
loadNamespace(package, lib.loc = NULL,
              keep.source = getOption("keep.source.pkgs"),
              partial = FALSE, declarativeOnly = FALSE)
loadedNamespaces()
unloadNamespace(ns)
```

Arguments

`ns`	string or namespace object.
`pos`	integer specifying position to attach.
`dataPath`	path to directory containing a database of datasets to be lazy-loaded into the attached environment.
`package`	string naming the package/name space to load.
`lib.loc`	character vector specifying library search path.
`keep.source`	logical specifying whether to retain source. This applies only to the specified name space, and not to other name spaces which might be loaded to satisfy imports. For more details see this argument to `library`.
`partial`	logical; if true, stop just after loading code.
`declarativeOnly`	logical; disables `.Import`, etc, if true.

Details

The functions `loadNamespace` and `attachNamespace` are usually called implicitly when `library` is used to load a name space and any imports needed. However it may be useful to call these functions directly at times.

`loadNamespace` loads the specified name space and registers it in an internal data base. A request to load a name space that is already loaded has no effect. The arguments have the same meaning as the corresponding arguments to `library`, whose help page explains the details of how a particular installed package comes to be chosen. After loading, `loadNamespace` looks for a hook function named `.onLoad` as an internal variable in the name space (it should not be exported). This function is called with the same arguments as `.First.lib`. Partial loading is used to support installation with the '`--save`' and '`--lazy`' options.

`loadNamespace` does not attach the name space it loads to the search path. `attachNamespace` can be used to attach a frame containing the exported values of a name space to the search path. The hook function `.onAttach` is run after the name space exports are attached.

`loadedNamespaces` returns a character vector of the names of the loaded name spaces.

`unloadNamespace` can be used to force a name space to be unloaded. An error is signaled if the name space is imported by other loaded name spaces. If defined, a hook function `.onUnload` is run before removing the name space from the internal registry. `unloadNamespace` will first `detach` a package of the same name if one is on the path, thereby running a `.Last.lib` function in the package if one is exported.

Author(s)

Luke Tierney

ns-topenv *Top Level Environment*

Description

Finding the top level environment.

Usage

```
topenv(envir = parent.frame(),
       matchThisEnv = getOption("topLevelEnvironment"))
```

Arguments

envir	environment.
matchThisEnv	return this environment, if it matches before any other criterion is satisfied. The default, the option "topLevelEnvironment", is set by sys.source, which treats a specific environment as the top level environment. Supplying the argument as NULL means it will never match.

Details

topenv returns the first top level environment found when searching envir and its parent environments. An environment is considered top level if it is the internal environment of a name space, a package environment in the search path, or .GlobalEnv.

Examples

```
topenv(.GlobalEnv)
topenv(new.env())
```

NULL *The Null Object*

Description

NULL represents the null object in R. NULL is used mainly to represent the lists with zero length, and is often returned by expressions and functions whose value is undefined.

as.null ignores its argument and returns the value NULL.

is.null returns TRUE if its argument is NULL and FALSE otherwise.

Usage

```
NULL
as.null(x, ...)
is.null(x)
```

Arguments

x	an object to be tested or coerced.
...	ignored.

Details

is.null is generic: you can write methods to handle specific classes of objects, see InternalMethods.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
is.null(list())    # FALSE (on purpose!)
is.null(integer(0))# F
is.null(logical(0))# F
as.null(list(a=1,b='c'))
```

---

numeric *Numeric Vectors*

Description

Creates or tests for objects of type "numeric".

Usage

```
numeric(length = 0)
as.numeric(x, ...)
is.numeric(x)
```

Arguments

length	desired length.
x	object to be coerced or tested.
...	further arguments passed to or from other methods.

Details

as.numeric is a generic function, but methods must be written for as.double, which it calls.

is.numeric is generic: you can write methods to handle specific classes of objects, see InternalMethods.
Note that is.numeric() returns FALSE for a factor via its "factor" method.

Value

`numeric` creates a real vector of the specified length. The elements of the vector are all equal to `0`.

`as.numeric` attempts to coerce its argument to a numeric type (either `"integer"` or `"double"`). `as.numeric` for factors yields the codes underlying the factor levels, not the numeric representation of the labels, see also `factor`.

`is.numeric` returns `TRUE` if its argument is of mode `"numeric"` (type `"double"` or type `"integer"`) and not a factor, and `FALSE` otherwise.

Note

R *has no single precision data type. All real numbers are stored in double precision format.*

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
as.numeric(c("-.1"," 2.7 ","B")) # (-0.1, 2.7, NA)  +  warning
as.numeric(factor(5:10))
```

NumericConstants *Numeric Constants*

Description

How R parses numeric constants.

Details

R parses numeric constants in its input in a very similar way to C floating-point constants.

`Inf` and `NaN` are numeric constants. All other numeric constants start with a digit or period.

Hexadecimal constants start with `0x` or `0X` followed by a non-empty sequence from `0-9 a-f A-F` which is interpreted as a hexadecimal number.

Decimal constants consists of a nonempty sequence of digits possibly containing a period (the decimal point), optionally followed by a decimal exponent. A decimal exponent consists of an `E` or `e` followed by an optional plus or minus sign followed by a non-empty sequence of digits, and indicates multiplication by a power of ten.

A numeric constant immediately followed by `i` is regarded as an imaginary complex number.

Only the ASCII digits 0–9 are recognized as digits, even in languages which have other representations of digits. The 'decimal separator' is always a period and never a comma.

Note that a leading plus or minus is not part of numeric constant but a unary operator applied to the constant.

See Also

`Syntax`.

`Quotes` for the parsing of character constants,

`octmode` *Display Numbers in Octal*

Description

Convert or print integers in octal format, with as many digits as are needed to display the largest, using leading zeroes as necessary.

Usage

```
## S3 method for class 'octmode':
as.character(x, ...)

## S3 method for class 'octmode':
format(x, ...)

## S3 method for class 'octmode':
print(x, ...)
```

Arguments

`x`	An object inheriting from class `"octmode"`.
`...`	further arguments passed to or from other methods.

Details

Class `"octmode"` consists of integer vectors with that class attribute, used merely to ensure that they are printed in octal notation, specifically for Unix-like file permissions such as `755`. Subsetting (`[`) works too.

See Also

These are auxiliary functions for `file.info`.

`hexmode`

Examples

```
(on <- structure(c(16,32, 127:129),  class = "octmode")) #-> print.*()
##-> "020" "040" "177" "200" "201"
unclass(on[3:4]) # subsetting
```

`on.exit` *Function Exit Code*

Description

`on.exit` records the expression given as its argument as needing to be executed when the current function exits (either naturally or as the result of an error). This is useful for resetting graphical parameters or performing other cleanup actions.

If no expression is provided, i.e., the call is `on.exit()`, then the current `on.exit` code is removed.

`on.exit` is a primitive function so positional matching is used and names of supplied arguments are ignored.

Usage

```
on.exit(expr, add = FALSE)
```

Arguments

`expr`	an expression to be executed.
`add`	if TRUE, add `expr` to be executed after any previously set expressions; otherwise (the default) `expr` will overwrite any previously set expressions.

Value

Invisible `NULL`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`sys.on.exit` which returns the expression stored for use by `on.exit()` in the function in which `sys.on.exit()` is evaluated.

Examples

```
opar <- par(mai = c(1,1,1,1))
on.exit(par(opar))
```

Ops.Date *Operators on the Date Class*

Description

Operators for the "Date" class.

There is an Ops method and specific methods for + and − for the Date class.

Usage

```
date + x
date - x
date1 lop date2
```

Arguments

date	date objects
date1, date2	date objects or character vectors. (Character vectors are converted by as.Date.
x	a numeric vector (in days) *or* an object of class "difftime".
lop	One of ==, !=, <, <=, > or >=.

Examples

```
(z <- Sys.Date())
z + 10
z < c("2006-06-01", "2007-01-01", "2010-01-01")
```

options *Options Settings*

Description

Allow the user to set and examine a variety of global "options" which affect the way in which R computes and displays its results.

Usage

```
options(...)

getOption(x)

.Options
```

Arguments

...	any options can be defined, using name = value or by passing a list of such tagged values. However, only the ones below are used in "base R". Further, options('name') == options()['name'], see the example.
x	a character string holding an option name.

Details

Invoking `options()` with no arguments returns a list with the current values of the options. Note that not all options listed below are set initially. To access the value of a single option, one should use `getOption("width")`, e.g., rather than `options("width")` which is a *list* of length one.

`.Options` also always contains the `options()` list (as a pairlist, unsorted), for S compatibility. Assigning to it will make a local copy and not change the original.

Value

For `getOption`, the current value set for option `x`, or `NULL` if the option is unset.

For `options()`, a list of all set options sorted by name. For `options(name)`, a list of length one containing the set value, or `NULL` if it is unset. For uses setting one or more options, a list with the previous values of the options changed (returned invisibly).

Options used in base R

`add.smooth`: typically logical, defaulting to `TRUE`. Could also be set to an integer for specifying how many (simulated) smooths should be added. This is currently only used by `plot.lm`.

`check.bounds`: logical, defaulting to `FALSE`. If true, a warning is produced whenever a "generalized vector" (atomic or `list`) is extended, by something like `x <- 1:3; x[5] <- 6`.

`continue`: a non-empty string setting the prompt used for lines which continue over one line.

`defaultPackages`: the packages that are attached by default when R starts up. Initially set from value of the environment variable `R_DEFAULT_PACKAGES`, or if that is unset to `c("datasets", "utils", "grDevices", "graphics", "stats", "methods")`. (Set `R_DEFAULT_PACKAGES` to `NULL` or a comma-separated list of package names.) A call to `options` should be in your '.Rprofile' file to ensure that the change takes effect before the base package is initialized (see `Startup`).

`deparse.max.lines`: controls the number of lines used when deparsing in `traceback` and `browser`. Initially unset, and only used if set to a positive integer.

`digits`: controls the number of digits to print when printing numeric values. It is a suggestion only. Valid values are 1...22 with default 7. See `print.default`.

`digits.secs`: controls the maximum number of digits to print when formatting time values in seconds. Valid values are 0...6 with default 0. See `strftime`.

`download.file.method`: Method to be used for `download.file`. Currently download methods `"internal"`, `"wget"` and `"lynx"` are available. There is no default for this option, when `method = "auto"` is chosen: see `download.file`.

`echo`: logical. Only used in non-interactive mode, when it controls whether input is echoed. Command-line option `-slave` sets this initially to `FALSE`.

`encoding`: The name of an encoding, default `"native.enc"`). See `connections`.

`error`: either a function or an expression governing the handling of non-catastrophic errors such as those generated by `stop` as well as by signals and internally detected errors. If the option is a function, a call to that function, with no arguments, is generated as the expression. The default value is `NULL`: see `stop` for the behaviour in that case. The functions `dump.frames` and `recover` provide alternatives that allow post-mortem debugging.

`expressions`: sets a limit on the number of nested expressions that will be evaluated. Valid values are 25...500000 with default 5000. If you increase it, you may also want to start R

with a larger protection stack; see `--max-ppsize` in `Memory`. Note too that you may cause a segfault from overflow of the C stack, and on OSes where it is possible you may want to increase that.

`keep.source`: When `TRUE`, the source code for functions (newly defined or loaded) is stored in their `"source"` attribute (see `attr`) allowing comments to be kept in the right places.

The default is `interactive()`, i.e., `TRUE` for interactive use.

`keep.source.pkgs`: As for `keep.source`, for functions in packages loaded by `library` or `require`. Defaults to `FALSE` unless the environment variable `R_KEEP_PKG_SOURCE` is set to `yes`.

Note this does not apply to packages using lazy-loading or saved images. Whether they have kept source is determined when they are installed (and is almost certainly false).

`mailer`: default mailer used by `bug.report()`. Can be `"none"`.

`max.contour.segments`: positive integer, defaulting to `250000` and usually not set. A limit on the number of segments in a single contour line in `contour` or `contourLines`.

`max.print`: integer, defaulting to `99999`. `print` or `show` methods can make use of this option, to limit the amount of information that is printed, to something in the order of (and typically slightly less than) `max.print` *entries*.

`OutDec`: one-character string. The character to be used as the decimal point in output conversions, that is in printing, plotting and `as.character` but not deparsing.

`pager`: the (stand-alone) program used for displaying text files on R's console, also used by `file.show` and sometimes `help`. Defaults to '$R_HOME/bin/pager'.

`papersize`: the default paper format used by `postscript`; set by environment variable `R_PAPERSIZE` when R is started: if that is unset or invalid it defaults to a value derived from the locale category `LC_PAPER`, or if that is unavailable to a default set when R was built.

`printcmd`: the command used by `postscript` for printing; set by environment variable `R_PRINTCMD` when R is started. This should be a command that expects either input to be piped to 'stdin' or to be given a single filename argument.

`prompt`: a non-empty string to be used for R's prompt; should usually end in a blank (`" "`).

`save.defaults`, `save.image.defaults`: see `save`.

`scipen`: integer. A penalty to be applied when deciding to print numeric values in fixed or exponential notation. Positive values bias towards fixed and negative towards scientific notation: fixed notation will be preferred unless it is more than `scipen` digits wider.

`show.error.messages`: a logical. Should error messages be printed? Intended for use with `try` or a user-installed error handler.

`stringsAsFactors`: The default setting for arguments of `data.frame` and `read.table`.

`texi2dvi`: used by the unexported function `texi2dvi` in namespace **tools**.

`timeout`: integer. The timeout for some Internet operations, in seconds. Default 60 seconds. See `download.file` and `connections`.

`topLevelEnvironment`: see `topenv` and `sys.source`.

`verbose`: logical. Should R report extra information on progress? Set to `TRUE` by the command-line option '`--verbose`'.

`warn`: sets the handling of warning messages. If `warn` is negative all warnings are ignored. If `warn` is zero (the default) warnings are stored until the top–level function returns. If fewer than 10 warnings were signalled they will be printed otherwise a message saying how many (max 50) were signalled. An object called `last.warning` is created and can be printed through the function `warnings`. If `warn` is one, warnings are printed as they occur. If `warn` is two or larger all warnings are turned into errors.

warning.expression: an R code expression to be called if a warning is generated, replacing the standard message. If non-null it is called irrespective of the value of option `warn`.

warnings.length: sets the truncation limit for error and warning messages. A non-negative integer, with allowed values 100...8192, default 1000.

width: controls the number of characters on a line. You may want to change this if you re-size the window that R is running in. Valid values are 10...10000 with default normally 80. (The limits on valid values are in file 'Print.h' and can be changed by re-compiling R.)

The 'factory-fresh' default settings of some of these options are

```
add.smooth            TRUE
check.bounds          FALSE
continue              "+ "
digits                7
echo                  TRUE
encoding              "native.enc"
error                 NULL
expressions           5000
keep.source           interactive()
keep.source.pkgs      FALSE
max.print             99999
OutDec                "."
prompt                "> "
scipen                0
show.error.messages   TRUE
timeout               60
verbose               FALSE
warn                  0
warnings.length       1000
width                 80
```

Others are set from environment variables or are platform-dependent.

Options set in package grDevices

These will be set when package **grDevices** (or its name space) is loaded if not already set.

device: a character string giving the default device for that session. This defaults to the normal screen device (e.g., `x11`, `windows` or `quartz`) for an interactive session, and `postscript` in batch use or if a screen is not available.

locatorBell: logical. Should selection in `locator` and `identify` be confirmed by a bell. Default `TRUE`. Honoured at least on `X11` and `windows` devices.

par.ask.default: logical. The default for `par("ask")` when a device is opened.

X11colortype: The default colour type for `X11` devices. Default `"true"`.

X11fonts: character vector of length 2. See `X11`.

gamma: double. The default value of `gamma` for `X11` devices, defaulting to 1 if unset (the default).

Options set in package stats

These will be set when package **stats** (or its name space) is loaded if not already set.

contrasts: the default `contrasts` used in model fitting such as with `aov` or `lm`. A character vector of length two, the first giving the function to be used with unordered factors and the second the function to be used with ordered factors. By default the elements are named `c("unordered", "ordered")`, but the names are unused.

na.action: the name of a function for treating missing values (`NA`'s) for certain situations.

show.coef.Pvalues: logical, affecting whether P values are printed in summary tables of coefficients. See `printCoefmat`.

show.signif.stars: logical, should stars be printed on summary tables of coefficients? See `printCoefmat`.

ts.eps: the relative tolerance for certain time series (`ts`) computations. Default `1e-05`.

ts.S.compat: logical. Used to select S compatibility for plotting time-series spectra. See the description of argument `log` in `plot.spec`.

Options set in package utils

These will be set when package **utils** (or its name space) is loaded if not already set.

browser: default HTML browser used by `help.start()` on UNIX, or a non-default browser on Windows.

de.cellwidth: integer: the cell widths (number of characters) to be used in the data editor `dataentry`. If this is unset (the default), 0, negative or `NA`, variable cell widths are used.

editor: a non-empty string. Sets the default text editor, e.g., for `edit`. Set from the environment variable `VISUAL` on UNIX.

example.ask: default for the `ask` argument of `example`.

help.try.all.packages: default for an argument of `help`.

HTTPUserAgent: string used as the user agent in HTTP requests. If `NULL`, HTTP requests will be made without a user agent header. The default is `R (<version> <platform> <arch> <os>)`

internet.info: The minimum level of information to be printed on URL downloads etc. Default is 2, for failure causes. Set to 1 or 0 to get more information.

menu.graphics: Logical: should graphical menus be used if available?. Defaults to `TRUE`. Currently applies to `chooseCRANmirror`, `setRepositories` and to select from multiple help files in `help`.

pkgType: The default type of packages to be downloaded and installed – see `install.packages`. Possible values are `"source"` (the default except under the CRAN Mac OS X build) and `"mac.binary"`.

repos: URLs of the repositories for use by `update.packages`. Defaults to `c(CRAN="@CRAN@")`, a value that causes some utilities to prompt for a CRAN mirror. To avoid this do set the CRAN mirror, by something like `local({r <- getOption("repos"); r["CRAN"] <- "http://my.local.cran"; options(repos=r)})`.

Note that you can add more repositories (Bioconductor and Omegahat, notably) using `setRepositories()`.

SweaveHooks, SweaveSyntax: see `Sweave`.

unzip: the command used for unzipping help files. Defaults to the value of `R_UNZIPCMD`, which is set in 'etc/Renviron' if an `unzip` command was found during configuration.

Options used on Unix only

`latexcmd, dvipscmd`: character strings giving commands to be used in off-line printing of help pages.

`pdfviewer`: default PDF viewer. Set from the environment variable `R_PDFVIEWER`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
options() # printing all current options
op <- options(); str(op) # nicer printing

options('width')[[1]] == options()$width # the latter needs more memory
options(digits = 20)
pi

# set the editor, and save previous value
old.o <- options(editor = "nedit")
old.o

options(check.bounds = TRUE, warn = 1)
x <- NULL; x[4] <- "yes" # gives a warning

options(digits=5)
print(1e5)
options(scipen=3); print(1e5)

options(op)     # reset (all) initial options
options('digits')

## Not run:
## set contrast handling to be like S
options(contrasts = c("contr.helmert", "contr.poly"))
## End(Not run)
## Not run:
## on error, terminate the R session with error status 66
options(error = quote(q("no", status=66, runLast=FALSE)))
stop("test it")
## End(Not run)
## Not run:
## Set error actions for debugging:
## enter browser on error, see ?recover:
options(error = recover)
## allows to call debugger() afterwards, see ?debugger:
options(error = dump.frames)
## A possible setting for non-interactive sessions
options(error = quote({dump.frames(to.file=TRUE); q()}))
## End(Not run)
```

`order` *Ordering Permutation*

Description

`order` returns a permutation which rearranges its first argument into ascending or descending order, breaking ties by further arguments. `sort.list` is the same, using only one argument. See the examples for how to use these functions to sort data frames, etc.

Usage

```
order(..., na.last = TRUE, decreasing = FALSE)

sort.list(x, partial = NULL, na.last = TRUE, decreasing = FALSE,
          method = c("shell", "quick", "radix"))
```

Arguments

`...`	a sequence of numeric, complex, character or logical vectors, all of the same length.
`x`	a vector.
`partial`	vector of indices for partial sorting. (Non-`NULL` values are not implemented.)
`decreasing`	logical. Should the sort order be increasing or decreasing?
`na.last`	for controlling the treatment of `NA`s. If `TRUE`, missing values in the data are put last; if `FALSE`, they are put first; if `NA`, they are removed.
`method`	the method to be used: partial matches are allowed.

Details

In the case of ties in the first vector, values in the second are used to break the ties. If the values are still tied, values in the later arguments are used to break the tie (see the first example). The sort used is *stable* (except for `method = "quick"`), so any unresolved ties will be left in their original ordering.

Complex values are sorted first by the real part, then the imaginary part.

The sort order for character vectors will depend on the collating sequence of the locale in use: see `Comparison`.

The default method for `sort.list` is a good compromise. Method `"quick"` is only supported for numeric `x` with `na.last=NA`, and is not stable, but will be faster for long vectors. Method `"radix"` is only implemented for integer `x` with a range of less than 100,000. For such `x` it is very fast (and stable), and hence is ideal for sorting factors.

`partial` is supplied for compatibility with other implementations of S, but no other values are accepted and ordering is always complete.

Note that these functions are only defined for vectors, so any class of the object supplied is ignored: this means factors are sorted on their internal codes and not their printed representation.

Note

`sort.list` can get called by mistake as a method for `sort` with a list argument, and gives a suitable error message for list `x`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`sort` and `rank`.

Examples

```
(ii <- order(x <- c(1,1,3:1,1:4,3), y <- c(9,9:1), z <-c(2,1:9)))
## 6  5  2  1  7  4 10  8  3  9
rbind(x,y,z)[,ii] # shows the reordering (ties via 2nd & 3rd arg)

## Suppose we wanted descending order on y. A simple solution is
rbind(x,y,z)[, order(x, -y, z)]
## For character vectors we can make use of rank:
cy <- as.character(y)
rbind(x,y,z)[, order(x, -rank(cy), z)]

## Sorting data frames:
dd <- transform(data.frame(x,y,z),
                z = factor(z, labels=LETTERS[9:1]))
## Either as above {for factor 'z' : using internal coding}:
dd[ order(x, -y, z) ,]
## or along 1st column, ties along 2nd, ... *arbitrary* no.{columns}:
dd[ do.call(order, dd) ,]

set.seed(1)# reproducible example:
d4 <- data.frame(x = round(   rnorm(100)), y = round(10*runif(100)),
                 z = round( 8*rnorm(100)), u = round(50*runif(100)))
(d4s <- d4[ do.call(order, d4) ,])
(i <- which(diff(d4s[,3]) == 0))# in 2 places, needed 3 cols to break ties:
d4s[ rbind(i,i+1), ]

## rearrange matched vectors so that the first is in ascending order
x <- c(5:1, 6:8, 12:9)
y <- (x - 5)^2
o <- order(x)
rbind(x[o], y[o])

## tests of na.last
a <- c(4, 3, 2, NA, 1)
b <- c(4, NA, 2, 7, 1)
z <- cbind(a, b)
(o <- order(a, b)); z[o, ]
(o <- order(a, b, na.last = FALSE)); z[o, ]
(o <- order(a, b, na.last = NA)); z[o, ]

## Not run:
##  speed examples for long vectors:
x <- factor(sample(letters, 1e6, replace=TRUE))
system.time(o <- sort.list(x)) ## 1.2 secs
stopifnot(!is.unsorted(x[o]))
system.time(o <- sort.list(x, method="quick", na.last=NA)) # 0.15 sec
stopifnot(!is.unsorted(x[o]))
```

```
system.time(o <- sort.list(x, method="radix")) # 0.02 sec
stopifnot(!is.unsorted(x[o]))
xx <- sample(1:26, 1e7, replace=TRUE)
system.time(o <- sort.list(xx, method="radix")) # 0.2 sec
xx <- sample(1:100000, 1e7, replace=TRUE)
system.time(o <- sort.list(xx, method="radix")) # 0.8 sec
system.time(o <- sort.list(xx, method="quick", na.last=NA)) # 1.4 sec
## End(Not run)
```

`outer` *Outer Product of Arrays*

Description

The outer product of the arrays `X` and `Y` is the array `A` with dimension `c(dim(X), dim(Y))` where element `A[c(arrayindex.x, arrayindex.y)] = FUN(X[arrayindex.x], Y[arrayindex.y], ...)`.

Usage

```
outer(X, Y, FUN="*", ...)
X %o% Y
```

Arguments

`X, Y`	First and second arguments for function `FUN`. Typically a vector or array.
`FUN`	a function to use on the outer products, found *via* `match.fun` (except for the special case `"*"`).
`...`	optional arguments to be passed to `FUN`.

Details

`FUN` must be a function (or the name of it) which expects at least two arguments and which operates elementwise.

`X` and `Y` must be suitable arguments for `FUN`. Each will be extended by `rep` to length the products of the lengths of `X` and `Y` before `FUN` is called.

Where they exist, the [dim]names of `X` and `Y` will be copied to the answer, and a dimension assigned which is the concatenation of the dimensions of `X` and `Y` (or lengths if dimensions do not exist).

`FUN = "*"` is handled internally as a special case, *via* `as.vector(X) %*% t(as.vector(Y))`, and is intended only for numeric vectors and arrays.

`%o%` is binary operator providing a wrapper for `outer(x, y, "*")`.

Author(s)

Jonathan Rougier

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`%*%` for usual (*inner*) matrix vector multiplication; `kronecker` which is based on `outer`.

Examples

```
x <- 1:9; names(x) <- x
# Multiplication & Power Tables
x %o% x
y <- 2:8; names(y) <- paste(y,":",sep="")
outer(y, x, "^")

outer(month.abb, 1999:2003, FUN = "paste")

## three way multiplication table:
x %o% x %o% y[1:3]
```

---

package-version *Package versions*

Description

A simple S3 class for representing package versions, and associated methods.

Usage

```
package_version(x, strict = TRUE)

getRversion()
```

Arguments

`x`	a character vector with package version strings, or an R version object as obtained by `R.version`.
`strict`	a logical indicating whether invalid package versions should results in an error (default) or not.

Details

R (package) versions are sequences of at least two non-negative integers, usually (e.g., in package 'DESCRIPTION' files) represented as character strings with the elements of the sequence concatenated and separated by single '`.`' or '`-`' characters.

`package_version` creates a representation from such strings which allows for coercion and testing, combination, comparison, summaries (min/max), inclusion in data frames, subscripting, and printing.

`getRversion` returns the version of the running R as an object of class `"package_version"`.

See Also

`compareVersion`

Examples

```
x <- package_version(c("1.2-4", "1.2-3", "2.1"))
x < "1.4-2.3"
c(min(x), max(x))
x[2, 2]
x$major
x$minor
```

`Paren` *Parentheses and Braces*

Description

Open parenthesis, `(`, and open brace, `{`, are `.Primitive` functions in R.

Effectively, `(` is semantically equivalent to the identity `function(x) x`, whereas `{` is slightly more interesting, see examples.

Usage

```
( ... )

{ ... }
```

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`if`, `return`, etc for other objects used in the R language itself.

`Syntax` for operator precedence.

Examples

```
f <- get("(")
e <- expression(3 + 2 * 4)
identical(f(e), e)

do <- get("{")
do(x <- 3, y <- 2*x-3, 6-x-y); x; y
```

parse *Parse Expressions*

Description

parse returns the parsed but unevaluated expressions in a list.

Usage

```
parse(file = "", n = NULL, text = NULL, prompt = "?")
```

Arguments

file	a connection, or a character string giving the name of a file or a URL to read the expressions from. If file is "" and text is missing or NULL then input is taken from the console.
n	integer (or coerced to integer). The maximum number of expressions to parse. If n is NULL or negative or NA the input is parsed in its entirety.
text	character vector. The text to parse. Elements are treated as if they were lines of a file. Other R objects will be coerced to character (without method dispatch) if possible.
prompt	the prompt to print when parsing from the keyboard. NULL means to use R's prompt, getOption("prompt").

Details

If text has length greater than zero (after coercion) it is used in preference to file.

All versions of R accept input from a connection with end of line marked by LF (as used on Unix), CRLF (as used on DOS/Windows) or CR (as used on classic MacOS). The final line can be incomplete, that is missing the final EOL marker.

See source for the limits on the size of functions that can be parsed (by default). There is also a limit of 8192 bytes on the size of strings which can be parsed.

When input is taken from the console, n = NULL is equivalent to n = 1, and n < 0 will read until an EOF character is read.

Value

An object of type "expression", with up to n elements if specified as a non-negative integer.

A syntax error (including an incomplete expression) will throw an error.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

scan, source, eval, deparse.

Examples

```
cat("x <- c(1,4)\n  x ^ 3 -10 ; outer(1:7,5:9)\n", file="xyz.Rdmped")
# parse 3 statements from the file "xyz.Rdmped"
parse(file = "xyz.Rdmped", n = 3)
unlink("xyz.Rdmped")
```

---

paste *Concatenate Strings*

Description

Concatenate vectors after converting to character.

Usage

```
paste(..., sep = " ", collapse = NULL)
```

Arguments

`...`	one or more R objects, to be converted to character vectors.
`sep`	a character string to separate the terms.
`collapse`	an optional character string to separate the results.

Details

`paste` converts its arguments (*via* `as.character`) to character strings, and concatenates them (separating them by the string given by `sep`). If the arguments are vectors, they are concatenated term-by-term to give a character vector result. Vector arguments are recycled as needed, with zero-length arguments being recycled to `""`.

If a value is specified for `collapse`, the values in the result are then concatenated into a single string, with the elements being separated by the value of `collapse`.

Value

A character vector of the concatenated values. This will be of length zero if all the objects are, unless `collapse` is non-NULL, in which case it is a single empty string.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

String manipulation with `as.character`, `substr`, `nchar`, `strsplit`; further, `cat` which concatenates and writes to a file, and `sprintf` for C like string construction.

Examples

```
paste(1:12) # same as as.character(1:12)
paste("A", 1:6, sep = "")
paste("Today is", date())
```

path.expand *Expand File Paths*

Description

Expand a path name, for example by replacing a leading tilde by the user's home directory (if defined on that platform).

Usage

```
path.expand(path)
```

Arguments

path	character vector containing one or more path names.

Details

On *some Unix* versions, a leading ~user will expand to the home directory of user, but not on Unix versions without readline installed.

See Also

basename

Examples

```
path.expand("~/foo")
```

pmatch *Partial String Matching*

Description

pmatch seeks matches for the elements of its first argument among those of its second.

Usage

```
pmatch(x, table, nomatch = NA, duplicates.ok = FALSE)
```

Arguments

x	the values to be matched: converted to a character vector by as.character.
table	the values to be matched against: converted to a character vector.
nomatch	the value to be returned at non-matching or multiply partially matching positions.
duplicates.ok	should elements be in table be used more than once?

Details

The behaviour differs by the value of `duplicates.ok`. Consider first the case if this is true. First exact matches are considered, and the positions of the first exact matches are recorded. Then unique partial matches are considered, and if found recorded. (A partial match occurs if the whole of the element of `x` matches the beginning of the element of `table`.) Finally, all remaining elements of `x` are regarded as unmatched. In addition, an empty string can match nothing, not even an exact match to an empty string. This is the appropriate behaviour for partial matching of character indices, for example.

If `duplicates.ok` is `FALSE`, values of `table` once matched are excluded from the search for subsequent matches. This behaviour is equivalent to the R algorithm for argument matching, except for the consideration of empty strings (which in argument matching are matched after exact and partial matching to any remaining arguments).

`charmatch` is similar to `pmatch` with `duplicates.ok` true, the differences being that it differentiates between no match and an ambiguous partial match, it does match empty strings, and it does not allow multiple exact matches.

`NA` values are treated as if they were the string constant `"NA"`.

Value

A numeric vector of integers (including `NA` if `nomatch = NA`) of the same length as `x`, giving the indices of the elements in `table` which matched, or `nomatch`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer.

See Also

`match`, `charmatch` and `match.arg`, `match.fun`, `match.call`, for function argument matching etc., `grep` etc for more general (regexp) matching of strings.

Examples

```
pmatch("", "")                              # returns NA
pmatch("m",   c("mean", "median", "mode")) # returns NA
pmatch("med", c("mean", "median", "mode")) # returns 2

pmatch(c("", "ab", "ab"), c("abc", "ab"), dup=FALSE)
pmatch(c("", "ab", "ab"), c("abc", "ab"), dup=TRUE)
## compare
charmatch(c("", "ab", "ab"), c("abc", "ab"))
```

`polyroot` *Find Zeros of a Real or Complex Polynomial*

Description

Find zeros of a real or complex polynomial.

Usage

```
polyroot(z)
```

Arguments

`z`	the vector of polynomial coefficients in increasing order.

Details

A polynomial of degree $n - 1$,

$$p(x) = z_1 + z_2 x + \cdots + z_n x^{n-1}$$

is given by its coefficient vector `z[1:n]`. `polyroot` returns the $n - 1$ complex zeros of $p(x)$ using the Jenkins-Traub algorithm.

If the coefficient vector `z` has zeroes for the highest powers, these are discarded.

Value

A complex vector of length $n - 1$, where n is the position of the largest non-zero element of `z`.

References

Jenkins and Traub (1972) TOMS Algorithm 419. *Comm. ACM*, **15**, 97–99.

See Also

`uniroot` for numerical root finding of arbitrary functions; `complex` and the `zero` example in the demos directory.

Examples

```
polyroot(c(1, 2, 1))
round(polyroot(choose(8, 0:8)), 11) # guess what!
for (n1 in 1:4) print(polyroot(1:n1), digits = 4)
polyroot(c(1, 2, 1, 0, 0)) # same as the first
```

pos.to.env *Convert Positions in the Search Path to Environments*

Description

Returns the environment at a specified position in the search path.

Usage

```
pos.to.env(x)
```

Arguments

x	an integer between 1 and `length(search())`, the length of the search path.

Details

Several R functions for manipulating objects in environments (such as `get` and `ls`) allow specifying environments via corresponding positions in the search path. `pos.to.env` is a convenience function for programmers which converts these positions to corresponding environments; users will typically have no need for it.

Examples

```
pos.to.env(1) # R_GlobalEnv
# the next returns the base environment
pos.to.env(length(search()))
```

pretty *Pretty Breakpoints*

Description

Compute a sequence of about `n+1` equally spaced nice values which cover the range of the values in `x`. The values are chosen so that they are 1, 2 or 5 times a power of 10.

Usage

```
pretty(x, n = 5, min.n = n %/% 3,  shrink.sml = 0.75,
       high.u.bias = 1.5, u5.bias = .5 + 1.5*high.u.bias,
       eps.correct = 0)
```

Arguments

x	numeric vector
n	integer giving the *desired* number of intervals. Non-integer values are rounded down.
min.n	nonnegative integer giving the *minimal* number of intervals. If `min.n == 0`, `pretty(.)` may return a single value.

`shrink.sml` positive numeric by a which a default scale is shrunk in the case when `range(x)` is "very small" (usually 0).

`high.u.bias` non-negative numeric, typically > 1. The interval unit is determined as {1,2,5,10} times `b`, a power of 10. Larger `high.u.bias` values favor larger units.

`u5.bias` non-negative numeric multiplier favoring factor 5 over 2. Default and "optimal": `u5.bias = .5 + 1.5*high.u.bias`.

`eps.correct` integer code, one of {0,1,2}. If non-0, an "*epsilon correction*" is made at the boundaries such that the result boundaries will be outside `range(x)`; in the *small* case, the correction is only done if `eps.correct >=2`.

Details

`pretty` ignores non-finite values in `x`.

Let `d <- max(x) - min(x)` ≥ 0. If `d` is not (very close) to 0, we let `c <- d/n`, otherwise more or less `c <- max(abs(range(x)))*shrink.sml / min.n`. Then, the *10 base* `b` is $10^{\lfloor \log_{10}(c) \rfloor}$ such that $b \leq c < 10b$.

Now determine the basic *unit* u as one of $\{1, 2, 5, 10\}b$, depending on $c/b \in [1, 10)$ and the two "*bias*" coefficients, $h =$`high.u.bias` and $f =$`u5.bias`.

.....

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`axTicks` for the computation of pretty axis tick locations in plots, particularly on the log scale.

Examples

```
pretty(1:15)      # 0  2  4  6  8 10 12 14 16
pretty(1:15, h=2)# 0  5 10 15
pretty(1:15, n=4)# 0  5 10 15
pretty(1:15 * 2) # 0  5 10 15 20 25 30
pretty(1:20)       # 0  5 10 15 20
pretty(1:20, n=2) # 0 10 20
pretty(1:20, n=10)# 0  2  4 ... 20

for(k in 5:11) {
  cat("k=",k,": "); print(diff(range(pretty(100 + c(0, pi*10^-k)))))}

##-- more bizarre, when  min(x) == max(x):
pretty(pi)

add.names <- function(v) { names(v) <- paste(v); v}
str(lapply(add.names(-10:20), pretty))
str(lapply(add.names(0:20),   pretty, min = 0))
sapply(    add.names(0:20),   pretty, min = 4)

pretty(1.234e100)
pretty(1001.1001)
```

```
pretty(1001.1001, shrink = .2)
for(k in -7:3)
  cat("shrink=",formatC(2^k,wid=9),":",
      formatC(pretty(1001.1001, shrink = 2^k), wid=6),"\n")
```

`Primitive` *Call a "Primitive" Internal Function*

Description

`.Primitive` returns an entry point to a "primitive" (internally implemented) function.

Usage

```
.Primitive(name)
```

Arguments

`name`	name of the R function.

Details

The advantage of `.Primitive` over `.Internal` functions is the potential efficiency of argument passing. However, this is done by ignoring argument names and using positional matching of arguments (unless arranged differently for specific primitives such as `rep`), so this is discouraged for functions of more than one argument.

All primitive functions are in the base name space.

See Also

`.Internal`.

Examples

```
mysqrt <- .Primitive("sqrt")
c
.Internal # this one *must* be primitive!
get("if") # just 'if' or 'print(if)' are not syntactically ok.
```

`print` *Print Values*

Description

`print` prints its argument and returns it *invisibly* (via `invisible(x)`). It is a generic function which means that new printing methods can be easily added for new `class`es.

Usage

```
print(x, ...)

## S3 method for class 'factor':
print(x, quote = FALSE, max.levels = NULL,
      width = getOption("width"), ...)

## S3 method for class 'table':
print(x, digits = getOption("digits"), quote = FALSE,
      na.print = "", zero.print = "0", justify = "none", ...)
```

Arguments

x	an object used to select a method.
...	further arguments passed to or from other methods.
quote	logical, indicating whether or not strings should be printed with surrounding quotes.
max.levels	integer, indicating how many levels should be printed for a factor; if 0, no extra "Levels" line will be printed. The default, NULL, entails choosing max.levels such that the levels print on one line of width width.
width	only used when max.levels is NULL, see above.
digits	minimal number of *significant* digits, see print.default.
na.print	character string (or NULL) indicating NA values in printed output, see print.default.
zero.print	character specifying how zeros (0) should be printed; for sparse tables, using "." can produce stronger results.
justify	character indicating if strings should left- or right-justified or left alone, passed to format.

Details

The default method, print.default has its own help page. Use methods("print") to get all the methods for the print generic.

print.factor allows some customization and is used for printing ordered factors as well.

print.table for printing tables allows other customization.

See noquote as an example of a class whose main purpose is a specific print method.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S.* Wadsworth & Brooks/Cole.

See Also

The default method print.default, and help for the methods above; further options, noquote.

For more customizable (but cumbersome) printing, see cat, format or also write.

Examples

```
ts(1:20)#-- print is the "Default function" --> print.ts(.) is called
rr <- for(i in 1:3) print(1:i)
rr

## Printing of factors
attenu$station ## 117 levels -> `max.levels' depending on width

## ordered factors: levels  "l1 < l2 < .."
esoph$agegp[1:12]
esoph$alcgp[1:12]

## Printing of sparse (contingency) tables
set.seed(521)
t1 <- round(abs(rt(200, df=1.8)))
t2 <- round(abs(rt(200, df=1.4)))
table(t1,t2) # simple
print(table(t1,t2), zero.print = ".")# nicer to read
```

---

print.data.frame *Printing Data Frames*

Description

Print a data frame.

Usage

```
## S3 method for class 'data.frame':
print(x, ..., digits = NULL, quote = FALSE, right = TRUE)
```

Arguments

x	object of class data.frame.
...	optional arguments to print or plot methods.
digits	the minimum number of significant digits to be used: see print.default.
quote	logical, indicating whether or not entries should be printed with surrounding quotes.
right	logical, indicating whether or not strings should be right-aligned. The default is right-alignment.

Details

This calls format which formats the data frame column-by-column, then converts to a character matrix and dispatches to the print method for matrices.

When quote = TRUE only the entries are quoted not the row names nor the column names.

See Also

data.frame.

print.default *Default Printing*

Description

print.default is the *default* method of the generic print function which prints its argument.

Usage

```
## Default S3 method:
print(x, digits = NULL, quote = TRUE, na.print = NULL,
      print.gap = NULL, right = FALSE, max = NULL, ...)
```

Arguments

x	the object to be printed.
digits	a non-null value for digits specifies the minimum number of significant digits to be printed in values. The default, NULL, uses getOption(digits). (For the interpretation for complex numbers see signif.) Non-integer values will be rounded down, and only values greater than or equal to one are accepted.
quote	logical, indicating whether or not strings (characters) should be printed with surrounding quotes.
na.print	a character string which is used to indicate NA values in printed output, or NULL (see Details)
print.gap	a non-negative integer $\leq$ 1024, or NULL (meaning 1), giving the spacing between adjacent "columns" in printed vectors, matrices and arrays.
right	logical, indicating whether or not strings should be right aligned. The default is left alignment.
max	a non-null value for max specifies the approximate maximum number of entries to be printed. The default, NULL, uses getOption(max.print); see that help page for more details.
...	further arguments to be passed to or from other methods. They are ignored in this function.

Details

The default for printing NAs is to print NA (without quotes) unless this is a character NA *and* quote = FALSE, when <NA> is printed.

The same number of decimal places is used throughout a vector, This means that digits specifies the minimum number of significant digits to be used, and that at least one entry will be encoded with that minimum number. However, if all the encoded elements then have trailing zeroes, the number of decimal places is reduced until at least one element has a non-zero final digit. Decimal points are only included if at least one decimal place is selected.

Attributes are printed respecting their class(es), using the values of digits to print.default, but using the default values (for the methods called) of the other arguments.

When the **methods** package is attached, print will call show for R objects with formal classes if called with no optional arguments.

Single-byte locales

If a non-printable character is encountered during output, it is represented as one of the ANSI escape sequences (
`\a`,
`\b`,
`\f`,
`\n`,
`\r`,
`\t`,
`\v`, `\\` and
`\0`: see Quotes), or failing that as a 3-digit octal code: for example the UK currency pound sign in the C locale (if implemented correctly) is printed as
`\243`. Which characters are non-printable depends on the locale.

Unicode and other multi-byte locales

In all locales, the characters in the ASCII range (`0x00` to `0x7f`) are printed in the same way, as-is if printable, otherwise via ANSI escape sequences or 3-digit octal escapes as described for single-byte locales.

Multi-byte non-printing characters are printed as an escape sequence of the form
`\uxxxx` or
`\Uxxxxxxxx` (in hexadecimal). This is the internal code for the wide-character representation of the character. If this is not known to be the Unicode point, a warning is issued. The only known exceptions are certain Japanese ISO2022 locales on commercial Unixes, which use a concatenation of the bytes: it is unlikely that R compiles on such a system.

It is possible to have a character string in a character vector that is not valid in the current locale. If a byte is encountered that is not part of a valid character it is printed in hex in the form `<xx>` and this is repeated until the start of a valid character. (This will rapidly recover from minor errors in UTF-8.)

See Also

The generic `print`, `options`. The `"noquote"` class and print method.

`encodeString`, which encodes a character vector the way it would be printed.

Examples

```
pi
print(pi, digits = 16)
LETTERS[1:16]
print(LETTERS, quote = FALSE)

M <- cbind(I=1, matrix(1:10000, nc=10, dimnames=list(NULL, LETTERS[1:10])))
head(M) # makes more sense than
print(M, max= 1000)# prints 90 rows and a message about omitting 910
```

prmatrix *Print Matrices, Old-style*

Description

An earlier method for printing matrices, provided for S compatibility.

Usage

```
prmatrix(x, rowlab =, collab =,
         quote = TRUE, right = FALSE, na.print = NULL, ...)
```

Arguments

x	numeric or character matrix.
rowlab,collab	(optional) character vectors giving row or column names respectively. By default, these are taken from dimnames(x).
quote	logical; if TRUE and x is of mode "character", *quotes* (") are used.
right	if TRUE and x is of mode "character", the output columns are *right*-justified.
na.print	how NAs are printed. If this is non-null, its value is used to represent NA.
...	arguments for print methods.

Details

prmatrix is an earlier form of print.matrix, and is very similar to the S function of the same name.

Value

Invisibly returns its argument, x.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

print.default, and other print methods.

Examples

```
prmatrix(m6 <- diag(6), row = rep("",6), coll=rep("",6))

chm <- matrix(scan(system.file("help", "AnIndex", package = "splines"),
                   what = ""), , 2, byrow = TRUE)
chm  # uses print.matrix()
prmatrix(chm, collab = paste("Column",1:3), right=TRUE, quote=FALSE)
```

`proc.time` *Running Time of R*

Description

`proc.time` determines how much time (in seconds) the currently running R process already consumed.

Usage

```
proc.time()
```

Value

A numeric vector of length 5, containing the user, system, and total elapsed times for the currently running R process, and the cumulative sum of user and system times of any child processes spawned by it.

The resolution of the times will be system-specific; and times are rounded to the nearest 1ms. On modern systems they will be that accurate, but on older systems they might be accurate to 1/100 or 1/60 sec.

It is most useful for "timing" the evaluation of R expressions, which can be done conveniently with `system.time`.

Note

It is possible to compile R without support for `proc.time`, when the function will throw an error.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`system.time` for timing a valid R expression, `gc.time` for how much of the time was spent in garbage collection.

Examples

```
## Not run:
## a way to time an R expression: system.time is preferred
ptm <- proc.time()
for (i in 1:50) mad(runif(500))
proc.time() - ptm
## End(Not run)
```

prod *Product of Vector Elements*

Description

prod returns the product of all the values present in its arguments.

Usage

```
prod(..., na.rm = FALSE)
```

Arguments

...	numeric or complex or logical vectors.
na.rm	logical. Should missing values be removed?

Details

If na.rm is FALSE an NA value in any of the arguments will cause a value of NA to be returned, otherwise NA values are ignored.

This is a generic function: methods can be defined for it directly or via the Summary group generic. For this to work properly, the arguments ... should be unnamed, and dispatch is on the first argument.

Logical true values are regarded as one, false values as zero. For historical reasons, NULL is accepted and treated as if it were numeric(0).

Value

The product, a numeric (of type "double") or complex vector of length one. **NB:** the product of an empty set is one, by definition.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

sum, cumprod, cumsum.

Examples

```
print(prod(1:7)) == print(gamma(8))
```

prop.table *Express Table Entries as Fraction of Marginal Table*

Description

This is really `sweep(x, margin, margin.table(x, margin), "/")` for newbies, except that if `margin` has length zero, then one gets `x/sum(x)`.

Usage

```
prop.table(x, margin=NULL)
```

Arguments

`x`	table
`margin`	index, or vector of indices to generate margin for

Value

Table like `x` expressed relative to `margin`

Author(s)

Peter Dalgaard

See Also

`margin.table`

Examples

```
m <- matrix(1:4,2)
m
prop.table(m,1)
```

pushBack *Push Text Back on to a Connection*

Description

Functions to push back text lines onto a connection, and to enquire how many lines are currently pushed back.

Usage

```
pushBack(data, connection, newLine = TRUE)
pushBackLength(connection)
```

Arguments

data	a character vector.
connection	A connection.
newLine	logical. If true, a newline is appended to each string pushed back.

Details

Several character strings can be pushed back on one or more occasions. The occasions form a stack, so the first line to be retrieved will be the first string from the last call to pushBack. Lines which are pushed back are read prior to the normal input from the connection, by the normal text-reading functions such as readLines and scan.

Pushback is only allowed for readable connections.

Not all uses of connections respect pushbacks, in particular the input connection is still wired directly, so for example parsing commands from the console and scan("") ignore pushbacks on stdin.

Value

pushBack returns nothing.

pushBackLength returns number of lines currently pushed back.

See Also

connections, readLines.

Examples

```
zz <- textConnection(LETTERS)
readLines(zz, 2)
pushBack(c("aa", "bb"), zz)
pushBackLength(zz)
readLines(zz, 1)
pushBackLength(zz)
readLines(zz, 1)
readLines(zz, 1)
close(zz)
```

---

qr *The QR Decomposition of a Matrix*

Description

qr computes the QR decomposition of a matrix. It provides an interface to the techniques used in the LINPACK routine DQRDC or the LAPACK routines DGEQP3 and (for complex matrices) ZGEQP3.

Usage

```
qr(x, tol = 1e-07 , LAPACK = FALSE)
qr.coef(qr, y)
qr.qy(qr, y)
qr.qty(qr, y)
qr.resid(qr, y)
qr.fitted(qr, y, k = qr$rank)
qr.solve(a, b, tol = 1e-7)
## S3 method for class 'qr':
solve(a, b, ...)

is.qr(x)
as.qr(x)
```

Arguments

x	a matrix whose QR decomposition is to be computed.
tol	the tolerance for detecting linear dependencies in the columns of x. Only used if LAPACK is false and x is real.
qr	a QR decomposition of the type computed by qr.
y, b	a vector or matrix of right-hand sides of equations.
a	A QR decomposition or (qr.solve only) a rectangular matrix.
k	effective rank.
LAPACK	logical. For real x, if true use LAPACK otherwise use LINPACK.
...	further arguments passed to or from other methods

Details

The QR decomposition plays an important role in many statistical techniques. In particular it can be used to solve the equation $\boldsymbol{Ax} = \boldsymbol{b}$ for given matrix $\boldsymbol{A}$, and vector $\boldsymbol{b}$. It is useful for computing regression coefficients and in applying the Newton-Raphson algorithm.

The functions qr.coef, qr.resid, and qr.fitted return the coefficients, residuals and fitted values obtained when fitting y to the matrix with QR decomposition qr. qr.qy and qr.qty return Q %*% y and t(Q) %*% y, where Q is the (complete) $\boldsymbol{Q}$ matrix.

All the above functions keep dimnames (and names) of x and y if there are.

solve.qr is the method for solve for qr objects. qr.solve solves systems of equations via the QR decomposition: if a is a QR decomposition it is the same as solve.qr, but if a is a rectangular matrix the QR decomposition is computed first. Either will handle over- and under-determined systems, providing a minimal-length solution or a least-squares fit if appropriate.

is.qr returns TRUE if x is a list with components named qr, rank and qraux and FALSE otherwise.

It is not possible to coerce objects to mode "qr". Objects either are QR decompositions or they are not.

Value

The QR decomposition of the matrix as computed by LINPACK or LAPACK. The components in the returned value correspond directly to the values returned by DQRDC/DGEQP3/ZGEQP3.

qr	a matrix with the same dimensions as x. The upper triangle contains the $\boldsymbol{R}$ of the decomposition and the lower triangle contains information on the $\boldsymbol{Q}$ of the decomposition (stored in compact form). Note that the storage used by DQRDC and DGEQP3 differs.
qraux	a vector of length ncol(x) which contains additional information on $\boldsymbol{Q}$.
rank	the rank of x as computed by the decomposition: always full rank in the LAPACK case.
pivot	information on the pivoting strategy used during the decomposition.

Non-complex QR objects computed by LAPACK have the attribute "useLAPACK" with value TRUE.

Note

To compute the determinant of a matrix (do you *really* need it?), the QR decomposition is much more efficient than using Eigen values (eigen). See det.

Using LAPACK (including in the complex case) uses column pivoting and does not attempt to detect rank-deficient matrices.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Dongarra, J. J., Bunch, J. R., Moler, C. B. and Stewart, G. W. (1978) *LINPACK Users Guide.* Philadelphia: SIAM Publications.

Anderson. E. and ten others (1999) *LAPACK Users' Guide*. Third Edition. SIAM.
Available on-line at http://www.netlib.org/lapack/lug/lapack_lug.html.

See Also

qr.Q, qr.R, qr.X for reconstruction of the matrices. lm.fit, lsfit, eigen, svd.

det (using qr) to compute the determinant of a matrix.

Examples

```
hilbert <- function(n) { i <- 1:n; 1 / outer(i - 1, i, "+") }
h9 <- hilbert(9); h9
qr(h9)$rank           #--> only 7
qrh9 <- qr(h9, tol = 1e-10)
qrh9$rank             #--> 9
##-- Solve linear equation system  H %*% x = y :
y <- 1:9/10
x <- qr.solve(h9, y, tol = 1e-10) # or equivalently :
x <- qr.coef(qrh9, y) #-- is == but much better than
                      #-- solve(h9) %*% y
h9 %*% x              # = y
```

`QR.Auxiliaries` *Reconstruct the Q, R, or X Matrices from a QR Object*

Description

Returns the original matrix from which the object was constructed or the components of the decomposition.

Usage

```
qr.X(qr, complete = FALSE, ncol =)
qr.Q(qr, complete = FALSE, Dvec =)
qr.R(qr, complete = FALSE)
```

Arguments

`qr`	object representing a QR decomposition. This will typically have come from a previous call to `qr` or `lsfit`.
`complete`	logical expression of length 1. Indicates whether an arbitrary orthogonal completion of the Q or X matrices is to be made, or whether the R matrix is to be completed by binding zero-value rows beneath the square upper triangle.
`ncol`	integer in the range `1:nrow(qr$qr)`. The number of columns to be in the reconstructed X. The default when `complete` is `FALSE` is the first `min(ncol(X), nrow(X))` columns of the original X from which the qr object was constructed. The default when `complete` is `TRUE` is a square matrix with the original X in the first `ncol(X)` columns and an arbitrary orthogonal completion (unitary completion in the complex case) in the remaining columns.
`Dvec`	vector (not matrix) of diagonal values. Each column of the returned Q will be multiplied by the corresponding diagonal value. Defaults to all `1`s.

Value

`qr.X` returns X, the original matrix from which the qr object was constructed, provided `ncol(X) <= nrow(X)`. If `complete` is `TRUE` or the argument `ncol` is greater than `ncol(X)`, additional columns from an arbitrary orthogonal (unitary) completion of `X` are returned.

`qr.Q` returns part or all of **Q**, the order-nrow(X) orthogonal (unitary) transformation represented by `qr`. If `complete` is `TRUE`, **Q** has `nrow(X)` columns. If `complete` is `FALSE`, **Q** has `ncol(X)` columns. When `Dvec` is specified, each column of **Q** is multiplied by the corresponding value in `Dvec`.

`qr.R` returns **R**. The number of rows of **R** is `nrow(X)` or `ncol(X)`, depending on whether `complete` is `TRUE` or `FALSE`.

See Also

`qr`, `qr.qy`.

Examples

```
p <- ncol(x <- LifeCycleSavings[,-1]) # not the 'sr'
qrstr <- qr(x)   # dim(x) == c(n,p)
qrstr $ rank # = 4 = p
Q <- qr.Q(qrstr) # dim(Q) == dim(x)
R <- qr.R(qrstr) # dim(R) == ncol(x)
X <- qr.X(qrstr) # X == x
range(X - as.matrix(x))# ~ < 6e-12
## X == Q %*% R if there has been no pivoting, as here.
Q %*% R
```

---

quit *Terminate an R Session*

Description

The function `quit` or its alias `q` terminate the current R session.

Usage

```
quit(save = "default", status = 0, runLast = TRUE)
   q(save = "default", status = 0, runLast = TRUE)
.Last <- function(x) { ..... }
```

Arguments

`save`	a character string indicating whether the environment (workspace) should be saved, one of `"no"`, `"yes"`, `"ask"` or `"default"`.
`status`	the (numerical) error status to be returned to the operating system, where relevant. Conventionally `0` indicates successful completion.
`runLast`	should `.Last()` be executed?

Details

`save` must be one of `"no"`, `"yes"`, `"ask"` or `"default"`. In the first case the workspace is not saved, in the second it is saved and in the third the user is prompted and can also decide *not* to quit. The default is to ask in interactive use but may be overridden by command-line arguments (which must be supplied in non-interactive use).

Immediately *before* terminating, the function `.Last()` is executed if it exists and `runLast` is true. If in interactive use there are errors in the `.Last` function, control will be returned to the command prompt, so do test the function thoroughly.

Some error statuses are used by R itself. The default error handler for non-interactive effectively calls `q("no", 1, FALSE)` and returns error code 1. Error status 2 is used for R 'suicide', that is a catastrophic failure, and other small numbers are used by specific ports for initialization failures. It is recommended that users choose statuses of 10 or more.

Valid values of `status` are system-dependent, but `0:255` are normally valid.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`.First` for setting things on startup.

Examples

```
## Not run:
## Unix-flavour example
.Last <- function() {
  cat("Now sending PostScript graphics to the printer:\n")
  system("lpr Rplots.ps")
  cat("bye bye...\n")
}
quit("yes")
## End(Not run)
```

`Quotes` *Quotes*

Description

Descriptions of the various uses of quoting in R.

Details

Three types of quote are part of the syntax of R: single and double quotation marks and the backtick (or back quote, `). In addition, backslash is used for quoting the following character(s) inside character constants.

Character constants

Single and double quotes delimit character constants. They can be used interchangeably but double quotes are preferred (and character constants are printed using double quotes), so single quotes are normally only used to delimit character constants containing double quotes.

Backslash is used to start an escape sequence inside character constants. Unless specified in the following table, an escaped character is interpreted as the character itself. Single quotes need to be escaped by backslash in single-quoted strings, and double quotes in double-quoted strings.

`\n`	newline
`\r`	carriage return
`\t`	tab
`\b`	backspace
`\a`	alert (bell)
`\f`	form feed
`\v`	vertical tab
`\\`	backslash \
`\nnn`	character with given octal code (1, 2 or 3 digits)
`\xnn`	character with given hex code (1 or 2 hex digits)
`\unnnn`	Unicode character with given code (1–4 hex digits)
`\Unnnnnnnn`	Unicode character with given code (1–8 hex digits)

The last two are only supported on versions of R built with MBCS support, and the last is not

supported on Windows. (They are an error if used where not supported.) All except the Unicode escape sequences are also supported when reading character strings from a connection by `scan`.

These forms will also be used by `print.default` when outputting non-printable characters (including backslash).

Names and Identifiers

Identifiers consist of a sequence of letters, digits, the period (`.`) and the underscore. They must not start with a digit nor underscore, nor with a period followed by a digit.

The definition of a *letter* depends on the current locale, but only ASCII digits are considered to be digits.

Such identifiers are also known as *syntactic names* and may be used directly in R code. Almost always, other names can be used provided they are quoted. The preferred quote is the backtick (`` ` ``), and `deparse` will normally use it, but under many circumstances single or double quotes can be used (as a character constant will often be converted to a name). One place where backticks may be essential is to delimit variable names in formulae: see `formula`.

See Also

`Syntax` for other aspects of the syntax.

`sQuote` for quoting English text.

`shQuote` for quoting OS commands.

The *R Language Definition* manual.

R.home *Return the R Home Directory*

Description

Return the R home directory.

Usage

```
R.home(component="home")
```

Arguments

component	As well as `"home"` which gives the R home directory, other known values are `"bin"`, `"doc"`, `"etc"` and `"share"` giving the paths to the corresponding parts of an R installation.

Details

The R home directory is the top-level directory of the R installation being run.

The R home directory is often referred to as `R_HOME`, and is the value of an environment variable of that name in an R session. It can be found outside an R session by `R RHOME`.

Value

A character string giving the R home directory or path to a particular component. Normally the components are all subdirectories of the R home directory, but this may not be the case in a Unix-like installation.

`R.Version` *Version Information*

Description

`R.Version()` provides detailed information about the version of R running.

`R.version` is a variable (a `list`) holding this information (and `version` is a copy of it for S compatibility).

Usage

```
R.Version()
R.version
R.version.string
version
```

Value

`R.Version` returns a list with character-string components

`platform`	the platform for which R was built. A triplet of the form CPU-VENDOR-OS, as determined by the configure script. E.g, `"i586-unknown-linux"` or `"i386-pc-mingw32"`.
`arch`	the architecture (CPU) R was built on/for.
`os`	the underlying operating system
`system`	CPU and OS, separated by a comma.
`status`	the status of the version (e.g., `"Alpha"`)
`major`	the major version number
`minor`	the minor version number, including the patchlevel
`year`	the year the version was released
`month`	the month the version was released
`day`	the day the version was released
`svn rev`	the Subversion revision number, which should be either `"unknown"` or a single number. (A range of numbers or a number with 'M' or 'S' appended indicates inconsistencies in the sources used to build this version of R.)
`language`	always `"R"`.
`version.string`	a `character` string concatenating some of the info above, useful for plotting, etc.

`R.version` and `version` are lists of class `"simple.list"` which has a `print` method.

Note

Do *not* use `R.version$os` to test the platform the code is running on: use `.Platform$OS.type` instead. Slightly different versions of the OS may report different values of `R.version$os`, as may different versions of R.

`R.version.string` is a copy of `R.version$version.string` for simplicity and backwards compatibility.

See Also

`sessionInfo` which provides additional information; `getRversion`, `.Platform`

Examples

```
R.version$os # to check how lucky you are ...
plot(0) # any plot
mtext(R.version.string, side=1,line=4,adj=1)# a useful bottom-right note
```

Random *Random Number Generation*

Description

`.Random.seed` is an integer vector, containing the random number generator (RNG) **state** for random number generation in R. It can be saved and restored, but should not be altered by the user.

`RNGkind` is a more friendly interface to query or set the kind of RNG in use.

`RNGversion` can be used to set the random generators as they were in an earlier R version (for reproducibility).

`set.seed` is the recommended way to specify seeds.

Usage

```
.Random.seed <- c(rng.kind, n1, n2, ...)
save.seed <- .Random.seed

RNGkind(kind = NULL, normal.kind = NULL)
RNGversion(vstr)
set.seed(seed, kind = NULL)
```

Arguments

`kind`	character or `NULL`. If `kind` is a character string, set R's RNG to the kind desired. If it is `NULL`, return the currently used RNG. Use `"default"` to return to the R default.
`normal.kind`	character string or `NULL`. If it is a character string, set the method of Normal generation. Use `"default"` to return to the R default.
`seed`	a single value, interpreted as an integer.
`vstr`	a character string containing a version number, e.g., `"1.6.2"`
`rng.kind`	integer code in `0:k` for the above `kind`.
`n1, n2, ...`	integers. See the details for how many are required (which depends on `rng.kind`).

Details

The currently available RNG kinds are given below. `kind` is partially matched to this list. The default is `"Mersenne-Twister"`.

`"Wichmann-Hill"` The seed, `.Random.seed[-1] == r[1:3]` is an integer vector of length 3, where each `r[i]` is in `1:(p[i] - 1)`, where `p` is the length 3 vector of primes, `p = (30269, 30307, 30323)`. The Wichmann–Hill generator has a cycle length of 6.9536×10^{12} (= `prod(p-1)/4`, see *Applied Statistics* (1984) **33**, 123 which corrects the original article).

`"Marsaglia-Multicarry"`: A *multiply-with-carry* RNG is used, as recommended by George Marsaglia in his post to the mailing list 'sci.stat.math'. It has a period of more than 2^{60} and has passed all tests (according to Marsaglia). The seed is two integers (all values allowed).

`"Super-Duper"`: Marsaglia's famous Super-Duper from the 70's. This is the original version which does *not* pass the MTUPLE test of the Diehard battery. It has a period of $\approx 4.6 \times 10^{18}$ for most initial seeds. The seed is two integers (all values allowed for the first seed: the second must be odd).

We use the implementation by Reeds et al. (1982–84).

The two seeds are the Tausworthe and congruence long integers, respectively. A one-to-one mapping to S's `.Random.seed[1:12]` is possible but we will not publish one, not least as this generator is **not** exactly the same as that in recent versions of S-PLUS.

`"Mersenne-Twister"`: From Matsumoto and Nishimura (1998). A twisted GFSR with period $2^{19937} - 1$ and equidistribution in 623 consecutive dimensions (over the whole period). The "seed" is a 624-dimensional set of 32-bit integers plus a current position in that set.

`"Knuth-TAOCP"`: From Knuth (1997). A GFSR using lagged Fibonacci sequences with subtraction. That is, the recurrence used is

$$X_j = (X_{j-100} - X_{j-37}) \bmod 2^{30}$$

and the "seed" is the set of the 100 last numbers (actually recorded as 101 numbers, the last being a cyclic shift of the buffer). The period is around 2^{129}.

`"Knuth-TAOCP-2002"`: The 2002 version which not backwards compatible with the earlier version: the initialization of the GFSR from the seed was altered. R did not allow you to choose consecutive seeds, the reported 'weakness', and already scrambled the seeds.

`"user-supplied"`: Use a user-supplied generator. See `Random.user` for details.

`normal.kind` can be `"Kinderman-Ramage"`, `"Buggy Kinderman-Ramage"`, `"Ahrens-Dieter"`, `"Box-Muller"`, `"Inversion"` (the default), or `"user-supplied"`. (For inversion, see the reference in `qnorm`.) The Kinderman-Ramage generator used in versions prior to 1.7.1 had several approximation errors and should only be used for reproduction of older results.

`set.seed` uses its single integer argument to set as many seeds as are required. It is intended as a simple way to get quite different seeds by specifying small integer arguments, and also as a way to get valid seed sets for the more complicated methods (especially `"Mersenne-Twister"` and `"Knuth-TAOCP"`).

Value

`.Random.seed` is an `integer` vector whose first element *codes* the kind of RNG and normal generator. The lowest two decimal digits are in `0:(k-1)` where `k` is the number of available RNGs. The hundreds represent the type of normal generator (starting at `0`).

In the underlying C, `.Random.seed[-1]` is `unsigned`; therefore in R `.Random.seed[-1]` can be negative, due to the representation of an unsigned integer by a signed integer.

`RNGkind` returns a two-element character vector of the RNG and normal kinds in use *before* the call, invisibly if either argument is not `NULL`. `RNGversion` returns the same information.

`set.seed` returns `NULL`, invisibly.

Note

Initially, there is no seed; a new one is created from the current time when one is required. Hence, different sessions will give different simulation results, by default.

`.Random.seed` saves the seed set for the uniform random-number generator, at least for the system generators. It does not necessarily save the state of other generators, and in particular does not save the state of the Box–Muller normal generator. If you want to reproduce work later, call `set.seed` rather than set `.Random.seed`.

The object `.Random.seed` is only looked for in the user's workspace.

All the supplied uniform generators return 32-bit integer values that are converted to doubles, so they take at most 2^{32} distinct values and long runs will return duplicated values.

Author(s)

of RNGkind: Martin Maechler. Current implementation, B. D. Ripley

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (`set.seed`, storing in `.Random.seed`.)

Wichmann, B. A. and Hill, I. D. (1982) *Algorithm AS 183: An Efficient and Portable Pseudo-random Number Generator*, Applied Statistics, **31**, 188–190; Remarks: **34**, 198 and **35**, 89.

De Matteis, A. and Pagnutti, S. (1993) *Long-range Correlation Analysis of the Wichmann-Hill Random Number Generator*, Statist. Comput., **3**, 67–70.

Marsaglia, G. (1997) *A random number generator for C.* Discussion paper, posting on Usenet newsgroup `sci.stat.math` on September 29, 1997.

Reeds, J., Hubert, S. and Abrahams, M. (1982–4) C implementation of SuperDuper, University of California at Berkeley. (Personal communication from Jim Reeds to Ross Ihaka.)

Marsaglia, G. and Zaman, A. (1994) Some portable very-long-period random number generators. *Computers in Physics*, **8**, 117–121.

Matsumoto, M. and Nishimura, T. (1998) Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator, *ACM Transactions on Modeling and Computer Simulation*, **8**, 3–30.
Source code at http://www.math.keio.ac.jp/~matumoto/emt.html.

Knuth, D. E. (1997) *The Art of Computer Programming.* Volume 2, third edition.
Source code at http://www-cs-faculty.stanford.edu/~knuth/taocp.html.

Knuth, D. E. (2002) *The Art of Computer Programming.* Volume 2, third edition, ninth printing.
See http://Sunburn.Stanford.EDU/~knuth/news02.html.

Kinderman, A. J. and Ramage, J. G. (1976) Computer generation of normal random variables. *Journal of the American Statistical Association* **71**, 893-896.

Ahrens, J.H. and Dieter, U. (1973) Extensions of Forsythe's method for random sampling from the normal distribution. *Mathematics of Computation* **27**, 927-937.

Box, G.E.P. and Muller, M.E. (1958) A note on the generation of normal random deviates. *Annals of Mathmatical Statistics* **29**, 610–611.

See Also

`runif`, `rnorm`, ...

Examples

```
## the default random seed is 626 integers, so only print a few
runif(1); .Random.seed[1:6]; runif(1); .Random.seed[1:6]
## If there is no seed, a "random" new one is created:
rm(.Random.seed); runif(1); .Random.seed[1:6]

RNGkind("Wich")# (partial string matching on 'kind')

## This shows how 'runif(.)' works for Wichmann-Hill,
## using only R functions:

p.WH <- c(30269, 30307, 30323)
a.WH <- c(  171,   172,   170)
next.WHseed <- function(i.seed = .Random.seed[-1])
  { (a.WH * i.seed) %% p.WH }
my.runif1 <- function(i.seed = .Random.seed)
  { ns <- next.WHseed(i.seed[-1]); sum(ns / p.WH) %% 1 }
rs <- .Random.seed
(WHs <- next.WHseed(rs[-1]))
u <- runif(1)
stopifnot(
 next.WHseed(rs[-1]) == .Random.seed[-1],
 all.equal(u, my.runif1(rs))
)

## ----
.Random.seed
ok <- RNGkind()
RNGkind("Super")#matches  "Super-Duper"
RNGkind()
.Random.seed # new, corresponding to  Super-Duper

## Reset:
RNGkind(ok[1])

## ----
sum(duplicated(runif(1e6))) # around 110
## and we would expect about almost sure duplicates beyond about
qbirthday(1-1e-6, classes=2e9) # 235,000
```

`Random.user` *User-supplied Random Number Generation*

Description

Function `RNGkind` allows user-coded uniform and normal random number generators to be supplied. The details are given here.

Details

A user-specified uniform RNG is called from entry points in dynamically-loaded compiled code. The user must supply the entry point `user_unif_rand`, which takes no arguments and returns a *pointer to* a double. The example below will show the general pattern.

Optionally, the user can supply the entry point `user_unif_init`, which is called with an `unsigned int` argument when `RNGkind` (or `set.seed`) is called, and is intended to be used to initialize the user's RNG code. The argument is intended to be used to set the "seeds"; it is the `seed` argument to `set.seed` or an essentially random seed if `RNGkind` is called.

If only these functions are supplied, no information about the generator's state is recorded in `.Random.seed`. Optionally, functions `user_unif_nseed` and `user_unif_seedloc` can be supplied which are called with no arguments and should return pointers to the number of "seeds" and to an integer array of "seeds". Calls to `GetRNGstate` and `PutRNGstate` will then copy this array to and from `.Random.seed`.

A user-specified normal RNG is specified by a single entry point `user_norm_rand`, which takes no arguments and returns a *pointer to* a double.

Warning

As with all compiled code, mis-specifying these functions can crash R. Do include the 'R_ext/Random.h' header file for type checking.

Examples

```
## Not run:
##  Marsaglia's conguential PRNG
#include <R_ext/Random.h>

static Int32 seed;
static double res;
static int nseed = 1;

double * user_unif_rand()
{
    seed = 69069 * seed + 1;
    res = seed * 2.32830643653869e-10;
    return &res;
}

void  user_unif_init(Int32 seed_in) { seed = seed_in; }
int * user_unif_nseed() { return &nseed; }
int * user_unif_seedloc() { return (int *) &seed; }

/*  ratio-of-uniforms for normal  */
#include <math.h>
static double x;

double * user_norm_rand()
{
    double u, v, z;
    do {
        u = unif_rand();
        v = 0.857764 * (2. * unif_rand() - 1);
        x = v/u; z = 0.25 * x * x;
        if (z < 1. - u) break;
```

```
        if (z > 0.259/u + 0.35) continue;
    } while (z > -log(u));
    return &x;
}

## Use under Unix:
R CMD SHLIB urand.c
R
> dyn.load("urand.so")
> RNGkind("user")
> runif(10)
> .Random.seed
> RNGkind(, "user")
> rnorm(10)
> RNGkind()
[1] "user-supplied" "user-supplied"
## End(Not run)
```

`range` *Range of Values*

Description

`range` returns a vector containing the minimum and maximum of all the given arguments.

Usage

```
range(..., na.rm = FALSE)

## Default S3 method:
range(..., na.rm = FALSE, finite = FALSE)
```

Arguments

`...`	any `numeric` objects.
`na.rm`	logical, indicating if `NA`'s should be omitted.
`finite`	logical, indicating if all non-finite elements should be omitted.

Details

`range` is a generic function: methods can be defined for it directly or via the `Summary` group generic. For this to work properly, the arguments `...` should be unnamed, and dispatch is on the first argument.

If `na.rm` is `FALSE`, `NA` and `NaN` values in any of the arguments will cause `NA` values to be returned, otherwise `NA` values are ignored.

If `finite` is `TRUE`, the minimum and maximum of all finite values is computed, i.e., `finite=TRUE` *includes* `na.rm=TRUE`.

A special situation occurs when there is no (after omission of `NA`s) nonempty argument left, see `min`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

The `extendrange()` utility; `min`, `max`, `Methods`.

Examples

```
(r.x <- range(rnorm(100)))
diff(r.x) # the SAMPLE range

x <- c(NA, 1:3, -1:1/0); x
range(x)
range(x, na.rm = TRUE)
range(x, finite = TRUE)
```

---

rank *Sample Ranks*

Description

Returns the sample ranks of the values in a vector. Ties, i.e., equal values, result in ranks being averaged, by default.

Usage

```
rank(x, na.last = TRUE,
     ties.method = c("average", "first", "random", "max", "min"))
```

Arguments

`x`	a numeric, complex, character or logical vector.
`na.last`	for controlling the treatment of `NA`s. If `TRUE`, missing values in the data are put last; if `FALSE`, they are put first; if `NA`, they are removed; if `"keep"` they are kept.
`ties.method`	a character string specifying how ties are treated, see below; can be abbreviated.

Details

If all components are different, the ranks are well defined, with values in `1:n` where `n <- length(x)` and we assume no `NA`s for the moment. Otherwise, with some values equal, called 'ties', the argument `ties.method` determines the result at the corresponding indices. The `"first"` method results in a permutation with increasing values at each index set of ties. The `"random"` method puts these in random order whereas the default, `"average"`, replaces them by their mean, and `"max"` and `"min"` replaces them by their maximum and minimum respectively, the latter being the typical "sports" ranking.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`order` and `sort`.

Examples

```
(r1 <- rank(x1 <- c(3, 1, 4, 15, 92)))
x2 <- c(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5)
names(x2) <- letters[1:11]
(r2 <- rank(x2)) # ties are averaged

## rank() is "idempotent": rank(rank(x)) == rank(x) :
stopifnot(rank(r1) == r1, rank(r2) == r2)

## ranks without averaging
rank(x2, ties.method= "first")  # first occurrence wins
rank(x2, ties.method= "random") # ties broken at random
rank(x2, ties.method= "random") # and again

## keep ties ties, no average
(rma <- rank(x2, ties.method= "max"))  # as used classically
(rmi <- rank(x2, ties.method= "min"))  # as in Sports
stopifnot(rma + rmi == round(r2 + r2))
```

`rapply` *Recursively Apply a Function to a List*

Description

`rapply` is a recursive version of `lapply`.

Usage

```
rapply(object, f, classes = "ANY", deflt = NULL,
       how = c("unlist", "replace", "list"), ...)
```

Arguments

`object`	A list.
`f`	A function of a single argument.
`classes`	A character vector of `class` names, or `"ANY"` to match any class.
`deflt`	The default result (not used if `how = "replace"`).
`how`	A character string matching the three possibilities given: see Details.
`...`	additional arguments passed to the call to `f`.

Details

This function has two basic modes. If `how = "replace"`, each element of the list which is not itself a list and has a class included in `classes` is replaced by the result of applying `f` to the element.

If the mode is `how = "list"` or `how = "unlist"`, the list is copied, all non-list elements which have a class included in `classes` are replaced by the result of applying `f` to the element and all others are replaced by `deflt`. Finally, if `how = "unlist"`, `unlist(recursive = TRUE)` is called on the result.

The semantics differ in detail from `lapply`: in particular the arguments are evaluated before calling the C code.

Value

If `how = "unlist"`, a vector, otherwise a list of similar structure to `object`.

References

Chambers, J. A. (1998) *Programming with Data*. Springer.
(`rapply` is only described briefly there.)

See Also

`lapply`, `dendrapply`.

Examples

```
X <- list(list(a=pi, b=list(c=1:1)), d="a test")
rapply(X, function(x) x, how="replace")
rapply(X, sqrt, classes="numeric", how="replace")
rapply(X, nchar, classes="character", deflt = as.integer(NA), how="list")
rapply(X, nchar, classes="character", deflt = as.integer(NA), how="unlist")
rapply(X, nchar, classes="character", how="unlist")
rapply(X, log, classes="numeric", how="replace", base=2)
```

`raw` *Raw Vectors*

Description

Creates or tests for objects of type `"raw"`.

Usage

```
raw(length = 0)
as.raw(x)
```

Arguments

`length`	desired length.
`x`	object to be coerced.

Details

The raw type is intended to hold raw bytes. It is possible to extract subsequences of bytes, and to replace elements (but only by elements of a raw vector). The relational operators (see Comparison) work, as do the logical operators (see Logic) with a bitwise interpretation.

A raw vector is printed with each byte separately represented as a pair of hex digits. If you want to see a character representation (with escape sequences for non-printing characters) use `rawToChar`.

Coercion to raw treats the input values as representing a small (decimal) integers, so the input is first coerced to integer, and then values which are outside the range `[0 ... 255]` or are `NA` are set to `0` (the `nul` byte).

Value

`raw` creates a raw vector of the specified length. Each element of the vector is equal to `0`. Raw vectors are used to store fixed-length sequences of bytes.

`as.raw` attempts to coerce its argument to be of raw type. The (elementwise) answer will be `0` unless the coercion succeeds.

See Also

`charToRaw`, `rawShift`, etc.

Examples

```
xx <- raw(2)
xx[1] <- as.raw(40)     # NB, not just 40.
xx[2] <- charToRaw("A")
xx

x <- "A test string"
(y <- charToRaw(x))
is.vector(y) # TRUE
rawToChar(y)

isASCII <-  function(txt) all(charToRaw(txt) <= as.raw(127))
isASCII(x)  # true
isASCII("\x9c25.63") # false  (in Latin-1, this is an amount in UK pounds)
```

`rawConversion` *Convert to or from Raw Vectors*

Description

Conversion and manipulation of objects of type `"raw"`.

Usage

```
charToRaw(x)
rawToChar(x, multiple = FALSE)

rawShift(x, n)
```

```
rawToBits(x)
intToBits(x)
packBits(x, type = c("raw", "integer"))
```

Arguments

x	object to be converted or shifted.
multiple	logical: should the conversion be to a single character string or multiple individual characters?
n	the number of bits to shift. Positive numbers shift right and negative numbers shift left: allowed values are `-8 ... 8`.
type	the result type.

Details

`packBits` accepts raw, integer or logical inputs, the last two without any NAs.

Note that 'bytes' are not necessarily the same as characters, e.g. in UTF-8 domains.

Value

`charToRaw` converts a length-one character string to raw bytes.

`rawToChar` converts raw bytes either to a single character string or a character vector of single bytes. (Note that a single character string could contain embedded nuls.)

`rawToBits` returns a raw vector of 8 times the length of a raw vector with entries 0 or 1. `intToBits` returns a raw vector of 32 times the length of an integer vector with entries 0 or 1. In both cases the unpacking is least-significant bit first.

`packbits` packs its input (using only the lowest bit for raw or integer vectors) least-significant bit first to a raw or integer vector.

Examples

```
x <- "A test string"
(y <- charToRaw(x))
is.vector(y) # TRUE

rawToChar(y)
rawToChar(y, multiple = TRUE)
(xx <- c(y, as.raw(0), charToRaw("more")))
rawToChar(xx)
xxx <- xx
xxx[length(y)+1] <- charToRaw("&")
xxx
rawToChar(xxx)

rawShift(y, 1)
rawShift(y, -2)

rawToBits(y)
```

`RdUtils` *Utilities for Processing Rd Files*

Description

Utilities for converting files in R documentation (Rd) format to other formats or create indices from them, and for converting documentation in other formats to Rd format.

Usage

```
R CMD Rdconv [options] file
R CMD Rd2dvi [options] files
R CMD Rd2txt [options] file
R CMD Sd2Rd [options] file
```

Arguments

`file`	the path to a file to be processed.
`files`	a list of file names specifying the R documentation sources to use, by either giving the paths to the files, or the path to a directory with the sources of a package.
`options`	further options to control the processing, or for obtaining information about usage and version of the utility.

Details

`Rdconv` converts Rd format to other formats. Currently, plain text, HTML, LaTeX, S version 3 (Sd), and S version 4 (.sgml) formats are supported. It can also extract the examples for run-time testing.

`Rd2dvi` and `Rd2txt` are user-level programs for producing DVI/PDF output or pretty text output from Rd sources.

`Sd2Rd` converts S (version 3 or 4) documentation formats to Rd format.

Use `R CMD foo --help` to obtain usage information on utility `foo`.

Note

Conversion to S version 3/4 formats is rough: there are some `.Rd` constructs for which there is no natural analogue. They are intended as a starting point for hand-tuning.

See Also

The chapter "Processing Rd format" in "Writing R Extensions" (see the 'doc/manual' subdirectory of the R source tree).

`read.table` *Data Input*

Description

Reads a file in table format and creates a data frame from it, with cases corresponding to lines and variables to fields in the file.

Usage

```
read.table(file, header = FALSE, sep = "", quote = "\"'",
           dec = ".", row.names, col.names,
           as.is = !stringsAsFactors,
           na.strings = "NA", colClasses = NA, nrows = -1,
           skip = 0, check.names = TRUE, fill = !blank.lines.skip,
           strip.white = FALSE, blank.lines.skip = TRUE,
           comment.char = "#", allowEscapes = FALSE, flush = FALSE,
           stringsAsFactors = default.stringsAsFactors())

read.csv(file, header = TRUE, sep = ",", quote="\"", dec=".",
         fill = TRUE, comment.char="", ...)

read.csv2(file, header = TRUE, sep = ";", quote="\"", dec=",",
          fill = TRUE, comment.char="", ...)

read.delim(file, header = TRUE, sep = "\t", quote="\"", dec=".",
           fill = TRUE, comment.char="", ...)

read.delim2(file, header = TRUE, sep = "\t", quote="\"", dec=",",
            fill = TRUE, comment.char="", ...)
```

Arguments

`file`	the name of the file which the data are to be read from. Each row of the table appears as one line of the file. If it does not contain an *absolute* path, the file name is *relative* to the current working directory, `getwd()`. Tilde-expansion is performed where supported. Alternatively, `file` can be a `connection`, which will be opened if necessary, and if so closed at the end of the function call. (If `stdin()` is used, the prompts for lines may be somewhat confusing. Terminate input with a blank line or an EOF signal, `Ctrl-D` on Unix and `Ctrl-Z` on Windows. Any pushback on `stdin()` will be cleared before return.) `file` can also be a complete URL. To read a data file not in the current encoding (for example a Latin-1 file in a UTF-8 locale or conversely) use a `file` connection setting the `encoding` argument.
`header`	a logical value indicating whether the file contains the names of the variables as its first line. If missing, the value is determined from the file format: `header` is set to `TRUE` if and only if the first row contains one fewer field than the number of columns.

`sep` the field separator character. Values on each line of the file are separated by this character. If `sep = ""` (the default for `read.table`) the separator is "white space", that is one or more spaces, tabs, newlines or carriage returns.

`quote` the set of quoting characters. To disable quoting altogether, use `quote = ""`. See `scan` for the behaviour on quotes embedded in quotes.

`dec` the character used in the file for decimal points.

`row.names` a vector of row names. This can be a vector giving the actual row names, or a single number giving the column of the table which contains the row names, or character string giving the name of the table column containing the row names.

If there is a header and the first row contains one fewer field than the number of columns, the first column in the input is used for the row names. Otherwise if `row.names` is missing, the rows are numbered.

Using `row.names = NULL` forces row numbering.

`col.names` a vector of optional names for the variables. The default is to use `"V"` followed by the column number.

`as.is` the default behavior of `read.table` is to convert character variables (which are not converted to logical, numeric or complex) to factors. The variable `as.is` controls the conversion of columns not otherwise specified by `colClasses`. Its value is either a vector of logicals (values are recycled if necessary), or a vector of numeric or character indices which specify which columns should not be converted to factors.

Note: to suppress all conversions including those of numeric columns, set `colClasses = "character"`.

Note that `as.is` is specified per column (not per variable) and so includes the column of row names (if any) and any columns to be skipped.

`na.strings` a character vector of strings which are to be interpreted as `NA` values. Blank fields are also considered to be missing values in logical, integer, numeric and complex fields.

`colClasses` character. A vector of classes to be assumed for the columns. Recycled as necessary, or if the character vector is named, unspecified values are taken to be `NA`.

Possible values are `NA` (when `type.convert` is used), `"NULL"` (when the column is skipped), one of the atomic vector classes (logical, integer, numeric, complex, character, raw), or `"factor"`, `"Date"` or `"POSIXct"`. Otherwise there needs to be an `as` method (from package **methods**) for conversion from `"character"` to the specified formal class.

Note that `colClasses` is specified per column (not per variable) and so includes the column of row names (if any).

`nrows` the maximum number of rows to read in. Negative values are ignored.

`skip` the number of lines of the data file to skip before beginning to read data.

`check.names` logical. If `TRUE` then the names of the variables in the data frame are checked to ensure that they are syntactically valid variable names. If necessary they are adjusted (by `make.names`) so that they are, and also to ensure that there are no duplicates.

`fill` logical. If `TRUE` then in case the rows have unequal length, blank fields are implicitly added. See Details.

`strip.white` logical. Used only when `sep` has been specified, and allows the stripping of leading and trailing white space from `character` fields (`numeric` fields are

always stripped). See scan for further details, remembering that the columns may include the row names.

blank.lines.skip
: logical: if TRUE blank lines in the input are ignored.

comment.char
: character: a character vector of length one containing a single character or an empty string. Use "" to turn off the interpretation of comments altogether.

allowEscapes
: logical. Should C-style escapes such as \n be processed or read verbatim (the default)? Note that if not within quotes these could be interpreted as a delimiter (but not as a comment character). For more details see scan.

flush
: logical: if TRUE, scan will flush to the end of the line after reading the last of the fields requested. This allows putting comments after the last field.

stringsAsFactors
: logical: should character vectors be converted to factors?

...
: Further arguments to read.table.

Details

A field or line is 'blank' if it contains nothing (except whitespace if no separator is specified) before a comment character or the end of the field or line.

If row.names is not specified and the header line has one less entry than the number of columns, the first column is taken to be the row names. This allows data frames to be read in from the format in which they are printed. If row.names is specified and does not refer to the first column, that column is discarded from such files.

The number of data columns is determined by looking at the first five lines of input (or the whole file if it has less than five lines), or from the length of col.names if it is specified and is longer. This could conceivably be wrong if fill or blank.lines.skip are true, so specify col.names if necessary.

read.csv and read.csv2 are identical to read.table except for the defaults. They are intended for reading "comma separated value" files ('.csv') or (read.csv2) the variant used in countries that use a comma as decimal point and a semicolon as field separator. Similarly, read.delim and read.delim2 are for reading delimited files, defaulting to the TAB character for the delimiter. Notice that header = TRUE and fill = TRUE in these variants, and that the comment character is disabled.

The rest of the line after a comment character is skipped; quotes are not processed in comments. Complete comment lines are allowed provided blank.lines.skip = TRUE; however, comment lines prior to the header must have the comment character in the first non-blank column.

Quoted fields with embedded newlines are supported except after a comment character.

Value

A data frame (data.frame) containing a representation of the data in the file. Empty input is an error unless col.names is specified, when a 0-row data frame is returned: similarly giving just a header line if header = TRUE results in a 0-row data frame.

This function is the principal means of reading tabular data into R.

Note

The columns referred to in `as.is` and `colClasses` include the column of row names (if any).

Less memory will be used if `colClasses` is specified as one of the six atomic vector classes.

Using `nrows`, even as a mild over-estimate, will help memory usage.

Using `comment.char = ""` will be appreciably faster.

`read.table` is not the right tool for reading large matrices, especially those with many columns: it is designed to read *data frames* which may have columns of very different classes. Use `scan` instead.

Prior to version 1.9.0, underscores were not valid in variable names, and code that relies on them being converted to dots will no longer work. The simplest workaround is to use `names(d) <- gsub("_", ".", names(d))`, or, avoiding the (small) risk of creating duplicate names, `names(d) <- make.names(gsub("_", ".", names(d)), unique=TRUE)`.

References

Chambers, J. M. (1992) *Data for models.* Chapter 3 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

The *R Data Import/Export* manual.

`scan`, `type.convert`, `read.fwf` for reading *f*ixed *w*idth *f*ormatted input; `write.table`; `data.frame`.

`count.fields` can be useful to determine problems with reading files which result in reports of incorrect record lengths.

---

readBin *Transfer Binary Data To and From Connections*

Description

Read binary data from a connection, or write binary data to a connection.

Usage

```
readBin(con, what, n = 1, size = NA, signed = TRUE,
        endian = .Platform$endian)

writeBin(object, con, size = NA, endian = .Platform$endian)
```

Arguments

`con`	A connection object or a character string naming a file or a raw vector.
`what`	Either an object whose mode will give the mode of the vector to be read, or a character vector of length one describing the mode: one of `"numeric"`, `"double"`, `"integer"`, `"int"`, `"logical"`, `"complex"`, `"character"`, `"raw"`.

n	integer. The (maximal) number of records to be read. You can use an overestimate here, but not too large as storage is reserved for n items.
size	integer. The number of bytes per element in the byte stream. The default, NA, uses the natural size. Size changing is not supported for raw and complex vectors.
signed	logical. Only used for integers of sizes 1 and 2, when it determines if the quantity on file should be regarded as a signed or unsigned integer.
endian	The endian-ness ("big" or "little" of the target system for the file. Using "swap" will force swapping endian-ness.
object	An R object to be written to the connection.

Details

If con is a character string, the functions call file to obtain an file connection which is opened for the duration of the function call.

If the connection is open it is read/written from its current position. If it is not open, it is opened for the duration of the call and then closed again. The connection must be open or open-able in binary mode.

If readBin is called con with a raw vector, the data in the vector is used as input. If writeBin is called with con a raw vector, it is just an indication that a raw vector should be returned.

If size is specified and not the natural size of the object, each element of the vector is coerced to an appropriate type before being written or as it is read. Possible sizes are 1, 2, 4 and possibly 8 for integer or logical vectors, and 4, 8 and possibly 12/16 for numeric vectors. (Note that coercion occurs as signed types except if signed = FALSE when reading integers of sizes 1 and 2.) Changing sizes is unlikely to preserve NAs, and the extended precision sizes are unlikely to be portable across platforms.

readBin and writeBin read and write C-style zero-terminated character strings. Input strings are limited to 10000 characters. readChar and writeChar can be used to read and write fixed-length strings.

Handling R's missing and special (Inf, -Inf and NaN) values is discussed in the *R Data Import/Export* manual.

Value

For readBin, a vector of appropriate mode and length the number of items read (which might be less than n).

For writeBin, a raw vector (if con is a raw vector) or invisibly NULL.

Note

Integer read/writes of size 8 will be available if either C type long is of size 8 bytes or C type long long exists and is of size 8 bytes.

Real read/writes of size sizeof(long double) (usually 12 or 16 bytes) will be available only if that type is available and different from double.

If readBin(what = character()) is used incorrectly on a file which does not contain C-style character strings, warnings (usually many) are given. From a file or connection, the input will be broken into pieces of length 10000 with any final part being discarded.

See Also

The *R Data Import/Export* manual.

`readChar` to read/write fixed-length strings.

`connections`, `readLines`, `writeLines`.

`.Machine` for the sizes of `long`, `long long` and `long double`.

Examples

```
zz <- file("testbin", "wb")
writeBin(1:10, zz)
writeBin(pi, zz, endian="swap")
writeBin(pi, zz, size=4)
writeBin(pi^2, zz, size=4, endian="swap")
writeBin(pi+3i, zz)
writeBin("A test of a connection", zz)
z <- paste("A very long string", 1:100, collapse=" + ")
writeBin(z, zz)
if(.Machine$sizeof.long == 8 || .Machine$sizeof.longlong == 8)
    writeBin(as.integer(5^(1:10)), zz, size = 8)
if((s <-.Machine$sizeof.longdouble) > 8)
    writeBin((pi/3)^(1:10), zz, size = s)
close(zz)

zz <- file("testbin", "rb")
readBin(zz, integer(), 4)
readBin(zz, integer(), 6)
readBin(zz, numeric(), 1, endian="swap")
readBin(zz, numeric(), size=4)
readBin(zz, numeric(), size=4, endian="swap")
readBin(zz, complex(), 1)
readBin(zz, character(), 1)
z2 <- readBin(zz, character(), 1)
if(.Machine$sizeof.long == 8 || .Machine$sizeof.longlong == 8)
    readBin(zz, integer(), 10,  size = 8)
if((s <-.Machine$sizeof.longdouble) > 8)
    readBin(zz, numeric(), 10, size = s)
close(zz)
unlink("testbin")
stopifnot(z2 == z)

## signed vs unsigned ints
zz <- file("testbin", "wb")
x <- as.integer(seq(0, 255, 32))
writeBin(x, zz, size=1)
writeBin(x, zz, size=1)
x <- as.integer(seq(0, 60000, 10000))
writeBin(x, zz, size=2)
writeBin(x, zz, size=2)
close(zz)
zz <- file("testbin", "rb")
readBin(zz, integer(), 8, size=1)
readBin(zz, integer(), 8, size=1, signed=FALSE)
readBin(zz, integer(), 7, size=2)
readBin(zz, integer(), 7, size=2, signed=FALSE)
close(zz)
```

```
unlink("testbin")

## use of raw
z <- writeBin(pi^{1:5}, raw(), size = 4)
readBin(z, numeric(), 5, size = 4)
z <- writeBin(c("a", "test", "of", "character"), raw())
rawToChar(z)
readBin(z, character(), 4)
```

`readChar` *Transfer Character Strings To and From Connections*

Description

Transfer character strings to and from connections, without assuming they are null-terminated on the connection.

Usage

```
readChar(con, nchars)

writeChar(object, con,
          nchars = nchar(object, type="chars"), eos = "")
```

Arguments

`con`	A connection object or a character string naming a file.
`nchars`	integer, giving the lengths in characters of (unterminated) character strings to be read or written. Must be >= 0 and not missing.
`object`	A character vector to be written to the connection, at least as long as `nchars`.
`eos`	'end of string': character string . The terminator to be written after each string, followed by an ASCII `nul`; use `NULL` for no terminator at all.

Details

These functions complement `readBin` and `writeBin` which read and write C-style zero-terminated character strings. They are for strings of known length, and can optionally write an end-of-string mark. They are intended only for character strings valid in the current locale.

If `con` is a character string, the functions call `file` to obtain an file connection which is opened for the duration of the function call.

If the connection is open it is read/written from its current position. If it is not open, it is opened for the duration of the call and then closed again. Connections can be open in either text or binary mode.

In a single-byte locale, character strings containing ASCII `nul`(s) will be read correctly by `readChar` and appear with embedded nuls in the character vector returned. This may not work for multi-byte locales, and does not work for `writeChar`.

If the character length requested for `readChar` is longer than the data available on the connection, what is available is returned. For `writeChar` if too many characters are requested the output is zero-padded, with a warning.

Missing strings are written as `NA`.

Value

For `readChar`, a character vector of length the number of items read (which might be less than `length(nchars)`).

For `writeChar` none (strictly, invisible `NULL`).

See Also

The *R Data Import/Export* manual.

`connections`, `readLines`, `writeLines`, `readBin`

Examples

```
## test fixed-length strings
zz <- file("testchar", "wb")
x <- c("a", "this will be truncated", "abc")
nc <- c(3, 10, 3)
writeChar(x, zz, nc, eos=NULL)
writeChar(x, zz, eos="\r\n")
close(zz)

zz <- file("testchar", "rb")
readChar(zz, nc)
readChar(zz, nchar(x)+3) # need to read the terminator explicitly
close(zz)
unlink("testchar")
```

`readline` *Read a Line from the Terminal*

Description

`readline` reads a line from the terminal

Usage

```
readline(prompt = "")
```

Arguments

`prompt`	the string printed when prompting the user for input. Should usually end with a space `" "`.

Details

The prompt string will be truncated to a maximum allowed length, normally 256 chars (but can be changed in the source code).

Value

A character vector of length one.

See Also

readLines for reading text lines of connections, including files.

Examples

```
fun <- function() {
  ANSWER <- readline("Are you a satisfied R user? ")
  if (substr(ANSWER, 1, 1) == "n")
    cat("This is impossible.  YOU LIED!\n")
  else
    cat("I knew it.\n")
}
fun()
```

---

readLines *Read Text Lines from a Connection*

Description

Read text lines from a connection.

Usage

```
readLines(con = stdin(), n = -1, ok = TRUE, warn = TRUE)
```

Arguments

con	a connection object or a character string.
n	integer. The (maximal) number of lines to read. Negative values indicate that one should read up to the end of the connection.
ok	logical. Is it OK to reach the end of the connection before n > 0 lines are read? If not, an error will be generated.
warn	logical. Warn if a text file is missing a final EOL.

Details

If the con is a character string, the function calls file to obtain a file connection which is opened for the duration of the function call.

If the connection is open it is read from its current position. If it is not open, it is opened for the duration of the call and then closed again.

If the final line is incomplete (no final EOL marker) the behaviour depends on whether the connection is blocking or not. For a blocking text-mode connection (or a non-text-mode connection) the line will be accepted, with a warning. For a non-blocking text-mode connection the incomplete line is pushed back, silently.

Whatever mode the connection is opened in, any of LF, CRLF or CR will be accepted as the EOL marker for a line.

Value

A character vector of length the number of lines read.

See Also

connections, writeLines, readBin, scan

Examples

```
cat("TITLE extra line", "2 3 5 7", "", "11 13 17", file="ex.data",
    sep="\n")
readLines("ex.data", n=-1)
unlink("ex.data") # tidy up

## difference in blocking
cat("123\nabc", file = "test1")
readLines("test1") # line with a warning

con <- file("test1", "r", blocking = FALSE)
readLines(con) # empty
cat(" def\n", file = "test1", append = TRUE)
readLines(con) # gets both
close(con)

unlink("test1") # tidy up
```

---

real *Real Vectors*

Description

`real` creates a double precision vector of the specified length. Each element of the vector is equal to `0`.

`as.real` attempts to coerce its argument to be of real type.

`is.real` returns `TRUE` or `FALSE` depending on whether its argument is of real type or not.

Usage

```
real(length = 0)
as.real(x, ...)
is.real(x)
```

Arguments

`length`	desired length.
`x`	object to be coerced or tested.
`...`	further arguments passed to or from other methods.

Note

R *has no single precision data type. All real numbers are stored in double precision format.*

Recall *Recursive Calling*

Description

`Recall` is used as a placeholder for the name of the function in which it is called. It allows the definition of recursive functions which still work after being renamed, see example below.

Usage

```
Recall(...)
```

Arguments

`...`	all the arguments to be passed.

Note

`Recall` will not work correctly when passed as a function argument, e.g. to the `apply` family of functions.

See Also

`do.call` and `call`.

`local` for another way to write anonymous recursive functions.

Examples

```
## A trivial (but inefficient!) example:
fib <- function(n) if(n<=2) {if(n>=0) 1 else 0} else Recall(n-1) + Recall(n-2)
fibonacci <- fib; rm(fib)
## renaming wouldn't work without Recall
fibonacci(10) # 55
```

reg.finalizer *Finalization of Objects*

Description

Registers an R function to be called upon garbage collection of object or (optionally) at the end of an R session.

Usage

```
reg.finalizer(e, f, onexit = FALSE)
```

Arguments

e	Object to finalize. Must be environment or external pointer.
f	Function to call on finalization. Must accept a single argument, which will be the object to finalize.
onexit	logical: should the finalizer be run if the object is still uncollected at the end of the R session?

Value

NULL.

Note

The purpose of this function is mainly to allow objects that refer to external items (a temporary file, say) to perform cleanup actions when they are no longer referenced from within R. This only makes sense for objects that are never copied on assignment, hence the restriction to environments and external pointers.

See Also

gc and Memory for garbage collection and memory management.

Examples

```
f <- function(e) print("cleaning....")
g <- function(x){ e <- environment(); reg.finalizer(e,f) }
g()
invisible(gc()) # trigger cleanup
```

---

regex *Regular Expressions as used in R*

Description

This help page documents the regular expression patterns supported by grep and related functions regexpr, gregexpr, sub and gsub, as well as by strsplit.

Details

A 'regular expression' is a pattern that describes a set of strings. Three types of regular expressions are used in R, *extended* regular expressions, used by grep(extended = TRUE) (its default), *basic* regular expressions, as used by grep(extended = FALSE), and *Perl-like* regular expressions used by grep(perl = TRUE).

Other functions which use regular expressions (often via the use of grep) include apropos, browseEnv, help.search, list.files, ls and strsplit. These will all use *extended* regular expressions, unless strsplit is called with argument extended = FALSE or perl = TRUE.

Patterns are described here as they would be printed by cat: do remember that backslashes need to be doubled in entering R character strings from the keyboard.

Extended Regular Expressions

This section covers the regular expressions allowed if `extended = TRUE` in `grep`, `regexpr`, `gregexpr`, `sub`, `gsub` and `strsplit`. They use the `glibc 2.3.6` implementation of the POSIX 1003.2 standard.

Regular expressions are constructed analogously to arithmetic expressions, by using various operators to combine smaller expressions.

The fundamental building blocks are the regular expressions that match a single character. Most characters, including all letters and digits, are regular expressions that match themselves. Any metacharacter with special meaning may be quoted by preceding it with a backslash. The metacharacters are `. \ | ( ) [ { ^ $ * + ?`.

A *character class* is a list of characters enclosed by `[` and `]` which matches any single character in that list; if the first character of the list is the caret `^`, then it matches any character *not* in the list. For example, the regular expression `[0123456789]` matches any single digit, and `[^abc]` matches anything except the characters `a`, `b` or `c`. A range of characters may be specified by giving the first and last characters, separated by a hyphen. (Because their interpretation is so system-dependent, they are best avoided.)

The precise way character ranges are interpreted depends on the values of `perl` and `ignore.case`. For basic and extended regular expressions the collation order is taken from the OS's implementation of the setting of the locale category `LC_COLLATE`, so `[W-Z]` may include `x` and if it does may or may not include `w`. (In most English locales the collation order is `wWxXyYzZ`.) For caseless matching the characters in a range are interpreted as if in lower case, so in an English locale `[W-z]` matches `WXYZwxyz`.

For Perl regexps, the ranges are interpreted in the numerical order of the characters, either as bytes in an 8-bit locale or as Unicode points in a UTF-8 locale.

Certain named classes of characters are predefined. Their interpretation depends on the *locale* (see locales); the interpretation below is that of the POSIX locale.

`[:alnum:]` Alphanumeric characters: `[:alpha:]` and `[:digit:]`.

`[:alpha:]` Alphabetic characters: `[:lower:]` and `[:upper:]`.

`[:blank:]` Blank characters: space and tab.

`[:cntrl:]` Control characters. In ASCII, these characters have octal codes 000 through 037, and 177 (`DEL`). In another character set, these are the equivalent characters, if any.

`[:digit:]` Digits: `0 1 2 3 4 5 6 7 8 9`.

`[:graph:]` Graphical characters: `[:alnum:]` and `[:punct:]`.

`[:lower:]` Lower-case letters in the current locale.

`[:print:]` Printable characters: `[:alnum:]`, `[:punct:]` and space.

`[:punct:]` Punctuation characters: `! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [ \ ] ^ _ ` { | } ~`.

`[:space:]` Space characters: tab, newline, vertical tab, form feed, carriage return, and space.

`[:upper:]` Upper-case letters in the current locale.

`[:xdigit:]` Hexadecimal digits: `0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f`.

For example, `[[:alnum:]]` means `[0-9A-Za-z]`, except the latter depends upon the locale and the character encoding, whereas the former is independent of locale and character set. (Note that the brackets in these class names are part of the symbolic names, and must be included in addition to the brackets delimiting the bracket list.) Most metacharacters lose their special meaning

inside lists. To include a literal `]`, place it first in the list. Similarly, to include a literal `^`, place it anywhere but first. Finally, to include a literal `-`, place it first or last. (Only these and `\` remain special inside character classes.)

The period `.` matches any single character. The symbol `\w` is documented to be synonym for `[[:alnum:]]` and `\W` is its negation. However, `\w` also matches underscore in the GNU grep code used in R.

The caret `^` and the dollar sign `$` are metacharacters that respectively match the empty string at the beginning and end of a line. The symbols `\<` and `\>` respectively match the empty string at the beginning and end of a word. The symbol `\b` matches the empty string at the edge of a word, and `\B` matches the empty string provided it is not at the edge of a word.

A regular expression may be followed by one of several repetition quantifiers:

`?` The preceding item is optional and will be matched at most once.

`*` The preceding item will be matched zero or more times.

`+` The preceding item will be matched one or more times.

`{n}` The preceding item is matched exactly `n` times.

`{n,}` The preceding item is matched `n` or more times.

`{n,m}` The preceding item is matched at least `n` times, but not more than `m` times.

Repetition is greedy, so the maximal possible number of repeats is used.

Two regular expressions may be concatenated; the resulting regular expression matches any string formed by concatenating two substrings that respectively match the concatenated subexpressions.

Two regular expressions may be joined by the infix operator `|`; the resulting regular expression matches any string matching either subexpression. For example, `abba|cde` matches either the string `abba` or the string `cde`. Note that alternation does not work inside character classes, where `|` has its literal meaning.

Repetition takes precedence over concatenation, which in turn takes precedence over alternation. A whole subexpression may be enclosed in parentheses to override these precedence rules.

The backreference `\N`, where N is a single digit, matches the substring previously matched by the Nth parenthesized subexpression of the regular expression.

Basic Regular Expressions

This section covers the regular expressions allowed if `extended = FALSE` in `grep`, `regexpr`, `gregexpr`, `sub`, `gsub` and `strsplit`.

In basic regular expressions the metacharacters `?`, `+`, `{`, `|`, `(`, and `)` lose their special meaning; instead use the backslashed versions `\?`, `\+`, `\ {`, `\|`, `\(`, and `\)`. Thus the metacharacters are `.` `\ [ ^ $ *`.

Perl Regular Expressions

The `perl = TRUE` argument to `grep`, `regexpr`, `gregexpr`, `sub`, `gsub` and `strsplit` switches to the PCRE library that 'implements regular expression pattern matching using the same syntax and semantics as Perl 5.6 or later, with just a few differences'.

For complete details please consult the man pages for PCRE, especially `man pcrepattern` and `man pcreapi`) on your system or from the sources at `ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/`. If PCRE support was compiled from the sources within R, the PCRE version is 6.7 as described here (version $\geq$ 4.0 is required if R is configured to use the system's PCRE library).

All the regular expressions described for extended regular expressions are accepted except \< and \>: in Perl all backslashed metacharacters are alphanumeric and backslashed symbols always are interpreted as a literal character. { is not special if it would be the start of an invalid interval specification. There can be more than 9 backreferences.

The construct (?...) is used for Perl extensions in a variety of ways depending on what immediately follows the ?.

Perl-like matching can work in several modes, set by the options (?i) (caseless, equivalent to Perl's /i), (?m) (multiline, equivalent to Perl's /m), (?s) (single line, so a dot matches all characters, even new lines: equivalent to Perl's /s) and (?x) (extended, whitespace data characters are ignored unless escaped and comments are allowed: equivalent to Perl's /x). These can be concatenated, so for example, (?im) sets caseless multiline matching. It is also possible to unset these options by preceding the letter with a hyphen, and to combine setting and unsetting such as (?im-sx). These settings can be applied within patterns, and then apply to the remainder of the pattern. Additional options not in Perl include (?U) to set 'ungreedy' mode (so matching is minimal unless ? is used, when it is greedy). Initially none of these options are set.

If you want to remove the special meaning from a sequence of characters, you can do so by putting them between \Q and \E. This is different from Perl in that $ and @ are handled as literals in \Q...\E sequences in PCRE, whereas in Perl, $ and @ cause variable interpolation.

The escape sequences \d, \s and \w represent any decimal digit, space character and 'word' character (letter, digit or underscore in the current locale) respectively, and their upper-case versions represent their negation. Unlike POSIX and earlier versions of Perl and PCRE, vertical tab is not regarded as a whitespace character.

Escape sequence \a is BEL, \e is ESC, \f is FF, \n is LF, \r is CR and \t is TAB. In addition \cx is cntrl-x for any x, \ddd is the octal character ddd (for up to three digits unless interpretable as a backreference), and \xhh specifies a character in hex.

Outside a character class, \b matches a word boundary, \B is its negation, \A matches at start of a subject (even in multiline mode, unlike ^), \Z matches at end of a subject or before newline at end, \z matches at end of a subject. and \G matches at first matching position in a subject. \C matches a single byte. including a newline.

The same repetition quantifiers as extended POSIX are supported. However, if a quantifier is followed by ?, the match is 'ungreedy', that is as short as possible rather than as long as possible (unless the meanings are reversed by the (?U) option.)

The sequence (?# marks the start of a comment which continues up to the next closing parenthesis. Nested parentheses are not permitted. The characters that make up a comment play no part at all in the pattern matching.

If the extended option is set, an unescaped # character outside a character class introduces a comment that continues up to the next newline character in the pattern.

The pattern (?:...) groups characters just as parentheses do but does not make a backreference.

Patterns (?=...) and (?!...) are zero-width positive and negative lookahead *assertions*: they match if an attempt to match the ... forward from the current position would succeed (or not), but use up no characters in the string being processed. Patterns (?<=...) and (?<!...) are the lookbehind equivalents: they do not allow repetition quantifiers nor \C in ....

Named subpatterns, atomic grouping, possessive qualifiers and conditional and recursive patterns are not covered here.

Author(s)

This help page is based on the documentation of GNU grep 2.4.2 and the pcrepattern man page from PCRE 6.7.

See Also

grep, apropos, browseEnv, glob2rx, help.search, list.files, ls and strsplit.

http://www.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap09.html

---

remove *Remove Objects from a Specified Environment*

Description

remove and rm can be used to remove objects. These can be specified successively as character strings, or in the character vector list, or through a combination of both. All objects thus specified will be removed.

If envir is NULL then the currently active environment is searched first.

If inherits is TRUE then parents of the supplied directory are searched until a variable with the given name is encountered. A warning is printed for each variable that is not found.

Usage

```
remove(..., list = character(0), pos = -1,
       envir = as.environment(pos), inherits = FALSE)

rm    (..., list = character(0), pos = -1,
       envir = as.environment(pos), inherits = FALSE)
```

Arguments

...	the objects to be removed, as names (unquoted) or character strings (quoted).
list	a character vector naming objects to be removed.
pos	where to do the removal. By default, uses the current environment. See the details for other possibilities.
envir	the environment to use. See the details section.
inherits	should the enclosing frames of the environment be inspected?

Details

The pos argument can specify the environment from which to remove the objects in any of several ways: as an integer (the position in the search list); as the character string name of an element in the search list; or as an environment (including using sys.frame to access the currently active function calls). The envir argument is an alternative way to specify an environment, but is primarily there for back compatibility.

It is not allowed to remove variables from the base environment and base namespace, nor from any environment which is locked (see lockEnvironment).

Earlier versions of R incorrectly claimed that supplying a character vector in ... removed the objects named in the character vector, but it removed the character vector. Use the list argument to specify objects *via* a character vector.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`ls`, `objects`

Examples

```
tmp <- 1:4
## work with tmp  and cleanup
rm(tmp)

## Not run:
## remove (almost) everything in the working environment.
## You will get no warning, so don't do this unless you are really sure.
rm(list = ls())
## End(Not run)
```

---

rep *Replicate Elements of Vectors and Lists*

Description

`rep` replicates the values in `x`. It is a generic function, and the (internal) default method is described here.

`rep.int` is a faster simplified version for the commonest case.

Usage

```
rep(x, ...)

rep.int(x, times)
```

Arguments

`x`	a vector (of any mode including a list) or a pairlist or a factor or (except for `rep.int`) a `POSIXct` or `POSIXlt` or `date` object.
`...`	further arguments to be passed to or from other methods. For the internal default method these can include: **`times`** A vector giving the number of times to repeat each element if of length `length(x)`, or to repeat the whole vector if of length 1. **`length.out`** non-negative integer. The desired length of the output vector. Ignored if `NA` or invalid. **`each`** non-negative integer. Each element of `x` is repeated `each` times. Treated as `1` if `NA` or invalid.
`times`	see `....`

Details

The default behaviour is as if the call was `rep(x, times=1, length.out=NA, each=1)`. Normally just one of the additional arguments is specified, but if codeeach is specified with either of the other two, its replication is performed first, and then that implied by `times` or `length.out`.

If `times` consists of a single integer, the result consists of the whole input repeated this many times. If `times` is a vector of the same length as `x` (after replication by `each`), the result consists of `x[1]` repeated `times[1]` times, `x[2]` repeated `times[2]` times and so on.

`length.out` may be given in place of `times`, in which case `x` is repeated as many times as is necessary to create a vector of this length. If both are given, `length.out` takes priority and `times` is ignored.

Non-integer values of `times` will be truncated towards zero. If `times` is a computed quantity it is prudent to add a small fuzz.

If `x` has length zero and `length.out` is supplied and is positive, the values are filled in using the extraction rules, that is by an `NA` of the appropriate class for an atomic vector (`0` for raw vectors) and `NULL` for a list.

Value

An object of the same type as `x` (except that `rep` will coerce pairlists to vector lists).

`rep.int` returns no attributes.

The default method of `rep` gives the result names (which will almost always contain duplicates) if `x` had names, but retains no other attributes except for factors.

Note

Function `rep.int` is a simple case handled by internal code, and provided as a separate function purely for S compatibility.

As from R 2.4.0, function `rep` is a primitive, but (partial) matching of argument names is performed as for normal functions. You can no longer pass a missing argument to. e.g. `length.out`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`seq`, `sequence`.

Examples

```
rep(1:4, 2)
rep(1:4, each = 2)       # not the same.
rep(1:4, c(2,2,2,2))     # same as second.
rep(1:4, c(2,1,2,1))
rep(1:4, each = 2, len = 4)    # first 4 only.
rep(1:4, each = 2, len = 10)   # 8 integers plus two recycled 1's.
rep(1:4, each = 2, times = 3)  # length 24, 3 complete replications

rep(1, 40*(1-.8)) # length 7 on most platforms
rep(1, 40*(1-.8)+1e-7) # better
```

```
## replicate a list
fred <- list(happy = 1:10, name = "squash")
rep(fred, 5)

# date-time objects
x <- .leap.seconds[1:3]
rep(x, 2)
rep(as.POSIXlt(x), rep(2, 3))

## named factor
x <- factor(LETTERS[1:4]); names(x) <- letters[1:4]
x
rep(x, 2)
rep(x, each=2)
rep.int(x, 2)  # no names
```

`replace` *Replace Values in a Vector*

Description

`replace` replaces the values in `x` with indexes given in `list` by those given in `values`. If necessary, the values in `values` are recycled.

Usage

```
replace(x, list, values)
```

Arguments

`x`	vector
`list`	an index vector
`values`	replacement values

Value

A vector with the values replaced.

Note

`x` is unchanged: remember to assign the result.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

rev *Reverse Elements*

Description

rev provides a reversed version of its argument. It is generic function with a default method for vectors and one for dendrograms.

Note that this is no longer needed (nor efficient) for obtaining vectors sorted into descending order, since that is now rather more directly achievable by sort(x, decreasing = TRUE).

Usage

```
rev(x)
```

Arguments

x	a vector or another object for which reversal is defined.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

seq, sort.

Examples

```
x <- c(1:5,5:3)
## sort into descending order; first more efficiently:
stopifnot(sort(x, decreasing = TRUE) == rev(sort(x)))
stopifnot(rev(1:7) == 7:1)#- don't need 'rev' here
```

rle *Run Length Encoding*

Description

Compute the lengths and values of runs of equal values in a vector – or the reverse operation.

Usage

```
rle(x)
inverse.rle(x, ...)
```

Arguments

x	a simple vector for rle() or an object of class "rle" for inverse.rle().
...	further arguments which are ignored in R.

Value

`rle()` returns an object of class `"rle"` which is a list with components

`lengths`	an integer vector containing the length of each run.
`values`	a vector of the same length as `lengths` with the corresponding values.

`inverse.rle()` is the inverse function of `rle()`.

Examples

```
x <- rev(rep(6:10, 1:5))
rle(x)
## lengths [1:5]  5 4 3 2 1
## values  [1:5] 10 9 8 7 6

z <- c(TRUE,TRUE,FALSE,FALSE,TRUE,FALSE,TRUE,TRUE,TRUE)
rle(z)
rle(as.character(z))

stopifnot(x == inverse.rle(rle(x)),
          z == inverse.rle(rle(z)))
```

Round *Rounding of Numbers*

Description

`ceiling` takes a single numeric argument `x` and returns a numeric vector containing the smallest integers not less than the corresponding elements of `x`.

`floor` takes a single numeric argument `x` and returns a numeric vector containing the largest integers not greater than the corresponding elements of `x`.

`trunc` takes a single numeric argument `x` and returns a numeric vector containing the integers formed by truncating the values in `x` toward `0`.

`round` rounds the values in its first argument to the specified number of decimal places (default 0).

`signif` rounds the values in its first argument to the specified number of significant digits.

`zapsmall` determines a `digits` argument `dr` for calling `round(x, digits = dr)` such that values "close to zero" (compared with the maximal absolute value) are "zapped", i.e., treated as `0`.

Usage

```
ceiling(x)
floor(x)
trunc(x)

round(x, digits = 0)
signif(x, digits = 6)
zapsmall(x, digits = getOption("digits"))
```

Arguments

`x`	a numeric vector. A complex vector is allowed for `round`, `signif` and `zapsmall`.
`digits`	integer indicating the precision to be used.

Details

All but `zapsmall` are generic functions: methods can be defined for them individually or via the `Math` group generic.

Note that for rounding off a 5, the IEC 60559 standard is expected to be used, "*go to the even digit*". Therefore `round(0.5)` is `0` and `round(-1.5)` is `-2`. However, this is dependent on OS services and on representation error (since e.g. `0.15` is not represented exactly, the rounding rule applies to the represented number and not to the printed number, and so `round(0.15, 1)` could be either `0.1` or `0.2`).

For `signif` the recognized values of `digits` are `1...22`. Complex numbers are rounded to retain the specified number of digits in the larger of the components. Each element of the vector is rounded individually, unlike printing.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (Except `zapsmall`.)

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer. (`zapsmall`.)

See Also

`as.integer`.

Examples

```
round(.5 + -2:4) # IEEE rounding: -2  0  0  2  2  4  4
( x1 <- seq(-2, 4, by = .5) )
round(x1)#-- IEEE rounding !
x1[trunc(x1) != floor(x1)]
x1[round(x1) != floor(x1 + .5)]
(non.int <- ceiling(x1) != floor(x1))

x2 <- pi * 100^(-1:3)
round(x2, 3)
signif(x2, 3)

print   (x2 / 1000, digits=4)
zapsmall(x2 / 1000, digits=4)
zapsmall(exp(1i*0:4*pi/2))
```

round.POSIXt *Round / Truncate Data-Time Objects*

Description

Round or truncate date-time objects.

Usage

```
## S3 method for class 'POSIXt':
round(x, units = c("secs", "mins", "hours", "days"))
## S3 method for class 'POSIXt':
trunc(x, units = c("secs", "mins", "hours", "days"))

## S3 method for class 'Date':
round(x, ...)
## S3 method for class 'Date':
trunc(x)
```

Arguments

x	an object inheriting from "POSIXt" or "Date".
units	one of the units listed. Can be abbreviated.
...	arguments to be passed to or from other methods, notably digits.

Details

The time is rounded or truncated to the second, minute, hour or day. Timezones are only relevant to days, when midnight in the current timezone is used.

The methods for class "Date" are of little use except to remove fractional days.

Value

An object of class "POSIXlt".

See Also

round for the generic function and default methods.

DateTimeClasses, Date

Examples

```
round(.leap.seconds + 1000, "hour")
trunc(Sys.time(), "day")
```

`row` *Row Indexes*

Description

Returns a matrix of integers indicating their row number in the matrix.

Usage

```
row(x, as.factor = FALSE)
```

Arguments

`x`	a matrix.
`as.factor`	a logical value indicating whether the value should be returned as a factor rather than as numeric.

Value

An integer matrix with the same dimensions as `x` and whose `ij`-th element is equal to `i`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`col` to get columns.

Examples

```
x <- matrix(1:12, 3, 4)
# extract the diagonal of a matrix
dx <- x[row(x) == col(x)]
dx

# create an identity 5-by-5 matrix
x <- matrix(0, nr = 5, nc = 5)
x[row(x) == col(x)] <- 1
x
```

`row.names` *Get and Set Row Names for Data Frames*

Description

All data frames have a row names attribute, a character vector of length the number of rows with no duplicates nor missing values.

For convenience, these are generic functions for which users can write other methods, and there are default methods for arrays. The description here is for the `data.frame` method.

Usage

```
row.names(x)
row.names(x) <- value
```

Arguments

`x`	object of class `"data.frame"`, or any other class for which a method has been defined.
`value`	an object to be coerced to character unless an integer vector. It should have (after coercion) the same length as the number of rows of `x` with no duplicated nor missing values. `NULL` is also allowed: see Details.

Details

A data frame has (by definition) a vector of *row names* which has length the number of rows in the data frame, and contains neither missing nor duplicated values.

Prior to R 2.4.0, row names were character. As from R 2.4.0, they are allowed to be integer or character, but for backwards compatibility `row.names` will always return a character vector. (Use `attr(x, "row.names")` if you need the actual value.)

Using `NULL` for the value resets the row names to `seq_len(nrow(x))`.

Value

`row.names` returns a character vector.

`row.names<-` returns a data frame with the row names changed.

Note

`row.names` is similar to `rownames` for arrays, and it has a method that calls `rownames` for an array argument.

Row names of the form `1:n` for `n > 2` are stored internally in a compact form, which might be seen from C code or by deparsing but never via `row.names` or `attr(x, "row.names")`.

References

Chambers, J. M. (1992) *Data for models.* Chapter 3 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

data.frame, rownames, names.

`row/colnames` *Row and Column Names*

Description

Retrieve or set the row or column names of a matrix-like object.

Usage

```
rownames(x, do.NULL = TRUE, prefix = "row")
rownames(x) <- value

colnames(x, do.NULL = TRUE, prefix = "col")
colnames(x) <- value
```

Arguments

`x`	a matrix-like R object, with at least two dimensions for `colnames`.
`do.NULL`	logical. Should this create names if they are `NULL`?
`prefix`	for created names.
`value`	a valid value for that component of `dimnames(x)`. For a matrix or array this is either `NULL` or a character vector of non-zero length equal to the appropriate dimension.

Details

The extractor functions try to do something sensible for any matrix-like object `x`. If the object has `dimnames` the first component is used as the row names, and the second component (if any) is used for the col names. For a data frame, `rownames` and `colnames` are equivalent to `row.names` and `names` respectively, but the latter are preferred (and can be much faster).

If `do.NULL` is `FALSE`, a character vector (of length `NROW(x)` or `NCOL(x)`) is returned in any case, prepending `prefix` to simple numbers, if there are no dimnames or the corresponding component of the dimnames is `NULL`.

The replacement methods for arrays/matrices coerce vector and factor values of `value` to character, but do not dispatch methods for `as.character`.

For a data frame, `value` for `rownames` should be a character vector of non-duplicated and non-missing names (this is enforced), and for `colnames` a character vector of (preferably) unique syntactically-valid names. In both cases, `value` will be coerced by `as.character`.

See Also

dimnames, case.names, variable.names.

Examples

```
m0 <- matrix(NA, 4, 0)
rownames(m0)

m2 <- cbind(1,1:4)
colnames(m2, do.NULL = FALSE)
colnames(m2) <- c("x","Y")
rownames(m2) <- rownames(m2, do.NULL = FALSE, prefix = "Obs.")
m2
```

---

rowsum — *Give column sums of a matrix or data frame, based on a grouping variable*

Description

Compute column sums across rows of a matrix-like object for each level of a grouping variable. `rowsum` is generic, with a method for data frames and a default method for vectors and matrices.

Usage

```
rowsum(x, group, reorder = TRUE, ...)

## S3 method for class 'data.frame':
rowsum(x, group, reorder = TRUE, na.rm = FALSE, ...)

## Default S3 method:
rowsum(x, group, reorder = TRUE, na.rm = FALSE, ...)
```

Arguments

`x`	a matrix, data frame or vector of numeric data. Missing values are allowed. A numeric vector will be treated as a column vector.
`group`	a vector or factor giving the grouping, with one element per row of `x`. Missing values will be treated as another group and a warning will be given.
`reorder`	if `TRUE`, then the result will be in order of `sort(unique(group))`, if `FALSE`, it will be in the order that groups were encountered.
`na.rm`	logical (`TRUE` or `FALSE`). Should `NA` values be discarded?
`...`	other arguments to be passed to or from methods

Details

The default is to reorder the rows to agree with `tapply` as in the example below. Reordering should not add noticeably to the time except when there are very many distinct values of `group` and `x` has few columns.

The original function was written by Terry Therneau, but this is a new implementation using hashing that is much faster for large matrices.

To sum over all the rows of a matrix (ie, a single `group`) use `colSums`, which should be even faster.

Value

A matrix or data frame containing the sums. There will be one row per unique value of `group`.

See Also

`tapply`, `aggregate`, `rowSums`

Examples

```
x <- matrix(runif(100), ncol=5)
group <- sample(1:8, 20, TRUE)
(xsum <- rowsum(x, group))
## Slower versions
tapply(x, list(group[row(x)], col(x)), sum)
t(sapply(split(as.data.frame(x), group), colSums))
aggregate(x, list(group), sum)[-1]
```

`sample` *Random Samples and Permutations*

Description

`sample` takes a sample of the specified size from the elements of `x` using either with or without replacement.

Usage

```
sample(x, size, replace = FALSE, prob = NULL)
```

Arguments

`x`	Either a (numeric, complex, character or logical) vector of more than one element from which to choose, or a positive integer.
`size`	non-negative integer giving the number of items to choose.
`replace`	Should sampling be with replacement?
`prob`	A vector of probability weights for obtaining the elements of the vector being sampled.

Details

If `x` has length 1 and `x >= 1`, sampling takes place from `1:x`. *Note* that this convenience feature may lead to undesired behaviour when `x` is of varying length `sample(x)`. See the `resample()` example below.

By default `size` is equal to `length(x)` so that `sample(x)` generates a random permutation of the elements of `x` (or `1:x`).

The optional `prob` argument can be used to give a vector of weights for obtaining the elements of the vector being sampled. They need not sum to one, but they should be nonnegative and not all zero. If `replace` is true, Walker's alias method (Ripley, 1987) is used when there are more that 250 reasonably probable values: this gives results incompatible with those from R < 2.2.0, and there will be a warning the first time this happens in a session.

If `replace` is false, these probabilities are applied sequentially, that is the probability of choosing the next item is proportional to the probabilities amongst the remaining items. The number of nonzero weights must be at least `size` in this case.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Ripley, B. D. (1987) *Stochastic Simulation*. Wiley.

Examples

```
x <- 1:12
# a random permutation
sample(x)
# bootstrap sampling -- only if length(x) > 1 !
sample(x,replace=TRUE)

# 100 Bernoulli trials
sample(c(0,1), 100, replace = TRUE)

## More careful bootstrapping --  Consider this when using sample()
## programmatically (i.e., in your function or simulation)!

# sample()'s surprise -- example
x <- 1:10
    sample(x[x >  8]) # length 2
    sample(x[x >  9]) # oops -- length 10!
try(sample(x[x > 10]))# error!

## This is safer, but only for sampling without replacement
resample <- function(x, size, ...)
  if(length(x) <= 1) { if(!missing(size) && size == 0) x[FALSE] else x
  } else sample(x, size, ...)

resample(x[x >  8])# length 2
resample(x[x >  9])# length 1
resample(x[x > 10])# length 0
```

`save` *Save R Objects*

Description

`save` writes an external representation of R objects to the specified file. The objects can be read back from the file at a later date by using the function `load` (or `data` in some cases).

`save.image()` is just a short-cut for "save my current workspace", i.e., `save(list = ls(all=TRUE), file = ".RData")`. It is also what happens with `q("yes")`.

Usage

```
save(..., list = character(0),
     file = stop("'file' must be specified"),
     ascii = FALSE, version = NULL, envir = parent.frame(),
     compress = !ascii, eval.promises = TRUE)

save.image(file = ".RData", version = NULL, ascii = FALSE,
           compress = !ascii, safe = TRUE)
```

Arguments

`...`	the names of the objects to be saved.
`list`	A character vector containing the names of objects to be saved.
`file`	a connection or the name of the file where the data will be saved. Must be a file name for workspace format version 1.
`ascii`	if `TRUE`, an ASCII representation of the data is written. The default value of `ascii` is `FALSE` which leads to a more compact binary file being written.
`version`	the workspace format version to use. `NULL` specifies the current default format. The version used from R 0.99.0 to R 1.3.1 was version 1. The default format as from R 1.4.0 is version 2.
`envir`	environment to search for objects to be saved.
`compress`	logical specifying whether saving to a named file is to use compression. Ignored when `file` is a connection and for workspace format version 1.
`eval.promises`	logical: should objects which are promises be forced before saving?
`safe`	logical. If `TRUE`, a temporary file is used for creating the saved workspace. The temporary file is renamed to `file` if the save succeeds. This preserves an existing workspace `file` if the save fails, but at the cost of using extra disk space during the save.

Details

The names of the objects specified either as symbols in `...` or as a character vector in `list` are used to look up the objects from environment `envir`. By default promises are evaluated, but if `eval.promises = FALSE` promises are saved (together with their evaluation environments). (Promises embedded in objects are always saved unevaluated.)

All R platforms use the XDR representation of binary objects in binary save-d files, and these are portable across all R platforms. (ASCII saves used to be useful for moving data between platforms but are now only of historical interest.)

Default values for the `ascii`, `compress`, `safe` and `version` arguments can be modified with the `save.defaults` option (used both by `save` and `save.image`), see also the example section below. If a `save.image.defaults` option is set it overrides `save.defaults` for function `save.image` (which allows this to have different defaults).

It is possible to compress later (with `gzip`) a file saved with `compress = FALSE`: the effect is the same as saving with `compress = TRUE`.

Warnings

The `...` arguments only give the *names* of the objects to be saved: they are searched for in the environment given by the `envir` argument, and the actual objects given as arguments need not be those found.

Saved R objects are binary files, even those saved with `ascii = TRUE`, so ensure that they are transferred without conversion of end of line markers and of 8-bit characters. The lines are delimited by LF on all platforms.

Although the default version has not changed since R 1.4.0, this does not mean that saved files are necessarily backwards compatible. You will be able to load a saved image into an earlier version of R unless use is made of later additions (for example, raw vectors or external pointers).

Note

The defaults were changed to use compressed saves for `save` in 2.3.0 and for `save.image` in 2.4.0. Any recent version of R can read compressed save files, and a compressed file can be uncompressed (by `gzip -d`) for use with very old versions of R.

See Also

`dput`, `dump`, `load`, `data`.

Examples

```
x <- runif(20)
y <- list(a = 1, b = TRUE, c = "oops")
save(x, y, file = "xy.Rdata")
save.image()
unlink("xy.Rdata")
unlink(".RData")

# set save defaults using option:
options(save.defaults=list(ascii=TRUE, safe=FALSE))
save.image()
unlink(".RData")
```

`scale` *Scaling and Centering of Matrix-like Objects*

Description

`scale` is generic function whose default method centers and/or scales the columns of a numeric matrix.

Usage

```
scale(x, center = TRUE, scale = TRUE)
```

Arguments

x	a numeric matrix(like object).
center	either a logical value or a numeric vector of length equal to the number of columns of x.
scale	either a logical value or a numeric vector of length equal to the number of columns of x.

Details

The value of center determines how column centering is performed. If center is a numeric vector with length equal to the number of columns of x, then each column of x has the corresponding value from center subtracted from it. If center is TRUE then centering is done by subtracting the column means (omitting NAs) of x from their corresponding columns, and if center is FALSE, no centering is done.

The value of scale determines how column scaling is performed (after centering). If scale is a numeric vector with length equal to the number of columns of x, then each column of x is divided by the corresponding value from scale. If scale is TRUE then scaling is done by dividing the (centered) columns of x by their root-mean-square, and if scale is FALSE, no scaling is done.

The root-mean-square for a column is obtained by computing the square-root of the sum-of-squares of the non-missing values in the column divided by the number of non-missing values minus one.

Value

For scale.default, the centered, scaled matrix. The numeric centering and scalings used (if any) are returned as attributes "scaled:center" and "scaled:scale"

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

sweep which allows centering (and scaling) with arbitrary statistics.

For working with the scale of a plot, see par.

Examples

```
require(stats)
x <- matrix(1:10, nc=2)
(centered.x <- scale(x, scale=FALSE))
cov(centered.scaled.x <- scale(x))# all 1
```

scan *Read Data Values*

Description

Read data into a vector or list from the console or file.

Usage

```
scan(file = "", what = double(0), nmax = -1, n = -1, sep = "",
     quote = if(identical(sep, "\n")) "" else "'\"", dec = ".",
     skip = 0, nlines = 0, na.strings = "NA",
     flush = FALSE, fill = FALSE, strip.white = FALSE,
     quiet = FALSE, blank.lines.skip = TRUE, multi.line = TRUE,
     comment.char = "", allowEscapes = FALSE)
```

Arguments

file	the name of a file to read data values from. If the specified file is "", then input is taken from the keyboard (or stdin if input is redirected). (In this case input can be terminated by a blank line or an EOF signal, Ctrl-D on Unix and Ctrl-Z on Windows.) Otherwise, the file name is interpreted *relative* to the current working directory (given by getwd()), unless it specifies an *absolute* path. Tilde-expansion is performed where supported. Alternatively, file can be a connection, which will be opened if necessary, and if so closed at the end of the function call. Whatever mode the connection is opened in, any of LF, CRLF or CR will be accepted as the EOL marker for a line and so will match sep = " n". file can also be a complete URL. To read a data file not in the current encoding (for example a Latin-1 file in a UTF-8 locale or conversely) use a file connection setting the encoding argument.
what	the type of what gives the type of data to be read. The supported types are logical, integer, numeric, complex, character, raw and list. If what is a list, it is assumed that the lines of the data file are records each containing length(what) items ("fields") and the list components should have elements which are one of the first six types listed or NULL, see section 'Details' below.
nmax	the maximum number of data values to be read, or if what is a list, the maximum number of records to be read. If omitted or not positive (and nlines is not set to a positive value), scan will read to the end of file.
n	the maximum number of data values to be read, defaulting to no limit.
sep	by default, scan expects to read white-space delimited input fields. Alternatively, sep can be used to specify a character which delimits fields. A field is always delimited by an end-of-line marker unless it is quoted. If specified this should be the empty character string (the default) or NULL or a character string containing just one single-byte character.

`quote` the set of quoting characters as a single character string or `NULL`. In a multibyte locale the quoting characters must be ASCII (single-byte).

`dec` decimal point character. This should be a character string containing just one single-byte character. (`NULL` and a zero-length character vector are also accepted, and taken as the default.)

`skip` the number of lines of the input file to skip before beginning to read data values.

`nlines` if positive, the maximum number of lines of data to be read.

`na.strings` character vector. Elements of this vector are to be interpreted as missing (`NA`) values. Blank fields are also considered to be missing values in logical, integer, numeric and complex fields.

`flush` logical: if `TRUE`, `scan` will flush to the end of the line after reading the last of the fields requested. This allows putting comments after the last field, but precludes putting more that one record on a line.

`fill` logical: if `TRUE`, `scan` will implicitly add empty fields to any lines with fewer fields than implied by `what`.

`strip.white` vector of logical value(s) corresponding to items in the `what` argument. It is used only when `sep` has been specified, and allows the stripping of leading and trailing white space from `character` fields (`numeric` fields are always stripped).

If `strip.white` is of length 1, it applies to all fields; otherwise, if `strip.white[i]` is `TRUE` *and* the `i`-th field is of mode character (because `what[i]` is) then the leading and trailing white space from field `i` is stripped.

`quiet` logical: if `FALSE` (default), scan() will print a line, saying how many items have been read.

`blank.lines.skip`
logical: if `TRUE` blank lines in the input are ignored, except when counting `skip` and `nlines`.

`multi.line` logical. Only used if `what` is a list. If `FALSE`, all of a record must appear on one line (but more than one record can appear on a single line). Note that using `fill = TRUE` implies that a record will terminated at the end of a line.

`comment.char` character: a character vector of length one containing a single character or an empty string. Use `""` to turn off the interpretation of comments altogether (the default).

`allowEscapes` logical. Should C-style escapes such as
`n` be processed (the default) or read verbatim? Note that if not within quotes these could be interpreted as a delimiter (but not as a comment character).

The escapes which are interpreted are the control characters
`a`,
`b`,
`f`,
`n`,
`r`,
`t`,
`v` and octal and hexadecimal representatons like
`040` and
`0x2A`. Any other escaped character is treated as itself, including backslash.

Details

The value of what can be a list of types, in which case scan returns a list of vectors with the types given by the types of the elements in what. This provides a way of reading columnar data. If any of the types is NULL, the corresponding field is skipped (but a NULL component appears in the result).

The type of what or its components can be one of the six atomic vector types or NULL (see is.atomic).

'White space' is defined for the purposes of this function as one or more contiguous characters from the set space, horizontal tab, carriage return and line feed. It does not include form feed, vertical tab or other non-ASCII space characters.

Empty numeric fields are always regarded as missing values. Empty character fields are scanned as empty character vectors, unless na.strings contains "" when they are regarded as missing values.

If sep is the default (""), the character \ in a quoted string escapes the following character, so quotes may be included in the string by escaping them.

If sep is non-default, the fields may be quoted in the style of '.csv' files where separators inside quotes (' or "") are ignored and quotes may be put inside strings by doubling them. However, if sep = "\n" it is assumed by default that one wants to read entire lines verbatim.

Quoting is only interpreted in character fields and in NULL fields (which might be skipping character fields).

Note that since sep is a separator and not a terminator, reading a file by scan("foo", sep="\n", blank.lines.skip=FALSE) will give an empty final line if the file ends in a linefeed and not if it does not. This might not be what you expected; see also readLines.

If comment.char occurs (except inside a quoted character field), it signals that the rest of the line should be regarded as a comment and be discarded. Lines beginning with a comment character (possibly after white space with the default separator) are treated as blank lines.

scan attempts to share storage with character strings that have already been read in the call. If an upper bound on the number of character strings cannot be deduced from nmax or n, sharing is used for the first 10000 unique strings which are read in.

There is a check for a user interrupt every 1000 lines if what is a list, otherwise every 10000 items.

Value

if what is a list, a list of the same length and same names (as any) as what.

Otherwise, a vector of the type of what.

Note

The default for multi.line differs from S. To read one record per line, use flush = TRUE and multi.line = FALSE. (Note that quoted character strings can still include embedded new lines.)

If number of items is not specified, the internal mechanism re-allocates memory in powers of two and so could use up to three times as much memory as needed. (It needs both old and new copies.) If you can, specify either n or nmax whenever inputting a large vector, and nmax or nlines when inputting a large list.

Using scan on an open connection to read partial lines can lose chars: use an explicit separator to avoid this.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`read.table` for more user-friendly reading of data matrices; `readLines` to read a file a line at a time. `write`.

`Quotes` for the details of C-style escape sequences.

`readChar` and `readBin` to read fixed or variable length character strings or binary representations of numbers a few at a time from a connection.

Examples

```
cat("TITLE extra line", "2 3 5 7", "11 13 17", file="ex.data", sep="\n")
pp <- scan("ex.data", skip = 1, quiet= TRUE)
    scan("ex.data", skip = 1)
    scan("ex.data", skip = 1, nlines=1)# only 1 line after the skipped one
scan("ex.data", what = list("","","")) # flush is F -> read "7"
scan("ex.data", what = list("","",""), flush = TRUE)
unlink("ex.data") # tidy up
```

---

search *Give Search Path for R Objects*

Description

Gives a list of `attach`ed *packages* (see `library`), and R objects, usually `data.frames`.

Usage

```
search()
searchpaths()
```

Value

A character vector, starting with `".GlobalEnv"`, and ending with `"package:base"` which is R's **base** package required always.

`searchpaths` gives a similar character vector, with the entries for packages being the path to the package used to load the code.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (`search`.)

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer. (`searchPaths`.)

See Also

`.packages` to list just the packages on search path.

`loadedNamespaces` to list loaded namespaces.

`attach` and `detach` to change the search "path", `objects` to find R objects in there.

Examples

```
search()
searchpaths()
```

`seek` *Functions to Reposition Connections*

Description

Functions to re-position connections.

Usage

```
seek(con, ...)
## S3 method for class 'connection':
seek(con, where = NA, origin = "start", rw = "", ...)

isSeekable(con)

truncate(con, ...)
```

Arguments

`con`	a connection.
`where`	numeric. A file position (relative to the origin specified by `origin`), or `NA`.
`rw`	character. Empty or `"read"` or `"write"`, partial matches allowed.
`origin`	character. One of `"start"`, `"current"`, `"end"`: see Details.
`...`	further arguments passed to or from other methods.

Details

`seek` with `where = NA` returns the current byte offset of a connection (from the beginning), and with a non-missing `where` argument the connection is re-positioned (if possible) to the specified position. `isSeekable` returns whether the connection in principle supports `seek`: currently only (possibly gz-compressed) file connections do. `gzfile` connections do not support `origin = "end"`; the file position they use is that of the uncompressed file.

`where` is stored as a real but should represent an integer: non-integer values are likely to be truncated. Note that the possible values can exceed the largest representable number in an R `integer` on 64-bit OSes, and on some 32-bit OSes.

File connections can be open for both writing/appending, in which case R keeps separate positions for reading and writing. Which `seek` refers to can be set by its `rw` argument: the default is the last mode (reading or writing) which was used. Most files are only opened for reading or writing and so

default to that state. If a file is open for reading and writing but has not been used, the default is to give the reading position (0).

The initial file position for reading is always at the beginning. The initial position for writing is at the beginning of the file for modes `"r+"` and `"r+b"`, otherwise at the end of the file. Some platforms only allow writing at the end of the file in the append modes. (The reported write position for a file opened in an append mode will typically be unreliable until the file has been written to.)

`truncate` truncates a file opened for writing at its current position. It works only for `file` connections, and is not implemented on all platforms: on others (including Windows) it will not work for large (> 2Gb) files.

Value

`seek` returns the current position (before any move), as a (numeric) byte offset from the origin, if relevant, or `0` if not. Note that the position can exceed the largest representable number in an R `integer` on 64-bit OSes, and on some 32-bit OSes.

`truncate` returns `NULL`: it stops with an error if it fails (or is not implemented).

`isSeekable` returns a logical value, whether the connection supports `seek`.

See Also

`connections`

`seq` *Sequence Generation*

Description

Generate regular sequences. `seq` is a standard generic with a default method. `seq.int` is an internal generic which can be much faster but has a few restrictions. `seq_along` and `seq_len` are very fast primitives for two common cases.

Usage

```
seq(...)

## Default S3 method:
seq(from = 1, to = 1, by = ((to - from)/(length.out - 1)),
    length.out = NULL, along.with = NULL, ...)

seq.int(from, to, by, length.out, along.with, ...)

seq_along(along.with)
seq_len(length.out)
```

Arguments

`...`	arguments passed to or from methods.
`from, to`	the starting and (maximal) end value of the sequence.
`by`	number: increment of the sequence.

length.out	desired length of the sequence. A non-negative number, which for seq and seq.int will be rounded up if fractional.
along.with	take the length from the length of this argument.

Details

The interpretation of the unnamed arguments of seq and seq.int is *not* standard, and it is recommended always to name the arguments when programming.

Both seq are seq.int are generic, and only the default method is described here. Typical usages are

```
seq(from, to)
seq(from, to, by= )
seq(from, to, length.out= )
seq(along.with= )
seq(from)
seq(length.out= )
```

The first form generates the sequence from, from+/−1, ..., to (identical to from:to).

The second form generates from, from+by, ..., up to the sequence value less than or equal to to. Specifying to - from and by of opposite signs is an error.

The third generates a sequence of length.out equally spaced values from from to to. (length.out is usually abbreviated to length or len, and seq_len is much faster.)

The fourth form generates the sequence 1, 2, ..., length(along.with). (along.with is usually abbreviated to along, and seq_along is much faster.)

The fifth form generates the sequence 1, 2, ..., length(from) (as if argument along.with had been specified), *unless* the argument is numeric of length 1 when it is interpreted as 1:from (even for seq(0) for compatibility with S).

The final form generates 1, 2, ..., length.out unless length.out = 0, when it generates integer(0).

Very small sequences (with from - to of the order of 10^{-14} times the larger of the ends) will return from.

For seq(only), up to two of from, to and by can be supplied as complex values provided length.out or along.with is specified.

Value

Currently, the default method returns a result of type "integer" if from is (numerically equal to an) integer and, e.g., only to is specified, or also if only length or only along.with is specified. **Note:** this may change in the future and programmers should not rely on it.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

The methods seq.Date and seq.POSIXt.

:, rep, sequence, row, col.

Examples

```
seq(0, 1, length=11)
seq(rnorm(20))
seq(1, 9, by = 2) # match
seq(1, 9, by = pi)# stay below
seq(1, 6, by = 3)
seq(1.575, 5.125, by=0.05)
seq(17) # same as 1:17
```

---

seq.Date *Generate Regular Sequences of Dates*

Description

The method for `seq` for date-time classes.

Usage

```
## S3 method for class 'Date':
seq(from, to, by, length.out = NULL, along.with = NULL, ...)
```

Arguments

`from`	starting date. Required
`to`	end date. Optional.
`by`	increment of the sequence. Optional. See Details.
`length.out`	integer, optional. desired length of the sequence.
`along.with`	take the length from the length of this argument.
`...`	arguments passed to or from other methods.

Details

`by` can be specified in several ways.

- A number, taken to be in days.
- A object of class `difftime`
- A character string, containing one of `"day"`, `"week"`, `"month"` or `"year"`. This can optionally be preceded by a (positive or negative) integer and a space, or followed by `"s"`. See `seq.POSIXt` for the details of `"month"`.

Value

A vector of class `"Date"`.

See Also

`Date`

Examples

```
## first days of years
seq(as.Date("1910/1/1"), as.Date("1999/1/1"), "years")
## by month
seq(as.Date("2000/1/1"), by="month", length=12)
## quarters
seq(as.Date("2000/1/1"), as.Date("2003/1/1"), by="3 months")

## find all 7th of the month between two dates, the last being a 7th.
st <- as.Date("1998-12-17")
en <- as.Date("2000-1-7")
ll <- seq.Date(en, st, by="-1 month")
rev(ll[ll > st & ll < en])
```

---

seq.POSIXt *Generate Regular Sequences of Dates*

Description

The method for seq for date-time classes.

Usage

```
## S3 method for class 'POSIXt':
seq(from, to, by, length.out = NULL, along.with = NULL, ...)
```

Arguments

from	starting date. Required.
to	end date. Optional.
by	increment of the sequence. Optional. See Details.
length.out	integer, optional. desired length of the sequence.
along.with	take the length from the length of this argument.
...	arguments passed to or from other methods.

Details

by can be specified in several ways.

- A number, taken to be in seconds.
- A object of class difftime
- A character string, containing one of "sec", "min", "hour", "day", "DSTday", "week", "month" or "year". This can optionally be preceded by a (positive or negative) integer and a space, or followed by "s".

The difference between "day" and "DSTday" is that the former ignores changes to/from daylight savings time and the latter takes the same clock time each day. ("week" ignores DST (it is a period of 144 hours), but "7 DSTdays") can be used as an alternative. "month" and "year" allow for DST.)

The timezone of the result is taken from `from`: remember than GMT does not have daylight savings time.

Using `"month"` first advances the month without changing the day: if this results in an invalid day of the month, it is counted forward into the next month: see the examples.

Value

A vector of class `"POSIXct"`.

See Also

`DateTimeClasses`

Examples

```
## first days of years
seq(ISOdate(1910,1,1), ISOdate(1999,1,1), "years")
## by month
seq(ISOdate(2000,1,1), by = "month", length = 12)
seq(ISOdate(2000,1,31), by = "month", length = 4)
## quarters
seq(ISOdate(1990,1,1), ISOdate(2000,1,1), by = "3 months")
## days vs DSTdays: use c() to lose the timezone.
seq(c(ISOdate(2000,3,20)), by = "day", length = 10)
seq(c(ISOdate(2000,3,20)), by = "DSTday", length = 10)
seq(c(ISOdate(2000,3,20)), by = "7 DSTdays", length = 4)
```

`sequence` *Create A Vector of Sequences*

Description

For each element of `nvec` the sequence `seq(nvec[i])` is created. These are appended and the result returned.

Usage

```
sequence(nvec)
```

Arguments

`nvec`	an integer vector each element of which specifies the upper bound of a sequence.

See Also

`gl`, `seq`, `rep`.

Examples

```
sequence(c(3,2))# the concatenated sequences 1:3 and 1:2.
#> [1] 1 2 3 1 2
```

sets *Set Operations*

Description

Performs **set** union, intersection, (asymmetric!) difference, equality and membership on two vectors.

Usage

```
union(x, y)
intersect(x, y)
setdiff(x, y)
setequal(x, y)
is.element(el, set)
```

Arguments

`x, y, el, set`
: vectors (of the same mode) containing a sequence of items (conceptually) with no duplicated values.

Details

Each of `union`, `intersect` and `setdiff` will remove any duplicated values in the arguments.

`is.element(x, y)` is identical to `x %in% y`.

Value

A vector of the same `mode` as `x` or `y` for `setdiff` and `intersect`, respectively, and of a common mode for `union`.

A logical scalar for `setequal` and a logical of the same length as `x` for `is.element`.

See Also

`%in%`

Examples

```
(x <- c(sort(sample(1:20, 9)),NA))
(y <- c(sort(sample(3:23, 7)),NA))
union(x, y)
intersect(x, y)
setdiff(x, y)
setdiff(y, x)
setequal(x, y)

## True for all possible x & y :
setequal( union(x,y),
          c(setdiff(x,y), intersect(x,y), setdiff(y,x)))

is.element(x, y)# length 10
is.element(y, x)# length  8
```

showConnections *Display Connections*

Description

Display aspects of connections.

Usage

```
showConnections(all = FALSE)
getConnection(what)
closeAllConnections()

stdin()
stdout()
stderr()
```

Arguments

all	logical: if true all connections, including closed ones and the standard ones are displayed. If false only open user-created connections are included.
what	integer: a row number of the table given by showConnections.

Details

stdin(), stdout() and stderr() are standard connections corresponding to input, output and error on the console respectively (and not necessarily to file streams). They are text-mode connections of class "terminal" which cannot be opened or closed, and are read-only, write-only and write-only respectively. The stdout() and stderr() connections can be re-directed by sink.

showConnections returns a matrix of information. If a connection object has been lost or forgotten, getConnection will take a row number from the table and return a connection object for that connection, which can be used to close the connection, for example.

closeAllConnections closes (and destroys) all open user connections, restoring all sink diversions as it does so.

Value

stdin(), stdout() and stderr() return connection objects.

showConnections returns a character matrix of information with a row for each connection, by default only for open non-standard connections.

getConnection returns a connection object, or NULL.

See Also

connections

Examples

```
showConnections(all = TRUE)

textConnection(letters)
# oops, I forgot to record that one
showConnections()
#  class     description      mode text   isopen   can read can write
#3 "letters" "textConnection" "r"  "text" "opened" "yes"    "no"
## Not run: close(getConnection(3))

showConnections()
```

---

shQuote *Quote Strings for Use in OS Shells*

Description

Quote a string to be passed to an operating system shell.

Usage

```
shQuote(string, type = c("sh", "csh", "cmd"))
```

Arguments

string	a character vector, usually of length one.
type	character: the type of shell. Partial matching is supported. "cmd" refers to the Windows NT shell, and is the default under Windows.

Details

The default type of quoting supported under Unix-alikes is that for the Bourne shell sh. If the string does not contain single quotes, we can just surround it with single quotes. Otherwise, the string is surrounded in double quotes, which suppresses all special meanings of metacharacters except dollar, backquote and backslash, so these (and of course double quote) are preceded by backslash. This type of quoting is also appropriate for ksh, zsh and bash.

The other type of quoting is for the C-shell (csh and tcsh). Once again, if the string does not contain single quotes, we can just surround it with single quotes. If it does contain single quotes, we can use double quotes provided it does not contain dollar or backquote (and we need to escape backslash, exclamation mark and double quote). As a last resort, we need to split the string into pieces not containing single quotes and surround each with single quotes, and the single quotes with double quotes.

References

Loukides, M. et al (2002) *Unix Power Tools* Third Edition. O'Reilly. Section 27.12.

http://www.mhuffman.com/notes/dos/bash_cmd.htm

See Also

Quotes for quoting R code.

sQuote for quoting English text.

Examples

```
test <- "abc$def`gh`i\\j"
cat(shQuote(test), "\n")
## Not run: system(paste("echo", shQuote(test)))
test <- "don't do it!"
cat(shQuote(test), "\n")

tryit <- "use the `-c' switch\nlike this"
cat(shQuote(tryit), "\n")
## Not run: system(paste("echo", shQuote(tryit)))
cat(shQuote(tryit, type="csh"), "\n")

## Windows-only example.
perlcmd <- 'print "Hello World\n";'
## Not run: shell(paste("perl -e", shQuote(perlcmd, type="cmd")))
```

---

sign *Sign Function*

Description

`sign` returns a vector with the signs of the corresponding elements of `x` (the sign of a real number is 1, 0, or -1 if the number is positive, zero, or negative, respectively).

Note that `sign` does not operate on complex vectors.

Usage

```
sign(x)
```

Arguments

`x`	a numeric vector

Details

This is a generic function: methods can be defined for it directly or via the `Math` group generic.

See Also

`abs`

Examples

```
sign(pi) # == 1
sign(-2:3)# -1 -1 0 1 1 1
```

Signals *Interrupting Execution of R*

Description

On receiving SIGUSR1 R will save the workspace and quit. SIGUSR2 has the same result except that the .Last function and on.exit expressions will not be called.

Usage

```
kill -USR1 pid
kill -USR2 pid
```

Arguments

pid	The process ID of the R process

Warning

It is possible that one or more R objects will be undergoing modification at the time the signal is sent. These objects could be saved in a corrupted form.

sink *Send R Output to a File*

Description

sink diverts R output to a connection.

sink.number() reports how many diversions are in use.

sink.number(type = "message") reports the number of the connection currently being used for error messages.

Usage

```
sink(file = NULL, append = FALSE, type = c("output", "message"),
     split = FALSE)

sink.number(type = c("output", "message"))
```

Arguments

file	a connection or a character string naming the file to write to, or NULL to stop sink-ing.
append	logical. If TRUE, output will be appended to file; otherwise, it will overwrite the contents of file.
type	character. Either the output stream or the messages stream.
split	logical: if TRUE, output will be sent to the new sink and to the current output stream, like the Unix program tee.

Details

`sink` diverts R output to a connection. If `file` is a character string, a file connection with that name will be established for the duration of the diversion.

Normal R output is diverted by the default `type = "output"`. Only prompts and warning/error messages continue to appear on the terminal. The latter can diverted by `type = "message"` (see below).

`sink()` or `sink(file=NULL)` ends the last diversion (of the specified type). There is a stack of diversions for normal output, so output reverts to the previous diversion (if there was one). The stack is of up to 21 connections (20 diversions).

If `file` is a connection it will be opened if necessary.

Sink-ing the messages stream should be done only with great care. For that stream `file` must be an already open connection, and there is no stack of connections.

Value

`sink` returns `NULL`.

For `sink.number()` the number (0, 1, 2, ...) of diversions of output in place.

For `sink.number("message")` the connection number used for messages, 2 if no diversion has been used.

Warning

Don't use a connection that is open for `sink` for any other purpose. The software will stop you closing one such inadvertently.

Do not sink the messages stream unless you understand the source code implementing it and hence the pitfalls.

Note

`sink(split = TRUE)` is only available on systems which support the C99 function `va_copy` (or under the name `__va_copy`), but we know of no current systems which do not.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer.

See Also

`capture.output`

Examples

```
sink("sink-examp.txt")
i <- 1:10
outer(i, i, "*")
sink()
unlink("sink-examp.txt")
## Not run:
## capture all the output to a file.
```

```
zz <- file("all.Rout", open="wt")
sink(zz)
sink(zz, type="message")
try(log("a"))
## back to the console
sink(type="message")
sink()
try(log("a"))
## End(Not run)
```

`slice.index` *Slice Indexes in an Array*

Description

Returns a matrix of integers indicating the number of their slice in a given array.

Usage

```
slice.index(x, MARGIN)
```

Arguments

`x`	an array. If `x` has no dimension attribute, it is considered a one-dimensional array.
`MARGIN`	an integer giving the dimension number to slice by.

Value

An integer array `y` with dimensions corresponding to those of `x` such that all elements of slice number `i` with respect to dimension `MARGIN` have value `i`.

See Also

`row` and `col` for determining row and column indexes; in fact, these are special cases of `slice.index` corresponding to `MARGIN` equal to 1 and 2, respectively when `x` is a matrix.

Examples

```
x <- array(1 : 24, c(2, 3, 4))
slice.index(x, 2)
```

slotOp *Extract Slots*

Description

Extract tbe contents of a slot in a object with a formal class structure.

Usage

```
object@name
```

Arguments

object	An object from a formally defined class.
name	The character-string name of the slot.

Details

This operator supports the formal classes of package **methods**, and is disabled unless the **methods** is loaded. See slot for further details.

Currently there is no checking that the object is an instance of a formal class, nor that name is a slot name.

See Also

Extract, slot

socketSelect *Wait on Socket Connections*

Description

Waits for the first of several socket connections to become available.

Usage

```
socketSelect(socklist, write = FALSE, timeout = NULL)
```

Arguments

socklist	list of open socket connections
write	logical. If TRUE wait for corresponding socket to become available for writing; otherwise wait for it to become available for reading.
timeout	numeric or NULL. Time in seconds to wait for a socket to become available; NULL means wait indefinitely.

Details

The values in `write` are recycled if necessary to make up a logical vector the same length as `socklist`. Socket connections can appear more than once in `socklist`; this can be useful if you want to determine whether a socket is available for reading or writing.

Value

Logical the same length as `socklist` indicating whether the corresponding socket connection is available for output or input, depending on the corresponding value of `write`.

Examples

```
## Not run:
## test whether socket connection s is available for writing or reading
socketSelect(list(s,s),c(TRUE,FALSE),timeout=0)
## End(Not run)
```

`solve` *Solve a System of Equations*

Description

This generic function solves the equation `a %*% x = b` for `x`, where `b` can be either a vector or a matrix.

Usage

```
solve(a, b, ...)

## Default S3 method:
solve(a, b, tol, LINPACK = FALSE, ...)
```

Arguments

`a`	a square numeric or complex matrix containing the coefficients of the linear system.
`b`	a numeric or complex vector or matrix giving the right-hand side(s) of the linear system. If missing, `b` is taken to be an identity matrix and `solve` will return the inverse of `a`.
`tol`	the tolerance for detecting linear dependencies in the columns of `a`. If `LINPACK` is `TRUE` the default is `1e-7`, otherwise it is `.Machine$double.eps`. Future versions of R may use a tighter tolerance. Not presently used with complex matrices `a`.
`LINPACK`	logical. Should LINPACK be used (for compatibility with R < 1.7.0)? Otherwise LAPACK is used.
`...`	further arguments passed to or from other methods

Details

a or b can be complex, but this uses double complex arithmetic which might not be available on all platforms and LAPACK will always be used.

The row and column names of the result are taken from the column names of a and of b respectively. As from R 1.7.0 if b is missing the column names of the result are the row names of a. No check is made that the column names of a and the row names of b are equal.

For back-compatibility a can be a (real) QR decomposition, although qr.solve should be called in that case. qr.solve can handle non-square systems.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

solve.qr for the qr method, chol2inv for inverting from the Choleski factor backsolve, qr.solve.

Examples

```
hilbert <- function(n) { i <- 1:n; 1 / outer(i - 1, i, "+") }
h8 <- hilbert(8); h8
sh8 <- solve(h8)
round(sh8 %*% h8, 3)

A <- hilbert(4)
A[] <- as.complex(A)
## might not be supported on all platforms
try(solve(A))
```

---

sort *Sorting or Ordering Vectors*

Description

Sort (or *order*) a vector or factor (partially) into ascending (or descending) order. For ordering along more than one variable, e.g., for sorting data frames, see order.

Usage

```
sort(x, decreasing = FALSE, ...)

## Default S3 method:
sort(x, decreasing = FALSE, na.last = NA, ...)

sort.int(x, partial = NULL, na.last = NA, decreasing = FALSE,
         method = c("shell", "quick"), index.return = FALSE)

is.unsorted(x, na.rm = FALSE)
```

Arguments

x	for sort, an R object with a class or a numeric, complex, character or logical vector. For sort.int, a numeric, complex, character or logical vector, or a factor.
decreasing	logical. Should the sort be increasing or decreasing? Not available for partial sorting.
...	arguments to be passed to or from methods or (for the default methods and objects without a class) to sim.int.
na.last	for controlling the treatment of NAs. If TRUE, missing values in the data are put last; if FALSE, they are put first; if NA, they are removed.
partial	NULL or an integer vector of indices for partial sorting.
method	character string specifying the algorithm used.
index.return	logical indicating if the ordering index vector should be returned as well; this is only available for a few cases, the default na.last = NA and full sorting of non-factors.
na.rm	logical. Should missing values be removed?

Details

sort is a generic function for which methods can be written, and sort.int is the internal method which is compatible with S if only the first three arguments are used.

If partial is not NULL, it is taken to contain indices of elements of x which are to be placed in their correct positions by partial sorting. After the sort, the values specified in partial are in their correct position in the sorted array. Any values smaller than these values are guaranteed to have a smaller index in the sorted array and any values which are greater are guaranteed to have a bigger index in the sorted array. (This is included for efficiency, and many of the options are not available for partial sorting. It is only substantially more efficient if partial has a handful of elements, and a full sort is done if there are more than 10.) Names are discarded for partial sorting.

Complex values are sorted first by the real part, then the imaginary part.

The sort order for character vectors will depend on the collating sequence of the locale in use: see Comparison. The sort order for factors is the order of their levels (which is particularly appropriate for ordered factors).

is.unsorted returns a logical indicating if x is sorted increasingly, i.e., is.unsorted(x) is true if any(x != sort(x)) (and there are no NAs).

Method "shell" uses Shellsort (an $O(n^{4/3})$ variant from Sedgewick (1996)). If x has names a stable sort is used, so ties are not reordered. (This only matters if names are present.)

Method "quick" uses Singleton's Quicksort implementation and is only available when x is numeric (double or integer) and partial is NULL. (For other types of x Shellsort is used, silently.) It is normally somewhat faster than Shellsort (perhaps twice as fast on vectors of length a million) but has poor performance in the rare worst case. (Peto's modification using a pseudo-random midpoint is used to make the worst case rarer.) This is not a stable sort, and ties may be reordered.

Value

For sort, the result depends on the S3 method which is dispatched. If x does not have a class the rest of this section applies. For classed objects which do not have a specific method the default method will be used and is equivalent to x[order(x, ...)]: this depends on the class having a suitable method for [(and also that order will work, which is not the case for a class based on a list).

For `sort.int` the sorted vector unless `index.return` is true, when the result is a list with components named `x` and `ix` containing the sorted numbers and the ordering index vector. In the latter case, if `method == "quick"` ties may be reversed in the ordering, unlike `sort.list`, as quicksort is not stable.

All attributes are removed from the return value (see Becker *et al*, 1988, p.146) except names, which are sorted. (If `partial` is specified even the names are removed.) Note that this means that the returned value has no class, except for factors and ordered factors (which are treated specially and whose result is transformed back to the original class).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Sedgewick, R. (1986) A new upper bound for Shell sort. *J. Algorithms* **7**, 159–173.

Singleton, R. C. (1969) An efficient algorithm for sorting with minimal storage: Algorithm 347. *Communications of the ACM* **12**, 185–187.

See Also

`order` for sorting on or reordering multiple variables.

`rank`.

Examples

```
require(stats)
x <- swiss$Education[1:25]
x; sort(x); sort(x, partial = c(10, 15))
stats:::median.default # shows you another example for 'partial'

## illustrate 'stable' sorting (of ties):
sort(c(10:3,2:12), method = "sh", index=TRUE) # is stable
## $x : 2  3  3  4  4  5  5  6  6  7  7  8  8  9  9 10 10 11 12
## $ix: 9  8 10  7 11  6 12  5 13  4 14  3 15  2 16  1 17 18 19
sort(c(10:3,2:12), method = "qu", index=TRUE) # is not
## $x : 2  3  3  4  4  5  5  6  6  7  7  8  8  9  9 10 10 11 12
## $ix: 9 10  8  7 11  6 12  5 13  4 14  3 15 16  2 17  1 18 19
##        ^^^^^

## Not run: ## Small speed comparison simulation:
N <- 2000
Sim <- 20
rep <- 1000 # << adjust to your CPU
c1 <- c2 <- numeric(Sim)
for(is in 1:Sim){
  x <- rnorm(N)
  c1[is] <- system.time(for(i in 1:rep) sort(x, method = "shell"))[1]
  c2[is] <- system.time(for(i in 1:rep) sort(x, method = "quick"))[1]
  stopifnot(sort(x, method = "s") == sort(x, method = "q"))
}
rbind(ShellSort = c1, QuickSort = c2)
cat("Speedup factor of quick sort():\n")
summary({qq <- c1 / c2; qq[is.finite(qq)]})

## A larger test
```

```
x <- rnorm(1e7)
system.time(x1 <- sort(x, method = "shell"))
system.time(x2 <- sort(x, method = "quick"))
stopifnot(identical(x1, x2))
## End(Not run)
```

`source` *Read R Code from a File or a Connection*

Description

`source` causes R to accept its input from the named file or URL (the name must be quoted) or connection. Input is read and `parse`d by from that file until the end of the file is reached, then the parsed expressions are evaluated sequentially in the chosen environment.

Usage

```
source(file, local = FALSE, echo = verbose, print.eval = echo,
       verbose = getOption("verbose"),
       prompt.echo = getOption("prompt"),
       max.deparse.length = 150, chdir = FALSE,
       encoding = getOption("encoding"))
```

Arguments

`file`	a connection or a character string giving the pathname of the file or URL to read from.
`local`	if `local` is `FALSE`, the statements scanned are evaluated in the user's workspace (the global environment), otherwise in the environment calling `source`.
`echo`	logical; if `TRUE`, each expression is printed after parsing, before evaluation.
`print.eval`	logical; if `TRUE`, the result of `eval(i)` is printed for each expression `i`; defaults to `echo`.
`verbose`	if `TRUE`, more diagnostics (than just `echo = TRUE`) are printed during parsing and evaluation of input, including extra info for **each** expression.
`prompt.echo`	character; gives the prompt to be used if `echo = TRUE`.
`max.deparse.length`	integer; is used only if `echo` is `TRUE` and gives the maximal length of the "echo" of a single expression.
`chdir`	logical; if `TRUE` and `file` is a pathname, the R working directory is temporarily changed to the directory containing `file` for evaluating.
`encoding`	character string. The encoding to be assumed when `file` is a character string: see `file`. A possible value is `"unknown"`: see the Details.

Details

All versions of R accept input from a connection with end of line marked by LF (as used on Unix), CRLF (as used on DOS/Windows) or CR (as used on classic MacOS). The final line can be incomplete, that is missing the final EOL marker.

If `options`("keep.source") is true (the default), the source of functions is kept so they can be listed exactly as input. This imposes a limit of 128K bytes on the function size and a nesting limit of 265. Use `option(keep.source = FALSE)` when these limits might take effect: if exceeded they generate an error.

This paragraph applies if `file` is a filename (rather than a connection). If `encoding = "unknown"`, an attempt is made to guess the encoding. The result of `localeToCharset()` is used a guide. If `encoding` has two or more elements, they are tried in turn until the file/URL can be read without error in the trial encoding.

Unlike input from a console, lines in the file or on a connection can contain an unlimited number of characters. However, there is a limit of 8192 bytes on the size of character strings.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`demo` which uses `source`; `eval`, `parse` and `scan`; `options("keep.source")`.

Examples

```
## If you want to source() a bunch of files, something like
## the following may be useful:
 sourceDir <- function(path, trace = TRUE, ...) {
    for (nm in list.files(path, pattern = "\\.[RrSsQq]$")) {
       if(trace) cat(nm,":")
       source(file.path(path, nm), ...)
       if(trace) cat("\n")
    }
 }
```

`Special` *Special Functions of Mathematics*

Description

Special mathematical functions related to the beta and gamma functions.

Usage

```
beta(a, b)
lbeta(a, b)
gamma(x)
lgamma(x)
psigamma(x, deriv = 0)
digamma(x)
```

```
trigamma(x)
choose(n, k)
lchoose(n, k)
factorial(x)
lfactorial(x)
```

Arguments

`a, b, x, n`	numeric vectors.
`k, deriv`	integer vectors.

Details

The functions `beta` and `lbeta` return the beta function and the natural logarithm of the beta function,

$$B(a,b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}.$$

The formal definition is

$$B(a,b) = \int_0^1 t^{a-1}(1-t)^{b-1}dt$$

(Abramowitz and Stegun (6.2.1), page 258).

The functions `gamma` and `lgamma` return the gamma function $\Gamma(x)$ and the natural logarithm of the absolute value of the gamma function. The gamma function is defined by (Abramowitz and Stegun (6.1.1), page 255)

$$\Gamma(x) = \int_0^\infty t^{a-1}e^{-t}dt$$

`factorial(x)` is $x!$ and identical to `gamma(x+1)` and `lfactorial` is `lgamma(x+1)`.

The functions `digamma` and `trigamma` return the first and second derivatives of the logarithm of the gamma function. `psigamma(x, deriv)` (`deriv >= 0`) is more generally computing the `deriv`-th derivative of $\psi(x)$.

$$\texttt{digamma(x)} = \psi(x) = \frac{d}{dx}\ln\Gamma(x) = \frac{\Gamma'(x)}{\Gamma(x)}$$

The functions `choose` and `lchoose` return binomial coefficients and their logarithms. Note that `choose(n,k)` is defined for all real numbers n and integer k. For $k \geq 1$ as $n(n-1)\cdots(n-k+1)/k!$, as 1 for $k = 0$ and as 0 for negative k.
`choose(*,k)` uses direct arithmetic (instead of `[l]gamma` calls) for small `k`, for speed and accuracy reasons. Note the function `combn` (package **utils**) for enumeration of all possible combinations.

The `gamma`, `lgamma`, `digamma` and `trigamma` functions are generic: methods can be defined for them individually or via the `Math` group generic.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (for `gamma` and `lgamma`.)

Abramowitz, M. and Stegun, I. A. (1972) *Handbook of Mathematical Functions.* New York: Dover. Chapter 6: Gamma and Related Functions.

See Also

`Arithmetic` for simple, `sqrt` for miscellaneous mathematical functions and `Bessel` for the real Bessel functions.

For the incomplete gamma function see `pgamma`.

Examples

```
choose(5, 2)
for (n in 0:10) print(choose(n, k = 0:n))

factorial(100)
lfactorial(10000)

## gamma has 1st order poles at 0, -1, -2, ...
## this will generate loss of precision warnings, so turn off
op <- options("warn")
options(warn = -1)
x <- sort(c(seq(-3,4, length=201), outer(0:-3, (-1:1)*1e-6, "+")))
plot(x, gamma(x), ylim=c(-20,20), col="red", type="l", lwd=2,
     main=expression(Gamma(x)))
abline(h=0, v=-3:0, lty=3, col="midnightblue")
options(op)

x <- seq(.1, 4, length = 201); dx <- diff(x)[1]
par(mfrow = c(2, 3))
for (ch in c("", "l","di","tri","tetra","penta")) {
  is.deriv <- nchar(ch) >= 2
  nm <- paste(ch, "gamma", sep = "")
  if (is.deriv) {
    dy <- diff(y) / dx # finite difference
    der <- which(ch == c("di","tri","tetra","penta")) - 1
    nm2 <- paste("psigamma(*, deriv = ", der,")",sep='')
    nm  <- if(der >= 2) nm2 else paste(nm, nm2, sep = " ==\n")
    y <- psigamma(x, deriv=der)
  } else {
    y <- get(nm)(x)
  }
  plot(x, y, type = "l", main = nm, col = "red")
  abline(h = 0, col = "lightgray")
  if (is.deriv) lines(x[-1], dy, col = "blue", lty = 2)
}
par(mfrow = c(1, 1))

## "Extended" Pascal triangle:
fN <- function(n) formatC(n, wid=2)
for (n in -4:10) cat(fN(n),":", fN(choose(n, k= -2:max(3,n+2))), "\n")

## R code version of choose()  [simplistic; warning for k < 0]:
mychoose <- function(r,k)
    ifelse(k <= 0, (k==0),
           sapply(k, function(k) prod(r:(r-k+1))) / factorial(k))
k <- -1:6
cbind(k=k, choose(1/2, k), mychoose(1/2, k))

## Binomial theorem for n=1/2 ;
## sqrt(1+x) = (1+x)^(1/2) = sum_{k=0}^Inf  choose(1/2, k) * x^k :
```

```
k <- 0:10 # 10 is sufficient for ~ 9 digit precision:
sqrt(1.25)
sum(choose(1/2, k)* .25^k)
```

`split` *Divide into Groups*

Description

`split` divides the data in the vector `x` into the groups defined by `f`. The replacement forms replace values corresponding to such a division. `unsplit` reverses the effect of `split`.

Usage

```
split(x, f, drop = FALSE, ...)
split(x, f, drop = FALSE, ...) <- value
unsplit(value, f, drop = FALSE)
```

Arguments

`x`	vector or data frame containing values to be divided into groups.
`f`	a "factor" in the sense that `as.factor(f)` defines the grouping, or a list of such factors in which case their interaction is used for the grouping.
`drop`	logical indicating if levels that do not occur should be dropped (if `f` is a `factor` or a list).
`value`	a list of vectors or data frames compatible with a splitting of `x`. Recycling applies if the lengths do not match.
`...`	further potential arguments passed to methods.

Details

`split` and `split<-` are generic functions with default and `data.frame` methods.

`f` is recycled as necessary and if the length of `x` is not a multiple of the length of `f` a warning is printed. `unsplit` works only with lists of vectors. The data frame method can also be used to split a matrix into a list of matrices, and the replacement form likewise, provided they are invoked explicitly.

Any missing values in `f` are dropped together with the corresponding values of `x`.

Value

The value returned from `split` is a list of vectors containing the values for the groups. The components of the list are named by the *used* factor levels given by `f`. (If `f` is longer than `x` then some of the components will be of zero length.)

The replacement forms return their right hand side. `unsplit` returns a vector for which `split(x, f)` equals `value`

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

cut

Examples

```
require(stats)
n <- 10; nn <- 100
g <- factor(round(n * runif(n * nn)))
x <- rnorm(n * nn) + sqrt(as.numeric(g))
xg <- split(x, g)
boxplot(xg, col = "lavender", notch = TRUE, varwidth = TRUE)
sapply(xg, length)
sapply(xg, mean)

## Calculate z-scores by group

z <- unsplit(lapply(split(x, g), scale), g)
tapply(z, g, mean)

# or

z <- x
split(z, g) <- lapply(split(x, g), scale)
tapply(z, g, sd)

## Split a matrix into a list by columns
ma <- cbind(x = 1:10, y = (-4:5)^2)
split(ma, col(ma))

split(1:10, 1:2)
```

---

sprintf *Use C-style String Formatting Commands*

Description

A wrapper for the C function `sprintf`, that returns a character vector containing a formatted combination of text and variable values.

Usage

```
sprintf(fmt, ...)
gettextf(fmt, ..., domain = NULL)
```

Arguments

fmt	a format string.
...	values to be passed into fmt. Only logical, integer, real and character vectors are supported, but some coercion will be done: see the Details section.
domain	see gettext.

Details

sprintf is a wrapper for the system sprintf C-library function. Attempts are made to check that the mode of the values passed match the format supplied, and R's special values (NA, Inf, -Inf and NaN) are handled correctly.

gettextf is a convenience function which provides C-style string formatting with possible translation of the format string.

The arguments (including fmt) are recycled if possible a whole number of times to the length of the longest, and then the formatting is done in parallel.

The following is abstracted from Kernighan and Ritchie (see References). The string fmt contains normal characters, which are passed through to the output string, and also special characters that operate on the arguments provided through ... . Special characters start with a % and end with one of the letters in the set difeEgGsxX%. These letters denote the following types:

d, i, x, X Integer value, x and X being hexadecimal (using the same case for a-f as the code). Numeric variables with exactly integer values will be coerced to integer.

f Double precision value, in decimal notation of the form "[-]mmm.ddd". The number of decimal places is specified by the precision: the default is 6; a precision of 0 suppresses the decimal point.

e, E Double precision value, in decimal notation of the form [-]m.ddde[+-]xx or [-]m.dddE[+-]xx.

g, G Double precision value, in %e or %E format if the exponent is less than -4 or greater than or equal to the precision, and %f format otherwise.

s Character string.

% Literal % (none of the formatting characters given below are permitted in this case).

as.character is used for non-character arguments with s and as.double for non-double arguments with f, e, E, g, G. NB: the length is determined before conversion, so do not rely on the internal coercion if this would change the length.

In addition, between the initial % and the terminating conversion character there may be, in any order:

m.n Two numbers separated by a period, denoting the field width (m) and the precision (n)

- Left adjustment of converted argument in its field

+ Always print number with sign

a space Prefix a space if the first number is not a sign

0 For numbers, pad to the field width with leading zeros

Further, immediately after % may come 1$ to 99$ to refer to the numbered argument: this allows arguments to be referenced out of order and is mainly intended for translators of error messages. If this is done it is best if all formats are numbered: if not the unnumbered ones process the arguments in order. See the examples.

A field width or precision (but not both) may be indicated by an asterisk `*`. In this case an argument specifies the desired number. A negative field width is taken as a '-' flag followed by a positive field width. A negative precision is taken as if the precision were omitted. The `*1$` to `*99$` notation for arguments referenced out of order is also supported.

The result has a length limit, probably 8192 bytes, and attempts to exceed this may result in an error, or truncation with a warning.

Value

A character vector of length that of the longest input. Character `NA`s are converted to `"NA"`.

Author(s)

Original code by Jonathan Rougier, ⟨J.C.Rougier@durham.ac.uk⟩.

References

Kernighan, B. W. and Ritchie, D. M. (1988) *The C Programming Language.* Second edition, Prentice Hall. describes the format options in table B-1 in the Appendix.

See Also

`formatC` for a way of formatting vectors of numbers in a similar fashion.

`paste` for another way of creating a vector combining text and values.

`gettext` for the mechanisms for the automated translation of text.

Examples

```
## be careful with the format: most things in R are floats
## only integer-valued reals get coerced to integer.

sprintf("%s is %f feet tall\n", "Sven", 7.1)      # OK
try(sprintf("%s is %i feet tall\n", "Sven", 7.1)) # not OK
try(sprintf("%s is %i feet tall\n", "Sven", 7))   # OK

## use a literal % :

sprintf("%.0f%% said yes (out of a sample of size %.0f)", 66.666, 3)

## various formats of pi :

sprintf("%f", pi)
sprintf("%.3f", pi)
sprintf("%1.0f", pi)
sprintf("%5.1f", pi)
sprintf("%05.1f", pi)
sprintf("%+f", pi)
sprintf("% f", pi)
sprintf("%-10f", pi) # left justified
sprintf("%e", pi)
sprintf("%E", pi)
sprintf("%g", pi)
sprintf("%g",   1e6 * pi) # -> exponential
sprintf("%.9g", 1e6 * pi) # -> "fixed"
sprintf("%G", 1e-6 * pi)
```

```
## no truncation:
sprintf("%1.f",101)

## re-use one argument three times, show difference between %x and %X
xx <- sprintf("%1$d %1$x %1$X", 0:15)
xx <- matrix(xx, dimnames=list(rep("", 16), "%d%x%X"))
noquote(format(xx, justify="right"))

## More sophisticated:

sprintf("min 10-char string '%10s'",
        c("a", "ABC", "and an even longer one"))

n <- 1:18
sprintf(paste("e with %2d digits = %.",n,"g",sep=""), n, exp(1))

## Using arguments out of order
sprintf("second %2$1.0f, first %1$5.2f, third %3$1.0f", pi, 2, 3)

## Using asterisk for width or precision
sprintf("precision %.*f, width '%*.3f'", 3, pi, 8, pi)

## Asterisk and argument re-use, 'e' example reiterated:
sprintf("e with %1$2d digits = %2$.*1$g", n, exp(1))

## re-cycle arguments
sprintf("%s %d", "test", 1:3)
```

`sQuote` *Quote Text*

Description

Single or double quote text by combining with appropriate single or double left and right quotation marks.

Usage

```
sQuote(x)
dQuote(x)
```

Arguments

`x`	an R object, to be coerced to a character vector.

Details

The purpose of the functions is to provide a simple means of markup for quoting text to be used in the R output, e.g., in warnings or error messages. Note that English quoting conventions are used, whatever the locale in use.

The choice of the appropriate quotation marks depends on both the locale and the available character sets. Older Unix/X11 fonts displayed the grave accent (0x60) and the apostrophe (0x27) in a way

that they could also be used as matching open and close single quotation marks. Using modern fonts, or non-Unix systems, these characters no longer produce matching glyphs. Unicode provides left and right single quotation mark characters (U+2018 and U+2019); if Unicode cannot be assumed, it seems reasonable to use the apostrophe as an undirectional single quotation mark.

Similarly, Unicode has left and right double quotation mark characters (U+201C and U+201D); if only ASCII's typewriter characteristics can be employed, than the ASCII quotation mark (0x22) should be used as both the left and right double quotation mark.

By default, `sQuote` and `dQuote` employ undirectional ASCII quotation style. In a UTF-8 locale (see `l10n_info`), the Unicode directional quotes are used.

References

Markus Kuhn, "ASCII and Unicode quotation marks". http://www.cl.cam.ac.uk/~mgk25/ucs/quotes.html

See Also

`Quotes` for quoting R code.

`shQuote` for quoting OS commands.

Examples

```
paste("argument", sQuote("x"), "must be non-zero")
```

---

stack *Stack or Unstack Vectors from a Data Frame or List*

Description

Stacking vectors concatenates multiple vectors into a single vector along with a factor indicating where each observation originated. Unstacking reverses this operation.

Usage

```
stack(x, ...)
## Default S3 method:
stack(x, ...)
## S3 method for class 'data.frame':
stack(x, select, ...)

unstack(x, ...)
## Default S3 method:
unstack(x, form, ...)
## S3 method for class 'data.frame':
unstack(x, form = formula(x), ...)
```

Arguments

x	object to be stacked or unstacked
select	expression, indicating variables to select from a data frame
form	a two-sided formula whose left side evaluates to the vector to be unstacked and whose right side evaluates to the indicator of the groups to create. Defaults to formula(x) in unstack.data.frame.
...	further arguments passed to or from other methods.

Details

The stack function is used to transform data available as separate columns in a data frame or list into a single column that can be used in an analysis of variance model or other linear model. The unstack function reverses this operation.

Value

unstack produces a list of columns according to the formula form. If all the columns have the same length, the resulting list is coerced to a data frame.

stack produces a data frame with two columns

values	the result of concatenating the selected vectors in x
ind	a factor indicating from which vector in x the observation originated

Author(s)

Douglas Bates

See Also

lm, reshape

Examples

```
require(stats)
formula(PlantGrowth)          # check the default formula
pg <- unstack(PlantGrowth)    # unstack according to this formula
pg
stack(pg)                     # now put it back together
stack(pg, select = -ctrl)     # omitting one vector
```

Startup *Initialization at Start of an R Session*

Description

In R, the startup mechanism is as follows.

Unless '`--no-environ`' was given on the command line, R searches for user and site files to process for setting environment variables. The name of the site file is the one pointed to by the environment variable `R_ENVIRON`; if this is unset or empty, '$R_HOME/etc/Renviron.site' is used (if it exists, which it does not in a "factory-fresh" installation). The user files searched for are '.Renviron' in the current or in the user's home directory (in that order). See **Details** for how the files are read.

Then R searches for the site-wide startup profile unless the command line option '`--no-site-file`' was given. The name of this file is taken from the value of the `R_PROFILE` environment variable. If this variable is unset, the default is '$R_HOME/etc/Rprofile.site', which is used if it exists (which it does not in a "factory-fresh" installation). This code is loaded into package **base**. Users need to be careful not to unintentionally overwrite objects in base, and it is normally advisable to use `local` if code needs to be executed: see the examples.

Then, unless '`--no-init-file`' was given, R searches for a file called '.Rprofile' in the current directory or in the user's home directory (in that order) and sources it into the user workspace.

It then loads a saved image of the user workspace from '.RData' if there is one (unless '`--no-restore-data`' was specified, or '`--no-restore`', on the command line).

Next, if a function `.First` is found on the search path, it is executed as `.First()`. Finally, function `.First.sys()` in the **base** package is run. This calls `require` to attach the default packages specified by `options("defaultPackages")`. If the **methods** package is included, this will have been attached earlier, by function `.OptRequireMethods()`, so that namespace initializations such as those from the user workspace will proceed correctly.

A function `.First` (and `.Last`) can be defined in appropriate '.Rprofile' or 'Rprofile.site' files or have been saved in '.RData'. If you want a different set of packages than the default ones when you start, insert a call to `options` in the '.Rprofile' or 'Rprofile.site' file. For example, `options(defaultPackages = character())` will attach no extra packages on startup. Alternatively, set `R_DEFAULT_PACKAGES=NULL` as an environment variable before running R. Using `options(defaultPackages = "")` or `R_DEFAULT_PACKAGES=""` enforces the R *system* default.

The commands history is read from the file specified by the environment variable `R_HISTFILE` (default '.Rhistory') unless '`--no-restore-history`' was specified (or '`--no-restore`').

The command-line flag '`--vanilla`' implies '`--no-site-file`', '`--no-init-file`', '`--no-restore`' and '`--no-environ`'.

Usage

```
.First <- function() { ..... }

.Rprofile <startup file>
```

Details

Note that there are two sorts of files used in startup: *environment files* which contain lists of environment variables to be set, and *profile files* which contain R code.

Lines in a site or user environment file should be either comment lines starting with `#`, or lines of the form `name=value`. The latter sets the environmental variable `name` to `value`, overriding an existing value. If `value` contains an expression of the form `${foo-bar}`, the value is that of the environmental variable `foo` if that exists and is set to a non-empty value, otherwise `bar`. (If it

is of the form `${foo}`, the default is `""`.) This construction can be nested, so `bar` can be of the same form (as in `${foo-${bar-blah}}`). Note that the braces are essential: `$HOME` will not be interpreted.

Leading and trailing white space in `value` are stripped. `value` is processed in a similar way to a Unix shell. In particular the outermost level of (single or double) quotes is stripped, and backslashes are removed except inside quotes.

Note

The file '$R_HOME/etc/Renviron' is always read very early in the start-up processing. It contains environment variables set by R in the configure process. Values in that file can be overriden in site or user environment files: do not change '$R_HOME/etc/Renviron' itself. Note that this is distinct from '$R_HOME/etc/Renviron.site'.

See Also

`.Last` for final actions before termination. `commandArgs` for accessing the command line arguments.

An Introduction to R for more command-line options: those affecting memory management are covered in the help file for Memory.

For profiling code, see `Rprof`.

Examples

```
## Not run:
# Example ~/.Renviron on Unix
R_LIBS=~/R/library
PAGER=/usr/local/bin/less

# Example .Renviron on Windows
R_LIBS=C:/R/library
MY_TCLTK=yes
TCL_LIBRARY=c:/packages/Tcl/lib/tcl8.4

# Example of .Rprofile
options(width=65, digits=5)
options(show.signif.stars=FALSE)
ps.options(horizontal=FALSE)
set.seed(1234)
.First <- function() cat("\n   Welcome to R!\n\n")
.Last <- function()  cat("\n   Goodbye!\n\n")

# Example of Rprofile.site
local({
  old <- getOption("defaultPackages")
  options(defaultPackages = c(old, "MASS"))
})

## if .Renviron contains
FOOBAR="coo\bar"doh\ex"abc\"def'"

## then we get
> cat(Sys.getenv("FOOBAR"), "\n")
coo\bardoh\exabc"def'
## End(Not run)
```

`stop` *Stop Function Execution*

Description

`stop` stops execution of the current expression and executes an error action.

`geterrmessage` gives the last error message.

Usage

```
stop(..., call. = TRUE, domain = NULL)
geterrmessage()
```

Arguments

`...`	character vectors (which are pasted together with no separator), a condition object, or `NULL`.
`call.`	logical, indicating if the call should become part of the error message.
`domain`	see `gettext`. If `NA`, messages will not be translated.

Details

The error action is controlled by error handlers established within the executing code and by the current default error handler set by `options(error=)`. The error is first signaled as if using `signalCondition()`. If there are no handlers or if all handlers return, then the error message is printed (if `options("show.error.messages")` is true) and the default error handler is used. The default behaviour (the `NULL` error-handler) in interactive use is to return to the top level prompt or the top level browser, and in non-interactive use to (effectively) call `q("no", status=1, runLast=FALSE)`. The default handler stores the error message in a buffer; it can be retrieved by `geterrmessage()`. It also stores a trace of the call stack that can be retrieved by `traceback()`.

Errors will be truncated to `getOption("warning.length")` characters, default 1000.

An attempt is made to coerce other types of inputs to character vectors.

Value

`geterrmessage` gives the last error message, as a character string ending in `"\n"`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`warning`, `try` to catch errors and retry, and `options` for setting error handlers. `stopifnot` for validity testing. `tryCatch` and `withCallingHandlers` can be used to establish custom handlers while executing an expression.

`gettext` for the mechanisms for the automated translation of messages.

Examples

```
options(error = expression(NULL))
# don't stop on stop(.)  << Use with CARE! >>

iter <- 12
if(iter > 10) stop("too many iterations")

tst1 <- function(...) stop("dummy error")
tst1(1:10, long, calling, expression)

tst2 <- function(...) stop("dummy error", call. = FALSE)
tst2(1:10, longcalling, expression, but.not.seen.in.Error)

options(error = NULL)# revert to default
```

---

stopifnot *Ensure the 'Truth' of R Expressions*

Description

If any of the expressions in `...` are not `all TRUE`, `stop` is called, producing an error message indicating the *first* of the elements of `...` which were not true.

Usage

```
stopifnot(...)
```

Arguments

`...`	any number of (`logical`) R expressions.

which should evaluate to `TRUE`.

Details

This function is intended for use in regression tests or also argument checking of functions, in particular to make them easier to read.

`stopifnot(A, B)` is conceptually equivalent to `{ if(any(is.na(A)) || !all(A)) stop(...) ; if(any(is.na(B)) || !all(B)) stop(...) }`.

Value

(`NULL` if all statements in `...` are `TRUE`.)

See Also

`stop`, `warning`.

Examples

```
stopifnot(1 == 1, all.equal(pi, 3.14159265), 1 < 2) # all TRUE

m <- matrix(c(1,3,3,1), 2,2)
stopifnot(m == t(m), diag(m) == rep(1,2)) # all(.) |=>  TRUE

op <- options(error = expression(NULL))
# "disable stop(.)"  << Use with CARE! >>

stopifnot(all.equal(pi, 3.141593),  2 < 2, all(1:10 < 12), "a" < "b")
stopifnot(all.equal(pi, 3.1415927), 2 < 2, all(1:10 < 12), "a" < "b")

options(op)# revert to previous error handler
```

---

strptime *Date-time Conversion Functions to and from Character*

Description

Functions to convert between character representations and objects of classes "POSIXlt" and "POSIXct" representing calendar dates and times.

Usage

```
## S3 method for class 'POSIXct':
format(x, format = "", tz = "", usetz = FALSE, ...)
## S3 method for class 'POSIXlt':
format(x, format = "", usetz = FALSE, ...)

## S3 method for class 'POSIXt':
as.character(x, ...)

strftime(x, format="", usetz = FALSE, ...)
strptime(x, format, tz = "")

ISOdatetime(year, month, day, hour, min, sec, tz = "")
ISOdate(year, month, day, hour = 12, min = 0, sec = 0, tz = "GMT")
```

Arguments

x	An object to be converted.
tz	A timezone specification to be used for the conversion. System-specific, but "" is the current time zone, and "GMT" is UTC.
format	A character string. The default is "%Y-%m-%d %H:%M:%S" if any component has a time component which is not midnight, and "%Y-%m-%d" otherwise. If options("digits.secs") is set, up to the specified number of digits will be printed for seconds.
...	Further arguments to be passed from or to other methods.
usetz	logical. Should the timezone be appended to the output? This is used in printing times, and as a workaround for problems with using "%Z" on most Linux systems.

`year, month, day`
: numerical values to specify a day.

`hour, min, sec`
: numerical values for a time within a day. Fractional seconds are allowed.

Details

`strftime` is an alias for `format.POSIXlt`, and `format.POSIXct` first converts to class `"POSIXlt"` by calling `as.POSIXlt`. Note that only that conversion depends on the time zone.

The usual vector re-cycling rules are applied to `x` and `format` so the answer will be of length that of the longer of the vectors.

Locale-specific conversions to and from character strings are used where appropriate and available. This affects the names of the days and months, the AM/PM indicator (if used) and the separators in formats such as `%x` and `%X`.

The details of the formats are system-specific, but the following are defined by the ISO C / POSIX standard for `strftime` and are likely to be widely available. A *conversion specification* is introduced by `%`, usually followed by a single letter or `O` or `E` and then a single letter. Any character in the format string not part of a conversion specification is interpreted literally (and `%%` gives `%`). Widely implemented conversion specifications include

`%a` Abbreviated weekday name in the current locale. (Also matches full name on input.)

`%A` Full weekday name in the current locale. (Also matches abbreviated name on input.)

`%b` Abbreviated month name in the current locale. (Also matches full name on input.)

`%B` Full month name in the current locale. (Also matches abbreviated name on input.)

`%c` Date and time, locale-specific.

`%d` Day of the month as decimal number (01–31).

`%H` Hours as decimal number (00–23).

`%I` Hours as decimal number (01–12).

`%j` Day of year as decimal number (001–366).

`%m` Month as decimal number (01–12).

`%M` Minute as decimal number (00–59).

`%p` AM/PM indicator in the locale. Used in conjuction with `%I` and **not** with `%H`.

`%S` Second as decimal number (00–61), allowing for up to two leap-seconds (but POSIX-compliant OSes will ignore leap seconds).

`%U` Week of the year as decimal number (00–53) using the first Sunday as day 1 of week 1.

`%w` Weekday as decimal number (0–6, Sunday is 0).

`%W` Week of the year as decimal number (00–53) using the first Monday as day 1 of week 1.

`%x` Date, locale-specific.

`%X` Time, locale-specific.

`%y` Year without century (00–99). If you use this on input, which century you get is system-specific. So don't! Often values up to 69 (or 68) are prefixed by 20 and 70(or 69) to 99 by 19.

`%Y` Year with century.

`%z` (output only.) Offset from Greenwich, so `-0800` is 8 hours west of Greenwich.

`%Z` (output only.) Time zone as a character string (empty if not available).

Where leading zeros are shown they will be used on output but are optional on input.

Also defined in the current standards but less widely implemented (e.g. not for output on Windows) are

%C Century (00–99): the integer part of the year divided by 100.

%D Locale-specific date format such as `%m/%d/%y`: ISO C99 says it should be that exact format.

%e Day of the month as decimal number (1–31), with a leading space for a single-digit number.

%F Equivalent to %Y-%m-%d (the ISO 8601 date format).

%g The last two digits of the week-based year (see `%V`). (Typically accepted but ignored on input.)

%G The week-based year (see `%V`) as a decimal number. (Typically accepted but ignored on input.)

%h Equivalent to `%b`.

%k The 24-hour clock time with single digits preceded by a blank.

%l The 12-hour clock time with single digits preceded by a blank.

%n Newline on output, arbitrary whitespace on input.

%r The 12-hour clock time (using the locale's AM or PM).

%R Equivalent to `%H:%M`.

%t Tab on output, arbitrary whitespace on input.

%T Equivalent to `%H:%M:%S`.

%u Weekday as a decimal number (1–7, Monday is 1).

%V Week of the year as decimal number (00–53). If the week (starting on Monday) containing 1 January has four or more days in the new year, then it is considered week 1. Otherwise, it is the last week of the previous year, and the next week is week 1. (Typically accepted but ignored on input.)

For output (and possibly input) there are also `%O[dHImMUVwWy]` which may emit numbers in an alternative locale-dependent format (e.g. roman numerals), and `%E[cCyYxX]` which can use an alternative 'era' (e.g. a different religious calendar). Which of these are supported is OS-dependent.

Specific to R is `%OSn`, which for output gives the seconds to `0 <= n <= 6` decimal places (and if `%OS` is not followed by a digit, it uses the setting of `getOption("digits.secs")`, or if that is unset, `n = 3`). Further, for `strptime` `%OS` will input seconds including fractional seconds.

The behaviour of other conversion specifications (and even if other character sequences commencing with `%` *are* conversion specifications) is system-specific.

`ISOdatetime` and `ISOdate` are convenience wrappers for `strptime`, that differ only in their defaults and that `ISOdate` sets a timezone. (For dates without times it would be better to use the `"Date"` class.)

Value

The `format` methods and `strftime` return character vectors representing the time.

`strptime` turns character representations into an object of class `"POSIXlt"`. The timezone is used to set the `isdst` component.

`ISOdatetime` and `ISOdate` return an object of class `"POSIXct"`.

Note

The default formats follow the rules of the ISO 8601 international standard which expresses a day as "2001-02-28" and a time as "14:01:02" using leading zeroes as here. The ISO form uses no space to separate dates and times.

If the date string does not specify the date completely, the returned answer may be system-specific. The most common behaviour is to assume that unspecified seconds, minutes or hours are zero, and a missing year, month or day is the current one. If it specifies a date incorrectly, reliable implementations will give an error and the date is reported as NA. Unfortunately some common implementations (such as 'glibc') are unreliable and guess at the intended meaning.

If the timezone specified is invalid on your system, what happens is system-specific but it will probably be ignored.

OS facilities will probably not print years before 1CE (aka 1AD) correctly.

Remember that in most timezones some times do not occur and some occur twice because of transitions to/from summer time. What happens in those cases is OS-specific.

References

International Organization for Standardization (2004, 1988, 1997, ...) *ISO 8601. Data elements and interchange formats – Information interchange – Representation of dates and times.* For links to versions available on-line see (at the time of writing) http://www.qsl.net/g1smd/isopdf.htm; for information on the current official version, see http://www.iso.org/iso/en/prods-services/popstds/datesandtime.html.

See Also

DateTimeClasses for details of the date-time classes; locales to query or set a locale.

Your system's help pages on strftime and strptime to see how to specify their formats. (On some systems strptime is replaced by corrected code from 'glibc', when all the conversion specifications described here are supported, but with no alternative number representation nor era available in any locale.

Examples

```
## locale-specific version of date()
format(Sys.time(), "%a %b %d %X %Y %Z")

## time to sub-second accuracy (if supported by the OS)
format(Sys.time(), "%H:%M:%OS3")

## read in date info in format 'ddmmmyyyy'
## This will give NA(s) in some locales; setting the C locale
## as in the commented lines will overcome this on most systems.
## lct <- Sys.getlocale("LC_TIME"); Sys.setlocale("LC_TIME", "C")
x <- c("1jan1960", "2jan1960", "31mar1960", "30jul1960")
z <- strptime(x, "%d%b%Y")
## Sys.setlocale("LC_TIME", lct)
z

## read in date/time info in format 'm/d/y h:m:s'
dates <- c("02/27/92", "02/27/92", "01/14/92", "02/28/92", "02/01/92")
times <- c("23:03:20", "22:29:56", "01:03:30", "18:21:03", "16:56:26")
x <- paste(dates, times)
strptime(x, "%m/%d/%y %H:%M:%S")
```

```
## time with fractional seconds
z <- strptime("20/2/06 11:16:16.683", "%d/%m/%y %H:%M:%OS")
z # prints without fractional seconds
op <- options(digits.secs=3)
z
options(op)
```

`strsplit` *Split the Elements of a Character Vector*

Description

Split the elements of a character vector `x` into substrings according to the presence of substring `split` within them.

Usage

```
strsplit(x, split, extended = TRUE, fixed = FALSE, perl = FALSE)
```

Arguments

`x`	character vector, each element of which is to be split. Other inputs, including a factor, will give an error.
`split`	character vector (or object which can be coerced to such) containing regular expression(s) (unless `fixed = TRUE`) to use as "split". If empty matches occur, in particular if `split` has length 0, `x` is split into single characters. If `split` has length greater than 1, it is re-cycled along `x`.
`extended`	logical. If `TRUE`, extended regular expression matching is used, and if `FALSE` basic regular expressions are used.
`fixed`	logical. If `TRUE` match string exactly, otherwise use regular expressions. Has priority over `perl` and `extended`.
`perl`	logical. Should perl-compatible regexps be used? Has priority over `extended`.

Details

Argument `split` will be coerced to character, so you will see uses with `split = NULL` to mean `split = character(0)`, including in the examples below.

Note that spltting into single characters can be done *via* `split=character(0)` or `split=""`; the two are equivalent. The definition of 'character' here depends on the locale (and perhaps OS): in a single-byte locale it is a byte, and in a multi-byte locale it is the unit represented by a 'wide character' (almost always a Unicode point).

A missing value of `split` does not split the corresponding element(s) of `x` at all.

The algorithm applied to each input string is

```
    repeat {
        if the string is empty
            break.
        if there is a match
            add the string to the left of the match to the output.
```

```
            remove the match and all to the left of it.
        else
            add the string to the output.
            break.
    }
```

Note that this means that if there is a match at the beginning of a (non-empty) string, the first element of the output is `""`, but if there is a match at the end of the string, the output is the same as with the match removed.

Value

A list of length `length(x)` the `i`-th element of which contains the vector of splits of `x[i]`.

Warning

The standard regular expression code has been reported to be very slow when applied to extremely long character strings (tens of thousands of characters or more): the code used when `perl = TRUE` seems much faster and more reliable for such usages.

The `perl = TRUE` option is only implemented for single-byte and UTF-8 encodings, and will warn if used in a non-UTF-8 multibyte locale.

See Also

`paste` for the reverse, `grep` and `sub` for string search and manipulation; further `nchar`, `substr`.

regular expression for the details of the pattern specification.

Examples

```
noquote(strsplit("A text I want to display with spaces", NULL)[[1]])

x <- c(as = "asfef", qu = "qwerty", "yuiop[", "b", "stuff.blah.yech")
# split x on the letter e
strsplit(x,"e")

unlist(strsplit("a.b.c", "."))
## [1] "" "" "" "" ""
## Note that 'split' is a regexp!
## If you really want to split on '.', use
unlist(strsplit("a.b.c", "\\."))
## [1] "a" "b" "c"
## or
unlist(strsplit("a.b.c", ".", fixed = TRUE))

## a useful function: rev() for strings
strReverse <- function(x)
        sapply(lapply(strsplit(x, NULL), rev), paste, collapse="")
strReverse(c("abc", "Statistics"))

## get the first names of the members of R-core
a <- readLines(file.path(R.home("doc"),"AUTHORS"))[-(1:8)]
a <- a[(0:2)-length(a)]
(a <- sub(" .*","", a))
# and reverse them
```

```
strReverse(a)

## Note that final empty strings are not produced:
strsplit(paste(c("", "a", ""), collapse="#"), split="#")[[1]]
# [1] ""  "a"
## and also an empty string is only produced before a definite match:
strsplit("", " ")[[1]]    # character(0)
strsplit(" ", " ")[[1]]   # [1] ""
```

`strtrim` *Trim Character Strings to Specified Widths*

Description

Trim character strings to specified display widths.

Usage

```
strtrim(x, width)
```

Arguments

`x`	a character vector, or an object which can be coerced to a character vector by `as.character`.
`width`	Positive integer values: recycled to the length of `x`.

Details

'Width' is interpreted as the display width in a monospaced font. What happens with non-printable characters (such as backspace, tab) is implementation-dependent and may depend on the locale (e.g. they may be included in the count or they may be omitted).

Using this function rather than `substr` is important when there might be double-width characters in character vectors

Value

A character vector of the same length and with the same attributes as `x` (after possible coercion).

Examples

```
strtrim(c("abcdef", "abcdef", "abcdef"), c(1,5,10))
```

`structure` *Attribute Specification*

Description

`structure` returns the given object with further attributes set.

Usage

```
structure(.Data, ...)
```

Arguments

`.Data`	an object which will have various attributes attached to it.
`...`	attributes, specified in `tag=value` form, which will be attached to data.

Details

Adding a `tsp` attribute will ensure that the object returned inherits from class `"ts"`, and adding a `levels` attribute ensures that the returned object is a factor. These conversions are deprecated: they give a warning and will be removed in R 2.5.0.

For historical reasons (these names are used when deparsing), attributes `".Dim"`, `".Dimnames"`, `".Names"`, `".Tsp"` and `".Label"` are renammed to `"dim"`, `"dimnames"`, `"names"`, `"tsp"` and `"levels"`.

It is possible to give the same tag more than once, in which case the last value assigned wins. As with other ways of assigning attributes, using `tag=NULL` removes attribute `tag` from `.Data` if it is present.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`attributes`, `attr`.

Examples

```
structure(1:6, dim = 2:3)
```

strwrap *Wrap Character Strings to Format Paragraphs*

Description

Each character string in the input is first split into paragraphs (on lines containing whitespace only). The paragraphs are then formatted by breaking lines at word boundaries. The target columns for wrapping lines and the indentation of the first and all subsequent lines of a paragraph can be controlled independently.

Usage

```
strwrap(x, width = 0.9 * getOption("width"), indent = 0,
        exdent = 0, prefix = "", simplify = TRUE)
```

Arguments

x	a character vector, or an object which can be converted to a character vector by as.character.
width	a positive integer giving the target column for wrapping lines in the output.
indent	a non-negative integer giving the indentation of the first line in a paragraph.
exdent	a non-negative integer specifying the indentation of subsequent lines in paragraphs.
prefix	a character string to be used as prefix for each line.
simplify	a logical. If TRUE, the result is a single character vector of line text; otherwise, it is a list of the same length as x the elements of which are character vectors of line text obtained from the corresponding element of x. (Hence, the result in the former case is obtained by unlisting that of the latter.)

Details

Whitespace in the input is destroyed. Double spaces after periods (thought as representing sentence ends) are preserved. Currently, possible sentence ends at line breaks are not considered specially.

Indentation is relative to the number of characters in the prefix string.

Examples

```
## Read in file 'THANKS'.
x <- paste(readLines(file.path(R.home("doc"), "THANKS")), collapse = "\n")
## Split into paragraphs and remove the first three ones
x <- unlist(strsplit(x, "\n[ \t\n]*\n"))[-(1:3)]
## Join the rest
x <- paste(x, collapse = "\n\n")
## Now for some fun:
writeLines(strwrap(x, width = 60))
writeLines(strwrap(x, width = 60, indent = 5))
writeLines(strwrap(x, width = 60, exdent = 5))
writeLines(strwrap(x, prefix = "THANKS> "))

## Note that messages are wrapped AT the target column indicated by
## 'width' (and not beyond it).
```

```
## From an R-devel posting by J. Hosking <jh910@juno.com>.
x <- paste(sapply(sample(10, 100, rep=TRUE),
           function(x) substring("aaaaaaaaaa", 1, x)), collapse = " ")
sapply(10:40,
       function(m)
       c(target = m, actual = max(nchar(strwrap(x, m)))))
```

`subset` *Subsetting Vectors, Matrices and Data Frames*

Description

Return subsets of vectors, matrices or data frames which meet conditions.

Usage

```
subset(x, ...)

## Default S3 method:
subset(x, subset, ...)

## S3 method for class 'matrix':
subset(x, subset, select, drop = FALSE, ...)

## S3 method for class 'data.frame':
subset(x, subset, select, drop = FALSE, ...)
```

Arguments

`x`	object to be subsetted.
`subset`	logical expression indicating elements or rows to keep: missing values are taken as false.
`select`	expression, indicating columns to select from a data frame.
`drop`	passed on to `[` indexing operator.
`...`	further arguments to be passed to or from other methods.

Details

This is a generic function, with methods supplied for matrices, data frames and vectors (including lists). Packages and users can add further methods.

For ordinary vectors, the result is simply `x[subset & !is.na(subset)]`.

For data frames, the `subset` argument works on the rows. Note that `subset` will be evaluated in the data frame, so columns can be referred to (by name) as variables in the expression (see the examples).

The `select` argument exists only for the methods for data frames and matrices. It works by first replacing column names in the selection expression with the corresponding column numbers in the data frame and then using the resulting integer vector to index the columns. This allows the use of the standard indexing conventions so that for example ranges of columns can be specified easily, or single columns can be dropped (see the examples).

The `drop` argument is passed on to the indexing method for matrices and data frames: note that the default for matrices is different from that for indexing.

Value

An object similar to `x` contain just the selected elements (for a vector), rows and columns (for a matrix or data frame), and so on.

Author(s)

Peter Dalgaard and Brian Ripley

See Also

`[`, `transform`

Examples

```
subset(airquality, Temp > 80, select = c(Ozone, Temp))
subset(airquality, Day == 1, select = -Temp)
subset(airquality, select = Ozone:Wind)

with(airquality, subset(Ozone, Temp > 80))

## sometimes requiring a logical 'subset' argument is a nuisance
nm <- rownames(state.x77)
start_with_M <- nm %in% grep("^M", nm, value=TRUE)
subset(state.x77, start_with_M, Illiteracy:Murder)
```

`substitute` *Substituting and Quoting Expressions*

Description

`substitute` returns the parse tree for the (unevaluated) expression `expr`, substituting any variables bound in `env`.

`quote` simply returns its argument. The argument is not evaluated and can be any R expression.

Usage

```
substitute(expr, env)
quote(expr)
```

Arguments

`expr`	Any syntactically valid R expression
`env`	An environment or a list object. Defaults to the current evaluation environment.

Details

The typical use of substitute is to create informative labels for data sets and plots. The myplot example below shows a simple use of this facility. It uses the functions deparse and substitute to create labels for a plot which are character string versions of the actual arguments to the function myplot.

Substitution takes place by examining each component of the parse tree as follows: If it is not a bound symbol in env, it is unchanged. If it is a promise object, i.e., a formal argument to a function or explicitly created using delayedAssign(), the expression slot of the promise replaces the symbol. If it is an ordinary variable, its value is substituted, unless env is .GlobalEnv in which case the symbol is left unchanged.

substitute is a primitive function so positional matching is used and names of supplied arguments are ignored.

Value

The mode of the result is generally "call" but may in principle be any type. In particular, single-variable expressions have mode "name" and constants have the appropriate base mode.

Note

Substitute works on a purely lexical basis. There is no guarantee that the resulting expression makes any sense.

Substituting and quoting often causes confusion when the argument is expression(...). The result is a call to the expression constructor function and needs to be evaluated with eval to give the actual expression object.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

missing for argument "missingness", bquote for partial substitution, sQuote and dQuote for adding quotation marks to strings.

Examples

```
(s.e <- substitute(expression(a + b), list(a = 1)))  #> expression(1 + b)
(s.s <- substitute( a + b,            list(a = 1)))  #> 1 + b
c(mode(s.e), typeof(s.e)) #  "call", "language"
c(mode(s.s), typeof(s.s)) #   (the same)
# but:
(e.s.e <- eval(s.e))          #>  expression(1 + b)
c(mode(e.s.e), typeof(e.s.e)) #  "expression", "expression"

substitute(x <- x + 1, list(x=1)) # nonsense

myplot <- function(x, y)
    plot(x, y, xlab=deparse(substitute(x)),
         ylab=deparse(substitute(y)))

## Simple examples about lazy evaluation, etc:
```

```
f1 <- function(x, y = x)             { x <- x + 1; y }
s1 <- function(x, y = substitute(x)) { x <- x + 1; y }
s2 <- function(x, y) { if(missing(y)) y <- substitute(x); x <- x + 1; y }
a <- 10
f1(a)# 11
s1(a)# 11
s2(a)# a
typeof(s2(a))# "symbol"
```

`substr` *Substrings of a Character Vector*

Description

Extract or replace substrings in a character vector.

Usage

```
substr(x, start, stop)
substring(text, first, last = 1000000)
substr(x, start, stop) <- value
substring(text, first, last = 1000000) <- value
```

Arguments

`x, text`	a character vector.
`start, first`	integer. The first element to be replaced.
`stop, last`	integer. The last element to be replaced.
`value`	a character vector, recycled if necessary.

Details

`substring` is compatible with S, with `first` and `last` instead of `start` and `stop`. For vector arguments, it expands the arguments cyclically to the length of the longest *provided* none are of zero length.

When extracting, if `start` is larger than the string length then `""` is returned.

For the extraction functions, `x` or `text` will be converted to a character vector by `as.character` if it is not already one.

For the replacement functions, if `start` is larger than the string length then no replacement is done. If the portion to be replaced is longer than the replacement string, then only the portion the length of the string is replaced.

Value

For `substr`, a character vector of the same length and with the same attributes as `x` (after possible coercion).

For `substring`, a character vector of length the longest of the arguments. This will have names taken from `x` (if it has any after coercion, repeated as needed), and other attributes copied from `x` if it is the longest of the arguments).

Note

The S4 version of `substring<-` ignores `last`; this version does not.

These functions are often used with `nchar` to truncate a display. That does not really work (you want to limit the width, not the number of characters, so it would be better to use `strtrim`), but at least make sure you use `nchar(type="c")`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (`substring`.)

See Also

`strsplit`, `paste`, `nchar`.

Examples

```
substr("abcdef",2,4)
substring("abcdef",1:6,1:6)
## strsplit is more efficient ...

substr(rep("abcdef",4),1:4,4:5)
x <- c("asfef", "qwerty", "yuiop[", "b", "stuff.blah.yech")
substr(x, 2, 5)
substring(x, 2, 4:6)

substring(x, 2) <- c("..", "+++")
x
```

`sum` *Sum of Vector Elements*

Description

`sum` returns the sum of all the values present in its arguments.

Usage

```
sum(..., na.rm = FALSE)
```

Arguments

`...`	numeric or complex or logical vectors.
`na.rm`	logical. Should missing values be removed?

Details

This is a generic function: methods can be defined for it directly or via the `Summary` group generic. For this to work properly, the arguments `...` should be unnamed, and dispatch is on the first argument.

If `na.rm` is `FALSE` an `NA` value in any of the arguments will cause a value of `NA` to be returned, otherwise `NA` values are ignored.

Logical true values are regarded as one, false values as zero. For historical reasons, `NULL` is accepted and treated as if it were `integer(0)`.

Value

The sum. If all of `...` are of type integer or logical, then the sum is integer, and in that case the result will be `NA` (with a warning) if integer overflow occurs. Otherwise it is a length-one numeric or complex vector.

NB: the sum of an empty set is zero, by definition.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

`summary` *Object Summaries*

Description

`summary` is a generic function used to produce result summaries of the results of various model fitting functions. The function invokes particular `methods` which depend on the `class` of the first argument.

Usage

```
summary(object, ...)

## Default S3 method:
summary(object, ..., digits = max(3, getOption("digits")-3))
## S3 method for class 'data.frame':
summary(object, maxsum = 7,
        digits = max(3, getOption("digits")-3), ...)

## S3 method for class 'factor':
summary(object, maxsum = 100, ...)

## S3 method for class 'matrix':
summary(object, ...)
```

Arguments

`object`	an object for which a summary is desired.
`maxsum`	integer, indicating how many levels should be shown for `factor`s.
`digits`	integer, used for number formatting with `signif()` (for `summary.default`) or `format()` (for `summary.data.frame`).
`...`	additional arguments affecting the summary produced.

Details

For `factor`s, the frequency of the first `maxsum - 1` most frequent levels is shown, where the less frequent levels are summarized in `"(Others)"` (resulting in `maxsum` frequencies).

The functions `summary.lm` and `summary.glm` are examples of particular methods which summarise the results produced by `lm` and `glm`.

Value

The form of the value returned by `summary` depends on the class of its argument. See the documentation of the particular methods for details of what is produced by that method.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

See Also

`anova`, `summary.glm`, `summary.lm`.

Examples

```
summary(attenu, digits = 4) #-> summary.data.frame(...), default precision
summary(attenu $ station, maxsum = 20) #-> summary.factor(...)

lst <- unclass(attenu$station) > 20 # logical with NAs
## summary.default() for logicals -- different from *.factor:
summary(lst)
summary(as.factor(lst))
```

`svd` — *Singular Value Decomposition of a Matrix*

Description

Compute the singular-value decomposition of a rectangular matrix.

Usage

```
svd(x, nu = min(n, p), nv = min(n, p), LINPACK = FALSE)

La.svd(x, nu = min(n, p), nv = min(n, p))
```

Arguments

`x`	a real or complex matrix whose SVD decomposition is to be computed.
`nu`	the number of left singular vectors to be computed. This must between `0` and `n = nrow(x)`.
`nv`	the number of right singular vectors to be computed. This must be between `0` and `p = ncol(x)`.
`LINPACK`	logical. Should LINPACK be used (for compatibility with R < 1.7.0)? In this case `nu` must be `0`, `nrow(x)` or `ncol(x)`.

Details

The singular value decomposition plays an important role in many statistical techniques. `svd` and `La.svd` provide two slightly different interfaces. The main functions used are the LAPACK routines DGESDD and ZGESVD; `svd(LINPACK = TRUE)` provides an interface to the LINPACK routine DSVDC, purely for backwards compatibility.

Computing the singular vectors is the slow part for large matrices. The computation will be more efficient if `nu <= min(n, p)` and `nv <= min(n, p)`, and even more efficient if one or both are zero.

Unsuccessful results from the underlying LAPACK code will result in an error giving a positive error code: these can only be interpreted by detailed study of the FORTRAN code.

Value

The SVD decomposition of the matrix as computed by LAPACK/LINPACK,

$$\boldsymbol{X} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}',$$

where $\boldsymbol{U}$ and $\boldsymbol{V}$ are orthogonal, $\boldsymbol{V}'$ means V *transposed*, and $\boldsymbol{D}$ is a diagonal matrix with the singular values D_{ii}. Equivalently, $\boldsymbol{D} = \boldsymbol{U}'\boldsymbol{X}\boldsymbol{V}$, which is verified in the examples, below.

The returned value is a list with components

`d`	a vector containing the singular values of `x`, of length `min(n, p)`.
`u`	a matrix whose columns contain the left singular vectors of `x`, present if `nu > 0`. Dimension `c(n, nu)`.
`v`	a matrix whose columns contain the right singular vectors of `x`, present if `nv > 0`. Dimension `c(p, nv)`.

For `La.svd` the return value replaces `v` by `vt`, the (conjugated if complex) transpose of `v`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Dongarra, J. J., Bunch, J. R., Moler, C. B. and Stewart, G. W. (1978) *LINPACK Users Guide*. Philadelphia: SIAM Publications.

Anderson. E. and ten others (1999) *LAPACK Users' Guide*. Third Edition. SIAM.
Available on-line at http://www.netlib.org/lapack/lug/lapack_lug.html.

See Also

`eigen`, `qr`.

Examples

```
hilbert <- function(n) { i <- 1:n; 1 / outer(i - 1, i, "+") }
X <- hilbert(9)[,1:6]
(s <- svd(X))
D <- diag(s$d)
s$u %*% D %*% t(s$v) #  X = U D V'
t(s$u) %*% X %*% s$v #  D = U' X V
```

---

sweep *Sweep out Array Summaries*

Description

Return an array obtained from an input array by sweeping out a summary statistic.

Usage

```
sweep(x, MARGIN, STATS, FUN="-", ...)
```

Arguments

x	an array.
MARGIN	a vector of indices giving the extents of x which correspond to STATS.
STATS	the summary statistic which is to be swept out.
FUN	the function to be used to carry out the sweep. In the case of binary operators such as "/" etc., the function name must backquoted or quoted. (FUN is found by a call to match.fun.)
...	optional arguments to FUN.

Value

An array with the same shape as x, but with the summary statistics swept out.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

apply on which sweep used to be based; scale for centering and scaling.

Examples

```
require(stats) # for median
med.att <- apply(attitude, 2, median)
sweep(data.matrix(attitude), 2, med.att)# subtract the column medians
```

`switch` *Select One of a List of Alternatives*

Description

`switch` evaluates `EXPR` and accordingly chooses one of the further arguments (in `...`).

Usage

```
switch(EXPR, ...)
```

Arguments

`EXPR`	an expression evaluating to a number or a character string.
`...`	the list of alternatives, given explicitly.

Details

If the value of `EXPR` is an integer between 1 and `nargs()-1` then the corresponding element of `...` is evaluated and the result returned.

If `EXPR` returns a character string then that string is used to match the names of the elements in `...`. If there is an exact match then that element is evaluated and returned if there is one, otherwise the next element is chosen, e.g., `switch("cc", a=1, cc=, d=2)` evaluates to `2`.

In the case of no match, if there's a further argument in `switch` that one is returned, otherwise `NULL`.

Warning

Beware of partial matching: an alternative `E = foo` will match the first argument `EXPR` unless that is named. See the examples for good practice in naming the first argument.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
require(stats)
centre <- function(x, type) {
  switch(type,
         mean = mean(x),
         median = median(x),
         trimmed = mean(x, trim = .1))
}
x <- rcauchy(10)
centre(x, "mean")
centre(x, "median")
centre(x, "trimmed")

ccc <- c("b","QQ","a","A","bb")
for(ch in ccc)
```

```
    cat(ch,":",switch(EXPR = ch, a=1,     b=2:3), "\n")
for(ch in ccc)
    cat(ch,":",switch(EXPR = ch, a=, A=1, b=2:3, "Otherwise: last"),"\n")

## Numeric EXPR don't allow an 'otherwise':
for(i in c(-1:3,9))  print(switch(i, 1,2,3,4))
```

`Syntax` *Operator Syntax and Precedence*

Description

Outlines R syntax and gives the precedence of operators

Details

The following unary and binary operators are defined. They are listed in precedence groups, from highest to lowest.

`[ [[`	indexing
`::  :::`	access variables in a name space
`$ @`	component / slot extraction
`^`	exponentiation (right to left)
`- +`	unary minus and plus
`:`	sequence operator
`%any%`	special operators
`* /`	multiply, divide
`+ -`	(binary) add, subtract
`< > <= >= == !=`	ordering and comparison
`!`	negation
`& &&`	and
`| ||`	or
`~`	as in formulae
`-> ->>`	rightwards assignment
`=`	assignment (right to left)
`<- <<-`	assignment (right to left)
`?`	help (unary and binary)

Within an expression operators of equal precedence are evaluated from left to right except where indicated.

The links in the **See Also** section covers most other aspects of the basic syntax.

Note

There are substantial precedence differences between R and S. In particular, in S `?` has the same precedence as (binary) `+ -` and `& && | ||` have equal precedence.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

Arithmetic, Comparison, Control, Extract, Logic, NumericConstants, Paren, Quotes.

The *R Language Definition* manual.

`Sys.getenv` *Get Environment Variables*

Description

`Sys.getenv` obtains the values of the environment variables named by `x`.

Usage

```
Sys.getenv(x)
```

Arguments

`x`	a character vector, or missing

Value

A vector of the same length as `x`, with the variable names as its `names` attribute. Each element holds the value of the environment variable named by the corresponding component of `x` (or `""` if no environment variable with that name was found).

On most platforms `Sys.getenv()` will return a named vector giving the values of all the environment variables.

See Also

`Sys.putenv`, `Sys.getlocale` for the locale "environment", `getwd` for the working directory.

Examples

```
## whether HOST is set will be shell-dependent e.g. Solaris' csh does not.
Sys.getenv(c("R_HOME", "R_PAPERSIZE", "R_PRINTCMD", "HOST"))

str(s <- Sys.getenv()) # all settings (rather do not print)

## Language and Locale settings -- but rather use Sys.getlocale()
s[grep("^L(C|ANG)", names(s))]
```

`Sys.info` *Extract System and User Information*

Description

Reports system and user information.

Usage

```
Sys.info()
```

Details

This function is not implemented on all R platforms, and returns `NULL` when not available. Where possible it is based on POSIX system calls.

`Sys.info()` returns details of the platform R is running on, whereas `R.version` gives details of the platform R was built on: they may well be different.

Value

A character vector with fields

`sysname`	The operating system.
`release`	The OS release.
`version`	The OS version.
`nodename`	A name by which the machine is known on the network (if any).
`machine`	A concise description of the hardware.
`login`	The user's login name, or `"unknown"` if it cannot be ascertained.
`user`	The name of the real user ID, or `"unknown"` if it cannot be ascertained.

The first five fields come from the `uname(2)` system call. The login name comes from `getlogin(2)`, and the user name from `getpwuid(getuid())`

Note

The meaning of OS "release" and "version" is highly system-dependent and there is no guarantee that the node or login or user names will be what you might reasonably expect. (In particular on some Linux distributions the login name is unknown from sessions with re-directed inputs.)

See Also

`.Platform`, and `R.version`.

Examples

```
Sys.info()
## An alternative (and probably better) way to get the login name on Unix
Sys.getenv("LOGNAME")
```

`sys.parent` *Functions to Access the Function Call Stack*

Description

These functions provide access to `environment`s ("frames" in S terminology) associated with functions further up the calling stack.

Usage

```
sys.call(which = 0)
sys.frame(which = 0)
sys.nframe()
sys.function(which = 0)
sys.parent(n = 1)

sys.calls()
sys.frames()
sys.parents()
sys.on.exit()
sys.status()
parent.frame(n = 1)
```

Arguments

`which`	the frame number if non-negative, the number of frames to go back if negative.
`n`	the number of generations to go back. (See the Details section.)

Details

`.GlobalEnv` is given number 0 in the list of frames. Each subsequent function evaluation increases the frame stack by 1 and the call, function definition and the environment for evaluation of that function are returned by `sys.call`, `sys.function` and `sys.frame` with the appropriate index.

`sys.call`, `sys.frame` and `sys.function` accept integer values for the argument `which`. Non-negative values of `which` are frame numbers whereas negative values are counted back from the frame number of the current evaluation.

The parent frame of a function evaluation is the environment in which the function was called. It is not necessarily numbered one less than the frame number of the current evaluation, nor is it the environment within which the function was defined. `sys.parent` returns the number of the parent frame if `n` is 1 (the default), the grandparent if `n` is 2, and so on.

`sys.nframe` returns an integer, the number of the current frame as described in the first paragraph.

`sys.calls` and `sys.frames` give a pairlist of all the active calls and frames, repectively, and `sys.parents` returns an integer vector of indices of the parent frames of each of those frames.

Notice that even though the `sys.`*xxx* functions (except `sys.status`) are interpreted, their contexts are not counted nor are they reported. There is no access to them.

`sys.status()` returns a list with components `sys.calls`, `sys.parents` and `sys.frames`, the results of calls to those three functions (which this will include the call to `sys.status`: see the first example).

`sys.on.exit()` returns the expression stored for use by `on.exit` in the function currently being evaluated. (Note that this differs from S, which returns a list of expressions for the current frame and its parents.)

`parent.frame(n)` is a convenient shorthand for `sys.frame(sys.parent(n))` (implemented slightly more efficiently).

Value

`sys.call` returns a call, `sys.function` a function definition, and `sys.frame`, `sys.parent` and `parent.frame` return an environment.

For the other functions, see the Details section.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (not `parent.frame`.)

See Also

`eval` for a usage of `sys.frame` and `parent.frame`.

Examples

```
## Note: the first two examples will give different results
## if run by example().
ff <- function(x) gg(x)
gg <- function(y) sys.status()
str(ff(1))

gg <- function(y) {
    ggg <- function() {
        cat("current frame is", sys.nframe(), "\n")
        cat("parents are", sys.parents(), "\n")
        print(sys.function(0)) # ggg
        print(sys.function(2)) # gg
    }
    if(y > 0) gg(y-1) else ggg()
}
gg(3)

t1 <- function() {
  aa <- "here"
  t2 <- function() {
    ## in frame 2 here
    cat("current frame is", sys.nframe(), "\n")
    str(sys.calls()) ## list with two components t1() and t2()
    cat("parents are frame numbers", sys.parents(), "\n") ## 0 1
    print(ls(envir=sys.frame(-1))) ## [1] "aa" "t2"
    invisible()
  }
  t2()
}
t1()

test.sys.on.exit <- function() {
```

```
  on.exit(print(1))
  ex <- sys.on.exit()
  str(ex)
  cat("exiting...\n")
}
test.sys.on.exit()
## gives 'language print(1)', prints 1 on exit

## An example where the parent is not the next frame up the stack
## since method dispatch uses a frame.
as.double.foo <- function(x)
{
    str(sys.calls())
    print(sys.frames())
    print(sys.parents())
    print(sys.frame(-1)); print(parent.frame())
    x
}
t2 <- function(x) as.double(x)
a <- structure(pi, class = "foo")
t2(a)
```

`Sys.putenv` *Set Environment Variables*

Description

`putenv` sets environment variables (for other processes called from within R or future calls to `Sys.getenv` from this R process).

Usage

```
Sys.putenv(...)
```

Arguments

`...`	arguments in `name=value` form, with `value` coercible to a character string.

Details

Non-standard R names must be quoted: see the Examples section.

Value

A logical vector of the same length as `x`, with elements being true if setting the corresponding variable succeeded.

Note

Not all systems need support `Sys.putenv` (although all known current platforms do).

See Also

`Sys.getenv`, Startup for ways to set environment variables for the R session.

`setwd` for the working directory.

Examples

```
print(Sys.putenv(R_TEST="testit", "A+C"=123))
Sys.getenv("R_TEST")
```

Sys.sleep *Suspend Execution for a Time Interval*

Description

Suspend execution of R expressions for a given number of seconds

Usage

```
Sys.sleep(time)
```

Arguments

`time`	The time interval to suspend execution for, in seconds.

Details

Using this function allows R to be given very low priority and hence not to interfere with more important foreground tasks. A typical use is to allow a process launched from R to set itself up and read its input files before R execution is resumed.

The intention is that this function suspends execution of R expressions but wakes the process up often enough to respond to GUI events, typically every 0.5 seconds.

There is no guarantee that the process will sleep for the whole of the specified interval, and it may well take slightly longer in real time to resume execution. The resolution of the time interval is system-dependent, but will normally be down to 0.02 secs or better. (On modern Unix-alikes it will be better than 1ms.)

Value

Invisible `NULL`.

Note

This function may not be implemented on all systems.

Examples

```
testit <- function(x)
{
    p1 <- proc.time()
    Sys.sleep(x)
    proc.time() - p1 # The cpu usage should be negligible
}
testit(3.7)
```

sys.source *Parse and Evaluate Expressions from a File*

Description

Parses expressions in the given file, and then successively evaluates them in the specified environment.

Usage

```
sys.source(file, envir = baseenv(), chdir = FALSE,
           keep.source = getOption("keep.source.pkgs"))
```

Arguments

file	a character string naming the file to be read from
envir	an R object specifying the environment in which the expressions are to be evaluated. May also be a list or an integer. The default value NULL corresponds to evaluation in the base environment. This is probably not what you want; you should typically supply an explicit envir argument.
chdir	logical; if TRUE, the R working directory is changed to the directory containing file for evaluating.
keep.source	logical. If TRUE, functions "keep their source" including comments, see options(keep.source = *) for more details.

Details

For large files, keep.source = FALSE may save quite a bit of memory. In order for the code being evaluated to use the correct environment (for example, in global assignments), source code in packages should call topenv(), which will return the namespace, if any, the environment set up by sys.source, or the global environment if a saved image is being used.

See Also

source, and library which uses sys.source.

Sys.time *Get Current Date, Time and Timezone*

Description

Sys.time and Sys.Date returns the system's idea of the current date with and without time, and Sys.timezone returns the current time zone.

Usage

```
Sys.time()
Sys.Date()
Sys.timezone()
```

Details

`Sys.time` returns an absolute date-time value which can be converted in various time zones and may return different days.

`Sys.Date` returns the day in the current timezone.

Value

`Sys.time` returns an object of class `"POSIXct"` (see DateTimeClasses). On some systems it will have sub-second accuracy, but on others it will increment in seconds. On systems conforming to POSIX 1003.1-2001 the time will be reported in microsecond increments.

`Sys.Date` returns an object of class `"Date"` (see Date).

`Sys.timezone` returns an OS-specific character string, possibly an empty string. It may be possible to set the timezone via the environment variable `"TZ"`: see `as.POSIXlt`.

See Also

`date` for the system time in a fixed-format character string; the elapsed time component of `proc.time` for possibly finer resolution in changes in time.

Examples

```
Sys.time()
## print with possibly greater accuracy:
op <- options(digits.secs=6)
Sys.time()
options(op)

## locale-specific version of date()
format(Sys.time(), "%a %b %d %X %Y")

Sys.Date()

Sys.timezone()
```

---

system *Invoke a System Command*

Description

`system` invokes the OS command specified by `command`.

Usage

```
system(command, intern = FALSE, ignore.stderr = FALSE)
```

Arguments

`command`	the system command to be invoked, as a string.
`intern`	a logical (not `NA`) which indicates whether to make the output of the command an R object. Not available unless `popen` is supported on the platform.
`ignore.stderr`	a logical indicating whether error messages (written to 'stderr') should be ignored.

Details

If intern is TRUE then popen is used to invoke the command and the output collected, line by line, into an R character vector which is returned as the value of system. Output lines of more than 8095 characters will be split.

If intern is FALSE then the C function system is used to invoke the command and the value returned by system is the exit status of this function.

Error messages written to 'stderr' will be sent by the shell to the terminal unless ignore.stderr = TRUE. They can be captured (in the most likely shells) by

```
system("some command 2>&1", intern=TRUE)
```

unix is a *deprecated* alternative, available for backwards compatibility.

Value

If intern = TRUE, a character vector giving the output of the command, one line per character string. If the command could not be run or gives an error this will be reported on the shell's 'stderr' (unless popen is not supported, when there is an R error). (Output lines of more than 8095 characters will be split.)

If intern = FALSE, the return value is a system error code (0 for success).

See Also

.Platform for platform specific variables.

Examples

```
# list all files in the current directory using the -F flag
## Not run: system("ls -F")

# t1 is a character vector, each one
# representing a separate line of output from who
# (if the platform has popen and who)
t1 <- try(system("who", TRUE))

try(system("ls fizzlipuzzli", TRUE, TRUE))
# empty since file doesn't exist
```

---

system.file *Find Names of R System Files*

Description

Finds the full file names of files in packages etc.

Usage

```
system.file(..., package = "base", lib.loc = NULL)
```

Arguments

`...`	character strings, specifying subdirectory and file(s) within some package. The default, none, returns the root of the package. Wildcards are not supported.
`package`	a character string with the name of a single package. An error occurs if more than one package name is given.
`lib.loc`	a character vector with path names of R libraries, or `NULL`. The default value of `NULL` corresponds to all libraries currently known. If the default is used, the loaded packages are searched before the libraries.

Value

A character vector of positive length, containing the file names that matched `...`, or the empty string, `""`, if none matched. If matching the root of a package, there is no trailing separator.

As a special case, `system.file()` gives the root of the **base** package only.

See Also

`list.files`

Examples

```
system.file()                   # The root of the 'base' package
system.file(package = "stats") # The root of package 'stats'
system.file("INDEX")
system.file("help", "AnIndex", package = "splines")
```

---

system.time *CPU Time Used*

Description

Return CPU (and other) times that `expr` used.

Usage

```
system.time(expr, gcFirst = TRUE)
unix.time(expr, gcFirst = TRUE)
```

Arguments

`expr`	Valid R expression to be "timed"
`gcFirst`	Logical - should a garbage collection be performed immediately before the timing? Default is `TRUE`.

Details

system.time calls the builtin proc.time, evaluates expr, and then calls proc.time once more, returning the difference between the two proc.time calls.

unix.time is an alias of system.time, for compatibility reasons.

Timings of evaluations of the same expression can vary considerably depending on whether the evaluation triggers a garbage collection. When gcFirst is TRUE a garbage collection (gc) will be performed immediately before the evaluation of expr. This will usually produce more consistent timings.

Value

A numeric vector of length 5 containing the user cpu, system cpu, elapsed, subproc1, subproc2 times. The subproc times are the user and system cpu time used by child processes (and so are usually zero).

The resolution of the times will be system-specific; see proc.time for details.

Note

It is possible to compile R without support for system.time, when the function will throw an error.

See Also

proc.time, time which is for time series.

Examples

```
require(stats)
system.time(for(i in 1:100) mad(runif(1000)))
## Not run:
exT <- function(n = 1000) {
  # Purpose: Test if system.time works ok;   n: loop size
  system.time(for(i in 1:n) x <- mean(rt(1000, df=4)))
}
#-- Try to interrupt one of the following (using Ctrl-C / Escape):
exT()                  #- about 3 secs on a 1GHz PIII
system.time(exT())     #~ +/- same
## End(Not run)
```

---

t *Matrix Transpose*

Description

Given a matrix or data.frame x, t returns the transpose of x.

Usage

```
t(x)
```

Arguments

x	a matrix or data frame, typically.

Details

This is a generic function for which methods can be written. The description here applies to the default and "data.frame" methods.

A data frame is first coerced to a matrix: see as.matrix. When x is a vector, it is treated as "column", i.e., the result is a 1-row matrix.

Value

A matrix, with dim and dimnames constructed appropriately from those of x, and other attributes except names copied across.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

aperm for permuting the dimensions of arrays.

Examples

```
a <- matrix(1:30, 5,6)
ta <- t(a) ##-- i.e.,  a[i, j] == ta[j, i] for all i,j :
for(j in seq(ncol(a)))
  if(! all(a[, j] == ta[j, ])) stop("wrong transpose")
```

---

table *Cross Tabulation and Table Creation*

Description

table uses the cross-classifying factors to build a contingency table of the counts at each combination of factor levels.

Usage

```
table(..., exclude = c(NA, NaN), dnn = list.names(...),
      deparse.level = 1)
as.table(x, ...)
is.table(x)

## S3 method for class 'table':
as.data.frame(x, row.names = NULL, ...,
              responseName = "Freq")
```

Arguments

`...`	objects which can be interpreted as factors (including character strings), or a list (or data frame) whose components can be so interpreted. (For `as.table` and `as.data.frame`, arguments passed to specific methods.)
`exclude`	values to use in the exclude argument of `factor` when interpreting non-factor objects; if specified, levels to remove from all factors in `...`.
`dnn`	the names to be given to the dimensions in the result (the *dimnames names*).
`deparse.level`	controls how the default `dnn` is constructed. See details.
`x`	an arbitrary R object, or an object inheriting from class `"table"` for the `as.data.frame` method.
`row.names`	a character vector giving the row names for the data frame.
`responseName`	The name to be used for the column of table entries, usually counts.

Details

If the argument `dnn` is not supplied, the internal function `list.names` is called to compute the 'dimname names'. If the arguments in `...` are named, those names are used. For the remaining arguments, `deparse.level = 0` gives an empty name, `deparse.level = 1` uses the supplied argument if it is a symbol, and `deparse.level = 2` will deparse the argument.

Only when `exclude` is specified (i.e., not by default), will `table` drop levels of factor arguments potentially.

Value

`table()` returns a *contingency table*, an object of `class` `"table"`, an array of integer values.

There is a `summary` method for objects created by `table` or `xtabs`, which gives basic information and performs a chi-squared test for independence of factors (note that the function `chisq.test` currently only handles 2-d tables).

`as.table` and `is.table` coerce to and test for contingency table, respectively.

The `as.data.frame` method for objects inheriting from class `"table"` can be used to convert the array-based representation of a contingency table to a data frame containing the classifying factors and the corresponding entries (the latter as component named by `responseName`). This is the inverse of `xtabs`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

Use `ftable` for printing (and more) of multidimensional tables. `margin.table`, `prop.table`, `addmargins`.

Examples

```
require(stats) # for rpois and xtabs
## Simple frequency distribution
table(rpois(100,5))
## Check the design:
with(warpbreaks, table(wool, tension))
table(state.division, state.region)

# simple two-way contingency table
with(airquality, table(cut(Temp, quantile(Temp)), Month))

a <- letters[1:3]
table(a, sample(a))                     # dnn is c("a", "")
table(a, sample(a), deparse.level = 0) # dnn is c("", "")
table(a, sample(a), deparse.level = 2) # dnn is c("a", "sample(a)")

## xtabs() <-> as.data.frame.table() :
UCBAdmissions ## already a contingency table
DF <- as.data.frame(UCBAdmissions)
class(tab <- xtabs(Freq ~ ., DF)) # xtabs & table
## tab *is* "the same" as the original table:
all(tab == UCBAdmissions)
all.equal(dimnames(tab), dimnames(UCBAdmissions))

a <- rep(c(NA, 1/0:3), 10)
table(a)
table(a, exclude=NULL)
b <- factor(rep(c("A","B","C"), 10))
table(b)
table(b, exclude="B")
d <- factor(rep(c("A","B","C"), 10), levels=c("A","B","C","D","E"))
table(d, exclude="B")
print(table(b,d), zero.print = ".")

## NA counting:
is.na(d) <- 3:4
d <- factor(d, exclude=NULL)
d[1:7]
table(d, exclude = NULL)
```

---

tabulate *Tabulation for Vectors*

Description

`tabulate` takes the integer-valued vector `bin` and counts the number of times each integer occurs in it.

Usage

```
tabulate(bin, nbins = max(1, bin))
```

Arguments

`bin`	a numeric vector (of positive integers), or a factor.
`nbins`	the number of bins to be used.

Details

`tabulate` is used as the basis of the `table` function.

If `bin` is a factor, its internal integer representation is tabulated.

If the elements of `bin` are numeric but not integers, they are truncated to the nearest integer.

Value

An integer vector (without names). There is a bin for each of the values `1, ..., nbins`; values outside that range are (silently) ignored.

See Also

`table`, `factor`.

Examples

```
tabulate(c(2,3,5))
tabulate(c(2,3,3,5), nbins = 10)
tabulate(c(-2,0,2,3,3,5))  # -2 and 0 are ignored
tabulate(c(-2,0,2,3,3,5), nbins = 3)
tabulate(factor(letters[1:10]))
```

`tapply` *Apply a Function Over a "Ragged" Array*

Description

Apply a function to each cell of a ragged array, that is to each (non-empty) group of values given by a unique combination of the levels of certain factors.

Usage

```
tapply(X, INDEX, FUN = NULL, ..., simplify = TRUE)
```

Arguments

`X`	an atomic object, typically a vector.
`INDEX`	list of factors, each of same length as `X`.
`FUN`	the function to be applied. In the case of functions like `+`, `%*%`, etc., the function name must be quoted. If `FUN` is `NULL`, tapply returns a vector which can be used to subscript the multi-way array `tapply` normally produces.
`...`	optional arguments to `FUN`.
`simplify`	If `FALSE`, `tapply` always returns an array of mode `"list"`. If `TRUE` (the default), then if `FUN` always returns a scalar, `tapply` returns an array with the mode of the scalar.

Value

When FUN is present, tapply calls FUN for each cell that has any data in it. If FUN returns a single atomic value for each cell (e.g., functions mean or var) and when simplify is TRUE, tapply returns a multi-way array containing the values. The array has the same number of dimensions as INDEX has components; the number of levels in a dimension is the number of levels (nlevels()) in the corresponding component of INDEX.

Note that contrary to S, simplify = TRUE always returns an array, possibly 1-dimensional.

If FUN does not return a single atomic value, tapply returns an array of mode list whose components are the values of the individual calls to FUN, i.e., the result is a list with a dim attribute.

Note that optional arguments to FUN supplied by the ... argument are not divided into cells. It is therefore inappropriate for FUN to expect additional arguments with the same length as X.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

the convenience functions by and aggregate (using tapply); apply, lapply with its versions sapply and mapply.

Examples

```
require(stats)
groups <- as.factor(rbinom(32, n = 5, p = .4))
tapply(groups, groups, length) #- is almost the same as
table(groups)

## contingency table from data.frame : array with named dimnames
tapply(warpbreaks$breaks, warpbreaks[,-1], sum)
tapply(warpbreaks$breaks, warpbreaks[, 3, drop = FALSE], sum)

n <- 17; fac <- factor(rep(1:3, len = n), levels = 1:5)
table(fac)
tapply(1:n, fac, sum)
tapply(1:n, fac, sum, simplify = FALSE)
tapply(1:n, fac, range)
tapply(1:n, fac, quantile)

## example of ... argument: find quarterly means
tapply(presidents, cycle(presidents), mean, na.rm = TRUE)

ind <- list(c(1, 2, 2), c("A", "A", "B"))
table(ind)
tapply(1:3, ind) #-> the split vector
tapply(1:3, ind, sum)
```

`taskCallback` *Add or remove a top-level task callback*

Description

`addTaskCallback` registers an R function that is to be called each time a top-level task is completed.

`removeTaskCallback` un-registers a function that was registered earlier via `addTaskCallback`.

These provide low-level access to the internal/native mechanism for managing task-completion actions. One can use `taskCallbackManager` at the S-language level to manage S functions that are called at the completion of each task. This is easier and more direct.

Usage

```
addTaskCallback(f, data = NULL, name = character(0))
removeTaskCallback(id)
```

Arguments

`f`	the function that is to be invoked each time a top-level task is successfully completed. This is called with 5 or 4 arguments depending on whether `data` is specified or not, respectively. The return value should be a logical value indicating whether to keep the callback in the list of active callbacks or discard it.
`data`	if specified, this is the 5-th argument in the call to the callback function `f`.
`id`	a string or an integer identifying the element in the internal callback list to be removed. Integer indices are 1-based, i.e the first element is 1. The names of currently registered handlers is available using `getTaskCallbackNames` and is also returned in a call to `addTaskCallback`.
`name`	character: names to be used.

Details

Top-level tasks are individual expressions rather than entire lines of input. Thus an input line of the form `expression1 ; expression2` will give rise to 2 top-level tasks.

A top-level task callback is called with the expression for the top-level task, the result of the top-level task, a logical value indicating whether it was successfully completed or not (always TRUE at present), and a logical value indicating whether the result was printed or not. If the `data` argument was specified in the call to `addTaskCallback`, that value is given as the fifth argument.

The callback function should return a logical value. If the value is FALSE, the callback is removed from the task list and will not be called again by this mechanism. If the function returns TRUE, it is kept in the list and will be called on the completion of the next top-level task.

Value

`addTaskCallback` returns an integer value giving the position in the list of task callbacks that this new callback occupies. This is only the current position of the callback. It can be used to remove the entry as long as no other values are removed from earlier positions in the list first.

`removeTaskCallback` returns a logical value indicating whether the specified element was removed. This can fail (i.e., return `FALSE`) if an incorrect name or index is given that does not correspond to the name or position of an element in the list.

Note

This is an experimental feature and the interface may be changed in the future.

There is also C-level access to top-level task callbacks to allow C routines rather than R functions be used.

See Also

`getTaskCallbackNames` `taskCallbackManager` `http://developer.r-project.org/TaskHandlers.pdf`

Examples

```
times <- function(total = 3, str="Task a") {
  ctr <- 0

  function(expr, value, ok, visible) {
   ctr <<- ctr + 1
   cat(str, ctr, "\n")
   if(ctr == total) {
     cat("handler removing itself\n")
   }
   return(ctr < total)
  }
}

# add the callback that will work for
# 4 top-level tasks and then remove itself.
n <- addTaskCallback(times(4))

# now remove it, assuming it is still first in the list.
removeTaskCallback(n)

## Not run:
# There is no point in running this
# as
 addTaskCallback(times(4))

 sum(1:10)
 sum(1:10)
 sum(1:10)
 sum(1:10)
 sum(1:10)
## End(Not run)
```

`taskCallbackManager`

Create an R-level task callback manager

Description

This provides an entirely S-language mechanism for managing callbacks or actions that are invoked at the conclusion of each top-level task. Essentially, we register a single R function from this manager with the underlying, native task-callback mechanism and this function handles invoking the other R callbacks under the control of the manager. The manager consists of a collection of functions that access shared variables to manage the list of user-level callbacks.

Usage

```
taskCallbackManager(handlers = list(), registered = FALSE,
                    verbose = FALSE)
```

Arguments

`handlers`	this can be a list of callbacks in which each element is a list with an element named `"f"` which is a callback function, and an optional element named `"data"` which is the 5-th argument to be supplied to the callback when it is invoked. Typically this argument is not specified, and one uses `add` to register callbacks after the manager is created.
`registered`	a logical value indicating whether the `evaluate` function has already been registered with the internal task callback mechanism. This is usually `FALSE` and the first time a callback is added via the `add` function, the `evaluate` function is automatically registered. One can control when the function is registered by specifying `TRUE` for this argument and calling `addTaskCallback` manually.
`verbose`	a logical value, which if `TRUE`, causes information to be printed to the console about certain activities this dispatch manager performs. This is useful for debugging callbacks and the handler itself.

Value

A list containing 6 functions:

`add`	register a callback with this manager, giving the function, an optional 5-th argument, an optional name by which the callback is stored in the list, and a `register` argument which controls whether the `evaluate` function is registered with the internal C-level dispatch mechanism if necessary.
`remove`	remove an element from the manager's collection of callbacks, either by name or position/index.
`evaluate`	the 'real' callback function that is registered with the C-level dispatch mechanism and which invokes each of the R-level callbacks within this manager's control.
`suspend`	a function to set the suspend state of the manager. If it is suspended, none of the callbacks will be invoked when a task is completed. One sets the state by specifying a logical value for the `status` argument.

register	a function to register the evaluate function with the internal C-level dispatch mechanism. This is done automatically by the add function, but can be called manually.
callbacks	returns the list of callbacks being maintained by this manager.

Note

This is an experimental feature and the interface may be changed in the future.

See Also

addTaskCallback, removeTaskCallback, getTaskCallbackNames\ http://developer.r-project.org/TaskHandlers.pdf

Examples

```
# create the manager
h <- taskCallbackManager()

# add a callback
h$add(function(expr, value, ok, visible) {
                        cat("In handler\n")
                        return(TRUE)
                      }, name = "simpleHandler")

# look at the internal callbacks.
getTaskCallbackNames()

# look at the R-level callbacks
names(h$callback())

getTaskCallbackNames()
removeTaskCallback("R-taskCallbackManager")
```

---

taskCallbackNames *Query the names of the current internal top-level task callbacks*

Description

This provides a way to get the names (or identifiers) for the currently registered task callbacks that are invoked at the conclusion of each top-level task. These identifiers can be used to remove a callback.

Usage

```
getTaskCallbackNames()
```

Value

A character vector giving the name for each of the registered callbacks which are invoked when a top-level task is completed successfully. Each name is the one used when registering the callbacks and returned as the in the call to addTaskCallback.

Note

One can use taskCallbackManager to manage user-level task callbacks, i.e., S-language functions, entirely within the S language and access the names more directly.

See Also

addTaskCallback, removeTaskCallback, taskCallbackManager\ http://developer.r-project.org/TaskHandlers.pdf

Examples

```
n <- addTaskCallback(function(expr, value, ok, visible) {
                       cat("In handler\n")
                       return(TRUE)
                     }, name = "simpleHandler")

getTaskCallbackNames()

  # now remove it by name
removeTaskCallback("simpleHandler")

h <- taskCallbackManager()
h$add(function(expr, value, ok, visible) {
                       cat("In handler\n")
                       return(TRUE)
                     }, name = "simpleHandler")
getTaskCallbackNames()
removeTaskCallback("R-taskCallbackManager")
```

---

tempfile *Create Names for Temporary Files*

Description

tempfile returns a vector of character strings which can be used as names for temporary files.

Usage

```
tempfile(pattern = "file", tmpdir = tempdir())
tempdir()
```

Arguments

pattern	a non-empty character vector giving the initial part of the name.
tmpdir	a non-empty character vector giving the directory name

Details

If `pattern` has length greater than one then the result is of the same length giving a temporary file name for each component of `pattern`.

The names are very likely to be unique among calls to `tempfile` in an R session and across simultaneous R sessions. The filenames are guaranteed not to be currently in use.

The file name is made of the pattern and a random suffix in hex. By default, the filenames will be in the directory given by `tempdir()`. This will be a subdirectory of the temporary directory found by the following rule. The environment variables `TMPDIR`, `TMP` and `TEMP` are checked in turn and the first found which points to a writable directory is used: if none succeeds '/tmp' is used.

Value

For `tempfile` a character vector giving the names of possible (temporary) files. Note that no files are generated by `tempfile`.

For `tempdir`, the path of the per-session temporary directory.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`unlink` for deleting files.

Examples

```
tempfile(c("ab", "a b c"))   # give file name with spaces in!

tempdir() # working on all platforms with quite platform dependent result
```

`textConnection` *Text Connections*

Description

Input and output text connections.

Usage

```
textConnection(object, open = "r", local = FALSE)

textConnectionValue(con)
```

Arguments

object	character. A description of the connection. For an input this is an R character vector object, and for an output connection the name for the R character vector to receive the output, or NULL (for none).
open	character. Either "r" (or equivalently "") for an input connection or "w" or "a" for an output connection.
local	logical. Used only for output connections. If TRUE, output is assigned to a variable in the calling environment. Otherwise the global environment is used.
con	An output text connection.

Details

An input text connection is opened and the character vector is copied at time the connection object is created, and close destroys the copy.

An output text connection is opened and creates an R character vector of the given name in the user's workspace or in the calling environment, depending on the value of the local argument. This object will at all times hold the completed lines of output to the connection, and isIncomplete will indicate if there is an incomplete final line. Closing the connection will output the final line, complete or not. (A line is complete once it has been terminated by end-of-line, represented by "\n" in R.) The output character vector has locked bindings (see lockBinding) until close is called on the connection. The character vector can also be retrieved *via* textConnectionValue, which is the only way to do so if object = NULL.

Opening a text connection with mode = "a" will attempt to append to an existing character vector with the given name in the user's workspace or the calling environment. If none is found (even if an object exists of the right name but the wrong type) a new character vector will be created, with a warning.

You cannot seek on a text connection, and seek will always return zero as the position.

Value

For textConnection, a connection object of class "textConnection" which inherits from class "connection".

For textConnectionValue, a character vector.

Note

As output text connections keep the character vector up to date line-by-line, they are relatively expensive to use, and it is often better to use an anonymous file() connection to collect output.

On platforms where vsnprintf does not return the needed length of output (e.g., Windows) there is a 100,000 character limit on the length of line for output connections: longer lines will be truncated with a warning.

References

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language.* Springer.
[S has input text conections only.]

See Also

connections, showConnections, pushBack, capture.output.

Examples

```
zz <- textConnection(LETTERS)
readLines(zz, 2)
scan(zz, "", 4)
pushBack(c("aa", "bb"), zz)
scan(zz, "", 4)
close(zz)

zz <- textConnection("foo", "w")
writeLines(c("testit1", "testit2"), zz)
cat("testit3 ", file=zz)
isIncomplete(zz)
cat("testit4\n", file=zz)
isIncomplete(zz)
close(zz)
foo

## Not run:
# capture R output: use part of example from help(lm)
zz <- textConnection("foo", "w")
ctl <- c(4.17, 5.58, 5.18, 6.11, 4.5, 4.61, 5.17, 4.53, 5.33, 5.14)
trt <- c(4.81, 4.17, 4.41, 3.59, 5.87, 3.83, 6.03, 4.89, 4.32, 4.69)
group <- gl(2, 10, 20, labels = c("Ctl", "Trt"))
weight <- c(ctl, trt)
sink(zz)
anova(lm.D9 <- lm(weight ~ group))
cat("\nSummary of Residuals:\n\n")
summary(resid(lm.D9))
sink()
close(zz)
cat(foo, sep = "\n")
## End(Not run)
```

`tilde` *Tilde Operator*

Description

Tilde is used to separate the left- and right-hand sides in model formula.

Usage

```
y ~ model
```

Arguments

`y, model`	symbolic expressions.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical models.* Chapter 2 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

formula

---

toString *Convert an R Object to a Character String*

Description

This is a helper function for format to produce a single character string describing an R object.

Usage

```
toString(x, ...)

## Default S3 method:
toString(x, width = NULL, ...)
```

Arguments

x	The object to be converted.
width	Suggestion for the maximum field width. Values of NULL or 0 indicate no maximum. The minimum value accepted is 6 and smaller values are taken as 6.
...	Optional arguments passed to or from methods.

Details

This is a generic function for which methods can be written: only the default method is described here. Most methods should honor the width argument to specify the maximum display width (as measured by nchar(type = "width") of the result.

The default method first converts x to character and then concatenates the elements separated by ", ". If width is supplied and is not NULL, the default method returns the first width - 4 characters of the result with .... appended, if the full result would use more than width characters.

Value

A character vector of length 1 is returned.

Author(s)

Robert Gentleman

See Also

format

Examples

```
x <- c("a", "b", "aaaaaaaaaaa")
toString(x)
toString(x, width=8)
```

`trace` *Interactive Tracing and Debugging of Calls to a Function or Method*

Description

A call to `trace` allows you to insert debugging code (e.g., a call to `browser` or `recover`) at chosen places in any function. A call to `untrace` cancels the tracing. Specified methods can be traced the same way, without tracing all calls to the function. Trace code can be any R expression. Tracing can be temporarily turned on or off globally by calling `tracingState`.

Usage

```
trace(what, tracer, exit, at, print, signature,
      where = topenv(parent.frame()), edit = FALSE)
untrace(what, signature = NULL, where = topenv(parent.frame()))

tracingState(on = NULL)
```

Arguments

`what`	The name (quoted or not) of a function to be traced or untraced. For `untrace` or for `trace` with more than one argument, more than one name can be given in the quoted form, and the same action will be applied to each one.
`tracer`	Either a function or an unevaluated expression. The function will be called or the expression will be evaluated either at the beginning of the call, or before those steps in the call specified by the argument `at`. See the details section.
`exit`	Either a function or an unevaluated expression. The function will be called or the expression will be evaluated on exiting the function. See the details section.
`at`	optional numeric vector. If supplied, `tracer` will be called just before the corresponding step in the body of the function. See the details section.
`print`	If `TRUE` (as per default), a descriptive line is printed before any trace expression is evaluated.
`signature`	If this argument is supplied, it should be a signature for a method for function `what`. In this case, the method, and *not* the function itself, is traced.
`edit`	For complicated tracing, such as tracing within a loop inside the function, you will need to insert the desired calls by editing the body of the function. If so, supply the `edit` argument either as `TRUE`, or as the name of the editor you want to use. Then `trace()` will call `edit` and use the version of the function after you edit it. See the details section for additional information.
`where`	where to look for the function to be traced; by default, the top-level environment of the call to `trace`. An important use of this argument is to trace a function when it is called from a package with a namespace. The current namespace mechanism imports the functions to be called (with the exception of functions in the base package). The functions being called are *not* the same objects seen from the top-level (in general, the imported packages may not even be attached). Therefore, you must ensure that the correct versions are being traced. The way to do this is to set argument `where` to a function in the namespace. The tracing computations will then start looking in the environment of that function (which will be the

namespace of the corresponding package). (Yes, it's subtle, but the semantics here are central to how namespaces work in R.)

`on` logical; a call to `tracingState` returns `TRUE` if tracing is globally turned on, `FALSE` otherwise. An argument of one or the other of those values sets the state. If the tracing state is `FALSE`, none of the trace actions will actually occur (used, for example, by debugging functions to shut off tracing during debugging).

Details

The `trace` function operates by constructing a revised version of the function (or of the method, if `signature` is supplied), and assigning the new object back where the original was found. If only the `what` argument is given, a line of trace printing is produced for each call to the function (back compatible with the earlier version of `trace`).

The object constructed by `trace` is from a class that extends `"function"` and which contains the original, untraced version. A call to `untrace` re-assigns this version.

If the argument `tracer` or `exit` is the name of a function, the tracing expression will be a call to that function, with no arguments. This is the easiest and most common case, with the functions `browser` and `recover` the likeliest candidates; the former browses in the frame of the function being traced, and the latter allows browsing in any of the currently active calls.

The `tracer` or `exit` argument can also be an unevaluated expression (such as returned by a call to `quote` or `substitute`). This expression itself is inserted in the traced function, so it will typically involve arguments or local objects in the traced function. An expression of this form is useful if you only want to interact when certain conditions apply (and in this case you probably want to supply `print=FALSE` in the call to `trace` also).

When the `at` argument is supplied, it should be a vector of integers referring to the substeps of the body of the function (this only works if the body of the function is enclosed in `{ ...}`. In this case `tracer` is *not* called on entry, but instead just before evaluating each of the steps listed in `at`. (Hint: you don't want to try to count the steps in the printed version of a function; instead, look at `as.list(body(f))` to get the numbers associated with the steps in function `f`.)

An intrinsic limitation in the `exit` argument is that it won't work if the function itself uses `on.exit`, since the existing calls will override the one supplied by `trace`.

Tracing does not nest. Any call to `trace` replaces previously traced versions of that function or method (except for edited versions as discussed below), and `untrace` always restores an untraced version. (Allowing nested tracing has too many potentials for confusion and for accidentally leaving traced versions behind.)

When the `edit` argument is used repeatedly with no call to `untrace` on the same function or method in between, the previously edited version is retained. If you want to throw away all the previous tracing and then edit, call `untrace` before the next call to `trace`. Editing may be combined with automatic tracing; just supply the other arguments such as `tracer`, and the `edit` argument as well. The `edit=TRUE` argument uses the default editor (see `edit`).

Tracing primitive functions (builtins and specials) from the base package works, but only by a special mechanism and not very informatively. Tracing a primitive causes the primitive to be replaced by a function with argument `...` (only). You can get a bit of information out, but not much. A warning message is issued when `trace` is used on a primitive.

The practice of saving the traced version of the function back where the function came from means that tracing carries over from one session to another, *if* the traced function is saved in the session image. (In the next session, `untrace` will remove the tracing.) On the other hand, functions that were in a package, not in the global environment, are not saved in the image, so tracing expires with the session for such functions.

Tracing a method is basically just like tracing a function, with the exception that the traced version is stored by a call to `setMethod` rather than by direct assignment, and so is the untraced version after a call to `untrace`.

The version of `trace` described here is largely compatible with the version in S-Plus, although the two work by entirely different mechanisms. The S-Plus `trace` uses the session frame, with the result that tracing never carries over from one session to another (R does not have a session frame). Another relevant distinction has nothing directly to do with `trace`: The browser in S-Plus allows changes to be made to the frame being browsed, and the changes will persist after exiting the browser. The R browser allows changes, but they disappear when the browser exits. This may be relevant in that the S-Plus version allows you to experiment with code changes interactively, but the R version does not. (A future revision may include a "destructive" browser for R.)

Value

In the simple version (just the first argument), invisible `NULL`. Otherwise, the traced function(s) name(s). The relevant consequence is the assignment that takes place.

Note

The version of function tracing that includes any of the arguments except for the function name requires the **methods** package (because it uses special classes of objects to store and restore versions of the traced functions).

If methods dispatch is not currently on, `trace` will load the methods namespace, but will not put the methods package on the search list.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`browser` and `recover`, the likeliest tracing functions; also, `quote` and `substitute` for constructing general expressions.

Examples

```
##  Very simple use
trace(sum)
hist(rnorm(100)) # shows about 3-4 calls to sum()
untrace(sum)

if(.isMethodsDispatchOn()) { # non-simple use needs 'methods' package

 f <- function(x, y) {
     y <- pmax(y, .001)
     x ^ y
 }

 ## arrange to call the browser on entering and exiting
 ## function f
 trace("f", browser, exit = browser)

 ## instead, conditionally assign some data, and then browse
```

```
## on exit, but only then.  Don't bother me otherwise

trace("f", quote(if(any(y < 0)) yOrig <- y),
      exit = quote(if(exists("yOrig")) browser()),
      print = FALSE)

## trace a utility function, with recover so we
## can browse in the calling functions as well.

trace("as.matrix", recover)

## turn off the tracing

untrace(c("f", "as.matrix"))

## Not run:
## trace calls to the function lm() that come from the nlme package
## (The function nlme is in that package, and the package has a namespace,
## so the where= argument must be used to get the right version of lm)

trace(lm, exit = recover, where = nlme)
## End(Not run)
}
```

`traceback` *Print Call Stacks*

Description

By default `traceback()` prints the call stack of the last uncaught error, i.e., the sequence of calls that lead to the error. This is useful when an error occurs with an unidentifiable error message. It can also be used to print arbitrary lists of deparsed calls.

Usage

```
traceback(x = NULL, max.lines = getOption("deparse.max.lines"))
```

Arguments

`x`	`NULL` (default, meaning `.Traceback`), or a list or pairlist of deparsed calls.
`max.lines`	The maximum number of lines to be printed *per call*. The default is unlimited.

Details

The stack of the last uncaught error is stored as a list of deparsed calls in `.Traceback`, which `traceback` prints in a user-friendly format. The stack of deparsed calls always contains all function calls and all foreign function calls (such as `.Call`): if profiling is in progress it will include calls to some primitive functions. (Calls to builtins are included, but not to specials.)

Errors which are caught *via* `try` or `tryCatch` do not generate a traceback, so what is printed is the call sequence for the last uncaught error, and not necessarily for the last error.

Value

`traceback()` returns nothing, but prints the deparsed call stack deepest call first. The calls may print on more than one line, and the first line for each call is labelled by the frame number. The number of lines printed per call can be limited via `max.lines`.

Warning

It is undocumented where `.Traceback` is stored nor that it is visible, and this is subject to change. Prior to R 2.4.0 it was stored in the workspace, but no longer.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
foo <- function(x) { print(1); bar(2) }
bar <- function(x) { x + a.variable.which.does.not.exist }
## Not run:
foo(2) # gives a strange error
traceback()
## End(Not run)
## 2: bar(2)
## 1: foo(2)
bar
## Ah, this is the culprit ...
```

`transform` *Transform an Object, for Example a Data Frame*

Description

`transform` is a generic function, which—at least currently—only does anything useful with data frames. `transform.default` converts its first argument to a data frame if possible and calls `transform.data.frame`.

Usage

```
transform(`_data`, ...)
```

Arguments

`_data`	The object to be transformed
`...`	Further arguments of the form `tag=value`

Details

The `...` arguments to `transform.data.frame` are tagged vector expressions, which are evaluated in the data frame `_data`. The tags are matched against `names(_data)`, and for those that match, the value replace the corresponding variable in `_data`, and the others are appended to `_data`.

Value

The modified value of `_data`.

Note

Prior to R 2.3.0, the first argument was named `x`, but this caused trouble if people wanted to create a variable of that name. Names starting with an underscore are syntactically invalid, so the current choice should be less problematic.

If some of the values are not vectors of the appropriate length, you deserve whatever you get!

Author(s)

Peter Dalgaard

See Also

`subset`, `list`, `data.frame`

Examples

```
transform(airquality, Ozone = -Ozone)
transform(airquality, new = -Ozone, Temp = (Temp-32)/1.8)

attach(airquality)
transform(Ozone, logOzone = log(Ozone)) # marginally interesting ...
detach(airquality)
```

`Trig` *Trigonometric Functions*

Description

These functions give the obvious trigonometric functions. They respectively compute the cosine, sine, tangent, arc-cosine, arc-sine, arc-tangent, and the two-argument arc-tangent.

Usage

```
cos(x)
sin(x)
tan(x)
acos(x)
asin(x)
atan(x)
atan2(y, x)
```

Arguments

`x, y`	numeric or complex vectors.

Details

The arc-tangent of two arguments `atan2(y, x)` returns the angle between the x-axis and the vector from the origin to (x, y), i.e., for positive arguments `atan2(y, x) == atan(y/x)`.

Angles are in radians, not degrees (i.e., a right angle is $\pi/2$).

All except `atan2` are generic functions: methods can be defined for them individually or via the `Math` group generic.

Complex values

For the inverse trigonometric functions, branch cuts are defined as in Abramowitz and Stegun, figure 4.4, page 79. Continuity on the branch cuts is standard.

For `asin()` and `acos()`, there are two cuts, both along the real axis: $(-\infty, -1]$ and $[1, \infty)$. Functions `asin()` and `acos()` are continuous from above on the interval $(-\infty, -1]$ and continuous from below on $[1, \infty)$.

For `atan()` there are two cuts, both along the pure imaginary axis: $(-\infty i, -1i]$ and $[1i, \infty i)$. It is continuous from the left on the interval $(-\infty i, -1i]$ and from the right on the interval $[1i, \infty i)$.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Abramowitz, M. and Stegun, I. A. (1972). *Handbook of Mathematical Functions,* New York: Dover. Chapter 4. Elementary Transcendental Functions: Logarithmic, Exponential, Circular and Hyperbolic Functions

---

`try` *Try an Expression Allowing Error Recovery*

Description

`try` is a wrapper to run an expression that might fail and allow the user's code to handle error-recovery.

Usage

```
try(expr, silent = FALSE)
```

Arguments

`expr`	an R expression to try.
`silent`	logical: should the report of error messages be suppressed?

Details

`try` evaluates an expression and traps any errors that occur during the evaluation. `try` establishes a handler for errors that uses the default error handling protocol. It also establishes a `tryRestart` restart that can be used by `invokeRestart`.

Value

The value of the expression if `expr` is evaluated without error, but an invisible object of class `"try-error"` containing the error message if it fails. The normal error handling will print the same message unless `options("show.error.messages")` is false or the call includes `silent = TRUE`.

See Also

`options` for setting error handlers and suppressing the printing of error messages; `geterrmessage` for retrieving the last error message. `tryCatch` provides another means of catching and handling errors.

Examples

```
## this example will not work correctly in example(try), but
## it does work correctly if pasted in
options(show.error.messages = FALSE)
try(log("a"))
print(.Last.value)
options(show.error.messages = TRUE)

## alternatively,
print(try(log("a"), TRUE))

## run a simulation, keep only the results that worked.
set.seed(123)
x <- rnorm(50)
doit <- function(x)
{
    x <- sample(x, replace=TRUE)
    if(length(unique(x)) > 30) mean(x)
    else stop("too few unique points")
}
## alternative 1
res <- lapply(1:100, function(i) try(doit(x), TRUE))
## alternative 2
## Not run:
res <- vector("list", 100)
for(i in 1:100) res[[i]] <- try(doit(x), TRUE)
## End(Not run)
unlist(res[sapply(res, function(x) !inherits(x, "try-error"))])
```

---

type.convert *Type Conversion on Character Variables*

Description

Convert a character vector to logical, integer, numeric, complex or factor as appropriate.

Usage

```
type.convert(x, na.strings = "NA", as.is = FALSE, dec = ".")
```

Arguments

x	a character vector.
na.strings	a vector of strings which are to be interpreted as NA values. Blank fields are also considered to be missing values in logical, integer, numeric or complex vectors.
as.is	logical. See Details.
dec	the character to be assumed for decimal points.

Details

This is principally a helper function for read.table. Given a character vector, it attempts to convert it to logical, integer, numeric or complex, and failing that converts it to factor unless as.is = TRUE. The first type that can accept all the non-missing values is chosen.

Vectors which are entirely missing values are converted to logical, since NA is primarily logical.

Value

A vector of the selected class, or a factor.

See Also

read.table

---

typeof *The Type of an Object*

Description

typeof determines the (R internal) type or storage mode of any object

Usage

```
typeof(x)
```

Arguments

x	any R object.

Value

A character string. The possible values are listed in the structure TypeTable in 'src/main/util.c'. Current values are the vector types "logical", "integer", "double", "complex", "character", "raw" and "list", "NULL", "closure" (function), "special" and "builtin" (basic functions and operators), "environment", and others that are unlikely to be seen at user level ("symbol", "pairlist", "promise", "language", "char", "...", "any", "expression", "externalptr", "bytecode" and "weakref").

As from R 2.4.0, "S4" can occur to refer to some objects with formal (S4) classes.

See Also

mode, storage.mode.

Examples

```
typeof(2)
mode(2)
```

`unique` *Extract Unique Elements*

Description

`unique` returns a vector, data frame or array like `x` but with duplicate elements removed.

Usage

```
unique(x, incomparables = FALSE, ...)

## S3 method for class 'matrix':
unique(x, incomparables = FALSE, MARGIN = 1, ...)

## S3 method for class 'array':
unique(x, incomparables = FALSE, MARGIN = 1, ...)
```

Arguments

`x`	a vector or a data frame or an array or `NULL`.
`incomparables`	a vector of values that cannot be compared. Currently, `FALSE` is the only possible value, meaning that all values can be compared.
`...`	arguments for particular methods.
`MARGIN`	the array margin to be held fixed: a single integer.

Details

This is a generic function with methods for vectors, data frames and arrays (including matrices).

The array method calculates for each element of the dimension specified by `MARGIN` if the remaining dimensions are identical to those for an earlier element (in row-major order). This would most commonly be used for matrices to find unique rows (the default) or columns (with `MARGIN = 2`).

Value

For a vector, an object of the same type of `x`, but with only one copy of each duplicated element. No attributes are copied (so the result has no names).

For a data frame, a data frame is returned with the same columns but possibly fewer rows (and with row names from the first occurrences of the unique rows).

A matrix or array is subsetted by `[, drop = FALSE]`, so dimensions and dimnames are copied appropriately, and the result always has the same number of dimensions as `x`.

Warning

Using this for lists is potentially slow, especially if the elements are not atomic vectors (see `vector`) or differ only in their attributes. In the worst case it is $O(n^2)$.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`duplicated` which gives the indices of duplicated elements.

Examples

```
unique(c(3:5, 11:8, 8 + 0:5))
length(unique(sample(100, 100, replace=TRUE)))
## approximately 100(1 - 1/e) = 63.21

unique(iris)
```

---

unlink *Delete Files and Directories*

Description

`unlink` deletes the file(s) or directories specified by `x`.

Usage

```
unlink(x, recursive = FALSE)
```

Arguments

`x`	a character vector with the names of the file(s) or directories to be deleted. Wildcards (normally '*' and '?') are allowed.
`recursive`	logical. Should directories be deleted recursively?

Details

If `recursive = FALSE` directories are not deleted, not even empty ones.

`file.remove` can only remove files, but gives more detailed error information.

Value

The return value of the corresponding system command, `rm -f`, normally `0` for success, `1` for failure. Not deleting a non-existent file is not a failure.

Note

Prior to R version 1.2.0 the default on Unix was `recursive = TRUE`, and on Windows empty directories could be deleted.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`file.remove`.

`unlist` *Flatten Lists*

Description

Given a list structure `x`, `unlist` simplifies it to produce a vector which contains all the atomic components which occur in `x`.

Usage

```
unlist(x, recursive = TRUE, use.names = TRUE)
```

Arguments

`x`	An R object, typically a list or vector.
`recursive`	logical. Should unlisting be applied to list components of `x`?
`use.names`	logical. Should names be preserved?

Details

`unlist` is generic: you can write methods to handle specific classes of objects, see InternalMethods.

If `x` has length zero the result is `NULL`.

If `recursive = FALSE`, the function will not recurse beyond the first level items in `x`.

Factors are treated specially. If all non-list elements of `x` are factors (or ordered factors) then the result will be a factor with levels the union of the level sets of the elements, in the order the levels occur in the level sets of the elements (which means that if all the elements have the same level set, that is the level set of the result).

`x` can be an atomic vector, but then `unlist` does nothing useful, not even drop names.

By default, `unlist` tries to retain the naming information present in `x`. If `use.names = FALSE` all naming information is dropped.

Where possible the list elements are coerced to a common mode during the unlisting, and so the result often ends up as a character vector. Vectors will be coerced to the highest type of the components in the hierarchy NULL < raw < logical < integer < real < complex < character < list < expression: pairlists are treated as lists.

A list is a (generic) vector, and the simplified vector might still be a list (and might be unchanged). Non-vector elements of the list (for example language elements such as names, formulas and calls) are not coerced, and so a list containing one or more of these remains a list. (The effect of unlisting an `lm` fit is a list which has individual residuals as components.)

Value

`NULL` or an expression or a vector of an appropriate mode to hold the list components.

The output type is determined from the highest type of the components in the hierarchy raw < logical < integer < real < complex < character < list < expression, after coercion of pairlists to lists.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`c`, `as.list`.

Examples

```
unlist(options())
unlist(options(), use.names=FALSE)

l.ex <- list(a = list(1:5, LETTERS[1:5]), b = "Z", c = NA)
unlist(l.ex, recursive = FALSE)
unlist(l.ex, recursive = TRUE)

l1 <- list(a="a", b=2, c=pi+2i)
unlist(l1) # a character vector
l2 <- list(a="a", b=as.name("b"), c=pi+2i)
unlist(l2) # remains a list
```

---

unname *Remove 'names' or 'dimnames'*

Description

Remove the `names` or `dimnames` attribute of an R object.

Usage

```
unname(obj, force = FALSE)
```

Arguments

`obj`	the R object which is wanted "nameless".
`force`	logical; if true, the `dimnames` are even removed from `data.frame`s. *This argument is currently* ***experimental*** *and hence might change!*

Value

Object as `obj` but without `names` or `dimnames`.

Examples

```
## Answering a question on R-help (14 Oct 1999):
col3 <- 750+ 100* rt(1500, df = 3)
breaks <- factor(cut(col3,breaks=360+5*(0:155)))
z <- table(breaks)
z[1:5] # The names are larger than the data ...
barplot(unname(z), axes= FALSE)
```

`UseMethod` *Class Methods*

Description

R possesses a simple generic function mechanism which can be used for an object-oriented style of programming. Method dispatch takes place based on the class(es) of the first argument to the generic function or of the object supplied as an argument to `UseMethod` or `NextMethod`.

Usage

```
UseMethod(generic, object)

NextMethod(generic = NULL, object = NULL, ...)
```

Arguments

`generic`	a character string naming a function (and not a built-in operator). Required for `UseMethod`.
`object`	for `UseMethod`: an object whose class will determine the method to be dispatched. Defaults to the first argument of the enclosing function.
`...`	further arguments to be passed to the next method.

Details

An R "object" is a data object which has a `class` attribute. A class attribute is a character vector giving the names of the classes which the object "inherits" from. If the object does not have a class attribute, it has an implicit class. Matrices and arrays have class `"matrix"` or`"array"` followed by the class of the underlying vector. Most vectors have class the result of `mode(x)`, except that integer vectors have class `c("integer", "numeric")` and real vectors have class `c("double", "numeric")`.

When a function calling `UseMethod("fun")` is applied to an object with class attribute `c("first", "second")`, the system searches for a function called `fun.first` and, if it finds it, applies it to the object. If no such function is found a function called `fun.second` is tried. If no class name produces a suitable function, the function `fun.default` is used, if it exists, or an error results.

Function `methods` can be used to find out about the methods for a particular generic function or class.

`UseMethod` is a primitive function so positional matching is used and names of supplied arguments are ignored. It is not the only means of dispatch of methods, for there are internal generic functions.

`NextMethod` invokes the next method (determined by the class vector, either of the object supplied to the generic, or of the first argument to the function containing `NextMethod` if a method was invoked directly). Normally `NextMethod` is used with only one argument, `generic`, but if further arguments are supplied these modify the call to the next method.

`NextMethod` should not be called except in methods called by `UseMethod` or from internal generics (see InternalGenerics). In particular it will not work inside anonymous calling functions (e.g. `get("print.ts")(AirPassengers)`).

Name spaces can register methods for generic functions. To support this, `UseMethod` and `NextMethod` search for methods in two places: first in the environment in which the generic function is called, and then in the registration data base for the environment in which the generic is defined (typically a name space). So methods for a generic function need to be available in the environment of the call to the generic, or they must be registered. (It does not matter whether they are visible in the environment in which the generic is defined.)

Technical Details

Now for some obscure details that need to appear somewhere. These comments will be slightly different than those in Chambers(1992). (See also the draft 'R Language Definition'.) `UseMethod` creates a "new" function call with arguments matched as they came in to the generic. Any local variables defined before the call to `UseMethod` are retained (unlike S). Any statements after the call to `UseMethod` will not be evaluated as `UseMethod` does not return. `UseMethod` can be called with more than two arguments: a warning will be given and additional arguments ignored. (They are not completely ignored in S.) If it is called with just one argument, the class of the first argument of the enclosing function is used as `object`: unlike S this is the first actual argument passed and not the current value of the object of that name.

`NextMethod` works by creating a special call frame for the next method. If no new arguments are supplied, the arguments will be the same in number, order and name as those to the current method but their values will be promises to evaluate their name in the current method and environment. Any named arguments matched to `...` are handled specially: they either replace existing arguments of the same name or are appended to the argument list. They are passed on as the promise that was supplied as an argument to the current environment. (S does this differently!) If they have been evaluated in the current (or a previous environment) they remain evaluated. (This is a complex area, and subject to change: see the draft 'R Language Definition'.)

The search for methods for `NextMethod` is slightly different from that for `UseMethod`. Finding no `fun.default` is not necessarily an error, as the search continues to the generic itself. This is to pick up an internal generic like `[` which has no separate default method, and succeeds only if the generic is a primitive function or a wrapper for a `.Internal` function of the same name. (When a primitive is called as the default method, argument matching may not work as described above due to the different semantics of primitives.)

You will see objects such as `.Generic`, `.Method`, and `.Class` used in methods. These are set in the environment within which the method is evaluated by the dispatch mechanism, which is as follows:

1. Find the context for the calling function (the generic): this gives us the unevaluated arguments for the original call.
2. Evaluate the object (usually an argument) to be used for dispatch, and find a method (possibly the default method) or throw an error.
3. Create an environment for evaluating the method and insert special variables (see below) into that environment. Also copy any variables in the environment of the generic that are not formal (or actual) arguments.
4. Fix up the argument list to be the arguments of the call matched to the formals of the method.

`.Generic` is a length-one character vector naming the generic function.

`.Method` is a character vector (normally of length one) naming the method function. (For functions in the group generic `Ops` it is of length two.)

`.Class` is a character vector of classes used to find the next method. `NextMethod` adds an attribute `"previous"` to `.Class` giving the `.Class` last used for dispatch, and shifts `.Class` along to that used for dispatch.

`.GenericCallEnv` and `.GenericDefEnv` are the environments of the call to be generic and defining the generic respectively. (The latter is used to find methods registered for the generic.)

Note that `.Class` is set when the generic is called, and is unchanged if the class of the dispatching argument is changed in a method. It is possible to change the method that `NextMethod` would dispatch by manipulating `.Class`, but 'this is not recommended unless you understand the inheritance mechanism thoroughly' (Chambers & Hastie, 1992, p. 469).

Historical note

Prior to R 2.1.0 `UseMethod` accepted a call with no arguments and tried to deduce the generic from the context. This was undocumented on the help page. It is allowed but 'strongly discouraged' in S-PLUS, and is no longer allowed in R.

Note

This scheme is called *S3* (S version 3). For new projects, it is recommended to use the more flexible and robust *S4* scheme provided in the **methods** package.

References

Chambers, J. M. (1992) *Classes and methods: object-oriented programming in S.* Appendix A of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

The draft 'R Language Definition'.

`methods`, `class`, `getS3method`.

`UserHooks` *Functions to Get and Set Hooks for Load, Attach, Detach and Unload*

Description

These functions allow users to set actions to be taken before packages are attached/detached and namespaces are (un)loaded.

Usage

```
getHook(hookName)
setHook(hookName, value, action = c("append", "prepend", "replace"))

packageEvent(pkgname,
             event = c("onLoad", "attach", "detach", "onUnload"))
```

Arguments

`hookName`	character string: the hook name
`pkgname`	character string: the package/namespace name. If versioned install has been used, `pkgname` should be the unversioned name of the package (but any version information will be stripped).
`event`	character string: an event for the package

value	A function, or for action="replace", NULL.
action	The action to be taken. The names can be appreviated.

Details

setHook provides a general mechanism for users to register hooks, a list of functions to be called from system (or user) functions. The initial set of hooks is associated with events on packages/namespaces: these hooks are named via calls to packageEvent.

To remove a hook completely, call setHook(hookName, NULL, "replace").

When an R package is attached by library, it can call initialization code via a function .First.lib, and when it is detach-ed it can tidy up via a function .Last.lib. Users can add their own initialization code via the hooks provided by these functions, functions which will be called as funname(pkgname, pkgpath) inside a try call. (The attach hook is called after .First.lib and the detach hook before .Last.lib.)

If a package has a namespace, there are two further actions, when the namespace is loaded (before being attached and after .onLoad is called) and when it is unloaded (after being detached and before .onUnload). Note that code in these hooks is run without the package being on the search path, so objects in the package need to be referred to using the double colon operator as in the example. (Unlike .onLoad, the user hook is run after the name space has been sealed.)

Hooks are normally run in the order shown by getHook, but the "detach" and "onUnload" hooks are run in reverse order so the default for package events is to add hooks 'inside' existing ones.

Note that when an R session is finished, packages are not detached and namespaces are not unloaded, so the corresponding hooks will not be run.

The hooks are stored in the environment .userHooksEnv in the base package, with 'mangled' names.

Value

For getHook function, a list of functions (possible empty). For setHook function, no return value. For packageEvent, the derived hook name (a character string).

See Also

library, detach, loadNamespace.

Other hooks may be added later: plot.new and persp already have them.

Examples

```
setHook(packageEvent("grDevices", "onLoad"),
        function(...) grDevices::ps.options(horizontal=FALSE))
```

utf8Conversion *Convert to or from UTF-8-enconded Character Vectors*

Description

Conversion of UTF-8 encoded character vectors to and from integer vectors.

Usage

```
utf8ToInt(x)
intToUtf8(x, multiple = FALSE)
```

Arguments

x	object to be converted.
multiple	logical: should the conversion be to a single character string or multiple individual characters?

Details

These will work in any locale, including on machines that do not otherwise support multi-byte character sets.

Value

utf8ToInt converts a length-one character string encoded in UTF-8 to an integer vector of (numeric) UTF-8 code points.

intToUtf8 converts a vector of (numeric) UTF-8 code points either to a single character string or a character vector of single characters. (Note that a single character string could contain embedded nuls.)

vector *Vectors*

Description

vector produces a vector of the given length and mode.

as.vector, a generic, attempts to coerce its argument into a vector of mode mode (the default is to coerce to whichever mode is most convenient).

is.vector returns TRUE if x is a vector (of mode logical, integer, real, complex, character, raw or list if not specified) or expression and FALSE otherwise.

Usage

```
vector(mode = "logical", length = 0)
as.vector(x, mode = "any")
is.vector(x, mode = "any")
```

Arguments

mode	A character string giving an atomic mode or "list", or (not for vector) "any".
length	A non-negative integer specifying the desired length.
x	An object.

Details

The atomic modes are "logical", "integer", "numeric", "complex", "character" and "raw".

is.vector returns FALSE if x has any attributes except names. (This is incompatible with S.) On the other hand, as.vector removes *all* attributes including names for results of atomic mode.

Note that factors are *not* vectors; is.vector returns FALSE and as.vector converts to a character vector for mode = "any".

Value

For vector, a vector of the given length and mode. Logical vector elements are initialized to FALSE, numeric vector elements to 0, character vector elements to "", raw vector elements to nul bytes and list elements to NULL.

All attributes are removed from the answer if it is of an atomic mode.

Note

as.vector and is.vector are quite distinct from the meaning of the formal class "vector" in the **methods** package, and hence as(x, "vector") and is(x, "vector").

modes of "symbol", "pairlist" and "expression" are allowed but have long been undocumented.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

c, is.numeric, is.list, etc.

Examples

```
df <- data.frame(x=1:3, y=5:7)
## Not run:
## Error:
  as.vector(data.frame(x=1:3, y=5:7), mode="numeric")
## End(Not run)

x <- c(a = 1, b = 2)
is.vector(x)
as.vector(x)
all.equal(x, as.vector(x)) ## FALSE

###-- All the following are TRUE:
```

```
is.list(df)
! is.vector(df)
! is.vector(df, mode="list")

is.vector(list(), mode="list")
is.vector(NULL,   mode="NULL")
```

`warning` *Warning Messages*

Description

Generates a warning message that corresponds to its argument(s) and (optionally) the expression or function from which it was called.

Usage

```
warning(..., call. = TRUE, immediate. = FALSE, domain = NULL)
suppressWarnings(expr)
```

Arguments

`...`	character vectors (which are pasted together with no separator), a condition object, or `NULL`.
`call.`	logical, indicating if the call should become part of the warning message.
`immediate.`	logical, indicating if the call should be output immediately, even if `getOption(warn) <= 0`.
`expr`	expression to evaluate.
`domain`	see `gettext`. If `NA`, messages will not be translated.

Details

The result *depends* on the value of `options("warn")` and on handlers established in the executing code.

`warning` signals a warning condition by (effectively) calling `signalCondition`. If there are no handlers or if all handlers return, then the value of `warn` is used to determine the appropriate action. If `warn` is negative warnings are ignored; if it is zero they are stored and printed after the top–level function has completed; if it is one they are printed as they occur and if it is 2 (or larger) warnings are turned into errors. Calling `warning(immediate. = TRUE)` turns `warn <= 0` into `warn = 1` for this call only.

If `warn` is zero (the default), a read-only variable `last.warning` is created. It contains the warnings which can be printed via a call to `warnings`.

Warnings will be truncated to `getOption("warning.length")` characters, default 1000.

While the warning is being processed, a `muffleWarning` restart is available. If this restart is invoked with `invokeRestart`, then `warning` returns immediately.

An attempt is made to coerce other types of inputs to `warning` to character vectors.

`suppressWarnings` evaluates its expression in a context that ignores all warnings.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`stop` for fatal errors, `message` for diagnostic messages, `warnings`, and `options` with argument `warn=`.

`gettext` for the mechanisms for the automated translation of messages.

Examples

```
testit <- function() warning("testit")
testit() ## shows call
testit <- function() warning("problem in testit", call. = FALSE)
testit() ## no call
suppressWarnings(warning("testit"))
```

---

warnings *Print Warning Messages*

Description

`warnings` and its `print` method print the variable `last.warning` in a pleasing form.

Usage

```
warnings(...)
```

Arguments

`...`	arguments to be passed to `cat`.

Details

See the decription of `options("warn")` for the circumstances under which there is a `last.warning` object and `warnings()` is used. In essence this is if `options(warn = 0)` and `warning` has been called at least once.

It is possible that `last.warning` refers to the last recorded warning and not to the last warning, for example if `options(warn)` has been changed or if a catastrophic error occurred.

Warning

It is undocumented where `last.warning` is stored nor that it is visible, and this is subject to change. Prior to R 2.4.0 it was stored in the workspace, but no longer.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

warning.

Examples

```
## NB this example is intended to be pasted in,
##    rather than run by example()
ow <- options("warn")
for(w in -1:1) {
   options(warn = w); cat("\n warn =",w,"\n")
   for(i in 1:3) { cat(i,"..\n"); m <- matrix(1:7, 3,4) }
}
warnings()
options(ow) # reset
```

---

weekdays *Extract Parts of a POSIXt or Date Object*

Description

Extract the weekday, month or quarter, or the Julian time (days since some origin). These are generic functions: the methods for the internal date-time classes are documented here.

Usage

```
weekdays(x, abbreviate)
## S3 method for class 'POSIXt':
weekdays(x, abbreviate = FALSE)
## S3 method for class 'Date':
weekdays(x, abbreviate = FALSE)

months(x, abbreviate)
## S3 method for class 'POSIXt':
months(x, abbreviate = FALSE)
## S3 method for class 'Date':
months(x, abbreviate = FALSE)

quarters(x, abbreviate)
## S3 method for class 'POSIXt':
quarters(x, ...)
## S3 method for class 'Date':
quarters(x, ...)

julian(x, ...)
## S3 method for class 'POSIXt':
julian(x, origin = as.POSIXct("1970-01-01", tz="GMT"), ...)
## S3 method for class 'Date':
julian(x, origin = as.Date("1970-01-01"), ...)
```

Arguments

`x`	an object inheriting from class `"POSIXt"` or `"Date"`.
`abbreviate`	logical. Should the names be abbreviated?
`origin`	an length-one object inheriting from class `"POSIXt"` or `"Date"`.
`...`	arguments for other methods.

Value

`weekdays` and `months` return a character vector of names in the locale in use.

`quarters` returns a character vector of `"Q1"` to `"Q4"`.

`julian` returns the number of days (possibly fractional) since the origin, with the origin as a `"origin"` attribute.

Note

Other components such as the day of the month or the year are very easy to compute: just use `as.POSIXlt` and extract the relevant component.

See Also

`DateTimeClasses`, `Date`

Examples

```
weekdays(.leap.seconds)
months(.leap.seconds)
quarters(.leap.seconds)
```

`which` *Which indices are TRUE?*

Description

Give the `TRUE` indices of a logical object, allowing for array indices.

Usage

```
which(x, arr.ind = FALSE)
```

Arguments

`x`	a `logical` vector or array. `NA`s are allowed and omitted (treated as if `FALSE`).
`arr.ind`	logical; should **arr**ay **ind**ices be returned when `x` is an array?

Value

If `arr.ind == FALSE` (the default), an integer vector with `length` equal to `sum(x)`, i.e., to the number of `TRUE`s in `x`; Basically, the result is `(1:length(x))[x]`.

If `arr.ind == TRUE` and `x` is an `array` (has a `dim` attribute), the result is a matrix whose rows each are the indices of one element of `x`; see Examples below.

Author(s)

Werner Stahel and Peter Holzer ⟨holzer@stat.math.ethz.ch⟩, for the array case.

See Also

`Logic`, `which.min` for the index of the minimum or maximum, and `match` for the first index of an element in a vector, i.e., for a scalar `a`, `match(a,x)` is equivalent to `min(which(x == a))` but much more efficient.

Examples

```
which(LETTERS == "R")
which(ll <- c(TRUE,FALSE,TRUE,NA,FALSE,FALSE,TRUE))#> 1 3 7
names(ll) <- letters[seq(ll)]
which(ll)
which((1:12)%%2 == 0) # which are even?
which(1:10 > 3, arr.ind=TRUE)

( m <- matrix(1:12,3,4) )
which(m %% 3 == 0)
which(m %% 3 == 0, arr.ind=TRUE)
rownames(m) <- paste("Case",1:3, sep="_")
which(m %% 5 == 0, arr.ind=TRUE)

dim(m) <- c(2,2,3); m
which(m %% 3 == 0, arr.ind=FALSE)
which(m %% 3 == 0, arr.ind=TRUE)

vm <- c(m)
dim(vm) <- length(vm) #-- funny thing with  length(dim(...)) == 1
which(vm %% 3 == 0, arr.ind=TRUE)
```

`which.min` *Where is the Min() or Max() ?*

Description

Determines the location, i.e., index of the (first) minimum or maximum of a numeric vector.

Usage

```
which.min(x)
which.max(x)
```

Arguments

`x`	numeric vector, whose `min` or `max` is searched (`NA`s are allowed).

Value

an `integer` of length 1 or 0 (iff `x` has no non-`NA`s) , giving the index of the *first* minimum or maximum respectively of `x`.

If this extremum is unique (or empty), the result is the same (but more efficient) as `which(x == min(x))` or `which(x == max(x))` respectively.

Author(s)

Martin Maechler

See Also

`which`, `max.col`, `max`, etc.

`which.is.max` in package **nnet** differs in breaking ties at random (and having a "fuzz" in the definition of ties).

Examples

```
x <- c(1:4,0:5,11)
which.min(x)
which.max(x)

## it *does* work with NA's present:
presidents[1:30]
range(presidents, na.rm = TRUE)
which.min(presidents)# 28
which.max(presidents)# 2
```

---

with *Evaluate an Expression in a Data Environment*

Description

Evaluate an R expression in an environment constructed from data.

Usage

```
with(data, expr, ...)
```

Arguments

`data`	data to use for constructing an environment. For the default method this may be an environment, a list, a data frame, or an integer as in `sys.call`.
`expr`	expression to evaluate.
`...`	arguments to be passed to future methods.

Details

`with` is a generic function that evaluates `expr` in a local environment constructed from `data`. The environment has the caller's environment as its parent. This is useful for simplifying calls to modeling functions.

Note that assignments within `expr` take place in the constructed environment and not in the user's workspace.

See Also

`evalq`, `attach`.

Examples

```
require(stats); require(graphics)
#examples from glm:
## Not run:
library(MASS)
with(anorexia, {
    anorex.1 <- glm(Postwt ~ Prewt + Treat + offset(Prewt),
                    family = gaussian)
    summary(anorex.1)
})
## End(Not run)

with(data.frame(u = c(5,10,15,20,30,40,60,80,100),
                lot1 = c(118,58,42,35,27,25,21,19,18),
                lot2 = c(69,35,26,21,18,16,13,12,12)),
    list(summary(glm(lot1 ~ log(u), family = Gamma)),
         summary(glm(lot2 ~ log(u), family = Gamma))))

# example from boxplot:
with(ToothGrowth, {
    boxplot(len ~ dose, boxwex = 0.25, at = 1:3 - 0.2,
            subset = (supp == "VC"), col = "yellow",
            main = "Guinea Pigs' Tooth Growth",
            xlab = "Vitamin C dose mg",
            ylab = "tooth length", ylim = c(0,35))
    boxplot(len ~ dose, add = TRUE, boxwex = 0.25, at = 1:3 + 0.2,
            subset = supp == "OJ", col = "orange")
    legend(2, 9, c("Ascorbic acid", "Orange juice"),
           fill = c("yellow", "orange"))
})

# alternate form that avoids subset argument:
with(subset(ToothGrowth, supp == "VC"),
     boxplot(len ~ dose, boxwex = 0.25, at = 1:3 - 0.2,
             col = "yellow", main = "Guinea Pigs' Tooth Growth",
             xlab = "Vitamin C dose mg",
             ylab = "tooth length", ylim = c(0,35)))
with(subset(ToothGrowth,  supp == "OJ"),
     boxplot(len ~ dose, add = TRUE, boxwex = 0.25, at = 1:3 + 0.2,
             col = "orange"))
legend(2, 9, c("Ascorbic acid", "Orange juice"),
       fill = c("yellow", "orange"))
```

---

write *Write Data to a File*

Description

The data (usually a matrix) `x` are written to file `file`. If `x` is a two-dimensional matrix you need to transpose it to get the columns in `file` the same as those in the internal representation.

Usage

```
write(x, file = "data",
```

```
        ncolumns = if(is.character(x)) 1 else 5,
        append = FALSE, sep = " ")
```

Arguments

x	the data to be written out.
file	A connection, or a character string naming the file to write to. If "", print to the standard output connection. If it is "|cmd", the output is piped to the command given by 'cmd'.
ncolumns	the number of columns to write the data in.
append	if TRUE the data x are appended to the connection.
sep	a string used to separate columns. Using sep = "\t" gives tab delimited output; default is " ".

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

write is a wrapper for cat, which gives further details on the format used.

save for writing any R objects, write.table for data frames, and scan for reading data.

Examples

```
# create a 2 by 5 matrix
x <- matrix(1:10,ncol=5)

# the file data contains x, two rows, five cols
# 1 3 5 7 9 will form the first row
write(t(x))

# Writing to the "console" 'tab-delimited'
# two rows, five cols but the first row is 1 2 3 4 5
write(x, "", sep = "\t")
unlink("data") # tidy up
```

---

write.table *Data Output*

Description

write.table prints its required argument x (after converting it to a data frame if it is not one nor a matrix) to a file or connection.

Usage

```
write.table(x, file = "", append = FALSE, quote = TRUE, sep = " ",
            eol = "\n", na = "NA", dec = ".", row.names = TRUE,
            col.names = TRUE, qmethod = c("escape", "double"))

write.csv(...)
write.csv2(...)
```

Arguments

x	the object to be written, preferably a matrix or data frame. If not, it is attempted to coerce x to a data frame.
file	either a character string naming a file or a connection. "" indicates output to the console.
append	logical. If TRUE, the output is appended to the file. If FALSE, any existing file of the name is destroyed.
quote	a logical value (TRUE or FALSE) or a numeric vector. If TRUE, any character or factor columns will be surrounded by double quotes. If a numeric vector, its elements are taken as the indices of columns to quote. In both cases, row and column names are quoted if they are written. If FALSE, nothing is quoted.
sep	the field separator string. Values within each row of x are separated by this string.
eol	the character(s) to print at the end of each line (row).
na	the string to use for missing values in the data.
dec	the string to use for decimal points in numeric or complex columns: must be a single character.
row.names	either a logical value indicating whether the row names of x are to be written along with x, or a character vector of row names to be written.
col.names	either a logical value indicating whether the column names of x are to be written along with x, or a character vector of column names to be written. See the section on 'CSV files' for the meaning of col.names = NA.
qmethod	a character string specifying how to deal with embedded double quote characters when quoting strings. Must be one of "escape" (default), in which case the quote character is escaped in C style by a backslash, or "double", in which case it is doubled. You can specify just the initial letter.
...	arguments to write.table: col.names, sep, dec and qmethod cannot be altered.

Details

If the table has no columns the rownames will be written only if row.names=TRUE, and *vice versa*.

Real and complex numbers are written to the maximal possible precision.

If a data frame has matrix-like columns these will be converted to multiple columns in the result (*via* as.matrix) and so a character col.names or a numeric quote should refer to the columns in the result, not the input. Such matrix-like columns are unquoted by default.

Any columns in a data frame which are lists or have a class (e.g. dates) will be converted by the appropriate as.character method: such columns are unquoted by default. On the other hand,

any class information for a matrix is discarded and non-atomic (e.g. list) matrices are coerced to character.

Only columns which have been converted to character will be quoted if specified by `quote`.

The `dec` argument only applies to columns that are not subject to conversion to character because they have a class or are part of a matrix-like column (or matrix), in particular to columns protected by `I()`. Use `options("OutDec")` to control such conversions.

In almost all cases the conversion of numeric quantities is governed by the options in force, in particular by `"digits"` and `"scipen"` (see `options`). For finer control, use `format` to make a character matrix/data frame, and call `write.table` on that.

These functions check for a user interrupt every 1000 lines of output.

CSV files

By default there is no column name for a column of row names. If `col.names = NA` and `row.names = TRUE` a blank column name is added, which is the convention used for CSV files to be read by spreadsheets.

`write.csv` and `write.csv2` provide convenience wrappers for writing CSV files. They set `sep`, `dec` and `qmethod`, and `col.names` to `NA` if `row.names = TRUE` and `TRUE` otherwise.

`write.csv` uses `"."` for the decimal point and a comma for the separator.

`write.csv2` uses a comma for the decimal point and a semicolon for the separator, the Excel convention for CSV files in some Western European locales.

These wrappers are deliberately inflexible: they are designed to ensure that the correct conventions are used to write a valid file. Attempts to change `col.names`, `sep`, `dec` or `qmethod` are ignored, with a warning.

Note

`write.table` can be slow for data frames with large numbers (hundreds or more) of columns: this is inevitable as each column could be of a different class and so must be handled separately. If they are all of the same class, consider using a matrix instead.

See Also

The 'R Data Import/Export' manual.

`read.table`, `write`.

`write.matrix` in package **MASS**.

Examples

```
## Not run:
## To write a CSV file for input to Excel one might use
x <- data.frame(a = I("a \" quote"), b = pi)
write.table(x, file = "foo.csv", sep = ",", col.names = NA,
            qmethod = "double")
## and to read this file back into R one needs
read.table("foo.csv", header = TRUE, sep = ",", row.names = 1)
## NB: you do need to specify a separator if qmethod = "double".

### Alternatively
write.csv(x, file = "foo.csv")
read.csv("foo.csv", row.names = 1)
```

```
## or without row names
write.csv(x, file = "foo.csv", row.names = FALSE)
read.csv("foo.csv")
## End(Not run)
```

`writeLines` *Write Lines to a Connection*

Description

Write text lines to a connection.

Usage

```
writeLines(text, con = stdout(), sep = "\n")
```

Arguments

`text`	A character vector
`con`	A connection object or a character string.
`sep`	character. A string to be written to the connection after each line of text.

Details

If the `con` is a character string, the function calls `file` to obtain a file connection which is opened for the duration of the function call.

If the connection is open it is written from its current position. If it is not open, it is opened for the duration of the call and then closed again.

Normally `writeLines` is used with a text connection, and the default separator is converted to the normal separator for that platform (LF on Unix/Linux, CRLF on Windows, CR on Classic MacOS). For more control, open a binary connection and specify the precise value you want written to the file in `sep`. For even more control, use `writeChar` on a binary connection.

See Also

`connections`, `writeChar`, `writeBin`, `readLines`, `cat`

`zip.file.extract` *Extract File from a Zip Archive*

Description

This will extract the file named `file` from the zip archive, if possible, and write it in a temporary location.

Usage

```
zip.file.extract(file, zipname = "R.zip")
```

Arguments

file	A file name.
zipname	The file name of a zip archive, including the ".zip" extension if required.

Details

The method used is selected by options(unzip=). All platforms support an "internal" unzip: this is the default under Windows and the fall-back under Unix if no unzip program was found during configuration and R_UNZIPCMD is not set.

The file will be extracted if it is in the archive and any required unzip utility is available. It will probably be extracted to the directory given by tempdir, overwriting an existing file of that name.

Value

The name of the original or extracted file. Success is indicated by returning a different name.

Note

The "internal" method is very simple, and will not set file dates.

---

zpackages *Listing of Packages*

Description

.packages returns information about package availability.

Usage

```
.packages(all.available = FALSE, lib.loc = NULL)
```

Arguments

all.available	logical; if TRUE return a character vector of all available packages in lib.loc.
lib.loc	a character vector describing the location of R library trees to search through, or NULL. The default value of NULL corresponds to all libraries currently known.

Details

.packages() returns the "base names" of the currently attached packages *invisibly* whereas .packages(all.available = TRUE) gives (visibly) *all* packages available in the library location path lib.loc. For a package to be regarded as being available it must have a 'DESCRIPTION' file containing a valid version field.

Value

A character vector of package "base names", invisible unless all.available = TRUE.

Author(s)

R core; Guido Masarotto for the all.available=TRUE part of .packages.

See Also

`library`, `.libPaths`.

Examples

```
(.packages())               # maybe just "base"
.packages(all = TRUE)       # return all available as character vector
require(splines)            # the same
(.packages())               # "splines", too
detach("package:splines")
```

---

zutils *Miscellaneous Internal/Programming Utilities*

Description

Miscellaneous internal/programming utilities.

Usage

```
.standard_regexps()
```

Details

`.standard_regexps` returns a list of "standard" regexps, including elements named `valid_package_name` and `valid_package_version` with the obvious meanings. The regexps are not anchored.

Chapter 2

The `datasets` package

`datasets-package` *The R Datasets Package*

Description

Base R datasets

Details

This package contains a variety of datasets. For a complete list, use `library(help="datasets")`.

Author(s)

R Development Core Team and contributors worldwide

Maintainer: R Core Team ⟨R-core@r-project.org⟩

`ability.cov` *Ability and Intelligence Tests*

Description

Six tests were given to 112 individuals. The covariance matrix is given in this object.

Usage

```
ability.cov
```

Details

The tests are described as

general: a non-verbal measure of general intelligence using Cattell's culture-fair test.

picture: a picture-completion test

blocks: block design

maze: mazes

reading: reading comprehension

vocab: vocabulary

Bartholomew gives both covariance and correlation matrices, but these are inconsistent. Neither are in the original paper.

Source

Barthlomew, D. J. (1987) *Latent Variable Analysis and Factor Analysis.* Griffin.

Barthlomew, D. J. and Knott, M. (1990) *Latent Variable Analysis and Factor Analysis.* Second Edition, Arnold.

References

Smith, G. A. and Stanley G. (1983) Clocking g: relating intelligence and measures of timed performance. *Intelligence*, **7**, 353–368.

Examples

```
require(stats)
(ability.FA <- factanal(factors = 1, covmat=ability.cov))
update(ability.FA, factors=2)
update(ability.FA, factors=2, rotation="promax")
```

---

airmiles *Passenger Miles on Commercial US Airlines, 1937–1960*

Description

The revenue passenger miles flown by commercial airlines in the United States for each year from 1937 to 1960.

Usage

```
airmiles
```

Format

A time series of 24 observations; yearly, 1937–1960.

Source

F.A.A. Statistical Handbook of Aviation.

References

Brown, R. G. (1963) *Smoothing, Forecasting and Prediction of Discrete Time Series*. Prentice-Hall.

Examples

```
plot(airmiles, main = "airmiles data",
     xlab = "Passenger-miles flown by U.S. commercial airlines", col = 4)
```

AirPassengers *Monthly Airline Passenger Numbers 1949-1960*

Description

The classic Box & Jenkins airline data. Monthly totals of international airline passengers, 1949 to 1960.

Usage

```
AirPassengers
```

Format

A monthly time series, in thousands.

Source

Box, G. E. P., Jenkins, G. M. and Reinsel, G. C. (1976) *Time Series Analysis, Forecasting and Control.* Third Edition. Holden-Day. Series G.

Examples

```
## Not run:
## These are quite slow and so not run by example(AirPassengers)

## The classic 'airline model', by full ML
(fit <- arima(log10(AirPassengers), c(0, 1, 1),
              seasonal = list(order=c(0, 1 ,1), period=12)))
update(fit, method = "CSS")
update(fit, x=window(log10(AirPassengers), start = 1954))
pred <- predict(fit, n.ahead = 24)
tl <- pred$pred - 1.96 * pred$se
tu <- pred$pred + 1.96 * pred$se
ts.plot(AirPassengers, 10^tl, 10^tu, log = "y", lty = c(1,2,2))

## full ML fit is the same if the series is reversed, CSS fit is not
ap0 <- rev(log10(AirPassengers))
attributes(ap0) <- attributes(AirPassengers)
arima(ap0, c(0, 1, 1), seasonal = list(order=c(0, 1 ,1), period=12))
arima(ap0, c(0, 1, 1), seasonal = list(order=c(0, 1 ,1), period=12),
      method = "CSS")

## Structural Time Series
ap <- log10(AirPassengers) - 2
```

```
(fit <- StructTS(ap, type= "BSM"))
par(mfrow=c(1,2))
plot(cbind(ap, fitted(fit)), plot.type = "single")
plot(cbind(ap, tsSmooth(fit)), plot.type = "single")
## End(Not run)
```

`airquality` *New York Air Quality Measurements*

Description

Daily air quality measurements in New York, May to September 1973.

Usage

```
airquality
```

Format

A data frame with 154 observations on 6 variables.

`[,1]`	`Ozone`	numeric	Ozone (ppb)
`[,2]`	`Solar.R`	numeric	Solar R (lang)
`[,3]`	`Wind`	numeric	Wind (mph)
`[,4]`	`Temp`	numeric	Temperature (degrees F)
`[,5]`	`Month`	numeric	Month (1–12)
`[,6]`	`Day`	numeric	Day of month (1–31)

Details

Daily readings of the following air quality values for May 1, 1973 (a Tuesday) to September 30, 1973.

- `Ozone`: Mean ozone in parts per billion from 1300 to 1500 hours at Roosevelt Island
- `Solar.R`: Solar radiation in Langleys in the frequency band 4000–7700 Angstroms from 0800 to 1200 hours at Central Park
- `Wind`: Average wind speed in miles per hour at 0700 and 1000 hours at LaGuardia Airport
- `Temp`: Maximum daily temperature in degrees Fahrenheit at La Guardia Airport.

Source

The data were obtained from the New York State Department of Conservation (ozone data) and the National Weather Service (meteorological data).

References

Chambers, J. M., Cleveland, W. S., Kleiner, B. and Tukey, P. A. (1983) *Graphical Methods for Data Analysis*. Belmont, CA: Wadsworth.

Examples

```
pairs(airquality, panel = panel.smooth, main = "airquality data")
```

`anscombe` *Anscombe's Quartet of "Identical" Simple Linear Regressions*

Description

Four x-y datasets which have the same traditional statistical properties (mean, variance, correlation, regression line, etc.), yet are quite different.

Usage

```
anscombe
```

Format

A data frame with 11 observations on 8 variables.

x1 == x2 == x3	the integers 4:14, specially arranged
x4	values 8 and 19
y1, y2, y3, y4	numbers in (3, 12.5) with mean 7.5 and sdev 2.03

Source

Tufte, Edward R. (1989) *The Visual Display of Quantitative Information*, 13–14. Graphics Press.

References

Anscombe, Francis J. (1973) Graphs in statistical analysis. *American Statistician*, **27**, 17–21.

Examples

```
require(stats)
summary(anscombe)

##-- now some "magic" to do the 4 regressions in a loop:
ff <- y ~ x
for(i in 1:4) {
  ff[2:3] <- lapply(paste(c("y","x"), i, sep=""), as.name)
  ## or   ff[[2]] <- as.name(paste("y", i, sep=""))
  ##      ff[[3]] <- as.name(paste("x", i, sep=""))
  assign(paste("lm.",i,sep=""), lmi <- lm(ff, data= anscombe))
  print(anova(lmi))
}

## See how close they are (numerically!)
sapply(objects(pat="lm\\.[1-4]$"), function(n) coef(get(n)))
lapply(objects(pat="lm\\.[1-4]$"), function(n) summary(get(n))$coef)

## Now, do what you should have done in the first place: PLOTS
op <- par(mfrow=c(2,2), mar=.1+c(4,4,1,1), oma= c(0,0,2,0))
for(i in 1:4) {
  ff[2:3] <- lapply(paste(c("y","x"), i, sep=""), as.name)
  plot(ff, data =anscombe, col="red", pch=21, bg = "orange", cex = 1.2,
       xlim=c(3,19), ylim=c(3,13))
```

```
  abline(get(paste("lm.",i,sep="")), col="blue")
}
mtext("Anscombe's 4 Regression data sets", outer = TRUE, cex=1.5)
par(op)
```

`attenu` *The Joyner–Boore Attenuation Data*

Description

This data gives peak accelerations measured at various observation stations for 23 earthquakes in California. The data have been used by various workers to estimate the attenuating affect of distance on ground acceleration.

Usage

```
attenu
```

Format

A data frame with 182 observations on 5 variables.

[,1]	event	numeric	Event Number
[,2]	mag	numeric	Moment Magnitude
[,3]	station	factor	Station Number
[,4]	dist	numeric	Station-hypocenter distance (km)
[,5]	accel	numeric	Peak acceleration (g)

Source

Joyner, W.B., D.M. Boore and R.D. Porcella (1981). Peak horizontal acceleration and velocity from strong-motion records including records from the 1979 Imperial Valley, California earthquake. USGS Open File report 81-365. Menlo Park, Ca.

References

Boore, D. M. and Joyner, W.B.(1982) The empirical prediction of ground motion, *Bull. Seism. Soc. Am.*, **72**, S269–S268.

Bolt, B. A. and Abrahamson, N. A. (1982) New attenuation relations for peak and expected accelerations of strong ground motion, *Bull. Seism. Soc. Am.*, **72**, 2307–2321.

Bolt B. A. and Abrahamson, N. A. (1983) Reply to W. B. Joyner & D. M. Boore's "Comments on: New attenuation relations for peak and expected accelerations for peak and expected accelerations of strong ground motion", *Bull. Seism. Soc. Am.*, **73**, 1481–1483.

Brillinger, D. R. and Preisler, H. K. (1984) An exploratory analysis of the Joyner-Boore attenuation data, *Bull. Seism. Soc. Am.*, **74**, 1441–1449.

Brillinger, D. R. and Preisler, H. K. (1984) *Further analysis of the Joyner-Boore attenuation data*. Manuscript.

Examples

```
## check the data class of the variables
```

```
sapply(attenu, data.class)
summary(attenu)
pairs(attenu, main = "attenu data")
coplot(accel ~ dist | as.factor(event), data = attenu, show = FALSE)
coplot(log(accel) ~ log(dist) | as.factor(event),
       data = attenu, panel = panel.smooth, show.given = FALSE)
```

`attitude` *The Chatterjee–Price Attitude Data*

Description

From a survey of the clerical employees of a large financial organization, the data are aggregated from the questionnaires of the approximately 35 employees for each of 30 (randomly selected) departments. The numbers give the percent proportion of favourable responses to seven questions in each department.

Usage

```
attitude
```

Format

A dataframe with 30 observations on 7 variables. The first column are the short names from the reference, the second one the variable names in the data frame:

Y	rating	numeric	Overall rating
X[1]	complaints	numeric	Handling of employee complaints
X[2]	privileges	numeric	Does not allow special privileges
X[3]	learning	numeric	Opportunity to learn
X[4]	raises	numeric	Raises based on performance
X[5]	critical	numeric	Too critical
X[6]	advancel	numeric	Advancement

Source

Chatterjee, S. and Price, B. (1977) *Regression Analysis by Example*. New York: Wiley. (Section 3.7, p.68ff of 2nd ed.(1991).)

Examples

```
require(stats)
pairs(attitude, main = "attitude data")
summary(attitude)
summary(fm1 <- lm(rating ~ ., data = attitude))
opar <- par(mfrow = c(2, 2), oma = c(0, 0, 1.1, 0),
            mar = c(4.1, 4.1, 2.1, 1.1))
plot(fm1)
summary(fm2 <- lm(rating ~ complaints, data = attitude))
plot(fm2)
par(opar)
```

austres *Quarterly Time Series of the Number of Australian Residents*

Description

Numbers (in thousands) of Australian residents measured quarterly from March 1971 to March 1994. The object is of class `"ts"`.

Usage

```
austres
```

Source

P. J. Brockwell and R. A. Davis (1996) *Introduction to Time Series and Forecasting.* Springer

beavers *Body Temperature Series of Two Beavers*

Description

Reynolds (1994) describes a small part of a study of the long-term temperature dynamics of beaver *Castor canadensis* in north-central Wisconsin. Body temperature was measured by telemetry every 10 minutes for four females, but data from a one period of less than a day for each of two animals is used there.

Usage

```
beaver1
beaver2
```

Format

The `beaver1` data frame has 114 rows and 4 columns on body temperature measurements at 10 minute intervals.

The `beaver2` data frame has 100 rows and 4 columns on body temperature measurements at 10 minute intervals.

The variables are as follows:

day Day of observation (in days since the beginning of 1990), December 12–13 (`beaver1`) and November 3–4 (`beaver2`).

time Time of observation, in the form `0330` for 3:30am

temp Measured body temperature in degrees Celsius.

activ Indicator of activity outside the retreat.

Note

The observation at 22:20 is missing in `beaver1`.

Source

P. S. Reynolds (1994) Time-series analyses of beaver body temperatures. Chapter 11 of Lange, N., Ryan, L., Billard, L., Brillinger, D., Conquest, L. and Greenhouse, J. eds (1994) *Case Studies in Biometry.* New York: John Wiley and Sons.

Examples

```
(yl <- range(beaver1$temp, beaver2$temp))

beaver.plot <- function(bdat, ...) {
  nam <- deparse(substitute(bdat))
  with(bdat, {
    # Hours since start of day:
    hours <- time %/% 100 + 24*(day - day[1]) + (time %% 100)/60
    plot (hours, temp, type = "l", ...,
          main = paste(nam, "body temperature"))
    abline(h = 37.5, col = "gray", lty = 2)
    is.act <- activ == 1
    points(hours[is.act], temp[is.act], col = 2, cex = .8)
  })
}
op <- par(mfrow = c(2,1), mar = c(3,3,4,2), mgp = .9* 2:0)
 beaver.plot(beaver1, ylim = yl)
 beaver.plot(beaver2, ylim = yl)
par(op)
```

BJsales *Sales Data with Leading Indicator*

Description

The sales time series `BJsales` and leading indicator `BJsales.lead` each contain 150 observations. The objects are of class `"ts"`.

Usage

```
BJsales
BJsales.lead
```

Source

The data are given in Box & Jenkins (1976). Obtained from the Time Series Data Library at http://www-personal.buseco.monash.edu.au/~hyndman/TSDL/

References

G. E. P. Box and G. M. Jenkins (1976): *Time Series Analysis, Forecasting and Control*, Holden-Day, San Francisco, p. 537.

P. J. Brockwell and R. A. Davis (1991): *Time Series: Theory and Methods*, Second edition, Springer Verlag, NY, pp. 414.

BOD *Biochemical Oxygen Demand*

Description

The BOD data frame has 6 rows and 2 columns giving the biochemical oxygen demand versus time in an evaluation of water quality.

Usage

```
BOD
```

Format

This data frame contains the following columns:

Time A numeric vector giving the time of the measurement (days).

demand A numeric vector giving the biochemical oxygen demand (mg/l).

Source

Bates, D.M. and Watts, D.G. (1988), *Nonlinear Regression Analysis and Its Applications*, Wiley, Appendix A1.4.

Originally from Marske (1967), *Biochemical Oxygen Demand Data Interpretation Using Sum of Squares Surface* M.Sc. Thesis, University of Wisconsin – Madison.

Examples

```
require(stats)
# simplest form of fitting a first-order model to these data
fm1 <- nls(demand ~ A*(1-exp(-exp(lrc)*Time)), data = BOD,
   start = c(A = 20, lrc = log(.35)))
coef(fm1)
print(fm1)
# using the plinear algorithm
fm2 <- nls(demand ~ (1-exp(-exp(lrc)*Time)), data = BOD,
   start = c(lrc = log(.35)), algorithm = "plinear", trace = TRUE)
# using a self-starting model
fm3 <- nls(demand ~ SSasympOrig(Time, A, lrc), data = BOD)
summary( fm3 )
```

cars *Speed and Stopping Distances of Cars*

Description

The data give the speed of cars and the distances taken to stop. Note that the data were recorded in the 1920s.

Usage

```
cars
```

Format

A data frame with 50 observations on 2 variables.

[,1]	speed	numeric	Speed (mph)
[,2]	dist	numeric	Stopping distance (ft)

Source

Ezekiel, M. (1930) *Methods of Correlation Analysis*. Wiley.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. Wiley.

Examples

```
require(stats)
plot(cars, xlab = "Speed (mph)", ylab = "Stopping distance (ft)",
     las = 1)
lines(lowess(cars$speed, cars$dist, f = 2/3, iter = 3), col = "red")
title(main = "cars data")
plot(cars, xlab = "Speed (mph)", ylab = "Stopping distance (ft)",
     las = 1, log = "xy")
title(main = "cars data (logarithmic scales)")
lines(lowess(cars$speed, cars$dist, f = 2/3, iter = 3), col = "red")
summary(fm1 <- lm(log(dist) ~ log(speed), data = cars))
opar <- par(mfrow = c(2, 2), oma = c(0, 0, 1.1, 0),
            mar = c(4.1, 4.1, 2.1, 1.1))
plot(fm1)
par(opar)

## An example of polynomial regression
plot(cars, xlab = "Speed (mph)", ylab = "Stopping distance (ft)",
    las = 1, xlim = c(0, 25))
d <- seq(0, 25, len = 200)
for(degree in 1:4) {
  fm <- lm(dist ~ poly(speed, degree), data = cars)
  assign(paste("cars", degree, sep="."), fm)
  lines(d, predict(fm, data.frame(speed=d)), col = degree)
}
anova(cars.1, cars.2, cars.3, cars.4)
```

ChickWeight *Weight versus age of chicks on different diets*

Description

The `ChickWeight` data frame has 578 rows and 4 columns from an experiment on the effect of diet on early growth of chicks.

Usage

```
ChickWeight
```

Format

This data frame contains the following columns:

weight a numeric vector giving the body weight of the chick (gm).

Time a numeric vector giving the number of days since birth when the measurement was made.

Chick an ordered factor with levels `18` < ... < `48` giving a unique identifier for the chick. The ordering of the levels groups chicks on the same diet together and orders them according to their final weight (lightest to heaviest) within diet.

Diet a factor with levels 1,...,4 indicating which experimental diet the chick received.

Details

The body weights of the chicks were measured at birth and every second day thereafter until day 20. They were also measured on day 21. There were four groups on chicks on different protein diets.

Source

Crowder, M. and Hand, D. (1990), *Analysis of Repeated Measures*, Chapman and Hall (example 5.3)

Hand, D. and Crowder, M. (1996), *Practical Longitudinal Data Analysis*, Chapman and Hall (table A.2)

Pinheiro, J. C. and Bates, D. M. (2000) *Mixed-effects Models in S and S-PLUS*, Springer.

Examples

```
require(stats)
coplot(weight ~ Time | Chick, data = ChickWeight,
    type = "b", show = FALSE)
## fit a representative chick
fm1 <- nls(weight ~ SSlogis( Time, Asym, xmid, scal ),
    data = ChickWeight, subset = Chick == 1)
summary( fm1 )
```

---

chickwts *Chicken Weights by Feed Type*

Description

An experiment was conducted to measure and compare the effectiveness of various feed supplements on the growth rate of chickens.

Usage

```
chickwts
```

Format

A data frame with 71 observations on 2 variables.

weight a numeric variable giving the chick weight.

feed a factor giving the feed type.

Details

Newly hatched chicks were randomly allocated into six groups, and each group was given a different feed supplement. Their weights in grams after six weeks are given along with feed types.

Source

Anonymous (1948) *Biometrika*, **35**, 214.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

Examples

```
require(stats)
boxplot(weight ~ feed, data = chickwts, col = "lightgray",
    varwidth = TRUE, notch = TRUE, main = "chickwt data",
    ylab = "Weight at six weeks (gm)")
anova(fm1 <- lm(weight ~ feed, data = chickwts))
opar <- par(mfrow = c(2, 2), oma = c(0, 0, 1.1, 0),
            mar = c(4.1, 4.1, 2.1, 1.1))
plot(fm1)
par(opar)
```

CO2 *Carbon Dioxide uptake in grass plants*

Description

The CO2 data frame has 84 rows and 5 columns of data from an experiment on the cold tolerance of the grass species *Echinochloa crus-galli*.

Usage

```
CO2
```

Format

This data frame contains the following columns:

Plant an ordered factor with levels Qn1 < Qn2 < Qn3 < ... < Mc1 giving a unique identifier for each plant.

Type a factor with levels Quebec Mississippi giving the origin of the plant

Treatment a factor with levels nonchilled chilled

conc a numeric vector of ambient carbon dioxide concentrations (mL/L).

uptake a numeric vector of carbon dioxide uptake rates (μmol/m^2 sec).

Details

The CO_2 uptake of six plants from Quebec and six plants from Mississippi was measured at several levels of ambient CO_2 concentration. Half the plants of each type were chilled overnight before the experiment was conducted.

Source

Potvin, C., Lechowicz, M. J. and Tardif, S. (1990) "The statistical analysis of ecophysiological response curves obtained from experiments involving repeated measures", *Ecology*, **71**, 1389–1400.

Pinheiro, J. C. and Bates, D. M. (2000) *Mixed-effects Models in S and S-PLUS*, Springer.

Examples

```
require(stats)
coplot(uptake ~ conc | Plant, data = CO2, show = FALSE, type = "b")
## fit the data for the first plant
fm1 <- nls(uptake ~ SSasymp(conc, Asym, lrc, c0),
   data = CO2, subset = Plant == 'Qn1')
summary(fm1)
## fit each plant separately
fmlist <- list()
for (pp in levels(CO2$Plant)) {
  fmlist[[pp]] <- nls(uptake ~ SSasymp(conc, Asym, lrc, c0),
      data = CO2, subset = Plant == pp)
}
## check the coefficients by plant
sapply(fmlist, coef)
```

---

co2 *Mauna Loa Atmospheric CO2 Concentration*

Description

Atmospheric concentrations of CO_2 are expressed in parts per million (ppm) and reported in the preliminary 1997 SIO manometric mole fraction scale.

Usage

```
co2
```

Format

A time series of 468 observations; monthly from 1959 to 1997.

Details

The values for February, March and April of 1964 were missing and have been obtained by interpolating linearly between the values for January and May of 1964.

Source

Keeling, C. D. and Whorf, T. P., Scripps Institution of Oceanography (SIO), University of California, La Jolla, California USA 92093-0220.

ftp://cdiac.esd.ornl.gov/pub/maunaloa-co2/maunaloa.co2.

References

Cleveland, W. S. (1993) *Visualizing Data*. New Jersey: Summit Press.

Examples

```
plot(co2, ylab = expression("Atmospheric concentration of CO"[2]),
     las = 1)
title(main = "co2 data set")
```

---

discoveries *Yearly Numbers of Important Discoveries*

Description

The numbers of "great" inventions and scientific discoveries in each year from 1860 to 1959.

Usage

```
discoveries
```

Format

A time series of 100 values.

Source

The World Almanac and Book of Facts, 1975 Edition, pages 315–318.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. Wiley.

Examples

```
plot(discoveries, ylab = "Number of important discoveries",
     las = 1)
title(main = "discoveries data set")
```

DNase *Elisa assay of DNase*

Description

The DNase data frame has 176 rows and 3 columns of data obtained during development of an ELISA assay for the recombinant protein DNase in rat serum.

Usage

```
DNase
```

Format

This data frame contains the following columns:

Run an ordered factor with levels 10 < ... < 3 indicating the assay run.

conc a numeric vector giving the known concentration of the protein.

density a numeric vector giving the measured optical density (dimensionless) in the assay. Duplicate optical density measurements were obtained.

Source

Davidian, M. and Giltinan, D. M. (1995) *Nonlinear Models for Repeated Measurement Data*, Chapman & Hall (section 5.2.4, p. 134)

Pinheiro, J. C. and Bates, D. M. (2000) *Mixed-effects Models in S and S-PLUS*, Springer.

Examples

```
require(stats)
coplot(density ~ conc | Run, data = DNase,
    show = FALSE, type = "b")
coplot(density ~ log(conc) | Run, data = DNase,
    show = FALSE, type = "b")
## fit a representative run
fm1 <- nls(density ~ SSlogis( log(conc), Asym, xmid, scal ),
    data = DNase, subset = Run == 1)
## compare with a four-parameter logistic
fm2 <- nls(density ~ SSfpl( log(conc), A, B, xmid, scal ),
    data = DNase, subset = Run == 1)
summary(fm2)
anova(fm1, fm2)
```

esoph *Smoking, Alcohol and (O)esophageal Cancer*

Description

Data from a case-control study of (o)esophageal cancer in Ile-et-Vilaine, France.

Usage

```
esoph
```

Format

A data frame with records for 88 age/alcohol/tobacco combinations.

[,1]	"agegp"	Age group	1 25–34 years
			2 35–44
			3 45–54
			4 55–64
			5 65–74
			6 75+
[,2]	"alcgp"	Alcohol consumption	1 0–39 gm/day
			2 40–79
			3 80–119
			4 120+
[,3]	"tobgp"	Tobacco consumption	1 0– 9 gm/day
			2 10–19
			3 20–29
			4 30+
[,4]	"ncases"	Number of cases	
[,5]	"ncontrols"	Number of controls	

Author(s)

Thomas Lumley

Source

Breslow, N. E. and Day, N. E. (1980) *Statistical Methods in Cancer Research. 1: The Analysis of Case-Control Studies.* IARC Lyon / Oxford University Press.

Examples

```
require(stats)
require(graphics) # for mosaicplot
summary(esoph)
## effects of alcohol, tobacco and interaction, age-adjusted
model1 <- glm(cbind(ncases, ncontrols) ~ agegp + tobgp * alcgp,
              data = esoph, family = binomial())
anova(model1)
## Try a linear effect of alcohol and tobacco
model2 <- glm(cbind(ncases, ncontrols) ~ agegp + unclass(tobgp)
```

```
                                          + unclass(alcgp),
              data = esoph, family = binomial())
summary(model2)
## Re-arrange data for a mosaic plot
ttt <- table(esoph$agegp, esoph$alcgp, esoph$tobgp)
ttt[ttt == 1] <- esoph$ncases
tt1 <- table(esoph$agegp, esoph$alcgp, esoph$tobgp)
tt1[tt1 == 1] <- esoph$ncontrols
tt <- array(c(ttt, tt1), c(dim(ttt),2),
            c(dimnames(ttt), list(c("Cancer", "control"))))
mosaicplot(tt, main = "esoph data set", color = TRUE)
```

`euro` *Conversion Rates of Euro Currencies*

Description

Conversion rates between the various Euro currencies.

Usage

```
euro
euro.cross
```

Format

`euro` is a named vector of length 11, `euro.cross` a matrix of size 11 by 11, with dimnames.

Details

The data set `euro` contains the value of 1 Euro in all currencies participating in the European monetary union (Austrian Schilling ATS, Belgian Franc BEF, German Mark DEM, Spanish Peseta ESP, Finnish Markka FIM, French Franc FRF, Irish Punt IEP, Italian Lira ITL, Luxembourg Franc LUF, Dutch Guilder NLG and Portugese Escudo PTE). These conversion rates were fixed by the European Union on December 31, 1998. To convert old prices to Euro prices, divide by the respective rate and round to 2 digits.

The data set `euro.cross` contains conversion rates between the various Euro currencies, i.e., the result of `outer(1 / euro, euro)`.

Examples

```
cbind(euro)

## These relations hold:
euro == signif(euro,6) # [6 digit precision in Euro's definition]
all(euro.cross == outer(1/euro, euro))

## Convert 20 Euro to Belgian Franc
20 * euro["BEF"]
## Convert 20 Austrian Schilling to Euro
20 / euro["ATS"]
## Convert 20 Spanish Pesetas to Italian Lira
20 * euro.cross["ESP", "ITL"]
```

```
require(graphics)
dotchart(euro,
         main = "euro data: 1 Euro in currency unit")
dotchart(1/euro,
         main = "euro data: 1 currency unit in Euros")
dotchart(log(euro, 10),
         main = "euro data: log10(1 Euro in currency unit)")
```

---

eurodist *Distances Between European Cities*

Description

The data give the road distances (in km) between 21 cities in Europe. The data are taken from a table in "The Cambridge Encyclopaedia".

Usage

```
eurodist
```

Format

A `dist` object based on 21 objects. (You must have the **stats** package loaded to have the methods for this kind of object available).

Source

Crystal, D. Ed. (1990) *The Cambridge Encyclopaedia*. Cambridge: Cambridge University Press,

---

EuStockMarkets *Daily Closing Prices of Major European Stock Indices, 1991–1998*

Description

Contains the daily closing prices of major European stock indices: Germany DAX (Ibis), Switzerland SMI, France CAC, and UK FTSE. The data are sampled in business time, i.e., weekends and holidays are omitted.

Usage

```
EuStockMarkets
```

Format

A multivariate time series with 1860 observations on 4 variables. The object is of class `"mts"`.

Source

The data were kindly provided by Erste Bank AG, Vienna, Austria.

`faithful` *Old Faithful Geyser Data*

Description

Waiting time between eruptions and the duration of the eruption for the Old Faithful geyser in Yellowstone National Park, Wyoming, USA.

Usage

```
faithful
```

Format

A data frame with 272 observations on 2 variables.

[,1]	eruptions	numeric	Eruption time in mins
[,2]	waiting	numeric	Waiting time to next eruption (in mins)

Details

A closer look at `faithful$eruptions` reveals that these are heavily rounded times originally in seconds, where multiples of 5 are more frequent than expected under non-human measurement. For a "better" version of the eruptions times, see the example below.

There are many versions of this dataset around: Azzalini and Bowman (1990) use a more complete version.

Source

W. Härdle.

References

Härdle, W. (1991) *Smoothing Techniques with Implementation in S*. New York: Springer.

Azzalini, A. and Bowman, A. W. (1990). A look at some data on the Old Faithful geyser. *Applied Statistics* **39**, 357–365.

See Also

`geyser` in package **MASS** for the Azzalini–Bowman version.

Examples

```
f.tit <-  "faithful data: Eruptions of Old Faithful"

ne60 <- round(e60 <- 60 * faithful$eruptions)
all.equal(e60, ne60)             # relative diff. ~ 1/10000
table(zapsmall(abs(e60 - ne60))) # 0, 0.02 or 0.04
faithful$better.eruptions <- ne60 / 60
te <- table(ne60)
te[te >= 4]                      # (too) many multiples of 5 !
plot(names(te), te, type="h", main = f.tit, xlab = "Eruption time (sec)")
```

```
plot(faithful[, -3], main = f.tit,
     xlab = "Eruption time (min)",
     ylab = "Waiting time to next eruption (min)")
lines(lowess(faithful$eruptions, faithful$waiting, f = 2/3, iter = 3),
      col = "red")
```

`Formaldehyde` *Determination of Formaldehyde*

Description

These data are from a chemical experiment to prepare a standard curve for the determination of formaldehyde by the addition of chromatropic acid and concentrated sulpuric acid and the reading of the resulting purple color on a spectophotometer.

Usage

```
Formaldehyde
```

Format

A data frame with 6 observations on 2 variables.

[,1]	carb	numeric	Carbohydrate (ml)
[,2]	optden	numeric	Optical Density

Source

Bennett, N. A. and N. L. Franklin (1954) *Statistical Analysis in Chemistry and the Chemical Industry*. New York: Wiley.

References

McNeil, D. R. (1977) *Interactive Data Analysis.* New York: Wiley.

Examples

```
require(stats)
plot(optden ~ carb, data = Formaldehyde,
     xlab = "Carbohydrate (ml)", ylab = "Optical Density",
     main = "Formaldehyde data", col = 4, las = 1)
abline(fm1 <- lm(optden ~ carb, data = Formaldehyde))
summary(fm1)
opar <- par(mfrow = c(2,2), oma = c(0, 0, 1.1, 0))
plot(fm1)
par(opar)
```

`freeny` *Freeny's Revenue Data*

Description

Freeny's data on quarterly revenue and explanatory variables.

Usage

```
freeny
freeny.x
freeny.y
```

Format

There are three 'freeny' data sets.

`freeny.y` is a time series with 39 observations on quarterly revenue from (1962,2Q) to (1971,4Q).

`freeny.x` is a matrix of explanatory variables. The columns are `freeny.y` lagged 1 quarter, price index, income level, and market potential.

Finally, `freeny` is a data frame with variables `y`, `lag.quarterly.revenue`, `price.index`, `income.level`, and `market.potential` obtained from the above two data objects.

Source

A. E. Freeny (1977) *A Portable Linear Regression Package with Test Programs*. Bell Laboratories memorandum.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
summary(freeny)
pairs(freeny, main = "freeny data") # gives warning: freeny$y has class "ts"
summary(fm1 <- lm(y ~ ., data = freeny))
opar <- par(mfrow = c(2, 2), oma = c(0, 0, 1.1, 0),
            mar = c(4.1, 4.1, 2.1, 1.1))
plot(fm1)
par(opar)
```

`HairEyeColor` *Hair and Eye Color of Statistics Students*

Description

Distribution of hair and eye color and sex in 592 statistics students.

Usage

```
HairEyeColor
```

Format

A 3-dimensional array resulting from cross-tabulating 592 observations on 3 variables. The variables and their levels are as follows:

No	Name	Levels
1	Hair	Black, Brown, Red, Blond
2	Eye	Brown, Blue, Hazel, Green
3	Sex	Male, Female

Details

This data set is useful for illustrating various techniques for the analysis of contingency tables, such as the standard chi-squared test or, more generally, log-linear modelling, and graphical methods such as mosaic plots, sieve diagrams or association plots.

References

Snee, R. D. (1974), Graphical display of two-way contingency tables. *The American Statistician*, **28**, 9–12.

Friendly, M. (1992), Graphical methods for categorical data. *SAS User Group International Conference Proceedings*, **17**, 190–200. http://www.math.yorku.ca/SCS/sugi/sugi17-paper.html

Friendly, M. (1992), Mosaic displays for loglinear models. *Proceedings of the Statistical Graphics Section*, American Statistical Association, pp. 61–68. http://www.math.yorku.ca/SCS/Papers/asa92.html

See Also

`chisq.test`, `loglin`, `mosaicplot`

Examples

```
require(graphics)
## Full mosaic
mosaicplot(HairEyeColor)
## Aggregate over sex:
x <- apply(HairEyeColor, c(1, 2), sum)
x
mosaicplot(x, main = "Relation between hair and eye color")
```

Harman23.cor *Harman Example 2.3*

Description

A correlation matrix of eight physical measurements on 305 girls between ages seven and seventeen.

Usage

```
Harman23.cor
```

Source

Harman, H. H. (1976) *Modern Factor Analysis*, Third Edition Revised, University of Chicago Press, Table 2.3.

Examples

```
require(stats)
(Harman23.FA <- factanal(factors = 1, covmat = Harman23.cor))
for(factors in 2:4) print(update(Harman23.FA, factors = factors))
```

Harman74.cor *Harman Example 7.4*

Description

A correlation matrix of 24 psychological tests given to 145 seventh and eight-grade children in a Chicago suburb by Holzinger and Swineford.

Usage

```
Harman74.cor
```

Source

Harman, H. H. (1976) *Modern Factor Analysis*, Third Edition Revised, University of Chicago Press, Table 7.4.

Examples

```
require(stats)
(Harman74.FA <- factanal(factors = 1, covmat = Harman74.cor))
for(factors in 2:5) print(update(Harman74.FA, factors = factors))
Harman74.FA <- factanal(factors = 5, covmat = Harman74.cor,
                        rotation="promax")
print(Harman74.FA$loadings, sort = TRUE)
```

`Indometh` *Pharmacokinetics of Indomethicin*

Description

The `Indometh` data frame has 66 rows and 3 columns of data on the pharmacokinetics of indomethicin.

Usage

```
Indometh
```

Format

This data frame contains the following columns:

Subject an ordered factor with containing the subject codes. The ordering is according to increasing maximum response.

time a numeric vector of times at which blood samples were drawn (hr).

conc a numeric vector of plasma concentrations of indomethicin (mcg/ml).

Details

Each of the six subjects were given an intravenous injection of indomethicin.

Source

Kwan, Breault, Umbenhauer, McMahon and Duggan (1976), "Kinetics of Indomethicin absorption, elimination, and enterohepatic circulation in man", *Journal of Pharmacokinetics and Biopharmaceutics*, **4**, 255–280.

Davidian, M. and Giltinan, D. M. (1995) *Nonlinear Models for Repeated Measurement Data*, Chapman & Hall (section 5.2.4, p. 134)

Pinheiro, J. C. and Bates, D. M. (2000) *Mixed-effects Models in S and S-PLUS*, Springer.

Examples

```
require(stats)
fm1 <- nls(conc ~ SSbiexp(time, A1, lrc1, A2, lrc2),
           data = Indometh, subset = Subject == 1)
summary(fm1)
```

`infert` *Infertility after Spontaneous and Induced Abortion*

Description

This is a matched case-control study dating from before the availability of conditional logistic regression.

Usage

```
infert
```

Format

1.	Education	0 = 0-5 years
		1 = 6-11 years
		2 = 12+ years
2.	age	age in years of case
3.	parity	count
4.	number of prior	0 = 0
	induced abortions	1 = 1
		2 = 2 or more
5.	case status	1 = case
		0 = control
6.	number of prior	0 = 0
	spontaneous abortions	1 = 1
		2 = 2 or more
7.	matched set number	1-83
8.	stratum number	1-63

Note

One case with two prior spontaneous abortions and two prior induced abortions is omitted.

Source

Trichopoulos et al. (1976) *Br. J. of Obst. and Gynaec.* **83**, 645–650.

Examples

```
require(stats)
model1 <- glm(case ~ spontaneous+induced, data=infert,family=binomial())
summary(model1)
## adjusted for other potential confounders:
summary(model2 <- glm(case ~ age+parity+education+spontaneous+induced,
                data=infert,family=binomial()))
## Really should be analysed by conditional logistic regression
## which is in the survival package
if(require(survival)){
```

```
  model3 <- clogit(case~spontaneous+induced+strata(stratum),data=infert)
  print(summary(model3))
  detach()# survival (conflicts)
}
```

`InsectSprays` *Effectiveness of Insect Sprays*

Description

The counts of insects in agricultural experimental units treated with different insecticides.

Usage

```
InsectSprays
```

Format

A data frame with 72 observations on 2 variables.

[,1]	count	numeric	Insect count
[,2]	spray	factor	The type of spray

Source

Beall, G., (1942) The Transformation of data from entomological field experiments, *Biometrika*, **29**, 243–262.

References

McNeil, D. (1977) *Interactive Data Analysis*. New York: Wiley.

Examples

```
require(stats)
boxplot(count ~ spray, data = InsectSprays,
        xlab = "Type of spray", ylab = "Insect count",
        main = "InsectSprays data", varwidth = TRUE, col = "lightgray")
fm1 <- aov(count ~ spray, data = InsectSprays)
summary(fm1)
opar <- par(mfrow = c(2,2), oma = c(0, 0, 1.1, 0))
plot(fm1)
fm2 <- aov(sqrt(count) ~ spray, data = InsectSprays)
summary(fm2)
plot(fm2)
par(opar)
```

`iris` *Edgar Anderson's Iris Data*

Description

This famous (Fisher's or Anderson's) iris data set gives the measurements in centimeters of the variables sepal length and width and petal length and width, respectively, for 50 flowers from each of 3 species of iris. The species are *Iris setosa*, *versicolor*, and *virginica*.

Usage

```
iris
iris3
```

Format

`iris` is a data frame with 150 cases (rows) and 5 variables (columns) named `Sepal.Length`, `Sepal.Width`, `Petal.Length`, `Petal.Width`, and `Species`.

`iris3` gives the same data arranged as a 3-dimensional array of size 50 by 4 by 3, as represented by S-PLUS. The first dimension gives the case number within the species subsample, the second the measurements with names `Sepal L.`, `Sepal W.`, `Petal L.`, and `Petal W.`, and the third the species.

Source

Fisher, R. A. (1936) The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, **7**, Part II, 179–188.

The data were collected by Anderson, Edgar (1935). The irises of the Gaspe Peninsula, *Bulletin of the American Iris Society*, **59**, 2–5.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (has `iris3` as `iris`.)

See Also

`matplot` some examples of which use `iris`.

Examples

```
dni3 <- dimnames(iris3)
ii <- data.frame(matrix(aperm(iris3, c(1,3,2)), ncol=4,
                        dimnames = list(NULL, sub(" L.",".Length",
                                                  sub(" W.",".Width", dni3[[2]])))),
                 Species = gl(3, 50,
                              lab=sub("S", "s", sub("V", "v", dni3[[3]]))))
all.equal(ii, iris) # TRUE
```

islands *Areas of the World's Major Landmasses*

Description

The areas in thousands of square miles of the landmasses which exceed 10,000 square miles.

Usage

```
islands
```

Format

A named vector of length 48.

Source

The World Almanac and Book of Facts, 1975, page 406.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. Wiley.

Examples

```
require(graphics)
dotchart(log(islands, 10),
   main = "islands data: log10(area) (log10(sq. miles))")
dotchart(log(islands[order(islands)], 10),
   main = "islands data: log10(area) (log10(sq. miles))")
```

JohnsonJohnson *Quarterly Earnings per Johnson & Johnson Share*

Description

Quarterly earnings (dollars) per Johnson & Johnson share 1960–80.

Usage

```
JohnsonJohnson
```

Format

A quarterly time series

Source

Shumway, R. H. and Stoffer, D. S. (2000) *Time Series Analysis and its Applications*. Second Edition. Springer. Example 1.1.

Examples

```
require(stats)
JJ <- log10(JohnsonJohnson)
plot(JJ)
(fit <- StructTS(JJ, type="BSM"))
tsdiag(fit)
sm <- tsSmooth(fit)
plot(cbind(JJ, sm[, 1], sm[, 3]-0.5), plot.type = "single",
     col = c("black", "green", "blue"))
abline(h = -0.5, col = "grey60")

monthplot(fit)
```

LakeHuron *Level of Lake Huron 1875–1972*

Description

Annual measurements of the level, in feet, of Lake Huron 1875–1972.

Usage

```
LakeHuron
```

Format

A time series of length 98.

Source

Brockwell, P. J. & Davis, R. A. (1991). *Time Series and Forecasting Methods*. Second edition. Springer, New York. Series A, page 555.

Brockwell, P. J. & Davis, R. A. (1996). *Introduction to Time Series and Forecasting*. Springer, New York. Sections 5.1 and 7.6.

---

lh *Luteinizing Hormone in Blood Samples*

Description

A regular time series giving the luteinizing hormone in blood samples at 10 mins intervals from a human female, 48 samples.

Usage

```
lh
```

Source

P.J. Diggle (1990) *Time Series: A Biostatistical Introduction.* Oxford, table A.1, series 3

`LifeCycleSavings` *Intercountry Life-Cycle Savings Data*

Description

Data on the savings ratio 1960–1970.

Usage

```
LifeCycleSavings
```

Format

A data frame with 50 observations on 5 variables.

[,1]	sr	numeric	aggregate personal savings
[,2]	pop15	numeric	% of population under 15
[,3]	pop75	numeric	% of population over 75
[,4]	dpi	numeric	real per-capita disposable income
[,5]	ddpi	numeric	% growth rate of dpi

Details

Under the life-cycle savings hypothesis as developed by Franco Modigliani, the savings ratio (aggregate personal saving divided by disposable income) is explained by per-capita disposable income, the percentage rate of change in per-capita disposable income, and two demographic variables: the percentage of population less than 15 years old and the percentage of the population over 75 years old. The data are averaged over the decade 1960–1970 to remove the business cycle or other short-term fluctuations.

Source

The data were obtained from Belsley, Kuh and Welsch (1980). They in turn obtained the data from Sterling (1977).

References

Sterling, Arnie (1977) Unpublished BS Thesis. Massachusetts Institute of Technology.

Belsley, D. A., Kuh. E. and Welsch, R. E. (1980) *Regression Diagnostics*. New York: Wiley.

Examples

```
require(stats)
pairs(LifeCycleSavings, panel = panel.smooth,
      main = "LifeCycleSavings data")
fm1 <- lm(sr ~ pop15 + pop75 + dpi + ddpi, data = LifeCycleSavings)
summary(fm1)
```

Loblolly *Growth of Loblolly pine trees*

Description

The `Loblolly` data frame has 84 rows and 3 columns of records of the growth of Loblolly pine trees.

Usage

```
Loblolly
```

Format

This data frame contains the following columns:

height a numeric vector of tree heights (ft).

age a numeric vector of tree ages (yr).

Seed an ordered factor indicating the seed source for the tree. The ordering is according to increasing maximum height.

Source

Kung, F. H. (1986), "Fitting logistic growth curve with predetermined carrying capacity", *Proceedings of the Statistical Computing Section, American Statistical Association*, 340–343.

Pinheiro, J. C. and Bates, D. M. (2000) *Mixed-effects Models in S and S-PLUS*, Springer.

Examples

```
require(stats)
plot(height ~ age, data = Loblolly, subset = Seed == 329,
     xlab = "Tree age (yr)", las = 1,
     ylab = "Tree height (ft)",
     main = "Loblolly data and fitted curve (Seed 329 only)")
fm1 <- nls(height ~ SSasymp(age, Asym, R0, lrc),
           data = Loblolly, subset = Seed == 329)
summary(fm1)
age <- seq(0, 30, len = 101)
lines(age, predict(fm1, list(age = age)))
```

longley *Longley's Economic Regression Data*

Description

A macroeconomic data set which provides a well-known example for a highly collinear regression.

Usage

```
longley
```

Format

A data frame with 7 economical variables, observed yearly from 1947 to 1962 ($n = 16$).

GNP.deflator: GNP implicit price deflator ($1954 = 100$)

GNP: Gross National Product.

Unemployed: number of unemployed.

Armed.Forces: number of people in the armed forces.

Population: 'noninstitutionalized' population $\geq$ 14 years of age.

Year: the year (time).

Employed: number of people employed.

The regression `lm(Employed ~ .)` is known to be highly collinear.

Source

J. W. Longley (1967) An appraisal of least-squares programs from the point of view of the user. *Journal of the American Statistical Association*, **62**, 819–841.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
## give the data set in the form it is used in S-PLUS:
longley.x <- data.matrix(longley[, 1:6])
longley.y <- longley[, "Employed"]
pairs(longley, main = "longley data")
summary(fm1 <- lm(Employed ~ ., data = longley))
opar <- par(mfrow = c(2, 2), oma = c(0, 0, 1.1, 0),
            mar = c(4.1, 4.1, 2.1, 1.1))
plot(fm1)
par(opar)
```

`lynx` *Annual Canadian Lynx trappings 1821–1934*

Description

Annual numbers of lynx trappings for 1821–1934 in Canada. Taken from Brockwell & Davis (1991), this appears to be the series considered by Campbell & Walker (1977).

Usage

```
lynx
```

Source

Brockwell, P. J. and Davis, R. A. (1991) *Time Series and Forecasting Methods.* Second edition. Springer. Series G (page 557).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Campbell, M. J.and A. M. Walker (1977). A Survey of statistical work on the Mackenzie River series of annual Canadian lynx trappings for the years 1821–1934 and a new analysis. *Journal of the Royal Statistical Society series A*, **140**, 411–431.

`morley` *Michaelson-Morley Speed of Light Data*

Description

The classical data of Michaelson and Morley on the speed of light. The data consists of five experiments, each consisting of 20 consecutive 'runs'. The response is the speed of light measurement, suitably coded.

Usage

```
morley
```

Format

A data frame contains the following components:

`Expt` The experiment number, from 1 to 5.

`Run` The run number within each experiment.

`Speed` Speed-of-light measurement.

Details

The data is here viewed as a randomized block experiment with 'experiment' and 'run' as the factors. 'run' may also be considered a quantitative variate to account for linear (or polynomial) changes in the measurement over the course of a single experiment.

Source

A. J. Weekes (1986) *A Genstat Primer*. London: Edward Arnold.

Examples

```
require(stats)
morley$Expt <- factor(morley$Expt)
morley$Run <- factor(morley$Run)
attach(morley)
plot(Expt, Speed, main = "Speed of Light Data", xlab = "Experiment No.")
fm <- aov(Speed ~ Run + Expt, data = morley)
summary(fm)
fm0 <- update(fm, . ~ . - Run)
anova(fm0, fm)
detach(morley)
```

mtcars *Motor Trend Car Road Tests*

Description

The data was extracted from the 1974 *Motor Trend* US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models).

Usage

```
mtcars
```

Format

A data frame with 32 observations on 11 variables.

[, 1]	mpg	Miles/(US) gallon
[, 2]	cyl	Number of cylinders
[, 3]	disp	Displacement (cu.in.)
[, 4]	hp	Gross horsepower
[, 5]	drat	Rear axle ratio
[, 6]	wt	Weight (lb/1000)
[, 7]	qsec	1/4 mile time
[, 8]	vs	V/S
[, 9]	am	Transmission (0 = automatic, 1 = manual)
[,10]	gear	Number of forward gears
[,11]	carb	Number of carburetors

Source

Henderson and Velleman (1981), Building multiple regression models interactively. *Biometrics*, **37**, 391–411.

Examples

```
pairs(mtcars, main = "mtcars data")
coplot(mpg ~ disp | as.factor(cyl), data = mtcars,
       panel = panel.smooth, rows = 1)
```

nhtemp *Average Yearly Temperatures in New Haven*

Description

The mean annual temperature in degrees Fahrenheit in New Haven, Connecticut, from 1912 to 1971.

Usage

```
nhtemp
```

Format

A time series of 60 observations.

Source

Vaux, J. E. and Brinker, N. B. (1972) *Cycles*, **1972**, 117–121.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

Examples

```
plot(nhtemp, main = "nhtemp data",
  ylab = "Mean annual temperature in New Haven, CT (deg. F)")
```

Nile — *Flow of the River Nile*

Description

Measurements of the annual flow of the river Nile at Ashwan 1871–1970.

Usage

```
Nile
```

Format

A time series of length 100.

Source

Durbin, J. and Koopman, S. J. (2001) *Time Series Analysis by State Space Methods.* Oxford University Press. http://www.ssfpack.com/dkbook/

References

Balke, N. S. (1993) Detecting level shifts in time series. *Journal of Business and Economic Statistics* **11**, 81–92.

Cobb, G. W. (1978) The problem of the Nile: conditional solution to a change-point problem. *Biometrika* **65**, 243–51.

Examples

```
require(stats)
par(mfrow = c(2,2))
plot(Nile)
acf(Nile)
pacf(Nile)
ar(Nile) # selects order 2
cpgram(ar(Nile)$resid)
par(mfrow = c(1,1))
arima(Nile, c(2, 0, 0))

## Now consider missing values, following Durbin & Koopman
NileNA <- Nile
NileNA[c(21:40, 61:80)] <- NA
arima(NileNA, c(2, 0, 0))
plot(NileNA)
pred <- predict(arima(window(NileNA, 1871, 1890), c(2,0,0)), n.ahead = 20)
lines(pred$pred, lty = 3, col = "red")
lines(pred$pred + 2*pred$se, lty=2, col="blue")
lines(pred$pred - 2*pred$se, lty=2, col="blue")
pred <- predict(arima(window(NileNA, 1871, 1930), c(2,0,0)), n.ahead = 20)
lines(pred$pred, lty = 3, col = "red")
lines(pred$pred + 2*pred$se, lty=2, col="blue")
lines(pred$pred - 2*pred$se, lty=2, col="blue")

## Structural time series models
par(mfrow = c(3, 1))
plot(Nile)
## local level model
(fit <- StructTS(Nile, type = "level"))
lines(fitted(fit), lty = 2)              # contempareneous smoothing
lines(tsSmooth(fit), lty = 2, col = 4)   # fixed-interval smoothing
plot(residuals(fit)); abline(h = 0, lty = 3)
## local trend model
(fit2 <- StructTS(Nile, type = "trend")) ## constant trend fitted
pred <- predict(fit, n.ahead = 30)
## with 50% confidence interval
ts.plot(Nile, pred$pred, pred$pred + 0.67*pred$se, pred$pred -0.67*pred$se)

## Now consider missing values
plot(NileNA)
(fit3 <- StructTS(NileNA, type = "level"))
lines(fitted(fit3), lty = 2)
lines(tsSmooth(fit3), lty = 3)
plot(residuals(fit3)); abline(h = 0, lty = 3)
```

`nottem` *Average Monthly Temperatures at Nottingham, 1920–1939*

Description

A time series object containing average air temperatures at Nottingham Castle in degrees Fahrenheit for 20 years.

Usage

```
nottem
```

Source

Anderson, O. D. (1976) *Time Series Analysis and Forecasting: The Box-Jenkins approach.* Butterworths. Series R.

Examples

```
## Not run:
nott <- window(nottem, end=c(1936,12))
fit <- arima(nott,order=c(1,0,0), list(order=c(2,1,0), period=12))
nott.fore <- predict(fit, n.ahead=36)
ts.plot(nott, nott.fore$pred, nott.fore$pred+2*nott.fore$se,
        nott.fore$pred-2*nott.fore$se, gpars=list(col=c(1,1,4,4)))
## End(Not run)
```

Orange *Growth of orange trees*

Description

The `Orange` data frame has 35 rows and 3 columns of records of the growth of orange trees.

Usage

```
Orange
```

Format

This data frame contains the following columns:

Tree an ordered factor indicating the tree on which the measurement is made. The ordering is according to increasing maximum diameter.

age a numeric vector giving the age of the tree (days since 1968/12/31)

circumference a numeric vector of trunk circumferences (mm). This is probably "circumference at breast height", a standard measurement in forestry.

Source

Draper, N. R. and Smith, H. (1998), *Applied Regression Analysis (3rd ed)*, Wiley (exercise 24.N).

Pinheiro, J. C. and Bates, D. M. (2000) *Mixed-effects Models in S and S-PLUS*, Springer.

Examples

```
require(stats)
coplot(circumference ~ age | Tree, data = Orange, show = FALSE)
fm1 <- nls(circumference ~ SSlogis(age, Asym, xmid, scal),
           data = Orange, subset = Tree == 3)
plot(circumference ~ age, data = Orange, subset = Tree == 3,
     xlab = "Tree age (days since 1968/12/31)",
     ylab = "Tree circumference (mm)", las = 1,
     main = "Orange tree data and fitted model (Tree 3 only)")
age <- seq(0, 1600, len = 101)
lines(age, predict(fm1, list(age = age)))
```

OrchardSprays *Potency of Orchard Sprays*

Description

An experiment was conducted to assess the potency of various constituents of orchard sprays in repelling honeybees, using a Latin square design.

Usage

```
OrchardSprays
```

Format

A data frame with 64 observations on 4 variables.

[,1]	rowpos	numeric	Row of the design
[,2]	colpos	numeric	Column of the design
[,3]	treatment	factor	Treatment level
[,4]	decrease	numeric	Response

Details

Individual cells of dry comb were filled with measured amounts of lime sulphur emulsion in sucrose solution. Seven different concentrations of lime sulphur ranging from a concentration of 1/100 to 1/1,562,500 in successive factors of 1/5 were used as well as a solution containing no lime sulphur.

The responses for the different solutions were obtained by releasing 100 bees into the chamber for two hours, and then measuring the decrease in volume of the solutions in the various cells.

An 8×8 Latin square design was used and the treatments were coded as follows:

A	highest level of lime sulphur
B	next highest level of lime sulphur
.	
.	
.	
G	lowest level of lime sulphur
H	no lime sulphur

Source

Finney, D. J. (1947) *Probit Analysis*. Cambridge.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

Examples

```
pairs(OrchardSprays, main = "OrchardSprays data")
```

PlantGrowth *Results from an Experiment on Plant Growth*

Description

Results from an experiment to compare yields (as measured by dried weight of plants) obtained under a control and two different treatment conditions.

Usage

```
PlantGrowth
```

Format

A data frame of 30 cases on 2 variables.

[, 1]	weight	numeric
[, 2]	group	factor

The levels of `group` are 'ctrl', 'trt1', and 'trt2'.

Source

Dobson, A. J. (1983) *An Introduction to Statistical Modelling*. London: Chapman and Hall.

Examples

```
## One factor ANOVA example from Dobson's book, cf. Table 7.4:
require(stats)
boxplot(weight ~ group, data = PlantGrowth, main = "PlantGrowth data",
        ylab = "Dried weight of plants", col = "lightgray",
        notch = TRUE, varwidth = TRUE)
anova(lm(weight ~ group, data = PlantGrowth))
```

---

precip *Annual Precipitation in US Cities*

Description

The average amount of precipitation (rainfall) in inches for each of 70 United States (and Puerto Rico) cities.

Usage

```
precip
```

Format

A named vector of length 70.

Source

Statistical Abstracts of the United States, 1975.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

Examples

```
require(graphics)
dotchart(precip[order(precip)], main = "precip data")
title(sub = "Average annual precipitation (in.)")
```

---

presidents — *Quarterly Approval Ratings of US Presidents*

Description

The (approximately) quarterly approval rating for the President of the United states from the first quarter of 1945 to the last quarter of 1974.

Usage

```
presidents
```

Format

A time series of 120 values.

Details

The data are actually a fudged version of the approval ratings. See McNeil's book for details.

Source

The Gallup Organisation.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

Examples

```
plot(presidents, las = 1, ylab = "Approval rating (%)",
     main = "presidents data")
```

---

pressure *Vapor Pressure of Mercury as a Function of Temperature*

Description

Data on the relation between temperature in degrees Celsius and vapor pressure of mercury in millimeters (of mercury).

Usage

```
pressure
```

Format

A data frame with 19 observations on 2 variables.

[, 1]	temperature	numeric	temperature (deg C)
[, 2]	pressure	numeric	pressure (mm)

Source

Weast, R. C., ed. (1973) *Handbook of Chemistry and Physics*. CRC Press.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

Examples

```
plot(pressure, xlab = "Temperature (deg C)",
     ylab = "Pressure (mm of Hg)",
     main = "pressure data: Vapor Pressure of Mercury")
plot(pressure, xlab = "Temperature (deg C)",  log = "y",
     ylab = "Pressure (mm of Hg)",
     main = "pressure data: Vapor Pressure of Mercury")
```

Puromycin *Reaction velocity of an enzymatic reaction*

Description

The `Puromycin` data frame has 23 rows and 3 columns of the reaction velocity versus substrate concentration in an enzymatic reaction involving untreated cells or cells treated with Puromycin.

Usage

```
Puromycin
```

Format

This data frame contains the following columns:

conc a numeric vector of substrate concentrations (ppm)

rate a numeric vector of instantaneous reaction rates (counts/min/min)

state a factor with levels `treated` `untreated`

Details

Data on the "velocity" of an enzymatic reaction were obtained by Treloar (1974). The number of counts per minute of radioactive product from the reaction was measured as a function of substrate concentration in parts per million (ppm) and from these counts the initial rate, or "velocity," of the reaction was calculated (counts/min/min). The experiment was conducted once with the enzyme treated with Puromycin, and once with the enzyme untreated.

Source

Bates, D.M. and Watts, D.G. (1988), *Nonlinear Regression Analysis and Its Applications*, Wiley, Appendix A1.3.

Treloar, M. A. (1974), *Effects of Puromycin on Galactosyltransferase in Golgi Membranes*, M.Sc. Thesis, U. of Toronto.

Examples

```
plot(rate ~ conc, data = Puromycin, las = 1,
     xlab = "Substrate concentration (ppm)",
     ylab = "Reaction velocity (counts/min/min)",
     pch = as.integer(Puromycin$state),
     col = as.integer(Puromycin$state),
     main = "Puromycin data and fitted Michaelis-Menten curves")
## simplest form of fitting the Michaelis-Menten model to these data
fm1 <- nls(rate ~ Vm * conc/(K + conc), data = Puromycin,
           subset = state == "treated",
           start = c(Vm = 200, K = 0.05), trace = TRUE)
fm2 <- nls(rate ~ Vm * conc/(K + conc), data = Puromycin,
           subset = state == "untreated",
           start = c(Vm = 160, K = 0.05), trace = TRUE)
summary(fm1)
summary(fm2)
## using partial linearity
fm3 <- nls(rate ~ conc/(K + conc), data = Puromycin,
           subset = state == "treated", start = c(K = 0.05),
           algorithm = "plinear", trace = TRUE)
## using a self-starting model
fm4 <- nls(rate ~ SSmicmen(conc, Vm, K), data = Puromycin,
           subset = state == "treated")
summary(fm4)
## add fitted lines to the plot
conc <- seq(0, 1.2, len = 101)
lines(conc, predict(fm1, list(conc = conc)), lty = 1, col = 1)
lines(conc, predict(fm2, list(conc = conc)), lty = 2, col = 2)
legend(0.8, 120, levels(Puromycin$state),
       col = 1:2, lty = 1:2, pch = 1:2)
```

quakes *Locations of Earthquakes off Fiji*

Description

The data set give the locations of 1000 seismic events of MB > 4.0. The events occurred in a cube near Fiji since 1964.

Usage

```
quakes
```

Format

A data frame with 1000 observations on 5 variables.

[,1]	lat	numeric	Latitude of event
[,2]	long	numeric	Longitude
[,3]	depth	numeric	Depth (km)
[,4]	mag	numeric	Richter Magnitude
[,5]	stations	numeric	Number of stations reporting

Details

There are two clear planes of seismic activity. One is a major plate junction; the other is the Tonga trench off New Zealand. These data constitute a subsample from a larger dataset of containing 5000 observations.

Source

This is one of the Harvard PRIM-H project data sets. They in turn obtained it from Dr. John Woodhouse, Dept. of Geophysics, Harvard University.

Examples

```
pairs(quakes, main = "Fiji Earthquakes, N = 1000", cex.main=1.2, pch=".")
```

randu *Random Numbers from Congruential Generator RANDU*

Description

400 triples of successive random numbers were taken from the VAX FORTRAN function RANDU running under VMS 1.5.

Usage

```
randu
```

Format

A data frame with 400 observations on 3 variables named `x`, `y` and `z` which give the first, second and third random number in the triple.

Details

In three dimensional displays it is evident that the triples fall on 15 parallel planes in 3-space. This can be shown theoretically to be true for all triples from the RANDU generator.

These particular 400 triples start 5 apart in the sequence, that is they are ((U[5i+1], U[5i+2], U[5i+3]), i= 0, . . . , 399), and they are rounded to 6 decimal places.

Under VMS versions 2.0 and higher, this problem has been fixed.

Source

David Donoho

Examples

```
## Not run:
## We could re-generate the dataset by the following R code
seed <- as.double(1)
RANDU <- function() {
    seed <<- ((2^16 + 3) * seed) %% (2^31)
    seed/(2^31)
}
for(i in 1:400) {
    U <- c(RANDU(), RANDU(), RANDU(), RANDU(), RANDU())
    print(round(U[1:3], 6))
}
## End(Not run)
```

`rivers` *Lengths of Major North American Rivers*

Description

This data set gives the lengths (in miles) of 141 "major" rivers in North America, as compiled by the US Geological Survey.

Usage

```
rivers
```

Format

A vector containing 141 observations.

Source

World Almanac and Book of Facts, 1975, page 406.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

`rock` *Measurements on Petroleum Rock Samples*

Description

Measurements on 48 rock samples from a petroleum reservoir.

Usage

```
rock
```

Format

A data frame with 48 rows and 4 numeric columns.

[,1]	area	area of pores space, in pixels out of 256 by 256
[,2]	peri	perimeter in pixels
[,3]	shape	perimeter/sqrt(area)
[,4]	perm	permeability in milli-Darcies

Details

Twelve core samples from petroleum reservoirs were sampled by 4 cross-sections. Each core sample was measured for permeability, and each cross-section has total area of pores, total perimeter of pores, and shape.

Source

Data from BP Research, image analysis by Ronit Katz, U. Oxford.

`sleep` *Student's Sleep Data*

Description

Data which show the effect of two soporific drugs (increase in hours of sleep) on groups consisting of 10 patients each.

Usage

```
sleep
```

Format

A data frame with 20 observations on 2 variables.

[, 1]	extra	numeric	increase in hours of sleep
[, 2]	group	factor	patient group

Source

Student (1908) The probable error of the mean. *Biometrika*, **6**, 20.

References

Scheffé, Henry (1959) *The Analysis of Variance*. New York, NY: Wiley.

Examples

```
require(stats)
## ANOVA
anova(lm(extra ~ group, data = sleep))
```

---

stackloss *Brownlee's Stack Loss Plant Data*

Description

Operational data of a plant for the oxidation of ammonia to nitric acid.

Usage

```
stackloss

stack.x
stack.loss
```

Format

`stackloss` is a data frame with 21 observations on 4 variables.

[,1]	`Air Flow`	Flow of cooling air
[,2]	`Water Temp`	Cooling Water Inlet Temperature
[,3]	`Acid Conc.`	Concentration of acid [per 1000, minus 500]
[,4]	`stack.loss`	Stack loss

For compatibility with S-PLUS, the data sets `stack.x`, a matrix with the first three (independent) variables of the data frame, and `stack.loss`, the numeric vector giving the fourth (dependent) variable, are provided as well.

Details

"Obtained from 21 days of operation of a plant for the oxidation of ammonia (NH_3) to nitric acid (HNO_3). The nitric oxides produced are absorbed in a countercurrent absorption tower". (Brownlee, cited by Dodge, slightly reformatted by MM.)

`Air Flow` represents the rate of operation of the plant. `Water Temp` is the temperature of cooling water circulated through coils in the absorption tower. `Acid Conc.` is the concentration of the acid circulating, minus 50, times 10: that is, 89 corresponds to 58.9 per cent acid. `stack.loss` (the dependent variable) is 10 times the percentage of the ingoing ammonia to the plant that escapes from the absorption column unabsorbed; that is, an (inverse) measure of the over-all efficiency of the plant.

Source

Brownlee, K. A. (1960, 2nd ed. 1965) *Statistical Theory and Methodology in Science and Engineering*. New York: Wiley. pp. 491–500.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Dodge, Y. (1996) The guinea pig of multiple regression. In: *Robust Statistics, Data Analysis, and Computer Intensive Methods; In Honor of Peter Huber's 60th Birthday*, 1996, *Lecture Notes in Statistics* **109**, Springer-Verlag, New York.

Examples

```
summary(lm.stack <- lm(stack.loss ~ stack.x))
```

---

state *US State Facts and Figures*

Description

Data sets related to the 50 states of the United States of America.

Usage

```
state.abb
state.area
state.center
state.division
state.name
state.region
state.x77
```

Details

R currently contains the following "state" data sets. Note that all data are arranged according to alphabetical order of the state names.

`state.abb`: character vector of 2-letter abbreviations for the state names.

`state.area`: numeric vector of state areas (in square miles).

`state.center`: list with components named `x` and `y` giving the approximate geographic center of each state in negative longitude and latitude. Alaska and Hawaii are placed just off the West Coast.

`state.division`: factor giving state divisions (New England, Middle Atlantic, South Atlantic, East South Central, West South Central, East North Central, West North Central, Mountain, and Pacific).

`state.name`: character vector giving the full state names.

`state.region`: factor giving the region (Northeast, South, North Central, West) that each state belongs to.

state.x77: matrix with 50 rows and 8 columns giving the following statistics in the respective columns.

Population: population estimate as of July 1, 1975

Income: per capita income (1974)

Illiteracy: illiteracy (1970, percent of population)

Life Exp: life expectancy in years (1969–71)

Murder: murder and non-negligent manslaughter rate per 100,000 population (1976)

HS Grad: percent high-school graduates (1970)

Frost: mean number of days with minimum temperature below freezing (1931–1960) in capital or large city

Area: land area in square miles

Source

U.S. Department of Commerce, Bureau of the Census (1977) *Statistical Abstract of the United States*.

U.S. Department of Commerce, Bureau of the Census (1977) *County and City Data Book*.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

---

sunspot.month *Monthly Sunspot Data, 1749–1997*

Description

Monthly numbers of sunspots.

Usage

```
sunspot.month
```

Format

The univariate time series `sunspot.year` and `sunspot.month` contain 289 and 2988 observations, respectively. The objects are of class `"ts"`.

Source

World Data Center-C1 For Sunspot Index Royal Observatory of Belgium, Av. Circulaire, 3, B-1180 BRUSSELS http://www.oma.be/KSB-ORB/SIDC/sidc_txt.html

See Also

`sunspot.month` is a longer version of `sunspots` that runs until 1988 rather than 1983.

Examples

```
require(stats)
## Compare the monthly series
plot (sunspot.month, main = "sunspot.month [stats]", col = 2)
lines(sunspots) # "very barely" see something

## Now look at the difference :
all(tsp(sunspots)     [c(1,3)] ==
    tsp(sunspot.month)[c(1,3)]) ## Start & Periodicity are the same
n1 <- length(sunspots)
table(eq <- sunspots == sunspot.month[1:n1]) #>  132  are different !
i <- which(!eq)
rug(time(eq)[i])
s1 <- sunspots[i] ; s2 <- sunspot.month[i]
cbind(i = i, sunspots = s1, ss.month = s2,
      perc.diff = round(100*2*abs(s1-s2)/(s1+s2), 1))
```

---

sunspot.year *Yearly Sunspot Data, 1700–1988*

Description

Yearly numbers of sunspots.

Usage

```
sunspot.year
```

Format

The univariate time series `sunspot.year` contains 289 observations, and is of class `"ts"`.

Source

H. Tong (1996) *Non-Linear Time Series*. Clarendon Press, Oxford, p. 471.

---

sunspots *Monthly Sunspot Numbers, 1749–1983*

Description

Monthly mean relative sunspot numbers from 1749 to 1983. Collected at Swiss Federal Observatory, Zurich until 1960, then Tokyo Astronomical Observatory.

Usage

```
sunspots
```

Format

A time series of monthly data from 1749 to 1983.

Source

Andrews, D. F. and Herzberg, A. M. (1985) *Data: A Collection of Problems from Many Fields for the Student and Research Worker*. New York: Springer-Verlag.

See Also

`sunspot.month` has a longer (and a bit different) series.

Examples

```
plot(sunspots, main = "sunspots data", xlab = "Year",
     ylab = "Monthly sunspot numbers")
```

`swiss` *Swiss Fertility and Socioeconomic Indicators (1888) Data*

Description

Standardized fertility measure and socio-economic indicators for each of 47 French-speaking provinces of Switzerland at about 1888.

Usage

```
swiss
```

Format

A data frame with 47 observations on 6 variables, *each* of which is in percent, i.e., in $[0, 100]$.

[,1]	Fertility	I_g, "common standardized fertility measure"
[,2]	Agriculture	% of males involved in agriculture as occupation
[,3]	Examination	% "draftees" receiving highest mark on army examination
[,4]	Education	% education beyond primary school for "draftees".
[,5]	Catholic	% catholic (as opposed to "protestant").
[,6]	Infant.Mortality	live births who live less than 1 year.

All variables but 'Fertility' give proportions of the population.

Details

(paraphrasing Mosteller and Tukey):

Switzerland, in 1888, was entering a period known as the "demographic transition"; i.e., its fertility was beginning to fall from the high level typical of underdeveloped countries.

The data collected are for 47 French-speaking "provinces" at about 1888.

Here, all variables are scaled to $[0, 100]$, where in the original, all but `"Catholic"` were scaled to $[0, 1]$.

Note

Files for all 182 districts in 1888 and other years have been available at `http://opr.princeton.edu/archive/eufert/switz.html` or `http://opr.`

princeton.edu/archive/pefp/switz.asp.

They state that variables Examination and Education are averages for 1887, 1888 and 1889.

Source

Project "16P5", pages 549–551 in

Mosteller, F. and Tukey, J. W. (1977) *Data Analysis and Regression: A Second Course in Statistics*. Addison-Wesley, Reading Mass.

indicating their source as "Data used by permission of Franice van de Walle. Office of Population Research, Princeton University, 1976. Unpublished data assembled under NICHD contract number No 1-HD-O-2077."

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
pairs(swiss, panel = panel.smooth, main = "swiss data",
      col = 3 + (swiss$Catholic > 50))
summary(lm(Fertility ~ . , data = swiss))
```

Theoph *Pharmacokinetics of theophylline*

Description

The Theoph data frame has 132 rows and 5 columns of data from an experiment on the pharmacokinetics of theophylline.

Usage

```
Theoph
```

Format

This data frame contains the following columns:

Subject an ordered factor with levels 1, ..., 12 identifying the subject on whom the observation was made. The ordering is by increasing maximum concentration of theophylline observed.

Wt weight of the subject (kg).

Dose dose of theophylline administered orally to the subject (mg/kg).

Time time since drug administration when the sample was drawn (hr).

conc theophylline concentration in the sample (mg/L).

Details

Boeckmann, Sheiner and Beal (1994) report data from a study by Dr. Robert Upton of the kinetics of the anti-asthmatic drug theophylline. Twelve subjects were given oral doses of theophylline then serum concentrations were measured at 11 time points over the next 25 hours.

These data are analyzed in Davidian and Giltinan (1995) and Pinheiro and Bates (2000) using a two-compartment open pharmacokinetic model, for which a self-starting model function, `SSfol`, is available.

Source

Boeckmann, A. J., Sheiner, L. B. and Beal, S. L. (1994), *NONMEM Users Guide: Part V*, NONMEM Project Group, University of California, San Francisco.

Davidian, M. and Giltinan, D. M. (1995) *Nonlinear Models for Repeated Measurement Data*, Chapman & Hall (section 5.5, p. 145 and section 6.6, p. 176)

Pinheiro, J. C. and Bates, D. M. (2000) *Mixed-effects Models in S and S-PLUS*, Springer (Appendix A.29)

See Also

`SSfol`

Examples

```
require(stats)
coplot(conc ~ Time | Subject, data = Theoph, show = FALSE)
Theoph.4 <- subset(Theoph, Subject == 4)
fm1 <- nls(conc ~ SSfol(Dose, Time, lKe, lKa, lCl),
           data = Theoph.4)
summary(fm1)
plot(conc ~ Time, data = Theoph.4,
     xlab = "Time since drug administration (hr)",
     ylab = "Theophylline concentration (mg/L)",
     main = "Observed concentrations and fitted model",
     sub  = "Theophylline data - Subject 4 only",
     las = 1, col = 4)
xvals <- seq(0, par("usr")[2], len = 55)
lines(xvals, predict(fm1, newdata = list(Time = xvals)),
      col = 4)
```

`Titanic` *Survival of passengers on the Titanic*

Description

This data set provides information on the fate of passengers on the fatal maiden voyage of the ocean liner 'Titanic', summarized according to economic status (class), sex, age and survival.

Usage

```
Titanic
```

Format

A 4-dimensional array resulting from cross-tabulating 2201 observations on 4 variables. The variables and their levels are as follows:

No	Name	Levels
1	Class	1st, 2nd, 3rd, Crew
2	Sex	Male, Female
3	Age	Child, Adult
4	Survived	No, Yes

Details

The sinking of the Titanic is a famous event, and new books are still being published about it. Many well-known facts—from the proportions of first-class passengers to the "women and children first" policy, and the fact that that policy was not entirely successful in saving the women and children in the third class—are reflected in the survival rates for various classes of passenger.

These data were originally collected by the British Board of Trade in their investigation of the sinking. Note that there is not complete agreement among primary sources as to the exact numbers on board, rescued, or lost.

Due in particular to the very successful film 'Titanic', the last years saw a rise in public interest in the Titanic. Very detailed data about the passengers is now available on the Internet, at sites such as *Encyclopedia Titanica* (http://www.rmplc.co.uk/eduweb/sites/phind).

Source

Dawson, Robert J. MacG. (1995), The 'Unusual Episode' Data Revisited. *Journal of Statistics Education*, **3**. http://www.amstat.org/publications/jse/v3n3/datasets.dawson.html

The source provides a data set recording class, sex, age, and survival status for each person on board of the Titanic, and is based on data originally collected by the British Board of Trade and reprinted in:

British Board of Trade (1990), *Report on the Loss of the 'Titanic' (S.S.)*. British Board of Trade Inquiry Report (reprint). Gloucester, UK: Allan Sutton Publishing.

Examples

```
require(graphics)
mosaicplot(Titanic, main = "Survival on the Titanic")
## Higher survival rates in children?
apply(Titanic, c(3, 4), sum)
## Higher survival rates in females?
apply(Titanic, c(2, 4), sum)
## Use loglm() in package 'MASS' for further analysis ...
```

`ToothGrowth` *The Effect of Vitamin C on Tooth Growth in Guinea Pigs*

Description

The response is the length of odontoblasts (teeth) in each of 10 guinea pigs at each of three dose levels of Vitamin C (0.5, 1, and 2 mg) with each of two delivery methods (orange juice or ascorbic acid).

Usage

```
ToothGrowth
```

Format

A data frame with 60 observations on 3 variables.

[,1]	len	numeric	Tooth length
[,2]	supp	factor	Supplement type (VC or OJ).
[,3]	dose	numeric	Dose in milligrams.

Source

C. I. Bliss (1952) *The Statistics of Bioassay*. Academic Press.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

Examples

```
coplot(len ~ dose | supp, data = ToothGrowth, panel = panel.smooth,
       xlab = "ToothGrowth data: length vs dose, given type of supplement")
```

`treering` *Yearly Treering Data, -6000–1979*

Description

Contains normalized tree-ring widths in dimensionless units.

Usage

```
treering
```

Format

A univariate time series with 7981 observations. The object is of class `"ts"`.

Each tree ring corresponds to one year.

Details

The data were recorded by Donald A. Graybill, 1980, from Gt Basin Bristlecone Pine 2805M, 3726-11810 in Methuselah Walk, California.

Source

Time Series Data Library: http://www-personal.buseco.monash.edu.au/~hyndman/TSDL/, series 'CA535.DAT'

References

For background on Bristlecone pines and Methuselah Walk, see http://www.sonic.net/bristlecone/; for some photos see http://www.ltrr.arizona.edu/~hallman/sitephotos/meth.html

---

trees — *Girth, Height and Volume for Black Cherry Trees*

Description

This data set provides measurements of the girth, height and volume of timber in 31 felled black cherry trees. Note that girth is the diameter of the tree (in inches) measured at 4 ft 6 in above the ground.

Usage

```
trees
```

Format

A data frame with 31 observations on 3 variables.

[,1]	Girth	numeric	Tree diameter in inches
[,2]	Height	numeric	Height in ft
[,3]	Volume	numeric	Volume of timber in cubic ft

Source

Ryan, T. A., Joiner, B. L. and Ryan, B. F. (1976) *The Minitab Student Handbook*. Duxbury Press.

References

Atkinson, A. C. (1985) *Plots, Transformations and Regression*. Oxford University Press.

Examples

```
pairs(trees, panel = panel.smooth, main = "trees data")
plot(Volume ~ Girth, data = trees, log = "xy")
coplot(log(Volume) ~ log(Girth) | Height, data = trees,
       panel = panel.smooth)
summary(fm1 <- lm(log(Volume) ~ log(Girth), data = trees))
summary(fm2 <- update(fm1, ~ . + log(Height), data = trees))
step(fm2)
## i.e., Volume ~= c * Height * Girth^2  seems reasonable
```

`UCBAdmissions` *Student Admissions at UC Berkeley*

Description

Aggregate data on applicants to graduate school at Berkeley for the six largest departments in 1973 classified by admission and sex.

Usage

```
UCBAdmissions
```

Format

A 3-dimensional array resulting from cross-tabulating 4526 observations on 3 variables. The variables and their levels are as follows:

No	Name	Levels
1	Admit	Admitted, Rejected
2	Gender	Male, Female
3	Dept	A, B, C, D, E, F

Details

This data set is frequently used for illustrating Simpson's paradox, see Bickel et al. (1975). At issue is whether the data show evidence of sex bias in admission practices. There were 2691 male applicants, of whom 1198 (44.5%) were admitted, compared with 1835 female applicants of whom 557 (30.4%) were admitted. This gives a sample odds ratio of 1.83, indicating that males were almost twice as likely to be admitted. In fact, graphical methods (as in the example below) or log-linear modelling show that the apparent association between admission and sex stems from differences in the tendency of males and females to apply to the individual departments (females used to apply "more" to departments with higher rejection rates).

This data set can also be used for illustrating methods for graphical display of categorical data, such as the general-purpose mosaic plot or the "fourfold display" for 2-by-2-by-k tables. See the home page of Michael Friendly (`http://www.math.yorku.ca/SCS/friendly.html`) for further information.

References

Bickel, P. J., Hammel, E. A., and O'Connell, J. W. (1975) Sex bias in graduate admissions: Data from Berkeley. *Science*, **187**, 398–403.

Examples

```
require(graphics)
## Data aggregated over departments
apply(UCBAdmissions, c(1, 2), sum)
mosaicplot(apply(UCBAdmissions, c(1, 2), sum),
           main = "Student admissions at UC Berkeley")
## Data for individual departments
opar <- par(mfrow = c(2, 3), oma = c(0, 0, 2, 0))
for(i in 1:6)
```

```
  mosaicplot(UCBAdmissions[,,i],
    xlab = "Admit", ylab = "Sex",
    main = paste("Department", LETTERS[i]))
mtext(expression(bold("Student admissions at UC Berkeley")),
      outer = TRUE, cex = 1.5)
par(opar)
```

`UKDriverDeaths` *Road Casualties in Great Britain 1969–84*

Description

`UKDriverDeaths` is a time series giving the monthly totals of car drivers in Great Britain killed or seriously injured Jan 1969 to Dec 1984. Compulsory wearing of seat belts was introduced on 31 Jan 1983.

`Seatbelts` is more information on the same problem.

Usage

```
UKDriverDeaths
Seatbelts
```

Format

`Seatbelts` is a multiple time series, with columns

`DriversKilled` car drivers killed.

`drivers` same as `UKDriverDeaths`.

`front` front-seat passengers killed or seriously injured.

`rear` rear-seat passengers killed or seriously injured.

`kms` distance driven.

`PetrolPrice` petrol price.

`VanKilled` number of van ('light goods vehicle') drivers.

`law` 0/1: was the law in effect that month?

Source

Harvey, A.C. (1989) *Forecasting, Structural Time Series Models and the Kalman Filter.* Cambridge University Press, pp. 519–523.

Durbin, J. and Koopman, S. J. (2001) *Time Series Analysis by State Space Methods.* Oxford University Press. http://www.ssfpack.com/dkbook/

References

Harvey, A. C. and Durbin, J. (1986) The effects of seat belt legislation on British road casualties: A case study in structural time series modelling. *Journal of the Royal Statistical Society* series B, **149**, 187–227.

Examples

```
require(stats)
## work with pre-seatbelt period to identify a model, use logs
work <- window(log10(UKDriverDeaths), end = 1982+11/12)
par(mfrow = c(3,1))
plot(work); acf(work); pacf(work)
par(mfrow = c(1,1))
(fit <- arima(work, c(1,0,0), seasonal = list(order= c(1,0,0))))
z <- predict(fit, n.ahead = 24)
ts.plot(log10(UKDriverDeaths), z$pred, z$pred+2*z$se, z$pred-2*z$se,
        lty = c(1,3,2,2), col = c("black", "red", "blue", "blue"))

## now see the effect of the explanatory variables
X <- Seatbelts[, c("kms", "PetrolPrice", "law")]
X[, 1] <- log10(X[, 1]) - 4
arima(log10(Seatbelts[, "drivers"]), c(1,0,0),
      seasonal = list(order= c(1,0,0)), xreg = X)
```

UKgas *UK Quarterly Gas Consumption*

Description

Quarterly UK gas consumption from 1960Q1 to 1986Q4, in millions of therms.

Usage

```
UKgas
```

Format

A quarterly time series of length 108.

Source

Durbin, J. and Koopman, S. J. (2001) *Time Series Analysis by State Space Methods.* Oxford University Press. http://www.ssfpack.com/dkbook/

Examples

```
## maybe str(UKgas) ; plot(UKgas) ...
```

UKLungDeaths *Monthly Deaths from Lung Diseases in the UK*

Description

Three time series giving the monthly deaths from bronchitis, emphysema and asthma in the UK, 1974–1979, both sexes (`ldeaths`), males (`mdeaths`) and females (`fdeaths`).

Usage

```
ldeaths
fdeaths
mdeaths
```

Source

P. J. Diggle (1990) *Time Series: A Biostatistical Introduction.* Oxford, table A.3

Examples

```
require(stats) # for time
plot(ldeaths)
plot(mdeaths, fdeaths)
## Better labels:
yr <- floor(tt <- time(mdeaths))
plot(mdeaths, fdeaths,
     xy.labels = paste(month.abb[12*(tt - yr)], yr-1900, sep="'"))
```

USAccDeaths *Accidental Deaths in the US 1973–1978*

Description

A time series giving the monthly totals of accidental deaths in the USA. The values for the first six months of 1979 are 7798 7406 8363 8460 9217 9316.

Usage

```
USAccDeaths
```

Source

P. J. Brockwell and R. A. Davis (1991) *Time Series: Theory and Methods.* Springer, New York.

USArrests *Violent Crime Rates by US State*

Description

This data set contains statistics, in arrests per 100,000 residents for assault, murder, and rape in each of the 50 US states in 1973. Also given is the percent of the population living in urban areas.

Usage

```
USArrests
```

Format

A data frame with 50 observations on 4 variables.

[,1]	Murder	numeric	Murder arrests (per 100,000)
[,2]	Assault	numeric	Assault arrests (per 100,000)
[,3]	UrbanPop	numeric	Percent urban population
[,4]	Rape	numeric	Rape arrests (per 100,000)

Source

World Almanac and Book of facts 1975. (Crime rates).

Statistical Abstracts of the United States 1975. (Urban rates).

References

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

See Also

The `state` data sets.

Examples

```
pairs(USArrests, panel = panel.smooth, main = "USArrests data")
```

USJudgeRatings *Lawyers' Ratings of State Judges in the US Superior Court*

Description

Lawyers' ratings of state judges in the US Superior Court.

Usage

```
USJudgeRatings
```

Format

A data frame containing 43 observations on 12 numeric variables.

[,1]	CONT	Number of contacts of lawyer with judge.
[,2]	INTG	Judicial integrity.
[,3]	DMNR	Demeanor.
[,4]	DILG	Diligence.
[,5]	CFMG	Case flow managing.
[,6]	DECI	Prompt decisions.
[,7]	PREP	Preparation for trial.
[,8]	FAMI	Familiarity with law.
[,9]	ORAL	Sound oral rulings.
[,10]	WRIT	Sound written rulings.
[,11]	PHYS	Physical ability.
[,12]	RTEN	Worthy of retention.

Source

New Haven Register, 14 January, 1977 (from John Hartigan).

Examples

```
pairs(USJudgeRatings, main = "USJudgeRatings data")
```

USPersonalExpenditure

Personal Expenditure Data

Description

This data set consists of United States personal expenditures (in billions of dollars) in the categories; food and tobacco, household operation, medical and health, personal care, and private education for the years 1940, 1945, 1950, 1955 and 1960.

Usage

```
USPersonalExpenditure
```

Format

A matrix with 5 rows and 5 columns.

Source

The World Almanac and Book of Facts, 1962, page 756.

References

Tukey, J. W. (1977) *Exploratory Data Analysis*. Addison-Wesley.

McNeil, D. R. (1977) *Interactive Data Analysis*. Wiley.

Examples

```
require(stats) # for medpolish
USPersonalExpenditure
medpolish(log10(USPersonalExpenditure))
```

uspop — *Populations Recorded by the US Census*

Description

This data set gives the population of the United States (in millions) as recorded by the decennial census for the period 1790–1970.

Usage

```
uspop
```

Format

A time series of 19 values.

Source

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

Examples

```
plot(uspop, log = "y", main = "uspop data", xlab = "Year",
     ylab = "U.S. Population (millions)")
```

VADeaths — *Death Rates in Virginia (1940)*

Description

Death rates per 1000 in Virginia in 1940.

Usage

```
VADeaths
```

Format

A matrix with 5 rows and 5 columns.

Details

The death rates are measured per 1000 population per year. They are cross-classified by age group (rows) and population group (columns). The age groups are: 50–54, 55–59, 60–64, 65–69, 70–74 and the population groups are Rural/Male, Rural/Female, Urban/Male and Urban/Female.

This provides a rather nice 3-way analysis of variance example.

Source

Moyneau, L., Gilliam, S. K., and Florant, L. C.(1947) Differences in Virginia death rates by color, sex, age, and rural or urban residence. *American Sociological Review*, **12**, 525–535.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. Wiley.

Examples

```
require(stats)
n <- length(dr <- c(VADeaths))
nam <- names(VADeaths)
d.VAD <- data.frame(
 Drate = dr,
 age = rep(ordered(rownames(VADeaths)),length=n),
 gender= gl(2,5,n, labels= c("M", "F")),
 site =  gl(2,10,  labels= c("rural", "urban")))
coplot(Drate ~ as.numeric(age) | gender * site, data = d.VAD,
       panel = panel.smooth, xlab = "VADeaths data - Given: gender")
summary(aov.VAD <- aov(Drate ~ .^2, data = d.VAD))
opar <- par(mfrow = c(2,2), oma = c(0, 0, 1.1, 0))
plot(aov.VAD)
par(opar)
```

---

volcano *Topographic Information on Auckland's Maunga Whau Volcano*

Description

Maunga Whau (Mt Eden) is one of about 50 volcanos in the Auckland volcanic field. This data set gives topographic information for Maunga Whau on a 10m by 10m grid.

Usage

```
volcano
```

Format

A matrix with 87 rows and 61 columns, rows corresponding to grid lines running east to west and columns to grid lines running south to north.

Source

Digitized from a topographic map by Ross Ihaka. These data should not be regarded as accurate.

See Also

`filled.contour` for a nice plot.

Examples

```
require(graphics)
filled.contour(volcano, color = terrain.colors, asp = 1)
title(main = "volcano data: filled contour map")
```

`warpbreaks` *The Number of Breaks in Yarn during Weaving*

Description

This data set gives the number of warp breaks per loom, where a loom corresponds to a fixed length of yarn.

Usage

```
warpbreaks
```

Format

A data frame with 54 observations on 3 variables.

`[,1]`	`breaks`	numeric	The number of breaks
`[,2]`	`wool`	factor	The type of wool (A or B)
`[,3]`	`tension`	factor	The level of tension (L, M, H)

There are measurements on 9 looms for each of the six types of warp (`AL`, `AM`, `AH`, `BL`, `BM`, `BH`).

Source

Tippett, L. H. C. (1950) *Technological Applications of Statistics*. Wiley. Page 106.

References

Tukey, J. W. (1977) *Exploratory Data Analysis*. Addison-Wesley.

McNeil, D. R. (1977) *Interactive Data Analysis*. Wiley.

See Also

`xtabs` for ways to display these data as a table.

Examples

```
summary(warpbreaks)
opar <- par(mfrow = c(1,2), oma = c(0, 0, 1.1, 0))
plot(breaks ~ tension, data = warpbreaks, col = "lightgray",
     varwidth = TRUE, subset = wool == "A", main = "Wool A")
plot(breaks ~ tension, data = warpbreaks, col = "lightgray",
     varwidth = TRUE, subset = wool == "B", main = "Wool B")
mtext("warpbreaks data", side = 3, outer = TRUE)
par(opar)
summary(fm1 <- lm(breaks ~ wool*tension, data = warpbreaks))
anova(fm1)
```

women *Average Heights and Weights for American Women*

Description

This data set gives the average heights and weights for American women aged 30–39.

Usage

```
women
```

Format

A data frame with 15 observations on 2 variables.

`[,1]`	`height`	numeric	Height (in)
`[,2]`	`weight`	numeric	Weight (lbs)

Details

The data set appears to have been taken from the American Society of Actuaries *Build and Blood Pressure Study* for some (unknown to us) earlier year.

The World Almanac notes: "The figures represent weights in ordinary indoor clothing and shoes, and heights with shoes".

Source

The World Almanac and Book of Facts, 1975.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. Wiley.

Examples

```
plot(women, xlab = "Height (in)", ylab = "Weight (lb)",
     main = "women data: American women aged 30-39")
```

WorldPhones *The World's Telephones*

Description

The number of telephones in various regions of the world (in thousands).

Usage

```
phones
```

Format

A matrix with 7 rows and 8 columns. The columns of the matrix give the figures for a given region, and the rows the figures for a year.

The regions are: North America, Europe, Asia, South America, Oceania, Africa, Central America.

The years are: 1951, 1956, 1957, 1958, 1959, 1960, 1961.

Source

AT&T (1961) *The World's Telephones*.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

Examples

```
matplot(rownames(WorldPhones), WorldPhones, type = "b", log = "y",
        xlab = "Year", ylab = "Number of telephones (1000's)")
legend(1951.5, 80000, colnames(WorldPhones), col = 1:6, lty = 1:5,
       pch = rep(21, 7))
title(main = "World phones data: log scale for response")
```

WWWusage *Internet Usage per Minute*

Description

A time series of the numbers of users connected to the Internet through a server every minute.

Usage

```
WWWusage
```

Format

A time series of length 100.

Source

Durbin, J. and Koopman, S. J. (2001) *Time Series Analysis by State Space Methods.* Oxford University Press. http://www.ssfpack.com/dkbook/

References

Makridakis, S., Wheelwright, S. C. and Hyndman, R. J. (1998) *Forecasting: Methods and Applications.* Wiley.

Examples

```
work <- diff(WWWusage)
par(mfrow = c(2,1)); plot(WWWusage); plot(work)
## Not run:
require(stats)
aics <- matrix(, 6, 6, dimnames=list(p=0:5, q=0:5))
for(q in 1:5) aics[1, 1+q] <- arima(WWWusage, c(0,1,q),
    optim.control = list(maxit = 500))$aic
for(p in 1:5)
   for(q in 0:5) aics[1+p, 1+q] <- arima(WWWusage, c(p,1,q),
       optim.control = list(maxit = 500))$aic
round(aics - min(aics, na.rm=TRUE), 2)
## End(Not run)
```

Chapter 3

The `grDevices` package

`grDevices-package` *The R Graphics Devices and Support for Colours and Fonts*

Description

Graphics devices and support for base and grid graphics

Details

This package contains functions which support both base and grid graphics.

For a complete list of functions, use `library(help="grDevices")`.

Author(s)

R Development Core Team and contributors worldwide

Maintainer: R Core Team ⟨R-core@r-project.org⟩

`boxplot.stats` *Box Plot Statistics*

Description

This function is typically called by another function to gather the statistics necessary for producing box plots, but may be invoked separately.

Usage

```
boxplot.stats(x, coef = 1.5, do.conf = TRUE, do.out = TRUE)
```

Arguments

x	a numeric vector for which the boxplot will be constructed (NAs and NaNs are allowed and omitted).
coef	this determines how far the plot "whiskers" extend out from the box. If coef is positive, the whiskers extend to the most extreme data point which is no more than coef times the length of the box away from the box. A value of zero causes the whiskers to extend to the data extremes (and no outliers be returned).
do.conf,do.out	logicals; if FALSE, the conf or out component respectively will be empty in the result.

Details

The two "hinges" are versions of the first and third quartile, i.e., close to `quantile(x, c(1,3)/4)`. The hinges equal the quartiles for odd n (where `n <- length(x)`) and differ for even n. Where the quartiles only equal observations for `n %% 4 == 1` ($n \equiv 1 \bmod 4$), the hinges do so *additionally* for `n %% 4 == 2` ($n \equiv 2 \bmod 4$), and are in the middle of two observations otherwise.

The notches (if requested) extend to `+/-1.58 IQR/sqrt(n)`. This seems to be based on same calculations as the formula with 1.57 in Chambers *et al.* (1983, p. 62), given in McGill *et al.* (1978, p. 16). They are based on asymptotic normality of the median and roughly equal sample sizes for the two medians being compared, and are said to be rather insensitive to the underlying distributions of the samples. The idea appears to be to give roughly a 95% confidence interval for the difference in two medians.

Value

List with named components as follows:

stats	a vector of length 5, containing the extreme of the lower whisker, the lower "hinge", the median, the upper "hinge" and the extreme of the upper whisker.
n	the number of non-NA observations in the sample.
conf	the lower and upper extremes of the "notch" (`if(do.conf)`). See the details.
out	the values of any data points which lie beyond the extremes of the whiskers (`if(do.out)`).

Note that `$stats` and `$conf` are sorted in *in*creasing order, unlike S, and that `$n` and `$out` include any `+- Inf` values.

References

Tukey, J. W. (1977) *Exploratory Data Analysis.* Section 2C.

McGill, R., Tukey, J. W. and Larsen, W. A. (1978) Variations of box plots. *The American Statistician* **32**, 12–16.

Velleman, P. F. and Hoaglin, D. C. (1981) *Applications, Basics and Computing of Exploratory Data Analysis.* Duxbury Press.

Emerson, J. D and Strenio, J. (1983). Boxplots and batch comparison. Chapter 3 of *Understanding Robust and Exploratory Data Analysis*, eds. D. C. Hoaglin, F. Mosteller and J. W. Tukey. Wiley.

Chambers, J. M., Cleveland, W. S., Kleiner, B. and Tukey, P. A. (1983) *Graphical Methods for Data Analysis.* Wadsworth & Brooks/Cole.

See Also

fivenum, boxplot, bxp.

Examples

```
x <- c(1:100, 1000)
(b1 <- boxplot.stats(x))
(b2 <- boxplot.stats(x, do.conf=FALSE, do.out=FALSE))
stopifnot(b1 $ stats == b2 $ stats) # do.out=F is still robust
boxplot.stats(x, coef = 3, do.conf=FALSE)
## no outlier treatment:
boxplot.stats(x, coef = 0)

boxplot.stats(c(x, NA)) # slight change : n is 101
(r <- boxplot.stats(c(x, -1:1/0)))
stopifnot(r$out == c(1000, -Inf, Inf))
```

---

check.options *Set Options with Consistency Checks*

Description

Utility function for setting options with some consistency checks. The attributes of the new settings in new are checked for consistency with the *model* (often default) list in name.opt.

Usage

```
check.options(new, name.opt, reset = FALSE, assign.opt = FALSE,
              envir = .GlobalEnv,
              check.attributes = c("mode", "length"),
              override.check = FALSE)
```

Arguments

new	a *named* list
name.opt	character with the name of R object containing the "model" (default) list.
reset	logical; if TRUE, reset the options from name.opt. If there is more than one R object with name name.opt, remove the first one in the search() path.
assign.opt	logical; if TRUE, assign the ...
envir	the environment used for get and assign.
check.attributes	character containing the attributes which check.options should check.
override.check	logical vector of length length(new) (or 1 which entails recycling). For those new[i] where override.check[i] == TRUE, the checks are overriden and the changes made anyway.

Value

A list of components with the same names as the one called `name.opt`. The values of the components are changed from the `new` list, as long as these pass the checks (when these are not overridden according to `override.check`).

Author(s)

Martin Maechler

See Also

`ps.options` which uses `check.options`.

Examples

```
(L1 <- list(a=1:3, b=pi, ch="CH"))
check.options(list(a=0:2), name.opt = "L1")
check.options(NULL, reset = TRUE, name.opt = "L1")
```

---

chull *Compute Convex Hull of a Set of Points*

Description

Computes the subset of points which lie on the convex hull of the set of points specified.

Usage

```
chull(x, y = NULL)
```

Arguments

`x, y`	coordinate vectors of points. This can be specified as two vectors `x` and `y`, a 2-column matrix `x`, a list `x` with two components, etc, see `xy.coords`.

Details

`xy.coords` is used to interpret the specification of the points. The algorithm is that given by Eddy (1977).

'Peeling' as used in the S function `chull` can be implemented by calling `chull` recursively.

Value

An integer vector giving the indices of the points lying on the convex hull, in clockwise order.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Eddy, W. F. (1977) A new convex hull algorithm for planar sets. *ACM Transactions on Mathematical Software*, **3**, 398–403.

Eddy, W. F. (1977) Algorithm 523. CONVEX, A new convex hull algorithm for planar sets[Z]. *ACM Transactions on Mathematical Software*, **3**, 411–412.

See Also

xy.coords,polygon

Examples

```
X <- matrix(rnorm(2000), ncol = 2)
chull(X)
## Not run:
  # Example usage from graphics package
  plot(X, cex = 0.5)
  hpts <- chull(X)
  hpts <- c(hpts, hpts[1])
  lines(X[hpts, ])
## End(Not run)
```

---

cm *Unit tansformation*

Description

Translates from inches to cm (centimeters).

Usage

```
cm(x)
```

Arguments

x	numeric vector

Examples

```
cm(1)# = 2.54
```

---

col2rgb *Color to RGB Conversion*

Description

"Any R color" to RGB (red/green/blue) conversion.

Usage

```
col2rgb(col, alpha = FALSE)
```

Arguments

col	vector of any of the three kind of R colors, i.e., either a color name (an element of colors()), a hexadecimal string of the form "#rrggbb", or an integer i meaning palette()[i].
alpha	logical value indicating whether alpha channel values should be returned.

Details

For integer colors, `0` is shorthand for the current `par("bg")`, and `NA` means "nothing" which effectively does not draw the corresponding item.

For character colors, `"NA"` is equivalent to `NA` above.

Value

an integer matrix with three or four rows and number of columns the length (and names if any) as `col`.

Author(s)

Martin Maechler

See Also

`rgb`, `colors`, `palette`, etc.

Examples

```
col2rgb("peachpuff")
col2rgb(c(blu = "royalblue", reddish = "tomato")) # names kept

col2rgb(1:8)# the ones from the palette() :

col2rgb(paste("gold", 1:4, sep=""))

col2rgb("#08a0ff")
## all three kind of colors mixed :
col2rgb(c(red="red", palette= 1:3, hex="#abcdef"))

##-- NON-INTRODUCTORY examples --

grC <- col2rgb(paste("gray",0:100,sep=""))
table(print(diff(grC["red",])))# '2' or '3': almost equidistant
## The 'named' grays are in between {"slate gray" is not gray, strictly}
col2rgb(c(g66="gray66", darkg= "dark gray", g67="gray67",
          g74="gray74", gray =      "gray", g75="gray75",
          g82="gray82", light="light gray", g83="gray83"))

crgb <- col2rgb(cc <- colors())
colnames(crgb) <- cc
t(crgb)## The whole table

ccodes <- c(256^(2:0) %*% crgb)## = internal codes
## How many names are 'aliases' of each other:
table(tcc <- table(ccodes))
length(uc <- unique(sort(ccodes))) # 502
## All the multiply named colors:
mult <- uc[tcc >= 2]
cl <- lapply(mult, function(m) cc[ccodes == m])
names(cl) <- apply(col2rgb(sapply(cl, function(x)x[1])),
                   2, function(n)paste(n, collapse=","))
utils::str(cl)
## Not run:
```

```
 if(require(xgobi)) { ## Look at the color cube dynamically :
   tc <- t(crgb[, !duplicated(ccodes)])
   table(is.gray <- tc[,1] == tc[,2] & tc[,2] == tc[,3])# (397, 105)
   xgobi(tc, color = c("gold", "gray")[1 + is.gray])
 }
## End(Not run)
```

`colorRamp` *Color interpolation*

Description

These functions return functions that interpolate a set of given colors to create new color palettes (like `topo.colors`) and color ramps, functions that map the interval $[0, 1]$ to colors (like `grey`.

Usage

```
colorRamp(colors, bias = 1, space = c("rgb", "Lab"),
          interpolate = c("linear", "spline"))
colorRampPalette(colors, ...)
```

Arguments

`colors`	Colors to interpolate
`bias`	A positive number. Higher values give more widely spaced colors at the high end.
`space`	Interpolation in RGB or CIE Lab color spaces
`interpolate`	Use spline or linear interpolation.
`...`	arguments to pass to `colorRamp`.

Details

The CIE Lab color space is approximately perceptually uniform, and so gives smoother and more uniform color ramps. On the other hand, palettes that vary from one hue to another via white may have a more symmetrical appearance in RGB space.

The conversion formulas in this function do not appear to be completely accurate and the color ramp may not reach the extreme values in Lab space. Future changes in the R color model may change the colors produced with `space="Lab"`.

Value

`colorRamp` returns a function that maps values between 0 and 1 to colors. `colorRampPalette` returns a function that takes an integer argument and returns that number of colors interpolating the given sequence (similar to `heat.colors` or `terrain.colors`.

See Also

Good starting points for interpolation are the "sequential" and "diverging" ColorBrewer palettes in the RColorBrewer package

Examples

```
## Here space="rgb" gives palettes that vary only in saturation,
## as intended.
## With space="Lab" the steps are more uniform, but the hues
## are slightly purple.
filled.contour(volcano,
               color = colorRampPalette(c("red", "white", "blue")),
               asp = 1)
filled.contour(volcano,
               color = colorRampPalette(c("red", "white", "blue"),
                                        space = "Lab"),
               asp = 1)

## Interpolating a 'sequential' ColorBrewer palette
YlOrBr <- c("#FFFFD4", "#FED98E", "#FE9929", "#D95F0E", "#993404")
filled.contour(volcano,
               color = colorRampPalette(YlOrBr, space = "Lab"),
               asp = 1)
filled.contour(volcano,
               color = colorRampPalette(YlOrBr, space = "Lab",
                                        bias = 0.5),
               asp = 1)

## 'jet.colors' is "as in Matlab" (and hurting the eye by oversaturation)
jet.colors <-
  colorRampPalette(c("#00007F", "blue", "#007FFF", "cyan",
                     "#7FFF7F", "yellow", "#FF7F00", "red", "#7F0000"))
filled.contour(volcano, color = jet.colors, asp = 1)

## space="Lab" helps when colors don't form a natural sequence
m <- outer(1:20,1:20,function(x,y) sin(sqrt(x*y)/3))
rgb.palette <- colorRampPalette(c("red", "orange", "blue"),
                                space = "rgb")
Lab.palette <- colorRampPalette(c("red", "orange", "blue"),
                                space = "Lab")
filled.contour(m,col = rgb.palette(20))
filled.contour(m,col = Lab.palette(20))
```

---

colors *Color Names*

Description

Returns the built-in color names which R knows about.

Usage

```
colors()
colours()
```

Details

These color names can be used with a `col=` specification in graphics functions.

An even wider variety of colors can be created with primitives `rgb` and `hsv` or the derived `rainbow`, `heat.colors`, etc.

Value

A character vector containing all the built-in color names.

See Also

`palette` for setting the "palette" of colors for `par(col=<num>)`; `rgb`, `hsv`, `hcl`, `gray`; `rainbow` for a nice example; and `heat.colors`, `topo.colors` for images.

`col2rgb` for translating to RGB numbers and extended examples.

Examples

```
cl <- colors()
length(cl); cl[1:20]
```

---

contourLines *Calculate Contour Lines*

Description

Calculate contour lines for a given set of data.

Usage

```
contourLines(x = seq(0, 1, len = nrow(z)),
             y = seq(0, 1, len = ncol(z)),
             z, nlevels = 10,
             levels = pretty(range(z, na.rm=TRUE), nlevels))
```

Arguments

`x,y`	locations of grid lines at which the values in `z` are measured. These must be in ascending order. By default, equally spaced values from 0 to 1 are used. If `x` is a `list`, its components `x$x` and `x$y` are used for `x` and `y`, respectively. If the list has component `z` this is used for `z`.
`z`	a matrix containing the values to be plotted (`NA`s are allowed). Note that `x` can be used instead of `z` for convenience.
`nlevels`	number of contour levels desired **iff** `levels` is not supplied.
`levels`	numeric vector of levels at which to draw contour lines.

Details

`contourLines` draws nothing, but returns a set of contour lines.

There is currently no documentation about the algorithm. The source code is in '$R_HOME/src/main/plot3d.c'.

Value

A list of contours. Each contour is a list with elements:

`level`	The contour level.
`x`	The x-coordinates of the contour.
`y`	The y-coordinates of the contour.

See Also

options("max.countour.segments") for the maximal complexity of a single contour line.

contour.

Examples

```
x <- 10*1:nrow(volcano)
y <- 10*1:ncol(volcano)
contourLines(x, y, volcano)
```

---

convertColor *Convert between colour spaces*

Description

Convert colours between standard colour space representations. This function is experimental.

Usage

```
convertColor(color, from, to, from.ref.white, to.ref.white,
             scale.in=1, scale.out=1, clip=TRUE)
```

Arguments

color	A matrix whose rows specify colors
from,to	Input and output color spaces. See Details below
from.ref.white,to.ref.white	Reference whites or NULL if these are built in to the definition, as for RGB spaces. D65 is the default, see Details for others
scale.in, scale.out	Input is divided by scale.in, output is multiplied by scale.out. Use NULL to suppress scaling when input or output is not numeric.
clip	If TRUE, truncate RGB output to [0,1], FALSE return out-of-range RGB, NA set out of range colors to NaN.

Details

Color spaces are specified by objects of class colorConverter, created by colorConverter or make.rgb. Built-in color spaces may be referenced by strings: "XYZ", "sRGB", "Apple RGB", "CIE RGB", "Lab", "Luv". The converters for these colour spaces are in the object colorspaces.

The "sRGB" color space is that used by standard PC monitors. "Apple RGB" is used by Apple monitors. "Lab" and "Luv" are approximately perceptually uniform spaces standardized by the Commission Internationale d'Eclairage. XYZ is a 1931 CIE standard capable of representing all visible colors (and then some), but not in a perceptually uniform way.

The Lab and Luv spaces describe colors of objects, and so require the specification of a reference "white light" color. Illuminant D65 is a standard indirect daylight, Illuminant D50 is close to direct sunlight, and Illuminant A is the light from a standard incandescent bulb. Other standard CIE

illuminants supported are `B`, `C`, `E` and `D55`. RGB colour spaces are defined relative to a particular reference white, and can be only approximately translated to other reference whites. The Bradford chromatic adaptation algorithm is used for this.

The RGB color spaces are specific to a particular class of display. An RGB space cannot represent all colors, and the `clip` option controls what is done to out-of-range colors.

Value

A 3-row matrix whose columns specify the colors.

References

For all the conversion equations http://www.brucelindbloom.com/

For the white points http://www.efg2.com/Lab/Graphics/Colors/Chromaticity.htm

See Also

`col2rgb` and `colors` for ways to specify colors in graphics.

`make.rgb` for specifying other colour spaces.

Examples

```
par(mfrow=c(2,2))
## The displayable colors from four planes of Lab space
ab <- expand.grid(a=(-10:15)*10,b=(-15:10)*10)

Lab <- cbind(L=20,ab)
srgb <- convertColor(Lab,from="Lab",to="sRGB",clip=NA)
clipped <- attr(na.omit(srgb),"na.action")
srgb[clipped,] <- 0
cols <- rgb(srgb[,1],srgb[,2],srgb[,3])
image((-10:15)*10,(-15:10)*10,matrix(1:(26*26),ncol=26),col=cols,
  xlab="a",ylab="b",main="Lab: L=20")

Lab <- cbind(L=40,ab)
srgb <- convertColor(Lab,from="Lab",to="sRGB",clip=NA)
clipped <- attr(na.omit(srgb),"na.action")
srgb[clipped,] <- 0
cols <- rgb(srgb[,1],srgb[,2],srgb[,3])
image((-10:15)*10,(-15:10)*10,matrix(1:(26*26),ncol=26),col=cols,
  xlab="a",ylab="b",main="Lab: L=40")

Lab <- cbind(L=60,ab)
srgb <- convertColor(Lab,from="Lab",to="sRGB",clip=NA)
clipped <- attr(na.omit(srgb),"na.action")
srgb[clipped,] <- 0
cols <- rgb(srgb[,1],srgb[,2],srgb[,3])
image((-10:15)*10,(-15:10)*10,matrix(1:(26*26),ncol=26),col=cols,
  xlab="a",ylab="b",main="Lab: L=60")

Lab <- cbind(L=80,ab)
srgb <- convertColor(Lab,from="Lab",to="sRGB",clip=NA)
clipped <- attr(na.omit(srgb),"na.action")
srgb[clipped,] <- 0
```

```
cols <- rgb(srgb[,1],srgb[,2],srgb[,3])
image((-10:15)*10,(-15:10)*10,matrix(1:(26*26),ncol=26),col=cols,
  xlab="a",ylab="b",main="Lab: L=80")

(cols <- t(col2rgb(palette())))
(lab <- convertColor(cols,from="sRGB",to="Lab",scale.in=255))
round(convertColor(lab,from="Lab",to="sRGB",scale.out=255))
```

`dev.interactive` *Is the Current Graphics Device Interactive ?*

Description

Test if the current graphics device is interactive.

Usage

```
dev.interactive(orNone = FALSE)
```

Arguments

`orNone`	logical; if `TRUE`, the function also returns `TRUE` when `.Device == "null device"` and `getOption("device")` is among the interactive devices.

Details

The `X11` (Unix), `windows` (Windows) and `quartz` (MacOS X) are regarded as interactive, together with `GTK` and `gnome` (used with the GNOME GUI: `GTK` is available in package **gtkDevice** and `gnome` is expected to be in a package **gnomeDevice**) and `JavaGD` (from the JGR project, see `http://stats.math.uni-augsburg.de/JGR/`).

Value

`dev.interactive()` returns a logical, `TRUE` iff an interactive (screen) device is in use.

See Also

`Devices` for the available devices on your platform.

Examples

```
dev.interactive # definition -> see list of "interactive" devices explicitly
```

`dev.xxx` *Control Multiple Devices*

Description

These functions provide control over multiple graphics devices.

Only one device is the *active* device. This is the device in which all graphics operations occur.

Devices are associated with a name (e.g., `"X11"` or `"postscript"`) and a number; the `"null device"` is always device 1.

`dev.off` shuts down the specified (by default the current) device. `graphics.off()` shuts down all open graphics devices.

`dev.set` makes the specified device the active device.

Usage

```
dev.cur()
dev.list()
dev.next(which = dev.cur())
dev.prev(which = dev.cur())
dev.off(which = dev.cur())
dev.set(which = dev.next())
graphics.off()
```

Arguments

`which`	An integer specifying a device number

Value

`dev.cur` returns the number and name of the active device, or 1, the null device, if none is active.

`dev.list` returns the numbers of all open devices, except device 1, the null device. This is a numeric vector with a names attribute giving the names, or `NULL` is there is no open device.

`dev.next` and `dev.prev` return the number and name of the next / previous device in the list of devices. The list is regarded as a circular list, and `"null device"` will be included only if there are no open devices.

`dev.off` returns the name and number of the new active device (after the specified device has been shut down).

`dev.set` returns the name and number of the new active device.

See Also

`Devices`, such as `postscript`, etc.

`layout` and its links for setting up plotting regions on the current device.

Examples

```
## Not run:
## Unix-specific example
x11()
plot(1:10)
x11()
plot(rnorm(10))
dev.set(dev.prev())
abline(0,1)# through the 1:10 points
dev.set(dev.next())
abline(h=0, col="gray")# for the residual plot
dev.set(dev.prev())
dev.off(); dev.off()#- close the two X devices
## End(Not run)
```

---

dev2 *Copy Graphics Between Multiple Devices*

Description

`dev.copy` copies the graphics contents of the current device to the device specified by `which` or to a new device which has been created by the function specified by `device` (it is an error to specify both `which` and `device`). (If recording is off on the current device, there are no contents to copy: this will result in no plot or an empty plot.) The device copied to becomes the current device.

`dev.print` copies the graphics contents of the current device to a new device which has been created by the function specified by `device` and then shuts the new device.

`dev.copy2eps` is similar to `dev.print` but produces an EPSF output file, in portrait orientation (`horizontal = FALSE`)

`dev.control` allows the user to control the recording of graphics operations in a device. If `displaylist` is `"inhibit"` (`"enable"`) then recording is turned off (on). It is only safe to change this at the beginning of a plot (just before or just after a new page). Initially recording is on for screen devices, and off for print devices.

Usage

```
dev.copy(device, ..., which = dev.next())
dev.print(device = postscript, ...)
dev.copy2eps(...)
dev.control(displaylist = c("inhibit", "enable"))
```

Arguments

`device`	A device function (e.g., `x11`, `postscript`, ...)
`...`	Arguments to the `device` function above. For `dev.print`, this includes `which` and by default any `postscript` arguments.
`which`	A device number specifying the device to copy to
`displaylist`	A character string: the only valid values are `"inhibit"` and `"enable"`.

Details

For dev.copy2eps, width and height are taken from the current device unless otherwise specified. If just one of width and height is specified, the other is adjusted to preserve the aspect ratio of the device being copied. The default file name is Rplot.eps.

The default for dev.print is to produce and print a postscript copy, if options("printcmd") is set suitably.

dev.print is most useful for producing a postscript print (its default) when the following applies. Unless file is specified, the plot will be printed. Unless width, height and pointsize are specified the plot dimensions will be taken from the current device, shrunk if necessary to fit on the paper. (pointsize is rescaled if the plot is shrunk.) If horizontal is not specified and the plot can be printed at full size by switching its value this is done instead of shrinking the plot region.

If dev.print is used with a specified device (even postscript) it sets the width and height in the same way as dev.copy2eps. This will not be appropriate unless the device specifies dimensions in inches, in particular not for png, jpeg and (Windows only) bmp.

Value

dev.copy returns the name and number of the device which has been copied to.

dev.print and dev.copy2eps return the name and number of the device which has been copied from.

Note

Most devices (including all screen devices) have a display list which records all of the graphics operations that occur in the device. dev.copy copies graphics contents by copying the display list from one device to another device. Also, automatic redrawing of graphics contents following the resizing of a device depends on the contents of the display list.

After the command dev.control("inhibit"), graphics operations are not recorded in the display list so that dev.copy and dev.print will not copy anything and the contents of a device will not be redrawn automatically if the device is resized.

The recording of graphics operations is relatively expensive in terms of memory so the command dev.control("inhibit") can be useful if memory usage is an issue.

See Also

dev.cur and other dev.xxx functions

Examples

```
## Not run:
x11()
plot(rnorm(10), main="Plot 1")
dev.copy(device=x11)
mtext("Copy 1", 3)
dev.print(width=6, height=6, horizontal=FALSE) # prints it
dev.off(dev.prev())
dev.off()
## End(Not run)
```

dev2bitmap *Graphics Device for Bitmap Files via GhostScript*

Description

bitmap generates a graphics file. dev2bitmap copies the current graphics device to a file in a graphics format.

Usage

```
bitmap(file, type = "png256", height = 6, width = 6, res = 72,
       pointsize, ...)

dev2bitmap(file, type = "png256", height = 6, width = 6, res = 72,
           pointsize, ...)
```

Arguments

file	The output file name, with an appropriate extension.
type	The type of bitmap. the default is "png256".
height	The plot height, in inches.
width	The plot width, in inches.
res	Resolution, in dots per inch.
pointsize	The pointsize to be used for text: defaults to something reasonable given the width and height
...	Other parameters passed to postscript.

Details

dev2bitmap works by copying the current device to a postscript device, and post-processing the output file using ghostscript. bitmap works in the same way using a postscript device and postprocessing the output as "printing".

You will need a version of ghostscript (5.10 and later have been tested): the full path to the executable can be set by the environment variable R_GSCMD. (If this is unset the command "gs" is used.)

The types available will depend on the version of ghostscript, but are likely to include "pcxmono", "pcxgray", "pcx16", "pcx256", "pcx24b", "pcxcmyk", "pbm", "pbmraw", "pgm", "pgmraw", "pgnm", "pgnmraw", "pnm", "pnmraw", "ppm", "ppmraw", "pkm", "pkmraw", "tiffcrle", "tiffg3", "tiffg32d", "tiffg4", "tifflzw", "tiffpack", "tiff12nc", "tiff24nc", "psmono", "psgray", "psrgb", "bit", "bitrgb", "bitcmyk", "pngmono", "pnggray", "png16", "png256", "png16m", "jpeg", "jpeggray", "pdfwrite".

Note: despite the name of the functions they can produce PDF *via* type = "pdfwrite", and the PDF produced is not bitmapped.

For formats which contain a single image, a file specification like Rplots%03d.png can be used: this is interpreted by GhostScript.

For dev2bitmap if just one of width and height is specified, the other is chosen to preserve aspect ratio of the device being copied.

Value

None.

See Also

`postscript`, `png` and `jpeg` and on Windows `bmp`.

`pdf` generate PDF directly.

To display an array of data, see `image`.

`Devices` *List of Graphical Devices*

Description

The following graphics devices are currently available:

- `postscript` Writes PostScript graphics commands to a file
- `pdf` Write PDF graphics commands to a file
- `pictex` Writes LaTeX/PicTeX graphics commands to a file
- `xfig` Device for XFIG graphics file format
- `bitmap` bitmap pseudo-device via `GhostScript` (if available).

The following devices will be available if R was compiled to use them and started with the appropriate '`--gui`' argument:

- `X11` The graphics driver for the X11 Window system
- `png` PNG bitmap device
- `jpeg` JPEG bitmap device

None of these are available under `R CMD BATCH`.

Details

If no device is open, using a high-level graphics function will cause a device to be opened. Which device is given by `options("device")` which is initially set as the most appropriate for each platform: a screen device for most interactive use and `postscript` otherwise. The exception is interactive use under Unix if no screen device is known to be available, when postscript() is used for most systems; pdf() for Mac OS X.

See Also

The individual help files for further information on any of the devices listed here;

`dev.interactive`, `dev.cur`, `dev.print`, `graphics.off`, `image`, `dev2bitmap`.

`capabilities` to see if `X11`, `jpeg` and `png` are available.

Examples

```
## Not run:
## open the default screen device on this platform if no device is
## open
if(dev.cur() == 1) get(getOption("device"))()
## End(Not run)
```

`embedFonts` *Embed Fonts in PostScript and PDF*

Description

Runs ghostscript to process a PDF or PostScript file and embed all fonts in the file.

Usage

```
embedFonts(file, format, outfile = file, fontpaths = "", options = "")
```

Arguments

`file`	a character string giving the name of the original file.
`format`	either `"pswrite"` or `"pdfwrite"`. If not specified, it is guessed from the suffix of `file`.
`outfile`	the name of the new file (with fonts embedded).
`fontpaths`	a character vector giving directories that ghostscript will search for fonts.
`options`	a character string containing further options to ghostscript.

Details

This function is not necessary if you just use the standard default fonts for PostScript and PDF output.

If you use a special font, this function is useful for embedding that font in your PostScript or PDF document so that it can be shared with others without them having to install your special font.

If the special font is not installed for ghostscript, you will need to tell ghostscript where the font is, using something like `options="-sFONTPATH=path/to/font"`.

This function relies on a suitable ghostscript executable being in your path, or the environment variable `R_GSCMD` (the same as `bitmap`) being set as the full path to the ghostscript executable.

Value

The shell command used to invoke ghostscript is returned invisibly. This may be useful for debugging purposes as you can run the command by hand in a shell to look for problems.

See Also

`postscriptFonts`, `Devices`.

extendrange *Extend a Numerical Range by a Small Percentage*

Description

Extends a numerical range by a small percentage, i.e., fraction, *on both sides*.

Usage

```
extendrange(x, r = range(x, na.rm = TRUE), f = 0.05)
```

Arguments

x	numeric vector; not used if r is specified.
r	numeric vector of length 2; defaults to the range of x.
f	number specifying the fraction by which the range should be extended.

Value

numeric vector of length 2, "r extended", namely simply `r + c(-f,f) * diff(r)`.

See Also

range; pretty which can be considered a sophisticated extension of extendrange.

Examples

```
x <- 1:5
(r <- range(x))          # 1    5
extendrange(x)           # 0.8  5.2
extendrange(x, f= 0.01) # 0.96 5.04
## Use 'r' if you have it already:
stopifnot(identical(extendrange(r=r),
                    extendrange(x)))
```

getGraphicsEvent *Wait for a mouse or keyboard event from a graphics window*

Description

This function waits for input from a graphics window in the form of a mouse or keyboard event.

Usage

```
getGraphicsEvent(prompt = "Waiting for input",
                 onMouseDown = NULL, onMouseMove = NULL, onMouseUp = NULL,
                 onKeybd = NULL)
```

Arguments

prompt	prompt to be displayed to the user
onMouseDown	a function to respond to mouse clicks
onMouseMove	a function to respond to mouse movement
onMouseUp	a function to respond to mouse button releases
onKeybd	a function to respond to key presses

Details

This function allows user input from some graphics devices (currently only the Windows screen display). When called, event handlers may be installed to respond to events involving the mouse or keyboard.

The mouse event handlers should be functions with header `function(buttons, x, y)`. The coordinates `x` and `y` will be passed to mouse event handlers in device independent coordinates (i.e. the lower left corner of the window is `(0,0)`, the upper right is `(1,1)`). The `buttons` argument will be a vector listing the buttons that are pressed at the time of the event, with 0 for left, 1 for middle, and 2 for right.

The keyboard event handler should be a function with header `function(key)`. A single element character vector will be passed to this handler, corresponding to the key press. Shift and other modifier keys will have been processed, so `shift-a` will be passed as `"A"`. The following special keys may also be passed to the handler:

- Control keys, passed as `"Ctrl-A"`, etc.
- Navigation keys, passed as one of `"Left", "Up", "Right", "Down", "PgUp", "PgDn", "End", "Home"`
- Edit keys, passed as one of `"Ins", "Del"`
- Function keys, passed as one of `"F1", "F2", ...`

The event handlers are standard R functions, and will be executed in an environment as though they had been called directly from `getGraphicsEvent`.

Events will be processed until

- one of the event handlers returns a non-`NULL` value which will be returned as the value of `getGraphicsEvent`, or
- the user interrupts the function from the console.

Value

A non-`NULL` value returned from one of the event handlers.

Author(s)

Duncan Murdoch

Examples

```
## Not run:
    mousedown <- function(buttons, x, y) {
        cat("Buttons ", paste(buttons, collapse=" "), " at ", x, y, "\n")
        points(x, y)
        if (x > 0.85 && y > 0.85) "Done"
```

```
        else NULL
    }

    mousemove <- function(buttons, x, y) {
        points(x, y)
        NULL
    }

    keybd <- function(key) {
        cat("Key <", key, ">\n", sep = "")
    }

    plot(0:1, 0:1, type='n')
    getGraphicsEvent("Click on upper right to quit",
                     onMouseDown = mousedown,
                     onMouseMove = mousemove,
                     onKeybd = keybd)
## End(Not run)
```

`gray` *Gray Level Specification*

Description

Create a vector of colors from a vector of gray levels.

Usage

```
gray(level)
grey(level)
```

Arguments

`level`	a vector of desired gray levels between `0` and `1`; zero indicates `"black"` and one indicates `"white"`.

Details

The values returned by `gray` can be used with a `col=` specification in graphics functions or in `par`.

`grey` is an alias for `gray`.

Value

A vector of "colors" of the same length as `level`.

See Also

`rainbow`, `hsv`, `hcl`, `rgb`.

Examples

```
gray(0:8 / 8)
```

`gray.colors` *Gray Color Palette*

Description

Create a vector of `n` gamma-corrected gray colors.

Usage

```
gray.colors(n, start = 0.3, end = 0.9, gamma = 2.2)
grey.colors(n, start = 0.3, end = 0.9, gamma = 2.2)
```

Arguments

`n`	the number of gray colors (≥ 1) to be in the palette.
`start`	starting gray level in the palette (should be between `0` and `1` where zero indicates `"black"` and one indicates `"white"`).
`end`	ending gray level in the palette.
`gamma`	the gamma correction.

Details

The function `gray.colors` chooses a series of `n` gamma-corrected gray levels between `start` and `end`: $(start^\gamma, \ldots, end^\gamma)^{(1/\gamma)}$. The returned palette contains the corresponding gray colors. This palette is used in `barplot.default`.

`grey.colors` is an alias for `gray.colors`.

Value

A vector of `n` gray colors.

See Also

`gray`, `rainbow`, `palette`.

Examples

```
pie(rep(1,12), col = gray.colors(12))
barplot(1:12, col = gray.colors(12))
```

`hcl` *HCL Color Specification*

Description

Create a vector of colors from vectors specifying hue, chroma and luminance.

Usage

```
hcl(h = 0, c = 35, l = 85, alpha, fixup = TRUE)
```

Arguments

`h`	The hue of the color specified as an angle in the range [0,360]. 0 yields red, 120 yields green 240 yields blue, etc.
`c`	The chroma of the color. The upper bound for chroma depends on hue and luminance.
`l`	A value in the range [0,100] giving the luminance of the colour. For a given combination of hue and chroma, only a subset of this range is possible.
`alpha`	numeric value in the range `[0,1]` for alpha transparency channel (0 means transparent and 1 means opaque).
`fixup`	a logical value which indicates whether the resulting RGB values should be corrected to ensure that a real color results. if `fixup` is `FALSE` RGB components lying outside the range [0,1] will result in an `NA` value.

Details

This function corresponds to polar coordinates in the CIE-LUV color space. Steps of equal size in this space correspond to approximately equal peceptual changes in color. Thus, `hcl` can be thought of as a perceptually based version of `hsv`.

The function is primarily intended as a way of computing colors for filling areas in plots where area corresponds to a numerical value (pie charts, bar charts, mosaic plots, histograms, etc). Choosing colors which have equal chroma and luminance provides a way of minimising the irradiation illusion which would otherwise produce a misleading impression of how large the areas are.

The default values of chroma and luminance make it possible to generate a full range of hues and have a relatively pleasant pastel appearance.

The RGB values produced by this function correspond to the sRGB color space used on most PC computer displays. There are other packages which provide more general color space facilities.

Value

A vector of character strings which can be used as color specifications by R graphics functions.

Note

At present there is no guarantee that the colours rendered by R graphics devices will correspond to their sRGB description. It is planned to adopt sRGB as the standard R color description in future.

Author(s)

Ross Ihaka

References

Ihaka, R. (2003). Colour for Presentation Graphics, Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003), March 20-22, 2003, Technische Universität Wien, Vienna, Austria. http://www.ci.tuwien.ac.at/Conferences/DSC-2003.

See Also

hsv, rgb.

Examples

```
# The Foley and Van Dam PhD Data.
csd <- matrix(c( 4,2,4,6, 4,3,1,4, 4,7,7,1,
                 0,7,3,2, 4,5,3,2, 5,4,2,2,
                 3,1,3,0, 4,4,6,7, 1,10,8,7,
                 1,5,3,2, 1,5,2,1, 4,1,4,3,
                 0,3,0,6, 2,1,5,5), nr=4)

csphd =
function(colors)
barplot(csd, col = colors, ylim = c(0,30),
        names = 72:85, xlab = "Year", ylab = "Students",
        legend = c("Winter", "Spring", "Summer", "Fall"),
        main = "Computer Science PhD Graduates", las = 1)

# The Original (Metaphorical) Colors (Ouch!)
csphd(c("blue", "green", "yellow", "orange"))

# A Color Tetrad (Maximal Color Differences)
csphd(hcl(h = c(30, 120, 210, 300)))

# Same, but lighter and less colorful
# Turn of automatic correction to make sure
# that we have defined real colors.
csphd(hcl(h = c(30, 120, 210, 300),
          c = 20, l = 90, fixup = FALSE))

# Analogous Colors
# Good for those with red/green color confusion
csphd(hcl(h = seq(60, 240, by = 60)))

# Metaphorical Colors
csphd(hcl(h = seq(210, 60, length = 4)))

# Cool Colors
csphd(hcl(h = seq(120, 0, length = 4) + 150))

# Warm Colors
csphd(hcl(h = seq(120, 0, length = 4) - 30))

# Single Color
hist(rnorm(1000), col = hcl(240))
```

`Hershey` *Hershey Vector Fonts in R*

Description

If the `family` graphical parameter (see `par`) has been set to one of the Hershey fonts (see below) Hershey vector fonts are used to render text.

These fonts have two advantages:

1. vector fonts describe each character in terms of a set of points; R renders the character by joining up the points with straight lines. This intimate knowledge of the outline of each character means that R can arbitrarily transform the characters, which can mean that the vector fonts look better for rotated and 3d text. This works for Latin-1 and Cyrillic characters, only.
2. this implementation was adapted from the GNU libplot library which provides support for non-ASCII and non-English fonts. This means that it is possible, for example, to produce weird plotting symbols and Japanese characters.

For historical reasons, when using the `text` function, Hershey fonts may also be selected via the `vfont` argument, which is a character vector of length 2 (see below for valid values).

Drawback:
You cannot use mathematical expressions (`plotmath`) with Hershey fonts.

Usage

```
Hershey
```

Details

The Hershey characters are organised into a set of fonts. A particular font is selected by specifying one of the following font families via `par(family)` and specifying the desired font face (plain, bold, italic, bold-italic) via `par(font)`.

family	faces available
"HersheySerif"	plain, bold, italic, bold-italic
"HersheySans"	plain, bold, italic, bold-italic
"HersheyScript"	plain, bold
"HersheyGothicEnglish"	plain
"HersheyGothicGerman"	plain
"HersheyGothicItalian"	plain
"HersheySymbol"	plain, bold, italic, bold-italic
"HersheySansSymbol"	plain, italic

In the old, `vfont` specification for the `text` function, the Hershey font is specified by a typeface (e.g., `serif` or `sans serif`) and a fontindex or "style" (e.g., `plain` or `italic`). The first element of `vfont` specifies the typeface and the second element specifies the fontindex. The first table produced by `demo(Hershey)` shows the character `a` produced by each of the different fonts.

The available `typeface` and `fontindex` values are available as list components of the variable `Hershey`. The allowed pairs for `(typeface, fontindex)` are:

serif	plain

serif	italic
serif	bold
serif	bold italic
serif	cyrillic
serif	oblique cyrillic
serif	EUC
sans serif	plain
sans serif	italic
sans serif	bold
sans serif	bold italic
script	plain
script	italic
script	bold
gothic english	plain
gothic german	plain
gothic italian	plain
serif symbol	plain
serif symbol	italic
serif symbol	bold
serif symbol	bold italic
sans serif symbol	plain
sans serif symbol	italic

and the indices of these are available as `Hershey$allowed`.

Escape sequences: The string to be drawn can include escape sequences, which all begin with a `\`. When R encounters a `\`, rather than drawing the `\`, it treats the subsequent character(s) as a coded description of what to draw.

One useful escape sequence (in the current context) is of the form: `\123`. The three digits following the `\` specify an octal code for a character. For example, the octal code for `p` is 160 so the strings `"p"` and `"\160"` are equivalent. This is useful for producing characters when there is not an appropriate key on your keyboard.

The other useful escape sequences all begin with `\\`. These are described below. Remember that backslashes have to be doubled in R character strings, so they need to be entered with *four* backslashes.

Symbols: an entire string of Greek symbols can be produced by selecting the Serif Symbol or Sans Serif Symbol typeface. To allow Greek symbols to be embedded in a string which uses a non-symbol typeface, there are a set of symbol escape sequences of the form `\\ab`. For example, the escape sequence `\\*a` produces a Greek alpha. The second table in `demo(Hershey)` shows all of the symbol escape sequences and the symbols that they produce.

ISO Latin-1: further escape sequences of the form `\\ab` are provided for producing ISO Latin-1 characters. Another option is to use the appropriate octal code. The (non-ASCII) ISO Latin-1 characters are in the range 241...377. For example, `\366` produces the character `o` with an umlaut. The third table in `demo(Hershey)` shows all of the ISO Latin-1 escape sequences.

These characters can be used directly in a Latin-1 or UTF-8 locale. (In the latter, non-Latin-1 characters are replaced by a dot.)

Special Characters: a set of characters are provided which do not fall into any standard font. These can only be accessed by escape sequence. For example, `\\LI` produces the zodiac sign for Libra, and `\\JU` produces the astronomical sign for Jupiter. The fourth table in `demo(Hershey)` shows all of the special character escape sequences.

Cyrillic Characters: cyrillic characters are implemented according to the K018-R encoding, and can be used directly in such a locale using the Serif typeface and Cyrillic (or Oblique Cyrillic) fontindex. Alternatively they can be specified via an octal code in the range 300 to 337 for lower case characters or 340 to 377 for upper case characters. The fifth table in `demo(Hershey)` shows the octal codes for the available cyrillic characters.

Cyrillic has to be selected via a `("serif", fontindex)` pair rather than via a font family.

Japanese Characters: 83 Hiragana, 86 Katakana, and 603 Kanji characters are implemented according to the EUC-JP (Extended Unix Code) encoding. Each character is identified by a unique hexadecimal code. The Hiragana characters are in the range 0x2421 to 0x2473, Katakana are in the range 0x2521 to 0x2576, and Kanji are (scattered about) in the range 0x3021 to 0x6d55.

When using the Serif typeface and EUC fontindex, these characters can be produced by a *pair* of octal codes. Given the hexadecimal code (e.g., 0x2421), take the first two digits and add 0x80 and do the same to the second two digits (e.g., 0x21 and 0x24 become 0xa4 and 0xa1), then convert both to octal (e.g., 0xa4 and 0xa1 become 244 and 241). For example, the first Hiragana character is produced by `\244\241`.

It is also possible to use the hexadecimal code directly. This works for all non-EUC fonts by specifying an escape sequence of the form `\\#J1234`. For example, the first Hiragana character is produced by `\\#J2421`.

The Kanji characters may be specified in a third way, using the so-called "Nelson Index", by specifying an escape sequence of the form `\\#N1234`. For example, the Kanji for “one” is produced by `\\#N0001`.

`demo(Japanese)` shows the available Japanese characters.

Raw Hershey Glyphs: all of the characters in the Hershey fonts are stored in a large array. Some characters are not accessible in any of the Hershey fonts. These characters can only be accessed via an escape sequence of the form `\\#H1234`. For example, the fleur-de-lys is produced by `\\#H0746`. The sixth and seventh tables of `demo(Hershey)` shows all of the available raw glyphs.

References

http://www.gnu.org/software/plotutils/plotutils.html

See Also

`demo(Hershey)`, `par`, `text`, `contour`.

`Japanese` for the Japanese characters in the Hershey fonts.

Examples

```
Hershey

## for tables of examples, see demo(Hershey)
```

`hsv` *HSV Color Specification*

Description

Create a vector of colors from vectors specifying hue, saturation and value.

Usage

```
hsv(h = 1, s = 1, v = 1, gamma = 1, alpha)
```

Arguments

`h,s,v`	numeric vectors of values in the range `[0,1]` for "hue", "saturation" and "value" to be combined to form a vector of colors. Values in shorter arguments are recycled.
`gamma`	a "gamma correction" exponent, γ
`alpha`	numeric value in the range `[0,1]` for alpha transparency channel (0 means transparent and 1 means opaque).

Value

This function creates a vector of "colors" corresponding to the given values in HSV space. The values returned by `hsv` can be used with a `col=` specification in graphics functions or in `par`.

Gamma correction

For each color, (r, g, b) in RGB space (with all values in $[0, 1]$), the final color corresponds to $(r^\gamma, g^\gamma, b^\gamma)$.

See Also

`hcl` for a perceptually based version of `hsv()`, `rgb` and `rgb2hsv` for RGB to HSV conversion; `rainbow`, `gray`.

Examples

```
hsv(.5,.5,.5)

## Look at gamma effect:
n <- 20;  y <- -sin(3*pi*((1:n)-1/2)/n)
op <- par(mfrow=c(3,2),mar=rep(1.5,4))
for(gamma in c(.4, .6, .8, 1, 1.2, 1.5))
  plot(y, axes = FALSE, frame.plot = TRUE,
       xlab = "", ylab = "", pch = 21, cex = 30,
       bg = rainbow(n, start=.85, end=.1, gamma = gamma),
       main = paste("Red tones;  gamma=",format(gamma)))
par(op)
```

Japanese *Japanese characters in R*

Description

The implementation of Hershey vector fonts provides a large number of Japanese characters (Hiragana, Katakana, and Kanji).

Details

Without keyboard support for typing Japanese characters, the only way to produce these characters is to use special escape sequences: see Hershey.

For example, the Hiragana character for the sound "ka" is produced by `\\#J242b` and the Katakana character for this sound is produced by `\\#J252b`. The Kanji ideograph for "one" is produced by `\\#J306c` or `\\#N0001`.

The output from `demo(Japanese)` shows tables of the escape sequences for the available Japanese characters.

References

http://www.gnu.org/software/plotutils/plotutils.html

See Also

`demo(Japanese)`, Hershey, text, contour

Examples

```
plot(1:9, type="n", axes=FALSE, frame=TRUE, ylab="",
     main= "example(Japanese)", xlab= "using Hershey fonts")
par(cex=3)
Vf <- c("serif", "plain")
text(4, 2, "\\#J2438\\#J2421\\#J2451\\#J2473", vfont = Vf)
text(4, 4, "\\#J2538\\#J2521\\#J2551\\#J2573", vfont = Vf)
text(4, 6, "\\#J467c\\#J4b5c", vfont = Vf)
text(4, 8, "Japan", vfont = Vf)
par(cex=1)
text(8, 2, "Hiragana")
text(8, 4, "Katakana")
text(8, 6, "Kanji")
text(8, 8, "English")
```

make.rgb *Create colour spaces*

Description

These functions specify colour spaces for use in convertColor.

Usage

```
make.rgb(red, green, blue, name = NULL, white = "D65", gamma = 2.2)
colorConverter(toXYZ, fromXYZ, name, white=NULL)
```

Arguments

red,green,blue	Chromaticity (xy or xyY) of RGB primaries
name	Name for the colour space
white	Character string specifying the reference white (see Details)
gamma	Display gamma (nonlinearity). A positive number or the string "sRGB"
fromXYZ	Function to convert from XYZ tristimulus coordinates to this space
toXYZ	Function to convert from this space to XYZ tristimulus coordinates.

Details

An RGB colour space is defined by the chromaticities of the red, green and blue primaries. These are given as vectors of length 2 or 3 in xyY coordinates (the Y component is not used and may be omitted). The chromaticities are defined relative to a reference white, which must be one of the CIE standard illuminants: "A", "B", "C", "D50", "D55", "D60", "E" (usually "D65").

The display gamma is most commonly 2.2, though 1.8 is used for Apple RGB. The sRGB standard specifies a more complicated function that is close to a gamma of 2.2; gamma="sRGB" uses this function.

Colour spaces other than RGB can be specified directly by giving conversions to and from XYZ tristimulus coordinates. The functions should take two arguments. The first is a vector giving the coordinates for one colour. The second argument is the reference white. If a specific reference white is included in the definition of the colour space (as for the RGB spaces) this second argument should be ignored and may be ....

Value

An object of class colorConverter

References

Conversion algorithms from http://www.brucelindbloom.com

See Also

convertColor

Examples

```
(pal <- make.rgb(red=  c(0.6400,0.3300),
                 green=c(0.2900,0.6000),
                 blue= c(0.1500,0.0600),
                 name = "PAL/SECAM RGB"))

## converter for sRGB in #rrggbb format
hexcolor <- colorConverter(toXYZ = function(hex,...) {
                           rgb <- t(col2rgb(hex))/255
                           colorspaces$sRGB$toXYZ(rgb,...) },
```

```
                                fromXYZ = function(xyz,...) {
                                   rgb <- colorspaces$sRGB$fromXYZ(xyz,..)
                                   rgb <- round(rgb,5)
                                   if (min(rgb) < 0 || max(rgb) > 1)
                                        as.character(NA)
                                   else
                                        rgb(rgb[1],rgb[2],rgb[3])},
                                white = "D65", name = "#rrggbb")

(cols <- t(col2rgb(palette())))
(luv <- convertColor(cols,from="sRGB", to="Luv", scale.in=255))
(hex <- convertColor(luv, from="Luv",  to=hexcolor, scale.out=NULL))

## must make hex a matrix before using it
(cc <- round(convertColor(as.matrix(hex), from= hexcolor, to= "sRGB",
                          scale.in=NULL, scale.out=255)))
stopifnot(cc == cols)
```

`n2mfrow` *Compute Default mfrow From Number of Plots*

Description

Easy setup for plotting multiple figures (in a rectangular layout) on one page. This computes a sensible default for `par(mfrow)`.

Usage

```
n2mfrow(nr.plots)
```

Arguments

`nr.plots`	integer; the number of plot figures you'll want to draw.

Value

A length two integer vector `nr, nc` giving the number of rows and columns, fulfilling `nr >= nc >= 1` and `nr * nc >= nr.plots`.

Author(s)

Martin Maechler

See Also

`par`, `layout`.

Examples

```
n2mfrow(8) # 3 x 3

n <- 5 ; x <- seq(-2,2, len=51)
## suppose now that 'n' is not known {inside function}
op <- par(mfrow = n2mfrow(n))
for (j in 1:n)
   plot(x, x^j, main = substitute(x^ exp, list(exp = j)), type = "l",
   col = "blue")

sapply(1:10, n2mfrow)
```

---

nclass *Compute the Number of Classes for a Histogram*

Description

Compute the number of classes for a histogram.

Usage

```
nclass.Sturges(x)
nclass.scott(x)
nclass.FD(x)
```

Arguments

x	A data vector.

Details

`nclass.Sturges` uses Sturges' formula, implicitly basing bin sizes on the range of the data.

`nclass.scott` uses Scott's choice for a normal distribution based on the estimate of the standard error.

`nclass.FD` uses the Freedman-Diaconis choice based on the inter-quartile range.

Value

The suggested number of classes.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S-PLUS.* Springer, page 112.

Freedman, D. and Diaconis, P. (1981) On the histogram as a density estimator: L_2 theory. *Zeitschrift für Wahrscheinlichkeitstheorie und verwandte Gebiete* **57**, 453–476.

Scott, D. W. (1979) On optimal and data-based histograms. *Biometrika* **66**, 605–610.

Scott, D. W. (1992) *Multivariate Density Estimation. Theory, Practice, and Visualization*. Wiley.

See Also

`hist`

palette *Set or View the Graphics Palette*

Description

View or manipulate the color palette which is used when a col= has a numeric index.

Usage

```
palette(value)
```

Arguments

value	an optional character vector.

Details

If value has length 1, it is taken to be the name of a built in color palette. If value has length greater than 1 it is assumed to contain a description of the colors which are to make up the new palette (either by name or by RGB levels).

If value is omitted or has length 0, no change is made the current palette.

Currently, the only built-in palette is "default".

Value

The palette which *was* in effect. This is invisible unless the argument is omitted.

See Also

colors for the vector of built-in "named" colors; hsv, gray, rainbow, terrain.colors,... to construct colors.

colorRamp to interpolate colors, making custom palettes; col2rgb for translating colors to RGB 3-vectors.

Examples

```
palette()               # obtain the current palette
palette(rainbow(6))     # six color rainbow

(palette(gray(seq(0,.9,len=25)))) # gray scales; print old palette
matplot(outer(1:100,1:30), type='l', lty=1,lwd=2, col=1:30,
        main = "Gray Scales Palette",
        sub = "palette(gray(seq(0,.9,len=25)))")
palette("default")      # reset back to the default
```

`Palettes` *Color Palettes*

Description

Create a vector of `n` "contiguous" colors.

Usage

```
rainbow(n, s = 1, v = 1, start = 0, end = max(1,n - 1)/n,
        gamma = 1, alpha = 1)
heat.colors(n, alpha = 1)
terrain.colors(n, alpha = 1)
topo.colors(n, alpha = 1)
cm.colors(n, alpha = 1)
```

Arguments

`n`	the number of colors (≥ 1) to be in the palette.
`s,v`	the "saturation" and "value" to be used to complete the HSV color descriptions.
`start`	the (corrected) hue in [0,1] at which the rainbow begins.
`end`	the (corrected) hue in [0,1] at which the rainbow ends.
`gamma`	the gamma correction, see argument `gamma` in `hsv`.
`alpha`	the alpha transparency, a number in [0,1], see argument `alpha` in `hsv`.

Details

Conceptually, all of these functions actually use (parts of) a line cut out of the 3-dimensional color space, parametrized by `hsv(h,s,v, gamma)`, where `gamma`$= 1$ for the *foo*`.colors` function, and hence, equispaced hues in RGB space tend to cluster at the red, green and blue primaries.

Some applications such as contouring require a palette of colors which do not "wrap around" to give a final color close to the starting one.

With `rainbow`, the parameters `start` and `end` can be used to specify particular subranges of hues. The following values can be used when generating such a subrange: red=0, yellow=$\frac{1}{6}$, green=$\frac{2}{6}$, cyan=$\frac{3}{6}$, blue=$\frac{4}{6}$ and magenta=$\frac{5}{6}$.

Value

A character vector, `cv`, of color names. This can be used either to create a user–defined color palette for subsequent graphics by `palette(cv)`, a `col=` specification in graphics functions or in `par`.

See Also

`colors`, `palette`, `hsv`, `hcl`, `rgb`, `gray` and `col2rgb` for translating to RGB numbers.

Examples

```
require(graphics)
# A Color Wheel
pie(rep(1,12), col=rainbow(12))

##------ Some palettes ------------
demo.pal <-
  function(n, border = if (n<32) "light gray" else NA,
           main = paste("color palettes;  n=",n),
           ch.col = c("rainbow(n, start=.7, end=.1)", "heat.colors(n)",
                      "terrain.colors(n)", "topo.colors(n)",
                      "cm.colors(n)"))
{
    nt <- length(ch.col)
    i <- 1:n; j <- n / nt; d <- j/6; dy <- 2*d
    plot(i,i+d, type="n", yaxt="n", ylab="", main=main)
    for (k in 1:nt) {
        rect(i-.5, (k-1)*j+ dy, i+.4, k*j,
             col = eval(parse(text=ch.col[k])), border = border)
        text(2*j,  k * j +dy/4, ch.col[k])
    }
}
n <- if(.Device == "postscript") 64 else 16
     # Since for screen, larger n may give color allocation problem
demo.pal(n)
```

`pdf` *PDF Graphics Device*

Description

`pdf` starts the graphics device driver for producing PDF graphics.

Usage

```
pdf(file = ifelse(onefile, "Rplots.pdf", "Rplot%03d.pdf"),
    width = 6, height = 6, onefile = TRUE, family = "Helvetica",
    title = "R Graphics Output", fonts = NULL, version = "1.1",
    paper = "special", encoding, bg, fg, pointsize, pagecentre)
```

Arguments

`file`	a character string giving the name of the file. For use with `onefile=FALSE` give a C integer format such as `"Rplot%03d.pdf"` (the default in that case). (See `postscript` for further details.)
`width, height`	the width and height of the graphics region in inches.
`onefile`	logical: if true (the default) allow multiple figures in one file. If false, generate a file name containing the page number.
`family`	the font family to be used, see `postscript`.
`title`	title string to embed in the file.

fonts	a character vector specifying R graphics font family names for fonts which will be included in the PDF file.
version	a string describing the PDF version that will be required to view the output. This is a minimum, and will be increased (with a warning) if necessary.
paper	the size of paper in the printer. The choices are "a4", "letter", "legal" and "executive" (and these can be capitalized). The default is "special", which means that the width and height specify the paper size. A further choice is "default"; if this is selected, the papersize is taken from the option "papersize" if that is set and to "a4" if it is unset or empty.
encoding	the name of an encoding file. See postscript for details.
bg	the default background color to be used.
fg	the default foreground color to be used.
pointsize	the default point size to be used. Strictly speaking, in bp, that is 1/72 of an inch, but approximately in points.
pagecentre	logical: should the device region be centred on the page? – defaults to true, but is only relevant for paper != "special".

Details

pdf() opens the file file and the PDF commands needed to plot any graphics requested are sent to that file.

The file argument is interpreted as a C integer format as used by sprintf, with integer argument the page number. The default gives files 'Rplot001.pdf', ..., 'Rplot999.pdf', 'Rplot1000.pdf', ...

The family argument can be used to specify a PDF-specific font family as the initial/default font for the device.

If a device-independent R graphics font family is specified (e.g., via par(family=) in the graphics package), the PDF device makes use of the PostScript font mappings to convert the R graphics font family to a PDF-specific font family description. (See the documentation for pdfFonts.)

R does *not* embed fonts in the PDF file though, so it is only possible straightforward to use mappings to the font families that are assumed to be available in any PDF viewer: "Times" (equivalently "serif"), "Helvetica" (equivalently "sans"), "Courier" (equivalently "mono") and "Symbol" (equivalently "symbol"). Other families may be specified, but it is the user's responsibility to ensure that these fonts are available on the system and third-party software, e.g., ghostscript, may be required to embed the fonts so that the PDF can be included in other documents (e.g., LaTeX). The URW-based families described for postscript can be used with viewers such as GSView which are based on URW fonts.

See postscript for details of encodings, as the internal code is shared between the drivers. The native PDF encoding is given in file 'PDFDoc.enc'.

pdf writes uncompressed PDF. It is primarily intended for producing PDF graphics for inclusion in other documents, and PDF-includers such as pdftex are usually able to handle compression.

At present the PDF is fairly simple, with each page being represented as a single stream. The R graphics model does not distinguish graphics objects at the level of the driver interface.

The version argument declares the version of PDF that gets produced. The version must be at least 1.4 for semitransparent output to be understood, and at least 1.3 if CID fonts are to be used: if these features are used the version number will be increased (with a warning). Specifying a low version number (as the default) is useful if you want to produce PDF output that can be viewed on older or non-Adobe PDF viewers. (PDF 1.4 requires Acrobat 5 or later.)

Line widths as controlled by `par(lwd=)` are in multiples of 1/96 inch. Multiples less than 1 are allowed. `pch="."` with `cex = 1` corresponds to a square of side 1/72 inch.

Note

Acrobat Reader does not use the fonts specified but rather emulates them from multiple-master fonts. This can be seen in imprecise centering of characters, for example the multiply and divide signs in Helvetica.

Acrobat Reader 5.x and later can be extended by support for Asian and (so-called) Central European fonts (the latter only for 7.x), and this will be needed for full use of encodings other than Latin-1. See http://www.adobe.com/products/acrobat/acrrasianfontpack.html.

See Also

pdfFonts, Devices, postscript

Examples

```
## Not run:
## Test function for encodings
TestChars <- function(encoding="ISOLatin1", ...)
{
    pdf(encoding=encoding, ...)
    par(pty="s")
    plot(c(-1,16), c(-1,16), type="n", xlab="", ylab="", xaxs="i", yaxs="i")
    title(paste("Centred chars in encoding", encoding))
    grid(17, 17, lty=1)
    for(i in c(32:255)) {
        x <- i %% 16
        y <- i %/% 16
        points(x, y, pch=i)
    }
    dev.off()
}
## there will be many warnings.
TestChars("ISOLatin2")
## this does not view properly in older viewers.
TestChars("ISOLatin2", family="URWHelvetica")
## works well for viewing in gs-based viewers, and often in xpdf.
## End(Not run)
```

---

pictex *A PicTeX Graphics Driver*

Description

This function produces graphics suitable for inclusion in TeX and LaTeX documents.

Usage

```
pictex(file = "Rplots.tex", width = 5, height = 4, debug = FALSE,
       bg = "white", fg = "black")
```

Arguments

file	the file where output will appear.
width	The width of the plot in inches.
height	the height of the plot in inches.
debug	should debugging information be printed.
bg	the background color for the plot. Ignored.
fg	the foreground color for the plot. Ignored.

Details

This driver does not have any font metric information, so the use of plotmath is not supported.

Multiple plots will be placed as separate environments in the output file.

Line widths are ignored except when setting the spacing of line textures. pch="." corresponds to a square of side 1pt.

This device does not support colour (nor does the PicTeX package), and all colour settings are ignored.

Author(s)

This driver was provided by Valerio Aimale ⟨valerio@svpop.com.dist.unige.it⟩ of the Department of Internal Medicine, University of Genoa, Italy.

References

Knuth, D. E. (1984) *The TeXbook.* Reading, MA: Addison-Wesley.

Lamport, L. (1994) *LATEX: A Document Preparation System.* Reading, MA: Addison-Wesley.

Goossens, M., Mittelbach, F. and Samarin, A. (1994) *The LATEX Companion.* Reading, MA: Addison-Wesley.

See Also

postscript, Devices.

Examples

```
pictex()
plot(1:11,(-5:5)^2, type='b', main="Simple Example Plot")
dev.off()
##--------------------
## Not run:
%% LaTeX Example
\documentclass{article}
\usepackage{pictex}
\begin{document}
%...
\begin{figure}[h]
  \centerline{\input{Rplots.tex}}
  \caption{}
\end{figure}
%...
\end{document}
```

```
%%-- plain TeX Example --
\input pictex
$$ \input Rplots.tex $$
## End(Not run)
##--------------------
unlink("Rplots.tex")
```

`plotmath` *Mathematical Annotation in R*

Description

If the `text` argument to one of the text-drawing functions (`text`, `mtext`, `axis`, `legend`) in R is an expression, the argument is interpreted as a mathematical expression and the output will be formatted according to TeX-like rules. Expressions can also be used for titles, subtitles and x- and y-axis labels (but not for axis labels on `persp` plots).

In most cases other language objects (names and calls) are coerced to expressions and so can also be used.

Details

A mathematical expression must obey the normal rules of syntax for any R expression, but it is interpreted according to very different rules than for normal R expressions.

It is possible to produce many different mathematical symbols, generate sub- or superscripts, produce fractions, etc.

The output from `demo(plotmath)` includes several tables which show the available features. In these tables, the columns of grey text show sample R expressions, and the columns of black text show the resulting output.

The available features are also described in the tables below:

Syntax	**Meaning**
`x + y`	x plus y
`x - y`	x minus y
`x*y`	juxtapose x and y
`x/y`	x forwardslash y
`x %+-% y`	x plus or minus y
`x %/% y`	x divided by y
`x %*% y`	x times y
`x %.% y`	x cdot y
`x[i]`	x subscript i
`x^2`	x superscript 2
`paste(x, y, z)`	juxtapose x, y, and z
`sqrt(x)`	square root of x
`sqrt(x, y)`	yth root of x
`x == y`	x equals y
`x != y`	x is not equal to y
`x < y`	x is less than y
`x <= y`	x is less than or equal to y
`x > y`	x is greater than y

`x >= y`	x is greater than or equal to y
`x %~~% y`	x is approximately equal to y
`x %=~% y`	x and y are congruent
`x %==% y`	x is defined as y
`x %prop% y`	x is proportional to y
`plain(x)`	draw x in normal font
`bold(x)`	draw x in bold font
`italic(x)`	draw x in italic font
`bolditalic(x)`	draw x in bolditalic font
`list(x, y, z)`	comma-separated list
`...`	ellipsis (height varies)
`cdots`	ellipsis (vertically centred)
`ldots`	ellipsis (at baseline)
`x %subset% y`	x is a proper subset of y
`x %subseteq% y`	x is a subset of y
`x %notsubset% y`	x is not a subset of y
`x %supset% y`	x is a proper superset of y
`x %supseteq% y`	x is a superset of y
`x %in% y`	x is an element of y
`x %notin% y`	x is not an element of y
`hat(x)`	x with a circumflex
`tilde(x)`	x with a tilde
`dot(x)`	x with a dot
`ring(x)`	x with a ring
`bar(xy)`	xy with bar
`widehat(xy)`	xy with a wide circumflex
`widetilde(xy)`	xy with a wide tilde
`x %<->% y`	x double-arrow y
`x %->% y`	x right-arrow y
`x %<-% y`	x left-arrow y
`x %up% y`	x up-arrow y
`x %down% y`	x down-arrow y
`x %<=>% y`	x is equivalent to y
`x %=>% y`	x implies y
`x %<=% y`	y implies x
`x %dblup% y`	x double-up-arrow y
`x %dbldown% y`	x double-down-arrow y
`alpha – omega`	Greek symbols
`Alpha – Omega`	uppercase Greek symbols
`theta1, phi1, sigma1, omega1`	cursive Greek symbols
`Upsilon1`	capital upsilon with hook
`infinity`	infinity symbol
`partialdiff`	partial differential symbol
`32*degree`	32 degrees
`60*minute`	60 minutes of angle
`30*second`	30 seconds of angle
`displaystyle(x)`	draw x in normal size (extra spacing)
`textstyle(x)`	draw x in normal size
`scriptstyle(x)`	draw x in small size
`scriptscriptstyle(x)`	draw x in very small size
`underline(x)`	draw x underlined
`x ~~ y`	put extra space between x and y

`x + phantom(0) + y`	leave gap for "0", but don't draw it
`x + over(1, phantom(0))`	leave vertical gap for "0" (don't draw)
`frac(x, y)`	x over y
`over(x, y)`	x over y
`atop(x, y)`	x over y (no horizontal bar)
`sum(x[i], i==1, n)`	sum x[i] for i equals 1 to n
`prod(plain(P)(X==x), x)`	product of P(X=x) for all values of x
`integral(f(x)*dx, a, b)`	definite integral of f(x) wrt x
`union(A[i], i==1, n)`	union of A[i] for i equals 1 to n
`intersect(A[i], i==1, n)`	intersection of A[i]
`lim(f(x), x %->% 0)`	limit of f(x) as x tends to 0
`min(g(x), x > 0)`	minimum of g(x) for x greater than 0
`inf(S)`	infimum of S
`sup(S)`	supremum of S
`x^y + z`	normal operator precedence
`x^(y + z)`	visible grouping of operands
`x^{y + z}`	invisible grouping of operands
`group("(",list(a, b),"]")`	specify left and right delimiters
`bgroup("(",atop(x,y),")")`	use scalable delimiters
`group(lceil, x, rceil)`	special delimiters

Note to TeX users: TeX's
`Upsilon` is `Upsilon1`, TeX's
`varepsilon` is close to `epsilon`, and there is no equivalent of TeX's
`epsilon`. TeX's
`varpi` is close to `omega1`. `vartheta`, `varphi` and `varsigma` are allowed as synonyms for `theta1`, `phi1` and `sigma1`.

`sigma1` is also known as `stigma`, its Unicode name.

Control characters (e.g. `\n`) are not interpreted in character strings in plotmath, unlike normal plotting.

References

Murrell, P. and Ihaka, R. (2000) An approach to providing mathematical annotation in plots. *Journal of Computational and Graphical Statistics*, **9**, 582–599.

See Also

`demo(plotmath)`, `axis`, `mtext`, `text`, `title`, `substitute` `quote`, `bquote`

Examples

```
x <- seq(-4, 4, len = 101)
y <- cbind(sin(x), cos(x))
matplot(x, y, type = "l", xaxt = "n",
        main = expression(paste(plain(sin) * phi, "  and  ",
                                plain(cos) * phi)),
        ylab = expression("sin" * phi, "cos" * phi), # only 1st is taken
        xlab = expression(paste("Phase Angle ", phi)),
        col.main = "blue")
axis(1, at = c(-pi, -pi/2, 0, pi/2, pi),
     labels = expression(-pi, -pi/2, 0, pi/2, pi))
```

```
## How to combine "math" and numeric variables :
plot(1:10, type="n", xlab="", ylab="", main = "plot math & numbers")
theta <- 1.23 ; mtext(bquote(hat(theta) == .(theta)))
for(i in 2:9)
    text(i,i+1, substitute(list(xi,eta) == group("(",list(x,y),")"),
                           list(x=i, y=i+1)))
## note that both of these use calls rather than expressions.

plot(1:10, 1:10)
text(4, 9, expression(hat(beta) == (X^t * X)^{-1} * X^t * y))
text(4, 8.4, "expression(hat(beta) == (X^t * X)^{-1} * X^t * y)",
     cex = .8)
text(4, 7, expression(bar(x) == sum(frac(x[i], n), i==1, n)))
text(4, 6.4, "expression(bar(x) == sum(frac(x[i], n), i==1, n))",
     cex = .8)
text(8, 5, expression(paste(frac(1, sigma*sqrt(2*pi)), " ",
                            plain(e)^{frac(-(x-mu)^2, 2*sigma^2)})),
     cex = 1.2)
```

`png` *JPEG and PNG graphics devices*

Description

A graphics device for JPEG or PNG format bitmap files.

Usage

```
jpeg(filename = "Rplot%03d.jpeg", width = 480, height = 480,
     pointsize = 12, quality = 75, bg = "white", res = NA, ...)

png(filename = "Rplot%03d.png", width = 480, height = 480,
    pointsize = 12, bg = "white",  res = NA, ...)
```

Arguments

`filename`	the name of the output file. The page number is substituted if a C integer format is included in the character string, as in the default. (The result must be less than `PATH_MAX` characters long, and may be truncated if not. See `postscript` for further details.) Tilde expansion is performed where supported by the platform.
`width`	the width of the device in pixels.
`height`	the height of the device in pixels.
`pointsize`	the default pointsize of plotted text.
`quality`	the 'quality' of the JPEG image, as a percentage. Smaller values will give more compression but also more degradation of the image.
`bg`	default background colour.
`res`	The nominal resolution in dpi which will be recorded in the bitmap file, if a positive integer.
`...`	additional arguments to the underlying `X11` device.

Details

Plots in PNG and JPEG format can easily be converted to many other bitmap formats, and both can be displayed in modern web browsers. The PNG format is lossless and is best for line diagrams and blocks of solid colour. The JPEG format is lossy, but may be useful for image plots, for example.

`png` supports transparent backgrounds: use `bg = "transparent"`. Not all PNG viewers render files with transparency correctly. When transparency is in use a very light grey is used as the background and so will appear as transparent if used in the plot. This allows opaque white to be used, as on the example.

R can be compiled without support for either or both of these devices: this will be reported if you attempt to use them on a system where they are not supported. They may not be usable unless the X11 display is available to the owner of the R process.

By default no resolution is recorded in the file. Readers will often assume nominal resolution of 72dpi when none is recorded. As resolutions in PNG files are recorded in pixels/metre, the dpi value will be changed slightly.

Value

A plot device is opened: nothing is returned to the R interpreter.

Warnings

Note that the `width` and `height` are in pixels not inches. A warning will be issued if both are less than 20.

If you plot more than one page on one of these devices and do not include something like `%d` for the sequence number in `file`, the file will contain the last page plotted.

Note

These are based on the `X11` device, so the additional arguments to that device work, but are rarely appropriate. The colour handling will be that of the `X11` device in use.

Author(s)

Guido Masarotto and Brian Ripley

See Also

`Devices`, `dev.print`

`capabilities` to see if these devices are supported by this build of R.

`bitmap` provides an alternative way to generate PNG and JPEG plots that does not depend on accessing the X11 display but does depend on having GhostScript installed. (Device `GDD` in CRAN package **GDD** is another alternative using several other additional pieces of software.)

Examples

```
## these examples will work only if the devices are available
## and the X11 display is available.

## copy current plot to a PNG file
## Not run: dev.print(png, file="myplot.png", width=480, height=480)

png(file="myplot.png", bg="transparent")
```

```
plot(1:10)
rect(1, 5, 3, 7, col="white")
dev.off()

jpeg(file="myplot.jpeg")
example(rect)
dev.off()
## End(Not run)
```

`postscript` *PostScript Graphics*

Description

`postscript` starts the graphics device driver for producing PostScript graphics.

Usage

```
postscript(file = ifelse(onefile, "Rplots.ps", "Rplot%03d.ps"),
           onefile = TRUE, family,
           title = "R Graphics Output", fonts = NULL,
           encoding, bg, fg,
           width, height, horizontal, pointsize,
           paper, pagecentre, print.it, command)
```

Arguments

`file`	a character string giving the name of the file. If it is `""`, the output is piped to the command given by the argument `command`. If it is of the form `"|cmd"`, the output is piped to the command given by 'cmd'. For use with `onefile = FALSE` give a `printf` format such as `"Rplot%03d.ps"` (the default in that case).
`onefile`	logical: if true (the default) allow multiple figures in one file. If false, generate a file name containing the page number and use an EPSF header and no `DocumentMedia` comment.
`family`	the initial font family to be used, normally as a character string. See the section 'Families'. This defaults to `"Helvetica"`.
`title`	title string to embed in the file.
`fonts`	a character vector specifying additional R graphics font family names for font families whose declarations will be included in the PostScript file and are available for use with the device. See Details.
`encoding`	the name of an encoding file. Defaults to '"ISOLatin1.enc"' unless the locale is recognized as corresponding to a language using ISO 8859-5,7,13,15 or KOI8-R,U, in the 'enc' directory of package **grDevices**, which is used if the path does not contain a path separator. An extension `".enc"` can be omitted.
`bg`	the default background color to be used. If `"transparent"` (or an equivalent specification), no background is painted.
`fg`	the default foreground color to be used.

`width, height`
: the width and height of the graphics region in inches. The default is to use the entire page less a 0.25 inch border on each side.

`horizontal`
: the orientation of the printed image, a logical. Defaults to true, that is landscape orientation on paper sizes with width less than height.

`pointsize`
: the default point size to be used. Strictly speaking, in bp, that is 1/72 of an inch, but approximately in points.

`paper`
: the size of paper in the printer. The choices are `"a4"`, `"letter"`, `"legal"` and `"executive"` (and these can be capitalized). Also, `"special"` can be used, when the `width` and `height` specify the paper size. A further choice is `"default"`, which is the default. If this is selected, the papersize is taken from the option `"papersize"` if that is set and to `"a4"` if it is unset or empty.

`pagecentre`
: logical: should the device region be centred on the page? – defaults to true.

`print.it`
: logical: should the file be printed when the device is closed? (This only applies if `file` is a real file name.)

`command`
: the command to be used for "printing". Defaults to option `"printcmd"`; this can also be selected as `"default"`.

Details

`postscript` opens the file `file` and the PostScript commands needed to plot any graphics requested are written to that file. This file can then be printed on a suitable device to obtain hard copy.

A postscript plot can be printed via `postscript` in two ways.

1. Setting `print.it = TRUE` causes the command given in argument `command` to be called with argument `"file"` when the device is closed. Note that the plot file is not deleted unless `command` arranges to delete it.
2. `file=""` or `file="|cmd"` can be used to print using a pipe on systems that support 'popen'. Failure to open the command will probably be reported to the terminal but not to 'popen', in which case close the device by `dev.off` immediately.

The `file` argument is interpreted as a C integer format as used by `sprintf`, with integer argument the page number. The default gives files 'Rplot001.ps', ..., 'Rplot999.ps', 'Rplot1000.ps', ...

The postscript produced for a single R plot is EPS (*Encapsulated PostScript*) compatible, and can be included into other documents, e.g., into LaTeX, using `includegraphics{<filename>}`. For use in this way you will probably want to set `horizontal = FALSE, onefile = FALSE, paper = "special"`. Note that the bounding box is for the device region: if you find the white space around the plot region excessive, reduce the margins of the figure region via `par(mar=)`.

Most of the PostScript prologue used is taken from the R character vector `.ps.prolog`. This is marked in the output, and can be changed by changing that vector. (This is only advisable for PostScript experts: the standard version is in `namespace:grDevices`.)

A PostScript device has a default family, which can be set by the user via `family`. If other font families are to be used when drawing to the PostScript device, these must be declared when the device is created via `fonts`; the font family names for this argument are R graphics font family names (see the documentation for `postscriptFonts`).

Line widths as controlled by `par(lwd=)` are in multiples of 1/96 inch: nultiples less than 1 are allowed. `pch="."` with `cex = 1` corresponds to a square of side 1/72 inch.

Families

Font families are collections of fonts covering the five font faces, (conventionally plain, bold, italic, bold-italic and symbol) selected by the graphics parameter `par(font=)` or the grid parameter `gpar(fontface=)`. Font families can be specified either as an an initial/default font family for the device via the `family` argument or after the device is opened by the graphics parameter `par(family=)` or the grid parameter `gpar(fontfamily=)`. Families which will be used in addition to the initial family must be specified in the `fonts` argument when the device is opened.

Font families are declared via a call to `postscriptFonts`.

The argument `family` specifies the initial/default font family to be used. In normal use it is one of `"AvantGarde"`, `"Bookman"`, `"Courier"`, `"Helvetica"`, `"Helvetica-Narrow"`, `"NewCenturySchoolbook"`, `"Palatino"` or `"Times"`, and refers to the standard Adobe PostScript fonts families of those names which are included (or cloned) in all common PostScript devices.

Many PostScript emulators (including those based on `ghostscript`) use the URW equivalents of these fonts, which are `"URWGothic"`, `"URWBookman"`, `"NimbusMon"`, `"NimbusSan"`, `"NimbusSanCond"`, `"CenturySch"`, `"URWPalladio"` and `"NimbusRom"` respectively. If your PostScript device is using URW fonts, you will obtain access to more characters and more appropriate metrics by using these names. To make these easier to remember, `"URWHelvetica" == "NimbusSan"` and `"URWTimes" == "NimbusRom"` are also supported.

Another type of family makes use of CID-keyed fonts for East Asian languages – see `postscriptFonts`.

The `family` argument is normally a character string naming a font family, but family objects generated by `Type1Font` and `CIDFont` are also accepted. For compatibiility with earlier versions of R, the initial family can also be specified as a vector of four or five afm files.

Encodings

Encodings describe which glyphs are used to display the character codes (in the range 0–255). Most commonly R uses ISOLatin1 encoding, and the examples for `text` are in that encoding. However, the encoding used on machines running R may well be different, and by using the `encoding` argument the glyphs can be matched to encoding in use. This suffices for European and Cyrillic languages, but not for CJK languages. For the latter, composite CID fonts are used. These fonts are useful for other languages: for example they may contain Greek glyphs. (The rest of this section applies only when CID fonts are not used.)

None of this will matter if only ASCII characters (codes 32–126) are used as all the encodings (except `"TeXtext"`) agree over that range. Some encodings are supersets of ISOLatin1, too. However, if accented and special characters do not come out as you expect, you may need to change the encoding. Some other encodings are supplied with R: `"WinAnsi.enc"` and `"MacRoman.enc"` correspond to the encodings normally used on Windows and Classic MacOS (at least by Adobe), and `"PDFDoc.enc"` is the first 256 characters of the Unicode encoding, the standard for PDF. There are also encodings `"ISOLatin2.enc"`, `"CP1250.enc"`, `"ISOLatin7.enc"` (ISO 8859-13), `"CP1257.enc"`, and `"ISOLatin9.enc"` (ISO 8859-15), `"Cyrillic.enc"` (ISO 8859-5), `"KOI8-R.enc"`, `"KOI8-U.enc"`, `"CP1251.enc"`, `"Greek.enc"` (ISO 8859-7) and `"CP1253.enc"`. Note that many glyphs in these encodings are not in the fonts corresponding to the standard families. (The Adobe ones for all but Courier, Helvetica and Times cover little more than Latin-1, whereas the URW ones also cover Latin-2, Latin-7, Latin-9 and Cyrillic but no Greek. The Adobe exceptions cover the Latin character sets, but not the Euro.)

If you specify the encoding, it is your responsibility to ensure that the PostScript font contains the glyphs used. One issue here is the Euro symbol which is in the WinAnsi and MacRoman encodings

but may well not be in the PostScript fonts. (It is in the URW variants; it is not in the supplied Adobe Font Metric files.)

There is an exception. Character 45 (`"-"`) is always set as minus (its value in Adobe ISOLatin1) even though it is hyphen in the other encodings. Hyphen is available as character 173 (octal 0255) in all the Latin encodings, Cyrillic and Greek. There are some discrepancies in accounts of glyphs 39 and 96: the supplied encodings (except CP1250 and CP1251) treat these as 'quoteright' and 'quoteleft' (rather than 'quotesingle'/'acute' and 'grave' respectively), as they are in the Adobe documentation.

TeX fonts

TeX has traditionally made use of fonts such as Computer Modern which are encoded rather differently, in a 7-bit encoding. This encoding can be specified by `encoding = "TeXtext.enc"`, taking care that the ASCII characters `< > \ _ { }` are not available in those fonts.

There are supplied families `"ComputerModern"` and `"ComputerModernItalic"` which use this encoding, and which are only supported for `postscript` (and not `pdf`). They are intended to use with the Type 1 versions of the TeX CM fonts. It will normally be possible to include such output in TeX or LaTeX provided it is processed with `dvips -Ppfb -j0` or the equivalent on your system. (`-j0` turns off font subsetting.) When `family = "ComputerModern"` is used, the italic/bold-italic fonts used are slanted fonts (`cmsl10` and `cmbxsl10`). To use text italic fonts instead, set `family = "ComputerModernItalic"`.

These families use the TeX math italic and symbol fonts for a comprehensive but incomplete coverage of the glyphs covered by the Adobe symbol font in other families. This is achieved by special-casing the postscript code generated from the supplied 'CM_symbol_10.afm'.

Author(s)

Support for Computer Modern fonts is based on a contribution by Brian D'Urso ⟨durso@hussle.harvard.edu⟩.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`postscriptFonts`, `Devices`, `check.options` which is called from both `ps.options` and `postscript`.

Examples

```
## Not run:
# open the file "foo.ps" for graphics output
postscript("foo.ps")
# produce the desired graph(s)
dev.off()              # turn off the postscript device
postscript("|lp -dlw")
# produce the desired graph(s)
dev.off()              # plot will appear on printer

# for URW PostScript devices
postscript("foo.ps", family = "NimbusSan")
```

```
## for inclusion in Computer Modern TeX documents, perhaps
postscript("cm_test.eps", width = 4.0, height = 3.0,
           horizontal = FALSE, onefile = FALSE, paper = "special",
           family = "ComputerModern", encoding = "TeXtext.enc")
## The resultant postscript file can be used by dvips -Ppfb -j0.

## To test out encodings, you can use
TestChars <- function(encoding="ISOLatin1", family="URWHelvetica")
{
    postscript(encoding=encoding, family=family)
    par(pty="s")
    plot(c(-1,16), c(-1,16), type="n", xlab="", ylab="",
         xaxs="i", yaxs="i")
    title(paste("Centred chars in encoding", encoding))
    grid(17, 17, lty=1)
    for(i in c(32:255)) {
        x <- i %% 16
        y <- i %/% 16
        points(x, y, pch=i)
    }
    dev.off()
}
## there will be many warnings.  We use URW to get a complete enough
## set of font metrics.
TestChars()
TestChars("ISOLatin2")
TestChars("WinAnsi")
## End(Not run)
```

`postscriptFonts` *PostScript and PDF Font Families*

Description

These functions handle the translation of a R graphics font family name to a PostScript or PDF font description, used by the `postscript` or `pdf` graphics devices.

Usage

```
postscriptFonts(...)
pdfFonts(...)
```

Arguments

`...`	either character strings naming mappings to display, or named arguments specifying mappings to add or change.

Details

If these functions are called with no argument they list all the existing mappings, whereas if they are called with named arguments they add (or change) mappings.

A PostScript or PDF device is created with a default font family (see the documentation for `postscript`), but it is also possible to specify a font family when drawing to the device (for

example, see the documentation for "family" in par and for "fontfamily" in gpar in the grid package).

The font family sent to the device is a simple string name, which must be mapped to a set of PostScript fonts. Separate lists of mappings for postscript and pdf devices are maintained for the current R session and can be added to by the user.

The postscriptFonts and pdfFonts functions can be used to list existing mappings and to define new mappings. The Type1Font and CIDFont functions can be used to create new mappings, when the xxxFonts function is used to add them to the database. See the examples.

Default mappings are provided for four device-independent family names: "sans" for a sans-serif font, "serif" for a serif font, "mono" for a monospaced font, and "symbol" for a symbol font.

Mappings for a number of standard Adobe fonts (and URW equivalents) are also provided: "AvantGarde", "Bookman", "Courier", "Helvetica", "Helvetica-Narrow", "NewCenturySchoolbook", "Palatino" and "Times"; "URWGothic", "URWBookman", "NimbusMon", "NimbusSan" (synonym "URWHelvetica"), "NimbusSanCond", "CenturySch", "URWPalladio" and "NimbusRom" (synonym "URWTimes").

There are also mappings for "ComputerModern" and "ComputerModernItalic".

Finally, there are some default mappings for East Asian locales described in a separate section.

The specification of font metrics and encodings is described in the help for the postscript function.

The fonts are not embedded in the resulting PostScript or PDF file, so software including the PostScript or PDF plot file should either embed the font outlines (usually from '.pfb' or '.pfa' files) or use DSC comments to instruct the print spooler or including application to do so (see also embedFonts).

A font family has both an R-level name, the argument name used when postscriptFonts was called, and an internal name, the family component. These two names are the same for all the pre-defined font families.

Once a font family is in use it cannot be changed. 'In use' means that it has been specified *via* a family or fonts argument to an invocation of the same graphics device already in the R session. (For these purposes xfig counts the same as postscript but only uses some of the predefined mappings.)

Value

A list of one or more font mappings.

East Asian fonts

There are some default mappings for East Asian locales:
"Japan1", "Japan1HeiMin", "Japan1GothicBBB", and "Japan1Ryumin" for Japanese; "Korea1" and "Korea1deb" for Korean; "GB1" (Simplified Chinese) for mainland China and Singapore; "CNS1" (Traditional Chinese) for Hong Kong and Taiwan.

These refer to the following fonts

Japan1 (PS)	HeiseiKakuGo-W5 Linotype Japanese printer font
Japan1 (PDF)	KozMinPro-Regular-Acro from Adobe Reader 7.0 Japanese Font Pack
Japan1HeiMin (PS)	HeiseiMin-W3 Linotype Japanese printer font

Japan1HeiMin (PDF)	`HeiseiMin-W3-Acro` from Adobe Reader 7.0 Japanese Font Pack
Japan1GothicBBB	`GothicBBB-Medium` Japanese-market PostScript printer font
Japan1Ryumin	`Ryumin-Light` Japanese-market PostScript printer font
Korea1 (PS)	`Baekmuk-Batang` TrueType font found on some Linux systems
Korea1 (PDF)	`HYSMyeongJoStd-Medium-Acro` from Adobe Reader 7.0 Korean Font Pack
Korea1deb (PS)	`Batang-Regular` another name for Baekmuk-Batang
Korea1deb (PDF)	`HYGothic-Medium-Acro` from Adobe Reader 4.0 Korean Font Pack
GB1 (PS)	`BousungEG-Light-GB` TrueType font found on some Linux systems
GB1 (PDF)	`STSong-Light-Acro` from Adobe Reader 7.0 Simplified Chinese Font Pack
CNS1 (PS)	`MOESung-Regular` Ken Lunde's CJKV resources
CNS1 (PDF)	`MSungStd-Light-Acro` from Adobe Reader 7.0 Traditional Chinese Font Pack

`Baekmuk-Batang` can be found at `ftp://ftp.mizi.com/pub/baekmuk/`. `BousungEG-Light-GB` can be found at `ftp://ftp.gnu.org/pub/non-gnu/chinese-fonts-truetype/`. Ken Lunde's CJKV resources are at `ftp://ftp.oreilly.com/pub/examples/nutshell/cjkv/adobe/samples/`. These will need to be installed or otherwise made available to the postscript/PDF interpreter such as ghostscript (and not all interpreters can handle TrueType fonts).

You may well find that your postscript/PDF interpreters has been set up to provide aliases for many of these fonts. For example, ghostscript on Windows can optionally be installed to map common CJK fonts names to Windows TrueType fonts. (You may want to add the `-Acro` versions as well.)

Adding a mapping for a CID-keyed font is for gurus only.

Author(s)

Support for Computer Modern fonts is based on a contribution by Brian D'Urso ⟨durso@hussle.harvard.edu⟩.

See Also

`postscript` and `pdf`; `Type1Font` and `CIDFont` for specifying new font mappings.

Examples

```
postscriptFonts()
## This duplicates "ComputerModernItalic".
CMitalic <- Type1Font("ComputerModern2",
                      c("CM_regular_10.afm", "CM_boldx_10.afm",
                        "cmti10.afm", "cmbxti10.afm",
                         "CM_symbol_10.afm"),
                      encoding = "TeXtext.enc")
postscriptFonts(CMitalic = CMitalic)
```

```
## A CID font for Japanese using a different CMap and
## corresponding cmapEncoding.
`Jp_UCS-2` <- CIDFont("TestUCS2",
                  c("Adobe-Japan1-UniJIS-UCS2-H.afm",
                    "Adobe-Japan1-UniJIS-UCS2-H.afm",
                    "Adobe-Japan1-UniJIS-UCS2-H.afm",
                    "Adobe-Japan1-UniJIS-UCS2-H.afm"),
                  "UniJIS-UCS2-H", "UCS-2")
pdfFonts(`Jp_UCS-2` = `Jp_UCS-2`)
names(pdfFonts())
```

`ps.options` *Auxiliary Function to Set/View Argument of postscript*

Description

The auxiliary function `ps.options` can be used to set and view (if called without arguments) default values for the arguments to `postscript`.

`ps.options` needs to be called before calling `postscript`, and the default values it sets can be overridden by supplying arguments to `postscript`.

Usage

```
ps.options(..., reset = FALSE, override.check = FALSE)
```

Arguments

`...`	arguments `paper`, `horizontal`, `width`, `height`, `family`, `encoding`, `pointsize`, `bg`, `fg`, `onefile`, `print.it`, and `append` (ignored) can be supplied.
`reset, override.check`	logical arguments passed to `check.options`. See the Examples.

Value

A named list of arguments.

See Also

`postscript`

Examples

```
ps.options(bg = "pink")
utils::str(ps.options(reset = TRUE))

### ---- error checking of arguments: ----
ps.options(width=0:12, onefile=0, bg=pi)
# override the check for 'onefile', but not the others:
utils::str(ps.options(width=0:12, onefile=1, bg=pi,
                      override.check = c(FALSE,TRUE,FALSE)))
```

quartz *MacOS X Quartz device*

Description

quartz starts a graphics device driver for the MacOS X System. This can only be done on machines that run MacOS X.

Usage

```
quartz(display = "", width = 5, height = 5, pointsize = 12,
       family = "Helvetica", antialias = TRUE, autorefresh = TRUE)
```

Arguments

display	the display on which the graphics window will appear. The default is to use the value in the user's environment variable DISPLAY.
width	the width of the plotting window in inches.
height	the height of the plotting window in inches.
pointsize	the default pointsize to be used.
family	this is the family name of the Postscript font that will be used by the device.
antialias	whether to use antialiasing. It is never the case to set it FALSE
autorefresh	logical specifying if realtime refreshing should be done. If FALSE, the system is charged to refresh the context of the device window.

Details

Quartz is the graphic engine based on the PDF format. It is used by the graphic interface of MacOS X to render high quality graphics. As PDF it is device independent and can be rescaled without loss of definition.

If a device-independent R graphics font family is specified (e.g., via par(family=) in the graphics package), the Quartz device makes use of the Quartz font database (see quartzFonts) to convert the R graphics font family to a Quartz-specific font family description.

Calling quartz() sets .Device to "quartz".

Line widths as controlled by par(lwd=) are in multiples of the 1/72 inch, and multiples < 1 are silently converted to 1.

See Also

quartzFonts, Devices.

quartzFonts *quartz Fonts*

Description

These functions handle the translation of a device-independent R graphics font family name to a quartz font description.

Usage

```
quartzFont(family)

quartzFonts(...)
```

Arguments

family	a character vector containing the four PostScript font names for plain, bold, italic, and bolditalic versions of a font family.
...	either character strings naming mappings to display, or new (named) mappings to define.

Details

A quartz device is created with a default font (see the documentation for quartz), but it is also possible to specify a font family when drawing to the device (for example, see the documentation for gpar in the grid package).

The font family sent to the device is a simple string name, which must be mapped to something more specific to quartz fonts. A list of mappings is maintained and can be modified by the user.

The quartzFonts function can be used to list existing mappings and to define new mappings. The quartzFont function can be used to create a new mapping.

Default mappings are provided for four device-independent font family names: "sans" for a sans-serif font, "serif" for a serif font, "mono" for a monospaced font, and "symbol" for a symbol font.

See Also

quartz

Examples

```
quartzFonts()
quartzFonts("mono")
```

`recordGraphics` *Record graphics operations*

Description

Records arbitrary code on the graphics engine display list. Useful for encapsulating calculations with graphical output that depends on the calculations. Intended *only* for expert use.

Usage

```
recordGraphics(expr, list, env)
```

Arguments

`expr`	object of mode `expression` or `call` or an "unevaluated expression".
`list`	a list defining the environment in which `expr` is to be evaluated.
`env`	An `environment` specifying where R looks for objects not found in `envir`.

Details

The code in `expr` is evaluated in an environment constructed from `list`, with `env` as the parent of that environment.

All three arguments are saved on the graphics engine display list so that on a device resize or copying between devices, the original evaluation environment can be recreated and the code can be re-evaluated to reproduce the graphical output.

Value

The value from evaluating `expr`.

Warning

This function is not intended for general use. Incorrect or improper use of this function could lead to unintended and/or undesirable results.

An example of acceptable use is querying the current state of a graphics device or graphics system setting and then calling a graphics function.

An example of improper use would be calling the `assign` function to performing assignments in the global environment.

See Also

`eval`

Examples

```
plot(1:10)
# This rectangle remains 1inch wide when the device is resized
recordGraphics(
  {
    rect(4, 2,
         4 + diff(par("usr")[1:2])/par("pin")[1], 3)
```

```
},
list(),
getNamespace("graphics"))
```

`recordPlot` *Record and Replay Plots*

Description

Functions to save the current plot in an R variable, and to replay it.

Usage

```
recordPlot()
replayPlot(x)
```

Arguments

`x`	A saved plot.

Details

These functions record and replay the displaylist of the current graphics device. The returned object is of class `"recordedplot"`, and `replayPlot` acts as a `print` method for that class.

Value

`recordPlot` returns an object of class `"recordedplot"`.

`replayPlot` has no return value.

Warning

The format of recorded plots may change between R versions. Recorded plots should **not** be used as a permanent storage format for R plots.

R will always attempt to replay a recorded plot, but if the plot was recorded with a different R version then bad things may happen.

`rgb` *RGB Color Specification*

Description

This function creates "colors" corresponding to the given intensities (between 0 and `max`) of the red, green and blue primaries.

An alpha transparency value can also be specified (0 means fully transparent and `max` means opaque). If `alpha` is not specified, an opaque colour is generated.

The names argument may be used to provide names for the colors.

The values returned by these functions can be used with a `col=` specification in graphics functions or in `par`.

Usage

```
rgb(red, green, blue, alpha, names = NULL, maxColorValue = 1)
```

Arguments

red, blue, green, alpha	vectors of same length with values in $[0, M]$ where M is maxColorValue. When this is 255, the red, blue, green, and alpha values are coerced to integers in 0:255 and the result is computed most efficiently.
names	character. The names for the resulting vector.
maxColorValue	number giving the maximum of the color values range, see above.

See Also

col2rgb the "inverse" for translating R colors to RGB vectors; rainbow, hsv, hcl, gray.

Examples

```
rgb(0,1,0)
(u01 <- seq(0,1, length=11))
stopifnot(rgb(u01,u01,u01) == gray(u01))
reds <- rgb((0:15)/15, g=0,b=0, names=paste("red",0:15,sep="."))
reds

rgb(0, 0:12, 0, max = 255)# integer input
```

---

rgb2hsv *RGB to HSV Conversion*

Description

rgb2hsv transforms colors from RGB space (red/green/blue) into HSV space (hue/saturation/value).

Usage

```
rgb2hsv(r, g = NULL, b = NULL, gamma = 1, maxColorValue = 255)
```

Arguments

r	vector of "red" values in $[0, M]$, (M =maxColorValue) or 3-row rgb matrix.
g	vector of "green" values, or NULL when r is a matrix.
b	vector of "blue" values, or NULL when r is a matrix.
gamma	a "gamma correction" (supposedly applied to the r,g,b values previously), see hsv(...., gamma).
maxColorValue	number giving the maximum of the RGB color values range. The default 255 corresponds to the typical 0:255 RGB coding as in col2rgb().

Details

Value (brightness) gives the amount of light in the color.
Hue describes the dominant wavelegth.
Saturation is the amount of Hue mixed into the color.

Value

A matrix with a column for each color. The three rows of the matrix indicate hue, saturation and value and are named `"h"`, `"s"`, and `"v"` accordingly.

Author(s)

R interface by Wolfram Fischer ⟨wolfram@fischer-zim.ch⟩;
C code mainly by Nicholas Lewin-Koh ⟨nikko@hailmail.net⟩.

See Also

`hsv`, `col2rgb`, `rgb`.

Examples

```
## These (saturated, bright ones) only differ by hue
(rc <- col2rgb(c("red", "yellow","green","cyan", "blue", "magenta")))
(hc <- rgb2hsv(rc))
6 * hc["h",] # the hues are equispaced


(rgb3 <- floor(256 * matrix(runif(3*12), 3,12)))
(hsv3 <- rgb2hsv(rgb3))
## Consistency :
stopifnot(rgb3 == col2rgb(hsv(h=hsv3[1,], s=hsv3[2,], v=hsv3[3,])),
          all.equal(hsv3, rgb2hsv(rgb3/255, maxC = 1)))

## A (simplified) pure R version -- originally by Wolfram Fischer --
## showing the exact algorithm:
rgb2hsvR <- function(rgb, gamma = 1, maxColorValue = 255)
{
    if(!is.numeric(rgb)) stop("rgb matrix must be numeric")
    d <- dim(rgb)
    if(d[1] != 3) stop("rgb matrix must have 3 rows")
    n <- d[2]
    if(n == 0) return(cbind(c(h=1,s=1,v=1))[,0])
    rgb <- rgb/maxColorValue
    if(gamma != 1) rgb <- rgb ^ (1/gamma)

    ## get the max and min
    v <- apply( rgb, 2, max)
    s <- apply( rgb, 2, min)
    D <- v - s # range

    ## set hue to zero for undefined values (gray has no hue)
    h <- numeric(n)
    notgray <- ( s != v )

    ## blue hue
    idx <- (v == rgb[3,] & notgray )
```

```
    if (any (idx))
        h[idx] <- 2/3 + 1/6 * (rgb[1,idx] - rgb[2,idx]) / D[idx]
    ## green hue
    idx <- (v == rgb[2,] & notgray )
    if (any (idx))
        h[idx] <- 1/3 + 1/6 * (rgb[3,idx] - rgb[1,idx]) / D[idx]
    ## red hue
    idx <- (v == rgb[1,] & notgray )
    if (any (idx))
        h[idx] <-       1/6 * (rgb[2,idx] - rgb[3,idx]) / D[idx]

    ## correct for negative red
    idx <- (h < 0)
    h[idx] <- 1+h[idx]

    ## set the saturation
    s[! notgray] <- 0;
    s[notgray] <- 1 - s[notgray] / v[notgray]

    rbind( h=h, s=s, v=v )
}

## confirm the equivalence:
all.equal(rgb2hsv (rgb3),
          rgb2hsvR(rgb3), tol=1e-14) # TRUE
```

`trans3d` *3D to 2D Transformation for Perspective Plots*

Description

Projection of 3-dimensional to 2-dimensional points using a 4x4 viewing transformation matrix. Mainly for adding to perspective plots such as `persp`.

Usage

```
trans3d(x,y,z, pmat)
```

Arguments

`x, y, z`	numeric vectors of equal length, specifying points in 3D space.
`pmat`	a 4×4 *viewing transformation matrix*, suitable for projecting the 3D coordinates (x, y, z) into the 2D plane using homogenous 4D coordinates (x, y, z, t); such matrices are returned by `persp()`.

Value

a list with two components

`x,y`	the projected 2d coordinates of the 3d input `(x,y,z)`.

See Also

`persp`

Examples

```
## See  help(persp) {after attaching the 'graphics' package}
##      -----------
```

`Type1Font` *Type 1 and CID Fonts*

Description

These functions are used to define the translation of a R graphics font family name to a Type 1 or CID font descriptions, used by both the `postscript` and `pdf` graphics devices.

Usage

```
Type1Font(family, metrics, encoding = "default")

CIDFont(family, cmap, cmapEncoding, pdfresource = "")
```

Arguments

`family`	a character string giving the name to be used internally for a Type 1 or CID-keyed font family. This needs to uniquely identify each family, so if you modify a family which is in use (see `postscriptFonts`) you need to change the family name.
`metrics`	a character vector of four or five strings giving paths to the afm (Adobe Font Metric) files for the font.
`cmap`	the name of a CMap file for a CID-keyed font.
`encoding`	for `Type1Font`, the name of an encoding file. Defaults to `"default"`, which maps on this platform to "ISOLatin1.enc". Otherwise, a file name in the 'enc' directory of the **grDevices** package, which is used if the path does not contain a path separator. An extension `".enc"` can be omitted.
`cmapEncoding`	The name of a charset encoding to be used with the named CMap file: strings will be translated to this encoding when written to the file.
`pdfresource`	A chunk of PDF code; only required for using a CID-keyed font on `pdf`; users should not be expected to provide this.

Details

For `Type1Fonts`, if four `.afm` files are supplied the fifth is taken to be `"Symbol.afm"`. Relative paths are taken relative to the directory 'R_HOME/library/grDevices/afm'. The fifth (symbol) font must be in `AdobeSym` encoding. However, the glyphs in the first four fonts are referenced by name and any encoding given within the `.afm` files is not used.

Glyphs in CID-keyed fonts are accessed by ID (number) and not by name. The CMap file maps encoded strings (usually in a MBCS) to IDs, so `cmap` and `cmapEncoding` specifications must match. There are no real bold or italic versions of CID fonts (bold/italic were very rarely used in traditional CJK topography), and for the `pdf` device all four font faces will be identical. However, for the `postscript` device, bold and italic (and bold italic) are emulated.

CID-keyed fonts are intended only for use for the glyphs of CJK languages, which are all monospaced and are all treated as filling the same bounding box. (Thus `plotmath` will work

with such characters, but the spacing will be less carefully controlled than with Western glyphs.) The CID-keyed fonts do contain other characters, including a Latin alphabet: non-CJK glyphs are regarded as monospaced with half the width of CJK glyphs. This is often the case, but sometimes Latin glyphs designed for proportional spacing are used (and may look odd). We strongly recommend that CID-keyed fonts are **only** used for CJK glyphs.

Value

A list of class `"Type1Font"` or `"CIDFont"`.

See Also

`postscript`, `pdf`, `postscriptFonts`, and `pdfFonts`.

Examples

```
## This duplicates "ComputerModernItalic".
CMitalic <- Type1Font("ComputerModern2",
                      c("CM_regular_10.afm", "CM_boldx_10.afm",
                        "cmti10.afm", "cmbxti10.afm",
                        "CM_symbol_10.afm"),
                      encoding = "TeXtext.enc")

## Not run:
## This could be used by
postscript(family = CMitalic)
## or
postscriptFonts(CMitalic = CMitalic)  # once in a session
postscript(family = "CMitalic", encoding = "TeXtext.enc")
## End(Not run)
```

---

x11 *X Window System Graphics*

Description

`X11` starts a graphics device driver for the X Window System (version 11). This can only be done on machines that run X. `x11` is recognized as a synonym for `X11`.

Usage

```
X11(display = "", width = 7, height = 7, pointsize = 12,
    gamma = getOption("gamma"), colortype = getOption("X11colortype"),
    maxcubesize = 256, bg = "transparent", canvas = "white",
    fonts = getOption("X11fonts"), xpos = NA, ypos = NA)
```

Arguments

`display`	the display on which the graphics window will appear. The default is to use the value in the user's environment variable `DISPLAY`.
`width, height`	the width and height of the plotting window, in inches. See also Resources.
`pointsize`	the default pointsize to be used.

gamma	the gamma correction factor. This value is used to ensure that the colors perceived are linearly related to RGB values (see hsv). By default this is taken from options("gamma"), or is 1 (no correction) if that is unset (which is the usual case).
colortype	the kind of color model to be used. The possibilities are "mono", "gray", "pseudo", "pseudo.cube" and "true". Ignored if an X11 device is already open.
maxcubesize	can be used to limit the size of color cube allocated for pseudocolor devices.
bg	color. The default background color.
canvas	color. The color of the canvas, which is visible only when the background color is transparent.
fonts	X11 font description strings into which weight, slant and size will be substituted. There are two, the first for fonts 1 to 4 and the second for font 5, the symbol font. See section Fonts.
xpos, ypos	initial position of the top left corner of the window, in pixels. Negative values are from the opposite corner, e.g. xpos=-100 says the top right corner should be 100 pixels from the right edge of the screen. If NA, successive devices are cascaded in 20 pixel steps from the top left. See also Resources.

Details

By default, an X11 device will use the best color rendering strategy that it can. The choice can be overriden with the colortype parameter. A value of "mono" results in black and white graphics, "gray" in grayscale and "true" in truecolor graphics (if this is possible). The values "pseudo" and "pseudo.cube" provide color strategies for pseudocolor displays. The first strategy provides on-demand color allocation which produces exact colors until the color resources of the display are exhausted. The second causes a standard color cube to be set up, and requested colors are approximated by the closest value in the cube. The default strategy for pseudocolor displays is "pseudo".

Note: All X11 devices share a colortype which is set by the first device to be opened. To change the colortype you need to close *all* open X11 devices then open one with the desired colortype.

With colortype equal to "pseudo.cube" or "gray" successively smaller palettes are tried until one is completely allocated. If allocation of the smallest attempt fails the device will revert to "mono".

Line widths as controlled by par(lwd=) are in multiples of the pixel size, and multiples < 1 are silently converted to 1.

pch="." with cex = 1 corresponds to a rectangle of sides the larger of one pixel and 0.01 inch.

The initial size and position are only hints, and may not be acted on by the window manager.

Fonts

An initial/default font family for the device can be specified via the fonts argument, but if a device-independent R graphics font family is specified (e.g., via par(family=) in the graphics package), the X11 device makes use of the X11 font database (see X11Fonts) to convert the R graphics font family to an X11-specific font family description.

X11 chooses fonts by matching to a pattern, and it is quite possible that it will choose a font in the wrong encoding or which does not contain glyphs for the your language (particularly common in iso10646-1 fonts.

The `fonts` argument is a two-element character vector, and the first element will be crucial in successfully using non-Western-European fonts. Settings that have proved useful include

`"-*-mincho-%s-%s-*-*-%d-*-*-*-*-*-*-*"` for CJK languages and `"-cronyx-helvetica-%s-%s-*-*-%d-*-*-*-*-*-*-*"` for Russian.

For UTF-8 locales, the `XLC_LOCALE` databases provide mappings between character encodings, and you may need to add an entry for your locale (e.g. Fedora Core 3 lacks one for `ru_RU.utf8`).

Resources

The standard X11 resource `geometry` can be use to specify the window position and/or size, but will be overridden by values specified as arguments. The class looked for is `R_x11`. Note that the resource specifies the width and height in pixels and not in inches.

See Also

`Devices`,`X11Fonts`.

`X11Fonts` *X11 Fonts*

Description

These functions handle the translation of a device-independent R graphics font family name to an X11 font description.

Usage

```
X11Font(font)

X11Fonts(...)
```

Arguments

`font`	a character string containing an X11 font description.
`...`	either character strings naming mappings to display, or new (named) mappings to define.

Details

An X11 device is created with a default font (see the documentation for `X11`), but it is also possible to specify a font family when drawing to the device (for example, see the documentation for `gpar` in the grid package).

The font family sent to the device is a simple string name, which must be mapped to something more specific to X11 fonts. A list of mappings is maintained and can be modified by the user.

The `X11Fonts` function can be used to list existing mappings and to define new mappings. The `X11Font` function can be used to create a new mapping.

Default mappings are provided for four device-independent font family names: `"sans"` for a sans-serif font, `"serif"` for a serif font, `"mono"` for a monospaced font, and `"symbol"` for a symbol font.

See Also

X11

Examples

```
X11Fonts()
X11Fonts("mono")
utopia <- X11Font("-*-utopia-*-*-*-*-*-*-*-*-*-*-*-*")
X11Fonts(utopia=utopia)
```

---

xfig *XFig Graphics Device*

Description

xfig starts the graphics device driver for producing XFig (version 3.2) graphics.

The auxiliary function ps.options can be used to set and view (if called without arguments) default values for the arguments to xfig and postscript.

Usage

```
xfig(file = ifelse(onefile, "Rplots.fig", "Rplot%03d.fig"),
     onefile = FALSE, encoding = "none", ...)
```

Arguments

file	a character string giving the name of the file. For use with onefile = FALSE give a C integer format such as "Rplot%03d.fig" (the default in that case). (See postscript for further details.)
onefile	logical: if true allow multiple figures in one file. If false, assume only one page per file and generate a file number containing the page number.
encoding	The encoding in which to write text strings. The default is not to re-encode. This can be any encoding recognized by iconv: in a Western UTF-8 locale you probably want to select an 8-bit encoding such as latin1, and in an East Asian locale an EUC encoding. If re-encoding fails, the text strings will be written in the current encoding with a warning.
...	further arguments to ps.options accepted by xfig():

paper the size of paper in the printer. The choices are "A4", "Letter" and "Legal" (and these can be lowercase). A further choice is "default", which is the default. If this is selected, the papersize is taken from the option "papersize" if that is set and to "A4" if it is unset or empty.

horizontal the orientation of the printed image, a logical. Defaults to true, that is landscape orientation.

width, height the width and height of the graphics region in inches. The default is to use the entire page less a 0.25 inch border.

family the font family to be used. This must be one of "AvantGarde", "Bookman", "Courier", "Helvetica", "Helvetica-Narrow", "NewCenturySchoolbook", "Palatino" or "Times". Any other value is replaced by "Helvetica" with a warning.

pointsize the default point size to be used.

bg the default background color to be used.

fg the default foreground color to be used.

pagecentre logical: should the device region be centred on the page: defaults to TRUE.

Details

Although xfig can produce multiple plots in one file, the XFig format does not say how to separate or view them. So onefile = FALSE is the default.

The file argument is interpreted as a C integer format as used by sprintf, with integer argument the page number. The default gives files 'Rplot001.fig', ..., 'Rplot999.fig', 'Rplot1000.fig', ...

Line widths as controlled by par(lwd=) are in multiples of 5/6*1/72 inch. Multiples less than 1 are allowed. pch="." with cex = 1 corresponds to a square of side 1/72 inch.

Windows users can make use of WinFIG (http://www.schmidt-web-berlin.de/WinFIG.htm).

Note

Only some line textures (0 <= lty < 4) are used. Eventually this will be partially remedied, but the XFig file format does not allow as general line textures as the R model. Unimplemented line textures are displayed as *dash-double-dotted*.

There is a limit of 512 colours (plus white and black) per file.

See Also

Devices, postscript, ps.options.

---

xy.coords *Extracting Plotting Structures*

Description

xy.coords is used by many functions to obtain x and y coordinates for plotting. The use of this common mechanism across all relevant R functions produces a measure of consistency.

Usage

```
xy.coords(x, y = NULL, xlab = NULL, ylab = NULL, log = NULL,
          recycle = FALSE)
```

Arguments

x, y	the x and y coordinates of a set of points. Alternatively, a single argument x can be provided.
xlab,ylab	names for the x and y variables to be extracted.
log	character, "x", "y" or both, as for plot. Sets negative values to NA and gives a warning.
recycle	logical; if TRUE, recycle (rep) the shorter of x or y if their lengths differ.

Details

An attempt is made to interpret the arguments `x` and `y` in a way suitable for bivariate plotting (or other bivariate procedures).

If `y` is `NULL` and `x` is a

formula: of the form `yvar ~ xvar`. `xvar` and `yvar` are used as x and y variables.

list: containing components `x` and `y`, these are used to define plotting coordinates.

time series: the x values are taken to be `time(x)` and the y values to be the time series.

matrix or `data.frame` with two or more columns: the first is assumed to contain the x values and the second the y values. *Note* that is also true if `x` has columns named `"x"` and `"y"`; these names will be irrelevant here.

In any other case, the `x` argument is coerced to a vector and returned as **y** component where the resulting `x` is just the index vector `1:n`. In this case, the resulting `xlab` component is set to `"Index"`.

If `x` (after transformation as above) inherits from class `"POSIXt"` it is coerced to class `"POSIXct"`.

Value

A list with the components

`x`	numeric (i.e., `"double"`) vector of abscissa values.
`y`	numeric vector of the same length as `x`.
`xlab`	`character(1)` or `NULL`, the 'label' of `x`.
`ylab`	`character(1)` or `NULL`, the 'label' of `y`.

See Also

`plot.default`, `lines`, `points` and `lowess` are examples of functions which use this mechanism.

Examples

```
xy.coords(stats::fft(c(1:10)), NULL)

with(cars, xy.coords(dist ~ speed, NULL)$xlab ) # = "speed"

xy.coords(1:3, 1:2, recycle=TRUE)
xy.coords(-2:10,NULL, log="y")
##> warning: 3 y values <=0 omitted ..
```

`xyz.coords` *Extracting Plotting Structures*

Description

Utility for obtaining consistent x, y and z coordinates and labels for three dimensional (3D) plots.

Usage

```
xyz.coords(x, y = NULL, z = NULL, xlab = NULL, ylab = NULL, zlab = NULL,
           log = NULL, recycle = FALSE)
```

Arguments

`x, y, z`	the x, y and z coordinates of a set of points. Both `y` and `z` can be left at `NULL`. In this case, an attempt is made to interpret `x` in a way suitable for plotting. If the argument is a formula `zvar ~ xvar + yvar`, `xvar`, `yvar` and `zvar` are used as x, y and z variables; if the argument is a list containing components `x`, `y` and `z`, these are assumed to define plotting coordinates; if the argument is a matrix or `data.frame` with three or more columns, the first is assumed to contain the x values, the 2nd the y ones, and the 3rd the z ones – independently of any column names that `x` may have. Alternatively two arguments `x` and `y` can be provided (leaving `z = NULL`). One may be real, the other complex; in any other case, the arguments are coerced to vectors and the values plotted against their indices.
`xlab, ylab, zlab`	names for the x, y and z variables to be extracted.
`log`	character, `"x"`, `"y"`, `"z"` or combinations. Sets negative values to `NA` and gives a warning.
`recycle`	logical; if `TRUE`, recycle (`rep`) the shorter ones of `x`, `y` or `z` if their lengths differ.

Value

A list with the components

`x`	numeric (i.e., `double`) vector of abscissa values.
`y`	numeric vector of the same length as `x`.
`z`	numeric vector of the same length as `x`.
`xlab`	`character(1)` or `NULL`, the axis label of `x`.
`ylab`	`character(1)` or `NULL`, the axis label of `y`.
`zlab`	`character(1)` or `NULL`, the axis label of `z`.

Author(s)

Uwe Ligges and Martin Maechler

See Also

`xy.coords` for 2D.

Examples

```
xyz.coords(data.frame(10*1:9, -4), y = NULL, z = NULL)

xyz.coords(1:6, stats::fft(1:6), z = NULL, xlab = "X", ylab = "Y")

y <- 2 * (x2 <- 10 + (x1 <- 1:10))
xyz.coords(y ~ x1 + x2, y = NULL, z = NULL)

xyz.coords(data.frame(x = -1:9, y = 2:12, z = 3:13), y = NULL, z = NULL,
           log = "xy")
##> Warning message: 2 x values <= 0 omitted ...
```

Chapter 4

The `graphics` package

`graphics-package` *The R Graphics Package*

Description

R functions for base graphics

Details

This package contains functions for base graphics. Base graphics are traditional S graphics, as opposed to the newer grid graphics.

For a complete list of functions with individual help pages, use `library(help="graphics")`.

Author(s)

R Development Core Team and contributors worldwide

Maintainer: R Core Team ⟨R-core@r-project.org⟩

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

`abline` *Add a Straight Line to a Plot*

Description

This function adds one or more straight lines through the current plot.

Usage

```
abline(a = NULL, b = NULL, h = NULL, v = NULL, reg = NULL,
       coef = NULL, untf = FALSE, ...)
```

Arguments

a, b	the intercept and slope, single values.
untf	logical asking whether to *untransform*. See Details.
h	the y-value(s) for horizontal line(s).
v	the x-value(s) for vertical line(s).
coef	a vector of length two giving the intercept and slope.
reg	an object with a coef method. See Details.
...	graphical parameters such as col, lty and lwd (possibly as vectors: see Details) and the line characteristics lend, ljoin and lmitre.

Details

Typical usages are

```
abline(a, b, untf = FALSE, ...)
abline(h=, untf = FALSE, ...)
abline(v=, untf = FALSE, ...)
abline(coef=, untf = FALSE, ...)
abline(reg=, untf = FALSE, ...)
```

The first form specifies the line in intercept/slope form (alternatively a can be specified on its own and is taken to contain the slope and intercept in vector form).

The h= and v= forms draw horizontal and vertical lines at the specified coordinates.

The coef form specifies the line by a vector containing the slope and intercept.

reg is a regression object with a coef method. If this returns a vector of length 1 then the value is taken to be the slope of a line through the origin, otherwise, the first 2 values are taken to be the intercept and slope.

If untf is true, and one or both axes are log-transformed, then a curve is drawn corresponding to a line in original coordinates, otherwise a line is drawn in the transformed coordinate system. The h and v parameters always refer to original coordinates.

The graphical parameters col, lty and lwd can be specified; see par for details. For the h= and v= usages they can be vectors of length greater than one, recycled as necessary.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

lines and segments for connected and arbitrary lines given by their *endpoints*. par.

Examples

```
z <- lm(dist ~ speed, data = cars)
plot(cars)
abline(z)
```

`arrows` *Add Arrows to a Plot*

Description

Draw arrows between pairs of points.

Usage

```
arrows(x0, y0, x1, y1, length = 0.25, angle = 30, code = 2,
       col = par("fg"), lty = par("lty"), lwd = par("lwd"),
       ...)
```

Arguments

`x0, y0`	coordinates of points **from** which to draw.
`x1, y1`	coordinates of points **to** which to draw.
`length`	length of the edges of the arrow head (in inches).
`angle`	angle from the shaft of the arrow to the edge of the arrow head.
`code`	integer code, determining *kind* of arrows to be drawn.
`col, lty, lwd`	graphical parameters, possible vectors. `NA` values in `col` cause the arrow to be omitted.
`...`	graphical parameters such as `xpd` and the line characteristics `lend`, `ljoin` and `lmitre`: see `par`.

Details

For each `i`, an arrow is drawn between the point `(x0[i], y0[i])` and the point `(x1[i],y1[i])`.

If `code=1` an arrowhead is drawn at `(x0[i],y0[i])` and if `code=2` an arrowhead is drawn at `(x1[i],y1[i])`. If `code=3` a head is drawn at both ends of the arrow. Unless `length = 0`, when no head is drawn.

The graphical parameters `col`, `lty` and `lwd` can be vectors of length greater than one and will be recycled if necessary.

The direction of a zero-length arrow is indeterminate, and hence so is the direction of the arrowheads. To allow for rounding error, arrowheads are omitted (with a warning) on any arrow of length less than 1/1000 inch.

Note

The first four arguments in the comparable S function are named `x1,y1,x2,y2`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

segments to draw segments.

Examples

```
x <- runif(12); y <- rnorm(12)
i <- order(x,y); x <- x[i]; y <- y[i]
plot(x,y, main="arrows(.) and segments(.)")
## draw arrows from point to point :
s <- seq(length(x)-1)# one shorter than data
arrows(x[s], y[s], x[s+1], y[s+1], col= 1:3)
s <- s[-length(s)]
segments(x[s], y[s], x[s+2], y[s+2], col= 'pink')
```

---

assocplot *Association Plots*

Description

Produce a Cohen-Friendly association plot indicating deviations from independence of rows and columns in a 2-dimensional contingency table.

Usage

```
assocplot(x, col = c("black", "red"), space = 0.3,
          main = NULL, xlab = NULL, ylab = NULL)
```

Arguments

x	a two-dimensional contingency table in matrix form.
col	a character vector of length two giving the colors used for drawing positive and negative Pearson residuals, respectively.
space	the amount of space (as a fraction of the average rectangle width and height) left between each rectangle.
main	overall title for the plot.
xlab	a label for the x axis. Defaults to the name (if any) of the row dimension in x.
ylab	a label for the y axis. Defaults to the name (if any) of the column dimension in x.

Details

For a two-way contingency table, the signed contribution to Pearson's χ^2 for cell i, j is $d_{ij} = (f_{ij} - e_{ij})/\sqrt{e_{ij}}$, where f_{ij} and e_{ij} are the observed and expected counts corresponding to the cell. In the Cohen-Friendly association plot, each cell is represented by a rectangle that has (signed) height proportional to d_{ij} and width proportional to $\sqrt{e_{ij}}$, so that the area of the box is proportional to the difference in observed and expected frequencies. The rectangles in each row are positioned relative to a baseline indicating independence ($d_{ij} = 0$). If the observed frequency of a cell is greater than the expected one, the box rises above the baseline and is shaded in the color specified by the first element of col, which defaults to black; otherwise, the box falls below the baseline and is shaded in the color specified by the second element of col, which defaults to red.

A more flexible and extensible implementation of association plots written in the grid graphics system is provided in the function `assoc` in the contributed package **vcd** (Meyer, Zeileis and Hornik, 2005).

References

Cohen, A. (1980), On the graphical display of the significant components in a two-way contingency table. *Communications in Statistics—Theory and Methods*, **A9**, 1025–1041.

Friendly, M. (1992), Graphical methods for categorical data. *SAS User Group International Conference Proceedings*, **17**, 190–200. http://www.math.yorku.ca/SCS/sugi/sugi17-paper.html

Meyer, D., Zeileis, A., and Hornik, K. (2005) The strucplot framework: Visualizing multi-way contingency tables with vcd. *Report 22*, Department of Statistics and Mathematics, Wirtschaftsuniversität Wien, Research Report Series. http://epub.wu-wien.ac.at/dyn/openURL?id=oai:epub.wu-wien.ac.at:epub-wu-01_8a1

See Also

`mosaicplot`, `chisq.test`.

Examples

```
## Aggregate over sex:
x <- margin.table(HairEyeColor, c(1, 2))
x
assocplot(x, main = "Relation between hair and eye color")
```

`Axis` *Generic function to add an Axis to a Plot*

Description

Generic function to add a suitable axis to the current plot.

Usage

```
Axis(x = NULL, at = NULL, ..., side, labels = NULL)
```

Arguments

`x`	an object which indicates the range over which an axis should be drawn
`at`	the points at which tick-marks are to be drawn.
`side`	an integer specifying which side of the plot the axis is to be drawn on. The axis is placed as follows: 1=below, 2=left, 3=above and 4=right.
`labels`	this can either be a logical value specifying whether (numerical) annotations are to be made at the tickmarks, or a character or expression vector of labels to be placed at the tickpoints. If this is specified as a character or expression vector, `at` should be supplied and they should be the same length.
`...`	Arguments to be passed to methods and perhaps then to `axis`.

Details

This is a generic function. It works in a slightly non-standard way: if x is supplied and non-NULL it dispatches on x, otherwise if at is supplied and non-NULL it dispatches on at, and the default action is to call axis, omitting argument x.

The idea is that for plots for which either or both of the axes are numerical but with a special interpretation, the standard plotting functions (including boxplot, contour, coplot, filled.contour, pairs, plot.default, rug and stripchart) will set up user coordinates and Axis will be called to label them appropriately.

There are "Date", "POSIXct" and "POSIXlt" methods which can pass an argument format onto the appropriate axis method.

Value

The numeric locations on the axis scale at which tick marks were drawn when the plot was first drawn (see Details).

This function is usually invoked for its side effect, which is to add an axis to an already existing plot.

See Also

axis.

---

axis *Add an Axis to a Plot*

Description

Adds an axis to the current plot, allowing the specification of the side, position, labels, and other options.

Usage

```
axis(side, at = NULL, labels = TRUE, tick = TRUE, line = NA,
     pos = NA, outer = FALSE, font = NA,
     lty = "solid", lwd = 1, col = NULL, hadj = NA, padj = NA,
     ...)
```

Arguments

side	an integer specifying which side of the plot the axis is to be drawn on. The axis is placed as follows: 1=below, 2=left, 3=above and 4=right.
at	the points at which tick-marks are to be drawn. Non-finite (infinite, NaN or NA) values are omitted. By default, when NULL, tickmark locations are computed, see Details below.
labels	this can either be a logical value specifying whether (numerical) annotations are to be made at the tickmarks, or a character or expression vector of labels to be placed at the tickpoints. If this is specified as a character or expression vector, at should be supplied and they should be the same length.
tick	a logical value specifying whether tickmarks and an axis line should be drawn

`line` the number of lines into the margin which the axis will be drawn. If not `NA` this overrides the value of the graphical parameter `mgp[3]`. The relative placing of tickmarks and tick labels is unchanged.

`pos` the coordinate at which the axis line is to be drawn: if not `NA` this overrides the values of both `line` and `mgp[3]`.

`outer` a logical value indicating whether the axis should be drawn in the outer plot margin, rather than the standard plot margin.

`font` font for text. Defaults to `par("font")`.

`lty, lwd` line type, width for the axis line and the tick marks.

`col` color for the axis line and the tick marks. Here `NULL` means to use `par("fg")`, possibly specified inline.

`hadj` adjustment (see `par("adj")`) for all labels *parallel* ('horizontal') to the reading direction. If this is not a finite value, the default is used (centring for strings parallel to the axis, justification of the end nearest the axis otherwise).

`padj` adjustment for each tick label *perpendicular* to the reading direction. For labels parallel to the axes, `padj=0` means right or top alignment, and `padj=1` means left or bottom alignment. This can be a vector given a value for each string, and will be recycled as necessary.

If `padj` is not a finite value (the default), the value of `par("las")` determines the adjustment. For strings plotted perpendicular to the axis the default is to centre the string.

`...` other graphical parameters may also be passed as arguments to this function, particularly, `cex.axis`, `col.axis` and `font.axis` for axis annotation, `mgp` and `xaxp` or `yaxp` for positioning, `tck` or `tcl` for tick mark length and direction, `las` for vertical/horizontal label orientation, or `fg` instead of `col`, see `par` on these.

Parameters `xaxt` (sides 1 and 3) and `yaxt` (sides 2 and 4) control if the axis is plotted at all.

Note that `xpd` is not accepted as clipping is always to the device region, and that `lab` will partial match to argument `labels` unless the latter is also supplied. (Since the default axes has already been set up by `plot.window`, `lab` will not be acted on by `axis`.)

Details

The axis line is drawn from the lowest to the highest value of `at`, but will be clipped at the plot region. Only ticks which are drawn from points within the plot region (up to a tolerance for rounding error) are plotted, but the ticks and their labels may well extend outside the plot region.

When `at = NULL`, pretty tick mark locations are computed internally (the same way `axTicks(side)` would) from `par("xaxp")` or `"yaxp"` and `par("xlog")` (or `"ylog"`). Note that these locations may change if an on-screen plot is resized (for example, if the `plot` argument `asp` (see `plot.window`) is set.)

If `labels` is not specified, the numeric values supplied or calculated for `at` are converted to character strings as if they were a numeric vector printed by `print.default(digits=7)`.

The code tries hard not to draw overlapping tick labels, and so will omit labels where they would abut or overlap previously drawn labels. This can result in, for example, every other tick being labelled. (The ticks are drawn left to right or bottom to top, and space at least the size of an 'm' is left between labels.)

Several of the graphics parameters affect the way axes are drawn. The vertical (for sides 1 and 3) positions of the axis and the tick labels are controlled by `mgp`, the size and direction of the ticks is controlled by `tck` and `tcl` and the appearance of the tick labels by `cex.axis`, `col.axis` and `font.axis` with orientation controlled by `las` (but not `srt`, unlike S which uses `srt` if `at` is supplied and `las` if it is not). Note that `adj` is not supported. See `par` for details.

Value

The numeric locations on the axis scale at which tick marks were drawn when the plot was first drawn (see Details).

This function is usually invoked for its side effect, which is to add an axis to an already existing plot.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`Axis` for a generic interface.

`axTicks` returns the axis tick locations corresponding to `at=NULL`; `pretty` is more flexible for computing pretty tick coordinates and does *not* depend on (nor adapt to) the coordinate system in use.

Several graphics parameters affecting the appearance are documented in `par`.

Examples

```
plot(1:4, rnorm(4), axes = FALSE)
axis(1, 1:4, LETTERS[1:4])
axis(2)
box() #- to make it look "as usual"

plot(1:7, rnorm(7), main = "axis() examples",
     type = "s", xaxt = "n", frame = FALSE, col = "red")
axis(1, 1:7, LETTERS[1:7], col.axis = "blue")
# unusual options:
axis(4, col = "violet", col.axis="dark violet", lwd = 2)
axis(3, col = "gold", lty = 2, lwd = 0.5)

# one way to have a custom x axis
plot(1:10, xaxt = "n")
axis(1, xaxp=c(2, 9, 7))
```

---

axis.POSIXct *Date and Date-time Plotting Functions*

Description

Functions to plot objects of classes `"POSIXlt"`, `"POSIXct"` and `"Date"` representing calendar dates and times.

Usage

```
axis.POSIXct(side, x, at, format, labels = TRUE, ...)
axis.Date(side, x, at, format, labels = TRUE, ...)

## S3 method for class 'POSIXct':
plot(x, y, xlab = "", ...)
## S3 method for class 'POSIXlt':
plot(x, y, xlab = "", ...)
## S3 method for class 'Date':
plot(x, y, xlab = "", ...)
```

Arguments

`x, at`	A date-time or date object.
`y`	numeric values to be plotted against `x`.
`xlab`	a character string giving the label for the x axis.
`side`	See `axis`.
`format`	See `strptime`.
`labels`	Either a logical value specifying whether annotations are to be made at the tick-marks, or a vector of character strings to be placed at the tickpoints.
`...`	Further arguments to be passed from or to other methods, typically graphical parameters or arguments of `plot.default`. For the `plot` methods, also `format`.

Details

The functions plot against an x-axis of date-times. `axis.POSIXct` and `axis.Date` work quite hard to choose suitable time units (years, months, days, hours, minutes or seconds) and a sensible output format, but this can be overridden by supplying a `format` specification.

If `at` is supplied it specifies the locations of the ticks and labels whereas if `x` is specified a suitable grid of labels is chosen. Printing of tick labels can be suppressed by using `labels = FALSE`.

Value

The locations on the axis scale at which tick marks were drawn.

See Also

DateTimeClasses, Dates for details of the classes.

Examples

```
with(beaver1, {
time <- strptime(paste(1990, day, time %/% 100, time %% 100),
                 "%Y %j %H %M")
plot(time, temp, type="l") # axis at 4-hour intervals.
# now label every hour on the time axis
plot(time, temp, type="l", xaxt="n")
r <- as.POSIXct(round(range(time), "hours"))
axis.POSIXct(1, at=seq(r[1], r[2], by="hour"), format="%H")
})
```

```
plot(.leap.seconds, 1:23, type="n", yaxt="n",
     xlab="leap seconds", ylab="", bty="n")
rug(.leap.seconds)
## or as dates
lps <- as.Date(.leap.seconds)
plot(lps, 1:23, type="n", yaxt="n", xlab="leap seconds", ylab="", bty="n")
rug(lps)

## 100 random dates in a 10-week period
random.dates <- as.Date("2001/1/1") + 70*sort(runif(100))
plot(random.dates, 1:100)
# or for a better axis labelling
plot(random.dates, 1:100, xaxt="n")
axis.Date(1, at=seq(as.Date("2001/1/1"), max(random.dates)+6, "weeks"))
axis.Date(1, at=seq(as.Date("2001/1/1"), max(random.dates)+6, "days"),
     labels = FALSE, tcl = -0.2)
```

`axTicks` *Compute Axis Tickmark Locations*

Description

Compute pretty tickmark locations, the same way as R does internally. This is only non-trivial when **log** coordinates are active. By default, gives the `at` values which `axis(side)` would use.

Usage

```
axTicks(side, axp = NULL, usr = NULL, log = NULL)
```

Arguments

`side`	integer in 1:4, as for `axis`.
`axp`	numeric vector of length three, defaulting to `par("Zaxp")` where "Z" is "x" or "y" depending on the `side` argument.
`usr`	numeric vector of length four, defaulting to `par("usr")` giving horizontal ('x') and vertical ('y') user coordinate limits.
`log`	logical indicating if log coordinates are active; defaults to `par("Zlog")` where 'Z' is as for the `axp` argument above.

Details

The `axp`, `usr`, and `log` arguments must be consistent as their default values (the `par(..)` results) are. If you specify all three (as non-NULL), the graphics environment is not used at all. Note that the meaning of `axp` alters very much when `log` is `TRUE`, see the documentation on `par(xaxp=.)`.

`axTicks()` can be regarded as an R implementation of the C function `CreateAtVector()` in '..../src/main/plot.c' which is called by `axis(side,*)` when no argument `at` is specified.

Value

numeric vector of coordinate values at which axis tickmarks can be drawn. By default, when only the first argument is specified, these values should be identical to those that `axis(side)` would use or has used.

See Also

`axis`, `par`. `pretty` uses the same algorithm (but independently of the graphics environment) and has more options. However it is not available for `log = TRUE`.

Examples

```
 plot(1:7, 10*21:27)
 axTicks(1)
 axTicks(2)
 stopifnot(identical(axTicks(1), axTicks(3)),
           identical(axTicks(2), axTicks(4)))

## Show how axTicks() and axis() correspond :
op <- par(mfrow = c(3,1))
for(x in 9999*c(1,2,8)) {
    plot(x,9, log = "x")
    cat(formatC(par("xaxp"),wid=5),";",T <- axTicks(1),"\n")
    rug(T, col="red")
}
par(op)
```

---

barplot *Bar Plots*

Description

Creates a bar plot with vertical or horizontal bars.

Usage

```
barplot(height, ...)

## Default S3 method:
barplot(height, width = 1, space = NULL,
        names.arg = NULL, legend.text = NULL, beside = FALSE,
        horiz = FALSE, density = NULL, angle = 45,
        col = NULL, border = par("fg"),
        main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
        xlim = NULL, ylim = NULL, xpd = TRUE, log = "",
        axes = TRUE, axisnames = TRUE,
        cex.axis = par("cex.axis"), cex.names = par("cex.axis"),
        inside = TRUE, plot = TRUE, axis.lty = 0, offset = 0,
        add = FALSE, ...)
```

Arguments

`height`	either a vector or matrix of values describing the bars which make up the plot. If `height` is a vector, the plot consists of a sequence of rectangular bars with heights given by the values in the vector. If `height` is a matrix and `beside` is `FALSE` then each bar of the plot corresponds to a column of `height`, with the values in the column giving the heights of stacked "sub-bars" making up the bar. If `height` is a matrix and `beside` is `TRUE`, then the values in each column are juxtaposed rather than stacked.

width	optional vector of bar widths. Re-cycled to length the number of bars drawn. Specifying a single value will no visible effect unless xlim is specified.
space	the amount of space (as a fraction of the average bar width) left before each bar. May be given as a single number or one number per bar. If height is a matrix and beside is TRUE, space may be specified by two numbers, where the first is the space between bars in the same group, and the second the space between the groups. If not given explicitly, it defaults to c(0,1) if height is a matrix and beside is TRUE, and to 0.2 otherwise.
names.arg	a vector of names to be plotted below each bar or group of bars. If this argument is omitted, then the names are taken from the names attribute of height if this is a vector, or the column names if it is a matrix.
legend.text	a vector of text used to construct a legend for the plot, or a logical indicating whether a legend should be included. This is only useful when height is a matrix. In that case given legend labels should correspond to the rows of height; if legend.text is true, the row names of height will be used as labels if they are non-null.
beside	a logical value. If FALSE, the columns of height are portrayed as stacked bars, and if TRUE the columns are portrayed as juxtaposed bars.
horiz	a logical value. If FALSE, the bars are drawn vertically with the first bar to the left. If TRUE, the bars are drawn horizontally with the first at the bottom.
density	a vector giving the density of shading lines, in lines per inch, for the bars or bar components. The default value of NULL means that no shading lines are drawn. Non-positive values of density also inhibit the drawing of shading lines.
angle	the slope of shading lines, given as an angle in degrees (counter-clockwise), for the bars or bar components.
col	a vector of colors for the bars or bar components. By default, grey is used if height is a vector, and a gamma-corrected grey palette if height is a matrix.
border	the color to be used for the border of the bars.
main,sub	overall and sub title for the plot.
xlab	a label for the x axis.
ylab	a label for the y axis.
xlim	limits for the x axis.
ylim	limits for the y axis.
xpd	logical. Should bars be allowed to go outside region?
log	string specifying if axis scales should be logarithmic; see plot.default.
axes	logical. If TRUE, a vertical (or horizontal, if horiz is true) axis is drawn.
axisnames	logical. If TRUE, and if there are names.arg (see above), the other axis is drawn (with lty=0) and labeled.
cex.axis	expansion factor for numeric axis labels.
cex.names	expansion factor for axis names (bar labels).
inside	logical. If TRUE, the lines which divide adjacent (non-stacked!) bars will be drawn. Only applies when space = 0 (which it partly is when beside = TRUE).
plot	logical. If FALSE, nothing is plotted.
axis.lty	the graphics parameter lty applied to the axis and tick marks of the categorical (default horizontal) axis. Note that by default the axis is suppressed.

offset	a vector indicating how much the bars should be shifted relative to the x axis.
add	logical specifying if bars should be added to an already existing plot; defaults to FALSE.
...	arguments to be passed to/from other methods. For the default method these can include further arguments (such as axes, asp and main) and graphical parameters (see par) which are passed to plot.window(), title() and axis.

Details

This is a generic function, it currently only has a default method. A formula interface may be added eventually.

Value

A numeric vector (or matrix, when beside = TRUE), say mp, giving the coordinates of *all* the bar midpoints drawn, useful for adding to the graph.

If beside is true, use colMeans(mp) for the midpoints of each *group* of bars, see example.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

plot(..., type="h"), dotchart, hist.

Examples

```
tN <- table(Ni <- rpois(100, lambda=5))
r <- barplot(tN, col=rainbow(20))
#- type = "h" plotting *is* 'bar'plot
lines(r, tN, type='h', col='red', lwd=2)

barplot(tN, space = 1.5, axisnames=FALSE,
        sub = "barplot(..., space= 1.5, axisnames = FALSE)")

barplot(VADeaths, plot = FALSE)
barplot(VADeaths, plot = FALSE, beside = TRUE)

mp <- barplot(VADeaths) # default
tot <- colMeans(VADeaths)
text(mp, tot + 3, format(tot), xpd = TRUE, col = "blue")
barplot(VADeaths, beside = TRUE,
        col = c("lightblue", "mistyrose", "lightcyan",
                "lavender", "cornsilk"),
        legend = rownames(VADeaths), ylim = c(0, 100))
title(main = "Death Rates in Virginia", font.main = 4)

hh <- t(VADeaths)[, 5:1]
mybarcol <- "gray20"
mp <- barplot(hh, beside = TRUE,
```

```
        col = c("lightblue", "mistyrose",
                "lightcyan", "lavender"),
        legend = colnames(VADeaths), ylim= c(0,100),
        main = "Death Rates in Virginia", font.main = 4,
        sub = "Faked upper 2*sigma error bars", col.sub = mybarcol,
        cex.names = 1.5)
segments(mp, hh, mp, hh + 2*sqrt(1000*hh/100), col = mybarcol, lwd = 1.5)
stopifnot(dim(mp) == dim(hh))# corresponding matrices
mtext(side = 1, at = colMeans(mp), line = -2,
      text = paste("Mean", formatC(colMeans(hh))), col = "red")

# Bar shading example
barplot(VADeaths, angle = 15+10*1:5, density = 20, col = "black",
        legend = rownames(VADeaths))
title(main = list("Death Rates in Virginia", font = 4))

# border :
barplot(VADeaths, border = "dark blue")

# log scales (not much sense here):
barplot(tN, col=heat.colors(12), log = "y")
barplot(tN, col=gray.colors(20), log = "xy")
```

`box` *Draw a Box around a Plot*

Description

This function draws a box around the current plot in the given color and linetype. The `bty` parameter determines the type of box drawn. See `par` for details.

Usage

```
box(which = "plot", lty = "solid", ...)
```

Arguments

`which`	character, one of `"plot"`, `"figure"`, `"inner"` and `"outer"`.
`lty`	line type of the box.
`...`	further graphical parameters, such as `bty`, `col`, or `lwd`, see `par`. Note that `xpd` is not accepted as clipping is always to the device region.

Details

The choice of colour is complicated. If `col` was supplied and is not `NA`, it is used. Otherwise, if `fg` was supplied and is not `NA`, it is used. The final default is `par("col")`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

rect for drawing of arbitrary rectangles.

Examples

```
plot(1:7, abs(rnorm(7)), type = 'h', axes = FALSE)
axis(1, at = 1:7, labels = letters[1:7])
box(lty = '1373', col = 'red')
```

---

boxplot *Box Plots*

Description

Produce box-and-whisker plot(s) of the given (grouped) values.

Usage

```
boxplot(x, ...)

## S3 method for class 'formula':
boxplot(formula, data = NULL, ..., subset, na.action = NULL)

## Default S3 method:
boxplot(x, ..., range = 1.5, width = NULL, varwidth = FALSE,
        notch = FALSE, outline = TRUE, names, plot = TRUE,
        border = par("fg"), col = NULL, log = "",
        pars = list(boxwex = 0.8, staplewex = 0.5, outwex = 0.5),
        horizontal = FALSE, add = FALSE, at = NULL)
```

Arguments

formula	a formula, such as y ~ grp, where y is a numeric vector of data values to be split into groups according to the grouping variable grp (usually a factor).
data	a data.frame (or list) from which the variables in formula should be taken.
subset	an optional vector specifying a subset of observations to be used for plotting.
na.action	a function which indicates what should happen when the data contain NAs. The default is to ignore missing values in either the response or the group.
x	for specifying data from which the boxplots are to be produced. Either a numeric vector, or a single list containing such vectors. Additional unnamed arguments specify further data as separate vectors (each corresponding to a component boxplot). NAs are allowed in the data.
...	For the formula method, named arguments to be passed to the default method. For the default method, unnamed arguments are additional data vectors (unless x is a list when they are ignored), and named arguments are arguments and graphical parameters to be passed to bxp in addition to the ones given by argument pars (and override those in pars).

range	this determines how far the plot whiskers extend out from the box. If range is positive, the whiskers extend to the most extreme data point which is no more than range times the interquartile range from the box. A value of zero causes the whiskers to extend to the data extremes.
width	a vector giving the relative widths of the boxes making up the plot.
varwidth	if varwidth is TRUE, the boxes are drawn with widths proportional to the square-roots of the number of observations in the groups.
notch	if notch is TRUE, a notch is drawn in each side of the boxes. If the notches of two plots do not overlap this is 'strong evidence' that the two medians differ (Chambers *et al.*, 1983, p. 62). See boxplot.stats for the calculations used.
outline	if outline is not true, the outliers are not drawn (as points whereas S+ uses lines).
names	group labels which will be printed under each boxplot.
boxwex	a scale factor to be applied to all boxes. When there are only a few groups, the appearance of the plot can be improved by making the boxes narrower.
staplewex	staple line width expansion, proportional to box width.
outwex	outlier line width expansion, proportional to box width.
plot	if TRUE (the default) then a boxplot is produced. If not, the summaries which the boxplots are based on are returned.
border	an optional vector of colors for the outlines of the boxplots. The values in border are recycled if the length of border is less than the number of plots.
col	if col is non-null it is assumed to contain colors to be used to colour the bodies of the box plots. By default they are in the background colour.
log	character indicating if x or y or both coordinates should be plotted in log scale.
pars	a list of (potentially many) more graphical parameters, e.g., boxwex or outpch; these are passed to bxp (if plot is true); for details, see there.
horizontal	logical indicating if the boxplots should be horizontal; default FALSE means vertical boxes.
add	logical, if true *add* boxplot to current plot.
at	numeric vector giving the locations where the boxplots should be drawn, particularly when add = TRUE; defaults to 1:n where n is the number of boxes.

Details

The generic function boxplot currently has a default method (boxplot.default) and a formula interface (boxplot.formula).

If multiple groups are supplied either as multiple arguments or via a formula, parallel boxplots will be plotted, in the order of the arguments or the order of the levels of the factor (see factor).

Missing values are ignored when forming boxplots.

Value

List with the following components:

stats	a matrix, each column contains the extreme of the lower whisker, the lower hinge, the median, the upper hinge and the extreme of the upper whisker for one group/plot. If all the inputs have the same class attribute, so will this component.

n	a vector with the number of observations in each group.
conf	a matrix where each column contains the lower and upper extremes of the notch.
out	the values of any data points which lie beyond the extremes of the whiskers.
group	a vector of the same length as out whose elements indicate to which group the outlier belongs.
names	a vector of names for the groups.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Chambers, J. M., Cleveland, W. S., Kleiner, B. and Tukey, P. A. (1983) *Graphical Methods for Data Analysis*. Wadsworth & Brooks/Cole.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See also boxplot.stats.

See Also

boxplot.stats which does the computation, bxp for the plotting and more examples; and stripchart for an alternative (with small data sets).

Examples

```
## boxplot on a formula:
boxplot(count ~ spray, data = InsectSprays, col = "lightgray")
# *add* notches (somewhat funny here):
boxplot(count ~ spray, data = InsectSprays,
        notch = TRUE, add = TRUE, col = "blue")

boxplot(decrease ~ treatment, data = OrchardSprays,
        log = "y", col = "bisque")

rb <- boxplot(decrease ~ treatment, data = OrchardSprays, col="bisque")
title("Comparing boxplot()s and non-robust mean +/- SD")

mn.t <- tapply(OrchardSprays$decrease, OrchardSprays$treatment, mean)
sd.t <- tapply(OrchardSprays$decrease, OrchardSprays$treatment, sd)
xi <- 0.3 + seq(rb$n)
points(xi, mn.t, col = "orange", pch = 18)
arrows(xi, mn.t - sd.t, xi, mn.t + sd.t,
       code = 3, col = "pink", angle = 75, length = .1)

## boxplot on a matrix:
mat <- cbind(Uni05 = (1:100)/21, Norm = rnorm(100),
             T5 = rt(100, df = 5), Gam2 = rgamma(100, shape = 2))
boxplot(data.frame(mat), main = "boxplot(data.frame(mat), main = ...)")
par(las=1)# all axis labels horizontal
boxplot(data.frame(mat), main = "boxplot(*, horizontal = TRUE)",
        horizontal = TRUE)

## Using 'at = ' and adding boxplots -- example idea by Roger Bivand :

boxplot(len ~ dose, data = ToothGrowth,
        boxwex = 0.25, at = 1:3 - 0.2,
```

```
        subset = supp == "VC", col = "yellow",
        main = "Guinea Pigs' Tooth Growth",
        xlab = "Vitamin C dose mg",
        ylab = "tooth length", ylim = c(0, 35), yaxs = "i")
boxplot(len ~ dose, data = ToothGrowth, add = TRUE,
        boxwex = 0.25, at = 1:3 + 0.2,
        subset = supp == "OJ", col = "orange")
legend(2, 9, c("Ascorbic acid", "Orange juice"),
       fill = c("yellow", "orange"))

## more examples in  help(bxp)
```

---

bxp *Draw Box Plots from Summaries*

Description

bxp draws box plots based on the given summaries in z. It is usually called from within boxplot, but can be invoked directly.

Usage

```
bxp(z, notch = FALSE, width = NULL, varwidth = FALSE,
    outline = TRUE, notch.frac = 0.5, log = "", border = par("fg"),
    pars = NULL, frame.plot = axes, horizontal = FALSE, add = FALSE,
    at = NULL, show.names = NULL, ...)
```

Arguments

z	a list containing data summaries to be used in constructing the plots. These are usually the result of a call to boxplot, but can be generated in any fashion.
notch	if notch is TRUE, a notch is drawn in each side of the boxes. If the notches of two plots do not overlap then the medians are significantly different at the 5 percent level.
width	a vector giving the relative widths of the boxes making up the plot.
varwidth	if varwidth is TRUE, the boxes are drawn with widths proportional to the square-roots of the number of observations in the groups.
outline	if outline is not true, the outliers are not drawn.
notch.frac	numeric in (0,1). When notch=TRUE, the fraction of the box width that the notches should use.
border	character or numeric (vector), the color of the box borders. Is recycled for multiple boxes. Is used as default for the boxcol, medcol, whiskcol, staplecol, and outcol options (see below).
log	character, indicating if any axis should be drawn in logarithmic scale, as in plot.default.
frame.plot	logical, indicating if a "frame" (box) should be drawn; defaults to TRUE, unless axes = FALSE is specified.
horizontal	logical indicating if the boxplots should be horizontal; default FALSE means vertical boxes.

`add` logical, if true *add* boxplot to current plot.

`at` numeric vector giving the locations where the boxplots should be drawn, particularly when `add = TRUE`; defaults to `1:n` where `n` is the number of boxes.

`show.names` Set to `TRUE` or `FALSE` to override the defaults on whether an x-axis label is printed for each group.

`pars, ...` graphical parameters (etc) can be passed as arguments to this function, either as a list (`pars`) or normally(`...`), see the following. (Those in `...` take precedence over those in `pars`.)

Currently, `yaxs` and `ylim` are used 'along the boxplot', i.e., vertically, when `horizontal` is false. `xaxt`, `yaxt`, `las`, `cex.axis`, and `col.axis` are passed to `axis`, and `main`, `cex.main`, `col.main`, `sub`, `cex.sub`, `col.sub`, `xlab`, `ylab`, `cex.lab`, and `col.lab` are passed to `title`.

In addition, `axes` is accepted (see `plot.window`), with default `TRUE`.

The following arguments (or `pars` components) allow further customization of the boxplot graphics. Their defaults are typically determined from the non-prefixed version (e.g., `boxlty` from `lty`), either from the specified argument or `pars` component or the corresponding `par` one.

boxwex: a scale factor to be applied to all boxes. When there are only a few groups, the appearance of the plot can be improved by making the boxes narrower.

staplewex, outwex: staple and outlier line width expansion, proportional to box width.

boxlty, boxlwd, boxcol, boxfill: box outline type, width, color, and fill color (which currently defaults to `col` and will in future default to `par("bg")`).

medlty, medlwd, medpch, medcex, medcol, medbg: median line type, line width, point character, point size expansion, color, and background color. The default `medpch= NA` suppresses the point, and `medlty="blank"` does so for the line. Note that`medlwd` defaults to $3\times$ the "default" `lwd`.

whisklty, whisklwd, whiskcol: whisker line type, width, and color.

staplelty, staplelwd, staplecol: staple (= end of whisker) line type, width, and color.

outlty, outlwd, outpch, outcex, outcol, outbg: outlier line type, line width, point character, point size expansion, color, and background color. The default `outlty= "blank"` suppresses the lines and `outpch=NA` suppresses points.

Value

An invisible vector, actually identical to the `at` argument, with the coordinates ("x" if horizontal is false, "y" otherwise) of box centers, useful for adding to the plot.

Author(s)

The R Core development team and Arni Magnusson ⟨arnima@u.washington.edu⟩ who has provided most changes for the box*, med*, whisk*, staple*, and out* arguments.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
set.seed(753)
(bx.p <- boxplot(split(rt(100, 4), gl(5,20))))
op <- par(mfrow= c(2,2))
bxp(bx.p, xaxt = "n")
bxp(bx.p, notch = TRUE, axes = FALSE, pch = 4, boxfill=1:5)
bxp(bx.p, notch = TRUE, boxfill= "lightblue", frame= FALSE, outl= FALSE,
    main = "bxp(*, frame= FALSE, outl= FALSE)")
bxp(bx.p, notch = TRUE, boxfill= "lightblue", border= 2:6, ylim = c(-4,4),
    pch = 22, bg = "green", log = "x", main = "... log='x', ylim=*")
par(op)
op <- par(mfrow= c(1,2))

## single group -- no label
boxplot (weight ~ group, data = PlantGrowth, subset = group=="ctrl")
## with label
bx <- boxplot(weight ~ group, data = PlantGrowth,
              subset = group=="ctrl", plot = FALSE)
bxp(bx,show.names=TRUE)
par(op)

z <- split(rnorm(1000), rpois(1000,2.2))
boxplot(z, whisklty=3, main="boxplot(z, whisklty = 3)")

## Colour support similar to plot.default:
op <- par(mfrow=1:2, bg="light gray", fg="midnight blue")
 boxplot(z,   col.axis="skyblue3", main="boxplot(*, col.axis=..,main=..)")
 plot(z[[1]], col.axis="skyblue3", main=   "plot(*, col.axis=..,main=..)")
 mtext("par(bg=\"light gray\", fg=\"midnight blue\")",
       outer = TRUE, line = -1.2)
par(op)

## Mimic S-Plus:
splus <- list(boxwex=0.4, staplewex=1, outwex=1, boxfill="grey40",
              medlwd=3, medcol="white", whisklty=3, outlty=1, outpch=NA)
boxplot(z, pars=splus)
## Recycled and "sweeping" parameters
op <- par(mfrow=c(1,2))
 boxplot(z, border=1:5, lty = 3, medlty = 1, medlwd = 2.5)
 boxplot(z, boxfill=1:3, pch=1:5, lwd = 1.5, medcol="white")
par(op)
## too many possibilities
boxplot(z, boxfill= "light gray", outpch = 21:25, outlty = 2,
        bg = "pink", lwd = 2, medcol = "dark blue", medcex = 2, medpch=20)
```

`cdplot` *Conditional Density Plots*

Description

Computes and plots conditional densities describing how the conditional distribution of a categorical variable `y` changes over a numerical variable `x`.

Usage

```
cdplot(x, ...)

## Default S3 method:
cdplot(x, y,
  plot = TRUE, tol.ylab = 0.05,
  bw = "nrd0", n = 512, from = NULL, to = NULL,
  col = NULL, border = 1, main = "", xlab = NULL, ylab = NULL,
  yaxlabels = NULL, xlim = NULL, ylim = c(0, 1), ...)

## S3 method for class 'formula':
cdplot(formula, data = list(),
  plot = TRUE, tol.ylab = 0.05,
  bw = "nrd0", n = 512, from = NULL, to = NULL,
  col = NULL, border = 1, main = "", xlab = NULL, ylab = NULL,
  yaxlabels = NULL, xlim = NULL, ylim = c(0, 1), ...,
  subset = NULL)
```

Arguments

`x`	an object, the default method expects either a single numerical variable.
`y`	a `"factor"` interpreted to be the dependent variable
`formula`	a `"formula"` of type `y ~ x` with a single dependent `"factor"` and a single numerical explanatory variable.
`data`	an optional data frame.
`plot`	logical. Should the computed conditional densities be plotted?
`tol.ylab`	convenience tolerance parameter for y-axis annotation. If the distance between two labels drops under this threshold, they are plotted equidistantly.
`bw, n, from, to, ...`	arguments passed to `density`
`col`	a vector of fill colors of the same length as `levels(y)`. The default is to call `gray.colors`.
`border`	border color of shaded polygons.
`main, xlab, ylab`	character strings for annotation
`yaxlabels`	character vector for annotation of y axis, defaults to `levels(y)`.
`xlim, ylim`	the range of x and y values with sensible defaults.
`subset`	an optional vector specifying a subset of observations to be used for plotting.

Details

`cdplot` computes the conditional densities of `x` given the levels of `y` weighted by the marginal distribution of `y`. The densities are derived cumulatively over the levels of `y`.

This visualization technique is similar to spinograms (see `spineplot`) and plots $P(y|x)$ against x. The conditional probabilities are not derived by descretization (as in the spinogram), but using a smoothing approach via `density`.

Note, that the estimates of the conditional densities are more reliable for high-density regions of x. Conversely, the are less reliable in regions with only few x observations.

Value

The conditional density functions (cumulative over the levels of `y`) are returned invisibly.

Author(s)

Achim Zeileis ⟨Achim.Zeileis@R-project.org⟩

References

Hofmann, H., Theus, M. (2005), *Interactive graphics for visualizing conditional distributions*, Unpublished Manuscript.

See Also

`spineplot`, `density`

Examples

```
## NASA space shuttle o-ring failures
fail <- factor(c(2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 2, 1, 2, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1),
               levels = 1:2, labels = c("no", "yes"))
temperature <- c(53, 57, 58, 63, 66, 67, 67, 67, 68, 69, 70, 70, 70, 70, 72, 73, 75,
                 75, 76, 76, 78, 79, 81)

## CD plot
cdplot(fail ~ temperature)
cdplot(fail ~ temperature, bw = 2)
cdplot(fail ~ temperature, bw = "SJ")

## compare with spinogram
(spineplot(fail ~ temperature, breaks = 3))

## scatter plot with conditional density
cdens <- cdplot(fail ~ temperature, plot = FALSE)
plot(I(as.numeric(fail) - 1) ~ jitter(temperature, factor = 2),
     xlab = "Temperature", ylab = "Conditional failure probability")
lines(53:81, 1 - cdens[[1]](53:81), col = 2)
```

`contour` *Display Contours*

Description

Create a contour plot, or add contour lines to an existing plot.

Usage

```
contour(x, ...)

## Default S3 method:
contour(x = seq(0, 1, len = nrow(z)),
        y = seq(0, 1, len = ncol(z)),
        z,
```

```
          nlevels = 10, levels = pretty(zlim, nlevels), labels = NULL,
          xlim = range(x, finite = TRUE),
          ylim = range(y, finite = TRUE),
          zlim = range(z, finite = TRUE),
          labcex = 0.6, drawlabels = TRUE, method = "flattest",
          vfont = c("sans serif", "plain"),
          axes = TRUE, frame.plot = axes,
          col = par("fg"), lty = par("lty"), lwd = par("lwd"),
          add = FALSE, ...)
```

Arguments

x,y	locations of grid lines at which the values in z are measured. These must be in ascending order. By default, equally spaced values from 0 to 1 are used. If x is a list, its components x$x and x$y are used for x and y, respectively. If the list has component z this is used for z.
z	a matrix containing the values to be plotted (NAs are allowed). Note that x can be used instead of z for convenience.
nlevels	number of contour levels desired **iff** levels is not supplied.
levels	numeric vector of levels at which to draw contour lines.
labels	a vector giving the labels for the contour lines. If NULL then the levels are used as labels.
labcex	cex for contour labelling. This is an absolute size, not a multiple of par("cex").
drawlabels	logical. Contours are labelled if TRUE.
method	character string specifying where the labels will be located. Possible values are "simple", "edge" and "flattest" (the default). See the Details section.
vfont	if a character vector of length 2 is specified, then Hershey vector fonts are used for the contour labels. The first element of the vector selects a typeface and the second element selects a fontindex (see text for more information).
xlim, ylim, zlim	x-, y- and z-limits for the plot.
axes, frame.plot	logical indicating whether axes or a box should be drawn, see plot.default.
col	color for the lines drawn.
lty	line type for the lines drawn.
lwd	line width for the lines drawn.
add	logical. If TRUE, add to a current plot.
...	additional arguments to plot.window and title.

Details

contour is a generic function with only a default method in base R.

The methods for positioning the labels on contours are "simple" (draw at the edge of the plot, overlaying the contour line), "edge" (draw at the edge of the plot, embedded in the contour line, with no labels overlapping) and "flattest" (draw on the flattest section of the contour, embedded in the contour line, with no labels overlapping). The second and third may not draw a label on every contour line.

For information about vector fonts, see the help for `text` and `Hershey`.

Notice that `contour` interprets the `z` matrix as a table of `f(x[i], y[j])` values, so that the x axis corresponds to row number and the y axis to column number, with column 1 at the bottom, i.e. a 90 degree clockwise rotation of the conventional textual layout.

Alternatively, use `contourplot` from the **lattice** package where the `formula` notation allows to use vectors `x,y,z` of the same length.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`options("max.countour.segments")` for the maximal complexity of a single contour line.

`contourLines`, `filled.contour` for "color-filled" contours, `contourplot` (and `levelplot`) from package **lattice**. Further, `image` and the graphics demo which can be invoked as `demo(graphics)`.

Examples

```
x <- -6:16
op <- par(mfrow = c(2, 2))
contour(outer(x, x), method = "edge", vfont = c("sans serif", "plain"))
z <- outer(x, sqrt(abs(x)), FUN = "/")
image(x, x, z)
contour(x, x, z, col = "pink", add = TRUE, method = "edge",
        vfont = c("sans serif", "plain"))
contour(x, x, z, ylim = c(1, 6), method = "simple", labcex = 1)
contour(x, x, z, ylim = c(-6, 6), nlev = 20, lty = 2, method = "simple")
par(op)

## Persian Rug Art:
x <- y <- seq(-4*pi, 4*pi, len = 27)
r <- sqrt(outer(x^2, y^2, "+"))
opar <- par(mfrow = c(2, 2), mar = rep(0, 4))
for(f in pi^(0:3))
  contour(cos(r^2)*exp(-r/f),
          drawlabels = FALSE, axes = FALSE, frame = TRUE)

rx <- range(x <- 10*1:nrow(volcano))
ry <- range(y <- 10*1:ncol(volcano))
ry <- ry + c(-1,1) * (diff(rx) - diff(ry))/2
tcol <- terrain.colors(12)
par(opar); opar <- par(pty = "s", bg = "lightcyan")
plot(x = 0, y = 0,type = "n", xlim = rx, ylim = ry, xlab = "", ylab = "")
u <- par("usr")
rect(u[1], u[3], u[2], u[4], col = tcol[8], border = "red")
contour(x, y, volcano, col = tcol[2], lty = "solid", add = TRUE,
        vfont = c("sans serif", "plain"))
title("A Topographic Map of Maunga Whau", font = 4)
abline(h = 200*0:4, v = 200*0:4, col = "lightgray", lty = 2, lwd = 0.1)

## contourLines produces the same contour lines as contour
```

```
line.list <- contourLines(x, y, volcano)
plot(x = 0, y = 0,type = "n", xlim = rx, ylim = ry, xlab = "", ylab = "")
u <- par("usr")
rect(u[1], u[3], u[2], u[4], col = tcol[8], border = "red")
contour(x, y, volcano, col = tcol[2], lty = "solid", add = TRUE,
            vfont = c("sans serif", "plain"))
templines <- function(clines) {
  lines(clines[[2]], clines[[3]])
}
invisible(lapply(line.list, templines))
par(opar)
```

`coplot` *Conditioning Plots*

Description

This function produces two variants of the **co**nditioning plots discussed in the reference below.

Usage

```
coplot(formula, data, given.values, panel = points, rows, columns,
       show.given = TRUE, col = par("fg"), pch = par("pch"),
       bar.bg = c(num = gray(0.8), fac = gray(0.95)),
       xlab = c(x.name, paste("Given :", a.name)),
       ylab = c(y.name, paste("Given :", b.name)),
       subscripts = FALSE,
       axlabels = function(f) abbreviate(levels(f)),
       number = 6, overlap = 0.5, xlim, ylim, ...)
co.intervals(x, number = 6, overlap = 0.5)
```

Arguments

`formula`	a formula describing the form of conditioning plot. A formula of the form `y ~ x | a` indicates that plots of `y` versus `x` should be produced conditional on the variable `a`. A formula of the form `y ~ x| a * b` indicates that plots of `y` versus `x` should be produced conditional on the two variables `a` and `b`. All three or four variables may be either numeric or factors. When `x` or `y` are factors, the result is almost as if `as.numeric()` was applied, whereas for factor `a` or `b`, the conditioning (and its graphics if `show.given` is true) are adapted.
`data`	a data frame containing values for any variables in the formula. By default the environment where `coplot` was called from is used.
`given.values`	a value or list of two values which determine how the conditioning on `a` and `b` is to take place. When there is no `b` (i.e., conditioning only on `a`), usually this is a matrix with two columns each row of which gives an interval, to be conditioned on, but is can also be a single vector of numbers or a set of factor levels (if the variable being conditioned on is a factor). In this case (no `b`), the result of `co.intervals` can be used directly as `given.values` argument.

panel	a function(x, y, col, pch, ...) which gives the action to be carried out in each panel of the display. The default is points.
rows	the panels of the plot are laid out in a rows by columns array. rows gives the number of rows in the array.
columns	the number of columns in the panel layout array.
show.given	logical (possibly of length 2 for 2 conditioning variables): should conditioning plots be shown for the corresponding conditioning variables (default TRUE)
col	a vector of colors to be used to plot the points. If too short, the values are recycled.
pch	a vector of plotting symbols or characters. If too short, the values are recycled.
bar.bg	a named vector with components "num" and "fac" giving the background colors for the (shingle) bars, for **num**eric and **fac**tor conditioning variables respectively.
xlab	character; labels to use for the x axis and the first conditioning variable. If only one label is given, it is used for the x axis and the default label is used for the conditioning variable.
ylab	character; labels to use for the y axis and any second conditioning variable.
subscripts	logical: if true the panel function is given an additional (third) argument subscripts giving the subscripts of the data passed to that panel.
axlabels	function for creating axis (tick) labels when x or y are factors.
number	integer; the number of conditioning intervals, for a and b, possibly of length 2. It is only used if the corresponding conditioning variable is not a factor.
overlap	numeric < 1; the fraction of overlap of the conditioning variables, possibly of length 2 for x and y direction. When overlap < 0, there will be *gaps* between the data slices.
xlim	the range for the x axis.
ylim	the range for the y axis.
...	additional arguments to the panel function.
x	a numeric vector.

Details

In the case of a single conditioning variable a, when both rows and columns are unspecified, a "close to square" layout is chosen with columns >= rows.

In the case of multiple rows, the *order* of the panel plots is from the bottom and from the left (corresponding to increasing a, typically).

A panel function should not attempt to start a new plot, but just plot within a given coordinate system: thus plot and boxplot are not panel functions.

The rendering of arguments xlab and ylab is not controlled by par arguments cex.lab and font.lab even though they are plotted by mtext rather than title.

Value

co.intervals(., number, .) returns a (number × 2) matrix, say ci, where ci[k,] is the range of x values for the k-th interval.

References

Chambers, J. M. (1992) *Data for models.* Chapter 3 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

Cleveland, W. S. (1993) *Visualizing Data.* New Jersey: Summit Press.

See Also

`pairs`, `panel.smooth`, `points`.

Examples

```
## Tonga Trench Earthquakes
coplot(lat ~ long | depth, data = quakes)
given.depth <- co.intervals(quakes$depth, number=4, overlap=.1)
coplot(lat ~ long | depth, data = quakes, given.v=given.depth, rows=1)

## Conditioning on 2 variables:
ll.dm <- lat ~ long | depth * mag
coplot(ll.dm, data = quakes)
coplot(ll.dm, data = quakes, number=c(4,7), show.given=c(TRUE,FALSE))
coplot(ll.dm, data = quakes, number=c(3,7),
       overlap=c(-.5,.1)) # negative overlap DROPS values

## given two factors
Index <- seq(length=nrow(warpbreaks)) # to get nicer default labels
coplot(breaks ~ Index | wool * tension, data = warpbreaks, show.given = 0:1)
coplot(breaks ~ Index | wool * tension, data = warpbreaks,
       col = "red", bg = "pink", pch = 21, bar.bg = c(fac = "light blue"))

## Example with empty panels:
with(data.frame(state.x77), {
coplot(Life.Exp ~ Income | Illiteracy * state.region, number = 3,
       panel = function(x, y, ...) panel.smooth(x, y, span = .8, ...))
## y ~ factor -- not really sensical, but 'show off':
coplot(Life.Exp ~ state.region | Income * state.division,
       panel = panel.smooth)
})
```

`curve` *Draw Function Plots*

Description

Draws a curve corresponding to the given function or expression (in `x`) over the interval `[from,to]`.

Usage

```
curve(expr, from, to, n = 101, add = FALSE, type = "l",
      ylab = NULL, log = NULL, xlim = NULL, ...)

## S3 method for class 'function':
plot(x, from = 0, to = 1, xlim = NULL, ...)
```

Arguments

`expr`	an expression written as a function of `x`, or alternatively the name of a function which will be plotted.
`x`	a 'vectorizing' numeric R function.
`from,to`	the range over which the function will be plotted.
`n`	integer; the number of x values at which to evaluate.
`add`	logical; if `TRUE` add to already existing plot.
`xlim`	numeric of length 2; if specified, it serves as default for `c(from, to)`.
`type`	plot type: see `plot.default`.
`ylab, log, ...`	labels and graphical parameters can also be specified as arguments. `plot.function` passes all these to `curve`.

Details

The evaluation of `expr` is at `n` points equally spaced over the range `[from, to]`, possibly adapted to log scale. The points determined in this way are then joined with straight lines. `x(t)` or `expr` (with `x` inside) must return a numeric of the same length as the argument `t` or `x`.

If `add = TRUE`, `c(from,to)` default to `xlim` which defaults to the current x-limits. Further, `log` is taken from the current plot when `add` is true.

This used to be a quick hack which now seems to serve a useful purpose, but can give bad results for functions which are not smooth.

For "expensive" `expr`essions, you should use smarter tools.

See Also

`splinefun` for spline interpolation, `lines`.

Examples

```
op <- par(mfrow=c(2,2))
curve(x^3-3*x, -2, 2)
curve(x^2-2, add = TRUE, col = "violet")

plot(cos, xlim = c(-pi,3*pi), n = 1001, col = "blue")

chippy <- function(x) sin(cos(x)*exp(-x/2))
curve(chippy, -8, 7, n=2001)
curve(chippy, -8, -5)

for(ll in c("","x","y","xy"))
   curve(log(1+x), 1,100, log=ll, sub=paste("log= '",ll,"'",sep=""))
par(op)
```

`dotchart` *Cleveland Dot Plots*

Description

Draw a Cleveland dot plot.

Usage

```
dotchart(x, labels = NULL, groups = NULL, gdata = NULL,
         cex = par("cex"), pch = 21, gpch = 21, bg = par("bg"),
         color = par("fg"), gcolor = par("fg"), lcolor = "gray",
         xlim = range(x[is.finite(x)]),
         main = NULL, xlab = NULL, ylab = NULL, ...)
```

Arguments

`x`	either a vector or matrix of numeric values (`NA`s are allowed). If `x` is a matrix the overall plot consists of juxtaposed dotplots for each row.
`labels`	a vector of labels for each point. For vectors the default is to use `names(x)` and for matrices the row labels `dimnames(x)[[1]]`.
`groups`	an optional factor indicating how the elements of `x` are grouped. If `x` is a matrix, `groups` will default to the columns of `x`.
`gdata`	data values for the groups. This is typically a summary such as the median or mean of each group.
`cex`	the character size to be used. Setting `cex` to a value smaller than one can be a useful way of avoiding label overlap. Unlike many other graphics functions, this sets the actual size, not a multiplet of `par("cex")`.
`pch`	the plotting character or symbol to be used.
`gpch`	the plotting character or symbol to be used for group values.
`bg`	the background color of plotting characters or symbols to be used; use `par(bg= *)` to set the background color of the whole plot.
`color`	the color(s) to be used for points and labels.
`gcolor`	the single color to be used for group labels and values.
`lcolor`	the color(s) to be used for the horizontal lines.
`xlim`	horizontal range for the plot, see `plot.window`, e.g.
`main`	overall title for the plot, see `title`.
`xlab, ylab`	axis annotations as in `title`.
`...`	graphical parameters can also be specified as arguments.

Value

This function is invoked for its side effect, which is to produce two variants of dotplots as described in Cleveland (1985).

Dot plots are a reasonable substitute for bar plots.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Cleveland, W. S. (1985) *The Elements of Graphing Data.* Monterey, CA: Wadsworth.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

Examples

```
dotchart(VADeaths, main = "Death Rates in Virginia - 1940")
op <- par(xaxs="i")# 0 -- 100%
dotchart(t(VADeaths), xlim = c(0,100),
         main = "Death Rates in Virginia - 1940")
par(op)
```

---

filled.contour *Level (Contour) Plots*

Description

This function produces a contour plot with the areas between the contours filled in solid color (Cleveland calls this a level plot). A key showing how the colors map to z values is shown to the right of the plot.

Usage

```
filled.contour(x = seq(0, 1, len = nrow(z)),
               y = seq(0, 1, len = ncol(z)),
               z,
               xlim = range(x, finite=TRUE),
               ylim = range(y, finite=TRUE),
               zlim = range(z, finite=TRUE),
               levels = pretty(zlim, nlevels), nlevels = 20,
               color.palette = cm.colors,
               col = color.palette(length(levels) - 1),
               plot.title, plot.axes, key.title, key.axes,
               asp = NA, xaxs = "i", yaxs = "i", las = 1,
               axes = TRUE, frame.plot = axes, ...)
```

Arguments

x,y	locations of grid lines at which the values in z are measured. These must be in ascending order. By default, equally spaced values from 0 to 1 are used. If x is a list, its components x$x and x$y are used for x and y, respectively. If the list has component z this is used for z.
z	a matrix containing the values to be plotted (NAs are allowed). Note that x can be used instead of z for convenience.
xlim	x limits for the plot.
ylim	y limits for the plot.
zlim	z limits for the plot.

`levels`	a set of levels which are used to partition the range of `z`. Must be **strictly** increasing (and finite). Areas with `z` values between consecutive levels are painted with the same color.
`nlevels`	if `levels` is not specified, the range of `z`, values is divided into approximately this many levels.
`color.palette`	a color palette function to be used to assign colors in the plot.
`col`	an explicit set of colors to be used in the plot. This argument overrides any palette function specification.
`plot.title`	statements which add titles to the main plot.
`plot.axes`	statements which draw axes (and a `box`) on the main plot. This overrides the default axes.
`key.title`	statements which add titles for the plot key.
`key.axes`	statements which draw axes on the plot key. This overrides the default axis.
`asp`	the y/x aspect ratio, see `plot.window`.
`xaxs`	the x axis style. The default is to use internal labeling.
`yaxs`	the y axis style. The default is to use internal labeling.
`las`	the style of labeling to be used. The default is to use horizontal labeling.
`axes, frame.plot`	logicals indicating if axes and a box should be drawn, as in `plot.default`.
`...`	additional graphical parameters, currently only passed to `title()`.

Note

This function currently uses the `layout` function and so is restricted to a full page display. As an alternative consider the `levelplot` and `contourplot` functions from the **lattice** package which work in multipanel displays.

The output produced by `filled.contour` is actually a combination of two plots; one is the filled contour and one is the legend. Two separate coordinate systems are set up for these two plots, but they are only used internally - once the function has returned these coordinate systems are lost. If you want to annotate the main contour plot, for example to add points, you can specify graphics commands in the `plot.axes` argument. An example is given below.

Author(s)

Ross Ihaka.

References

Cleveland, W. S. (1993) *Visualizing Data.* Summit, New Jersey: Hobart.

See Also

`contour`, `image`, `palette`; `contourplot` from package **lattice**.

Examples

```
filled.contour(volcano, color = terrain.colors, asp = 1)# simple

x <- 10*1:nrow(volcano)
y <- 10*1:ncol(volcano)
filled.contour(x, y, volcano, color = terrain.colors,
    plot.title = title(main = "The Topography of Maunga Whau",
    xlab = "Meters North", ylab = "Meters West"),
    plot.axes = { axis(1, seq(100, 800, by = 100))
                  axis(2, seq(100, 600, by = 100)) },
    key.title = title(main="Height\n(meters)"),
    key.axes = axis(4, seq(90, 190, by = 10)))# maybe also asp=1
mtext(paste("filled.contour(.) from", R.version.string),
      side = 1, line = 4, adj = 1, cex = .66)

# Annotating a filled contour plot
a <- expand.grid(1:20, 1:20)
b <- matrix(a[,1] + a[,2], 20)
filled.contour(x = 1:20, y = 1:20, z = b,
               plot.axes={ axis(1); axis(2); points(10,10) })

## Persian Rug Art:
x <- y <- seq(-4*pi, 4*pi, len = 27)
r <- sqrt(outer(x^2, y^2, "+"))
filled.contour(cos(r^2)*exp(-r/(2*pi)), axes = FALSE)
## rather, the key *should* be labeled:
filled.contour(cos(r^2)*exp(-r/(2*pi)), frame.plot = FALSE, plot.axes = {})
```

---

fourfoldplot *Fourfold Plots*

Description

Creates a fourfold display of a 2 by 2 by k contingency table on the current graphics device, allowing for the visual inspection of the association between two dichotomous variables in one or several populations (strata).

Usage

```
fourfoldplot(x, color = c("#99CCFF", "#6699CC"), conf.level = 0.95,
             std = c("margins", "ind.max", "all.max"),
             margin = c(1, 2), space = 0.2, main = NULL,
             mfrow = NULL, mfcol = NULL)
```

Arguments

x	a 2 by 2 by k contingency table in array form, or as a 2 by 2 matrix if k is 1.
color	a vector of length 2 specifying the colors to use for the smaller and larger diagonals of each 2 by 2 table.
conf.level	confidence level used for the confidence rings on the odds ratios. Must be a single nonnegative number less than 1; if set to 0, confidence rings are suppressed.

std	a character string specifying how to standardize the table. Must be one of "margins", "ind.max", or "all.max", and can be abbreviated by the initial letter. If set to "margins", each 2 by 2 table is standardized to equate the margins specified by margin while preserving the odds ratio. If "ind.max" or "all.max", the tables are either individually or simultaneously standardized to a maximal cell frequency of 1.
margin	a numeric vector with the margins to equate. Must be one of 1, 2, or c(1, 2) (the default), which corresponds to standardizing the row, column, or both margins in each 2 by 2 table. Only used if std equals "margins".
space	the amount of space (as a fraction of the maximal radius of the quarter circles) used for the row and column lebals.
main	character string for the fourfold title.
mfrow	a numeric vector of the form c(nr, nc), indicating that the displays for the 2 by 2 tables should be arranged in an nr by nc layout, filled by rows.
mfcol	a numeric vector of the form c(nr, nc), indicating that the displays for the 2 by 2 tables should be arranged in an nr by nc layout, filled by columns.

Details

The fourfold display is designed for the display of 2 by 2 by k tables.

Following suitable standardization, the cell frequencies f_{ij} of each 2 by 2 table are shown as a quarter circle whose radius is proportional to $\sqrt{f_{ij}}$ so that its area is proportional to the cell frequency. An association (odds ratio different from 1) between the binary row and column variables is indicated by the tendency of diagonally opposite cells in one direction to differ in size from those in the other direction; color is used to show this direction. Confidence rings for the odds ratio allow a visual test of the null of no association; the rings for adjacent quadrants overlap iff the observed counts are consistent with the null hypothesis.

Typically, the number k corresponds to the number of levels of a stratifying variable, and it is of interest to see whether the association is homogeneous across strata. The fourfold display visualizes the pattern of association. Note that the confidence rings for the individual odds ratios are not adjusted for multiple testing.

References

Friendly, M. (1994). A fourfold display for 2 by 2 by k tables. Technical Report 217, York University, Psychology Department. http://www.math.yorku.ca/SCS/Papers/4fold/4fold.ps.gz

See Also

mosaicplot

Examples

```
## Use the Berkeley admission data as in Friendly (1995).
x <- aperm(UCBAdmissions, c(2, 1, 3))
dimnames(x)[[2]] <- c("Yes", "No")
names(dimnames(x)) <- c("Sex", "Admit?", "Department")
stats::ftable(x)

## Fourfold display of data aggregated over departments, with
## frequencies standardized to equate the margins for admission
```

```
## and sex.
## Figure 1 in Friendly (1994).
fourfoldplot(margin.table(x, c(1, 2)))

## Fourfold display of x, with frequencies in each table
## standardized to equate the margins for admission and sex.
## Figure 2 in Friendly (1994).
fourfoldplot(x)

## Fourfold display of x, with frequencies in each table
## standardized to equate the margins for admission. but not
## for sex.
## Figure 3 in Friendly (1994).
fourfoldplot(x, margin = 2)
```

`frame` *Create / Start a New Plot Frame*

Description

This function (`frame` is an alias for `plot.new`) causes the completion of plotting in the current plot (if there is one) and an advance to a new graphics frame. This is used in all high-level plotting functions and also useful for skipping plots when a multi-figure region is in use.

Usage

```
plot.new()
frame()
```

Details

There is a hook called `"plot.new"` (see `setHook`) called immediately after advancing the frame, which is used in the testing code to annotate the new page. The hook function(s) are called with no argument. (If the value is a character string, `get` is called on it from within the **graphics** namespace.)

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (`frame`.)

See Also

`plot.window`, `plot.default`.

`grid` *Add Grid to a Plot*

Description

`grid` adds an `nx` by `ny` rectangular grid to an existing plot.

Usage

```
grid(nx = NULL, ny = nx, col = "lightgray", lty = "dotted",
     lwd = par("lwd"), equilogs = TRUE)
```

Arguments

`nx,ny`	number of cells of the grid in x and y direction. When `NULL`, as per default, the grid aligns with the tick marks on the corresponding *default* axis (i.e., tick-marks as computed by `axTicks`). When `NA`, no grid lines are drawn in the corresponding direction.
`col`	character or (integer) numeric; color of the grid lines.
`lty`	character or (integer) numeric; line type of the grid lines.
`lwd`	non-negative numeric giving line width of the grid lines.
`equilogs`	logical, only used when *log* coordinates and alignment with the axis tick marks are active. Setting `equilogs = FALSE` in that case gives *non equidistant* tick aligned grid lines.

Note

If more fine tuning is required, use `abline(h = ., v = .)` directly.

References

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

`plot`, `abline`, `lines`, `points`.

Examples

```
plot(1:3)
grid(NA, 5, lwd = 2) # grid only in y-direction

## maybe change the desired number of tick marks:  par(lab=c(mx,my,7))
op <- par(mfcol = 1:2)
with(iris,
     {
     plot(Sepal.Length, Sepal.Width, col = as.integer(Species),
          xlim = c(4, 8), ylim = c(2, 4.5), panel.first = grid(),
          main = "with(iris,  plot(...., panel.first = grid(), ..) )")
     plot(Sepal.Length, Sepal.Width, col = as.integer(Species),
          panel.first = grid(3, lty=1,lwd=2),
          main = "... panel.first = grid(3, lty=1,lwd=2), ..")
```

```
      }
    )
par(op)
```

`hist` *Histograms*

Description

The generic function `hist` computes a histogram of the given data values. If `plot=TRUE`, the resulting object of `class "histogram"` is plotted by `plot.histogram`, before it is returned.

Usage

```
hist(x, ...)

## Default S3 method:
hist(x, breaks = "Sturges", freq = NULL, probability = !freq,
     include.lowest = TRUE, right = TRUE,
     density = NULL, angle = 45, col = NULL, border = NULL,
     main = paste("Histogram of" , xname),
     xlim = range(breaks), ylim = NULL,
     xlab = xname, ylab,
     axes = TRUE, plot = TRUE, labels = FALSE,
     nclass = NULL, ...)
```

Arguments

`x`	a vector of values for which the histogram is desired.
`breaks`	one of: • a vector giving the breakpoints between histogram cells, • a single number giving the number of cells for the histogram, • a character string naming an algorithm to compute the number of cells (see Details), • a function to compute the number of cells. In the last three cases the number is a suggestion only.
`freq`	logical; if `TRUE`, the histogram graphic is a representation of frequencies, the `counts` component of the result; if `FALSE`, probability densities, component `density`, are plotted (so that the histogram has a total area of one). Defaults to `TRUE` *iff* `breaks` are equidistant (and `probability` is not specified).
`probability`	an *alias* for `!freq`, for S compatibility.
`include.lowest`	logical; if `TRUE`, an `x[i]` equal to the `breaks` value will be included in the first (or last, for `right = FALSE`) bar. This will be ignored (with a warning) unless `breaks` is a vector.
`right`	logical; if `TRUE`, the histograms cells are right-closed (left open) intervals.

`density`	the density of shading lines, in lines per inch. The default value of `NULL` means that no shading lines are drawn. Non-positive values of `density` also inhibit the drawing of shading lines.
`angle`	the slope of shading lines, given as an angle in degrees (counter-clockwise).
`col`	a colour to be used to fill the bars. The default of `NULL` yields unfilled bars.
`border`	the color of the border around the bars. The default is to use the standard foreground color.
`main, xlab, ylab`	these arguments to `title` have useful defaults here.
`xlim, ylim`	the range of x and y values with sensible defaults. Note that `xlim` is *not* used to define the histogram (breaks), but only for plotting (when `plot = TRUE`).
`axes`	logical. If `TRUE` (default), axes are draw if the plot is drawn.
`plot`	logical. If `TRUE` (default), a histogram is plotted, otherwise a list of breaks and counts is returned. In the latter case, a warning is used if (typically graphical) arguments are specified that only apply to the `plot = TRUE` case.
`labels`	logical or character. Additionally draw labels on top of bars, if not `FALSE`; see `plot.histogram`.
`nclass`	numeric (integer). For S(-PLUS) compatibility only, `nclass` is equivalent to `breaks` for a scalar or character argument.
`...`	further graphical parameters passed to `plot.histogram` and their to `title` and `axis` (if `plot=TRUE`).

Details

The definition of "histogram" differs by source (with country-specific biases). R's default with equi-spaced breaks (also the default) is to plot the counts in the cells defined by `breaks`. Thus the height of a rectangle is proportional to the number of points falling into the cell, as is the area *provided* the breaks are equally-spaced.

The default with non-equi-spaced breaks is to give a plot of area one, in which the *area* of the rectangles is the fraction of the data points falling in the cells.

If `right = TRUE` (default), the histogram cells are intervals of the form `(a, b]`, i.e., they include their right-hand endpoint, but not their left one, with the exception of the first cell when `include.lowest` is `TRUE`.

For `right = FALSE`, the intervals are of the form `[a, b)`, and `include.lowest` really has the meaning of "*include highest*".

A numerical tolerance of 10^{-7} times the median bin size is applied when counting entries on the edges of bins.

The default for `breaks` is `"Sturges"`: see `nclass.Sturges`. Other names for which algorithms are supplied are `"Scott"` and `"FD"` / `"Freedman-Diaconis"` (with corresponding functions `nclass.scott` and `nclass.FD`). Case is ignored and partial matching is used. Alternatively, a function can be supplied which will compute the intended number of breaks as a function of `x`.

Value

an object of class `"histogram"` which is a list with components:

`breaks`	the $n + 1$ cell boundaries (= `breaks` if that was a vector).
`counts`	n integers; for each cell, the number of `x[]` inside.

`density`	values $\hat{f}(x_i)$, as estimated density values. If `all(diff(breaks) == 1)`, they are the relative frequencies `counts/n` and in general satisfy $\sum_i \hat{f}(x_i)(b_{i+1} - b_i) = 1$, where b_i = `breaks[i]`.
`intensities`	same as `density`. Deprecated, but retained for compatibility.
`mids`	the n cell midpoints.
`xname`	a character string with the actual `x` argument name.
`equidist`	logical, indicating if the distances between `breaks` are all the same.

Note

The resulting value does *not* depend on the values of the arguments `freq` (or `probability`) or `plot`. This is intentionally different from S.

Prior to R 1.7.0, the element `breaks` of the result was adjusted for numerical tolerances. The nominal values are now returned even though tolerances are still used when counting.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Venables, W. N. and Ripley. B. D. (2002) *Modern Applied Statistics with S*. Springer.

See Also

`nclass.Sturges`, `stem`, `density`, `truehist` in package **MASS**.

Typical plots with vertical bars are *not* histograms. Consider `barplot` or `plot(*, type = "h")` for such bar plots.

Examples

```
op <- par(mfrow=c(2, 2))
hist(islands)
utils::str(hist(islands, col="gray", labels = TRUE))

hist(sqrt(islands), br = 12, col="lightblue", border="pink")
##-- For non-equidistant breaks, counts should NOT be graphed unscaled:
r <- hist(sqrt(islands), br = c(4*0:5, 10*3:5, 70, 100, 140), col='blue1')
text(r$mids, r$density, r$counts, adj=c(.5, -.5), col='blue3')
sapply(r[2:3], sum)
sum(r$density * diff(r$breaks)) # == 1
lines(r, lty = 3, border = "purple") # -> lines.histogram(*)
par(op)

utils::str(hist(islands, br=12, plot= FALSE)) #-> 10 (~= 12) breaks
utils::str(hist(islands, br=c(12,20,36,80,200,1000,17000), plot = FALSE))

hist(islands, br=c(12,20,36,80,200,1000,17000), freq = TRUE,
     main = "WRONG histogram") # and warning

set.seed(14)
x <- rchisq(100, df = 4)

## Comparing data with a model distribution should be done with qqplot()!
qqplot(x, qchisq(ppoints(x), df = 4)); abline(0,1, col = 2, lty = 2)
```

```
## if you really insist on using hist() ... :
hist(x, freq = FALSE, ylim = c(0, 0.2))
curve(dchisq(x, df = 4), col = 2, lty = 2, lwd = 2, add = TRUE)
```

`hist.POSIXt` *Histogram of a Date or Date-Time Object*

Description

Method for `hist` applied to date or date-time objects.

Usage

```
## S3 method for class 'POSIXt':
hist(x, breaks, ...,
     xlab = deparse(substitute(x)),
     plot = TRUE, freq = FALSE,
     start.on.monday = TRUE, format)

## S3 method for class 'Date':
hist(x, breaks, ...,
     xlab = deparse(substitute(x)),
     plot = TRUE, freq = FALSE,
     start.on.monday = TRUE, format)
```

Arguments

`x`	an object inheriting from class `"POSIXt"` or `"Date"`.
`breaks`	a vector of cut points *or* number giving the number of intervals which `x` is to be cut into *or* an interval specification, one of `"days"`, `"weeks"`, `"months"` or `"years"`, plus `"secs"`, `"mins"`, `"hours"` for date-time objects.
`...`	graphical parameters, or arguments to `hist.default` such as `include.lowest`, `right` and `labels`.
`xlab`	a character string giving the label for the x axis, if plotted.
`plot`	logical. If `TRUE` (default), a histogram is plotted, otherwise a list of breaks and counts is returned.
`freq`	logical; if `TRUE`, the histogram graphic is a representation of frequencies, i.e, the `counts` component of the result; if `FALSE`, *relative* frequencies ("probabilities") are plotted.
`start.on.monday`	logical. If `breaks = "weeks"`, should the week start on Mondays or Sundays?
`format`	for the x-axis labels. See `strptime`.

Value

An object of class `"histogram"`: see `hist`.

See Also

seq.POSIXt, axis.POSIXct, hist

Examples

```
hist(.leap.seconds, "years", freq = TRUE)
hist(.leap.seconds,
     seq(ISOdate(1970, 1, 1), ISOdate(2010, 1, 1), "5 years"))

## 100 random dates in a 10-week period
random.dates <- as.Date("2001/1/1") + 70*runif(100)
hist(random.dates, "weeks", format = "%d %b")
```

---

identify *Identify Points in a Scatter Plot*

Description

identify reads the position of the graphics pointer when the (first) mouse button is pressed. It then searches the coordinates given in x and y for the point closest to the pointer. If this point is close enough to the pointer, its index will be returned as part of the value of the call.

Usage

```
identify(x, ...)

## Default S3 method:
identify(x, y = NULL, labels = seq(along = x), pos = FALSE,
         n = length(x), plot = TRUE, atpen = FALSE, offset = 0.5,
         tolerance = 0.25, ...)
```

Arguments

x,y	coordinates of points in a scatter plot. Alternatively, any object which defines coordinates (a plotting structure, time series etc: see xy.coords) can be given as x, and y left undefined.
labels	an optional character vector, the same length as x and y, giving labels for the points. Will be coerced using as.character.
pos	if pos is TRUE, a component is added to the return value which indicates where text was plotted relative to each identified point: see Value.
n	the maximum number of points to be identified.
plot	logical: if plot is TRUE, the labels are printed at the points and if FALSE they are omitted.
atpen	logical: if TRUE and plot = TRUE, the lower-left corners of the labels are plotted at the points clicked rather than relative to the points.
offset	the distance (in character widths) which separates the label from identified points. Ignored if atpen = TRUE or pos = 0.
tolerance	the maximal distance (in inches) for the pointer to be 'close enough' to a point.
...	further arguments passed to par such as cex, col and font.

Details

`identify` is only supported on screen devices such as `X11`, `windows` and `quartz`. On other devices the call will do nothing.

If `plot` is `TRUE`, the point is labelled with the corresponding element of `text`. The labels are placed below, to the left, above or to the right of the identified point, depending on where the cursor was relative to the point.

For the usual `X11` device the identification process is terminated by pressing any mouse button other than the first. For the `quartz` device the process is terminated by pressing the `ESC` key.

On most devices which support `identify`, successful selection of a point is indicated by a bell sound unless `options(locatorBell = FALSE)` has been set.

If the window is resized or hidden and then exposed before the identification process has terminated, any labels drawn by `identify` will disappear. These will reappear once the identification process has terminated and the window is resized or hidden and exposed again. This is because the labels drawn by `identify` are not recorded in the device's display list until the identification process has terminated.

Value

If `pos` is `FALSE`, an integer vector containing the indexes of the identified points.

If `pos` is `TRUE`, a list containing a component `ind`, indicating which points were identified and a component `pos`, indicating where the labels were placed relative to the identified points (1=below, 2=left, 3=above, 4=right and 0=no offset, used if `atpen = TRUE`).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`locator`

`image` *Display a Color Image*

Description

Creates a grid of colored or gray-scale rectangles with colors corresponding to the values in `z`. This can be used to display three-dimensional or spatial data aka "images". This is a generic function.

The functions `heat.colors`, `terrain.colors` and `topo.colors` create heat-spectrum (red to white) and topographical color schemes suitable for displaying ordered data, with `n` giving the number of colors desired.

Usage

```
image(x, ...)

## Default S3 method:
image(x, y, z, zlim, xlim, ylim, col = heat.colors(12),
      add = FALSE, xaxs = "i", yaxs = "i", xlab, ylab,
      breaks, oldstyle = FALSE, ...)
```

Arguments

x,y	locations of grid lines at which the values in z are measured. These must be finite, non-missing and in (strictly) ascending order. By default, equally spaced values from 0 to 1 are used. If x is a list, its components x$x and x$y are used for x and y, respectively. If the list has component z this is used for z.
z	a matrix containing the values to be plotted (NAs are allowed). Note that x can be used instead of z for convenience.
zlim	the minimum and maximum z values for which colors should be plotted, defaulting to the range of the finite values of z. Each of the given colors will be used to color an equispaced interval of this range. The *midpoints* of the intervals cover the range, so that values just outside the range will be plotted.
xlim, ylim	ranges for the plotted x and y values, defaulting to the ranges of x and y.
col	a list of colors such as that generated by rainbow, heat.colors, topo.colors, terrain.colors or similar functions.
add	logical; if TRUE, add to current plot (and disregard the following arguments). This is rarely useful because image "paints" over existing graphics.
xaxs, yaxs	style of x and y axis. The default "i" is appropriate for images. See par.
xlab, ylab	each a character string giving the labels for the x and y axis. Default to the 'call names' of x or y, or to "" if these were unspecified.
breaks	a set of breakpoints for the colours: must give one more breakpoint than colour.
oldstyle	logical. If true the midpoints of the colour intervals are equally spaced, and zlim[1] and zlim[2] were taken to be midpoints. The default is to have colour intervals of equal lengths between the limits.
...	graphical parameters for plot may also be passed as arguments to this function, as can the plot aspect ratio asp and axes (see plot.window).

Details

The length of x should be equal to the nrow(z)+1 or nrow(z). In the first case x specifies the boundaries between the cells: in the second case x specifies the midpoints of the cells. Similar reasoning applies to y. It probably only makes sense to specify the midpoints of an equally-spaced grid. If you specify just one row or column and a length-one x or y, the whole user area in the corresponding direction is filled.

Rectangles corresponding to missing values are not plotted (and so are transparent and (unless add=TRUE) the default background painted in par("bg") will show though and if that is transparent, the canvas colour will be seen).

If breaks is specified then zlim is unused and the algorithm used follows cut, so intervals are closed on the right and open on the left except for the lowest interval.

Notice that image interprets the z matrix as a table of f(x[i], y[j]) values, so that the x axis corresponds to row number and the y axis to column number, with column 1 at the bottom, i.e. a 90 degree counter-clockwise rotation of the conventional printed layout of a matrix.

Note

Based on a function by Thomas Lumley ⟨tlumley@u.washington.edu⟩.

See Also

filled.contour or heatmap which can look nicer (but are less modular), contour; The **lattice** equivalent of image is levelplot.

heat.colors, topo.colors, terrain.colors, rainbow, hsv, par.

Examples

```
x <- y <- seq(-4*pi, 4*pi, len=27)
r <- sqrt(outer(x^2, y^2, "+"))
image(z = z <- cos(r^2)*exp(-r/6), col=gray((0:32)/32))
image(z, axes = FALSE, main = "Math can be beautiful ...",
      xlab = expression(cos(r^2) * e^{-r/6}))
contour(z, add = TRUE, drawlabels = FALSE)

# Volcano data visualized as matrix. Need to transpose and flip
# matrix horizontally.
image(t(volcano)[ncol(volcano):1,])

# A prettier display of the volcano
x <- 10*(1:nrow(volcano))
y <- 10*(1:ncol(volcano))
image(x, y, volcano, col = terrain.colors(100), axes = FALSE)
contour(x, y, volcano, levels = seq(90, 200, by = 5),
        add = TRUE, col = "peru")
axis(1, at = seq(100, 800, by = 100))
axis(2, at = seq(100, 600, by = 100))
box()
title(main = "Maunga Whau Volcano", font.main = 4)
```

---

layout *Specifying Complex Plot Arrangements*

Description

layout divides the device up into as many rows and columns as there are in matrix mat, with the column-widths and the row-heights specified in the respective arguments.

Usage

```
layout(mat, widths = rep(1, ncol(mat)),
       heights = rep(1, nrow(mat)), respect = FALSE)

layout.show(n = 1)
lcm(x)
```

Arguments

mat	a matrix object specifying the location of the next N figures on the output device. Each value in the matrix must be 0 or a positive integer. If N is the largest positive integer in the matrix, then the integers $\{1, \ldots, N-1\}$ must also appear at least once in the matrix.

`widths`	a vector of values for the widths of columns on the device. Relative widths are specified with numeric values. Absolute widths (in centimetres) are specified with the `lcm()` function (see examples).
`heights`	a vector of values for the heights of rows on the device. Relative and absolute heights can be specified, see `widths` above.
`respect`	either a logical value or a matrix object. If the latter, then it must have the same dimensions as `mat` and each value in the matrix must be either `0` or `1`.
`n`	number of figures to plot.
`x`	a dimension to be interpreted as a number of centimetres.

Details

Figure i is allocated a region composed from a subset of these rows and columns, based on the rows and columns in which i occurs in `mat`.

The `respect` argument controls whether a unit column-width is the same physical measurement on the device as a unit row-height.

There is a limit (currently 50) for the numbers of rows and columns in the layout, and also for the total number of cells (500).

`layout.show(n)` plots (part of) the current layout, namely the outlines of the next `n` figures.

`lcm` is a trivial function, to be used as *the* interface for specifying absolute dimensions for the `widths` and `heights` arguments of `layout()`.

Value

`layout` returns the number of figures, N, see above.

Warnings

These functions are totally incompatible with the other mechanisms for arranging plots on a device: `par(mfrow)`, `par(mfcol)` and `split.screen`.

Author(s)

Paul R. Murrell

References

Murrell, P. R. (1999) Layouts: A mechanism for arranging plots on a page. *Journal of Computational and Graphical Statistics*, **8**, 121-134.

Chapter 5 of Paul Murrell's Ph.D. thesis.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

`par` with arguments `mfrow`, `mfcol`, or `mfg`.

Examples

```
def.par <- par(no.readonly = TRUE) # save default, for resetting...

## divide the device into two rows and two columns
## allocate figure 1 all of row 1
## allocate figure 2 the intersection of column 2 and row 2
layout(matrix(c(1,1,0,2), 2, 2, byrow = TRUE))
## show the regions that have been allocated to each plot
layout.show(2)

## divide device into two rows and two columns
## allocate figure 1 and figure 2 as above
## respect relations between widths and heights
nf <- layout(matrix(c(1,1,0,2), 2, 2, byrow=TRUE), respect=TRUE)
layout.show(nf)

## create single figure which is 5cm square
nf <- layout(matrix(1), widths=lcm(5), heights=lcm(5))
layout.show(nf)

##-- Create a scatterplot with marginal histograms -----

x <- pmin(3, pmax(-3, rnorm(50)))
y <- pmin(3, pmax(-3, rnorm(50)))
xhist <- hist(x, breaks=seq(-3,3,0.5), plot=FALSE)
yhist <- hist(y, breaks=seq(-3,3,0.5), plot=FALSE)
top <- max(c(xhist$counts, yhist$counts))
xrange <- c(-3,3)
yrange <- c(-3,3)
nf <- layout(matrix(c(2,0,1,3),2,2,byrow=TRUE), c(3,1), c(1,3), TRUE)
layout.show(nf)

par(mar=c(3,3,1,1))
plot(x, y, xlim=xrange, ylim=yrange, xlab="", ylab="")
par(mar=c(0,3,1,1))
barplot(xhist$counts, axes=FALSE, ylim=c(0, top), space=0)
par(mar=c(3,0,1,1))
barplot(yhist$counts, axes=FALSE, xlim=c(0, top), space=0, horiz=TRUE)

par(def.par)#- reset to default
```

`legend` *Add Legends to Plots*

Description

This function can be used to add legends to plots. Note that a call to the function `locator(1)` can be used in place of the `x` and `y` arguments.

Usage

```
legend(x, y = NULL, legend, fill = NULL, col = par("col"),
       lty, lwd, pch,
       angle = 45, density = NULL, bty = "o", bg = par("bg"),
```

```
box.lwd = par("lwd"), box.lty = par("lty"),
pt.bg = NA, cex = 1, pt.cex = cex, pt.lwd = lwd,
xjust = 0, yjust = 1, x.intersp = 1, y.intersp = 1,
adj = c(0, 0.5), text.width = NULL, text.col = par("col"),
merge = do.lines && has.pch, trace = FALSE,
plot = TRUE, ncol = 1, horiz = FALSE, title = NULL,
inset = 0)
```

Arguments

`x, y`	the x and y co-ordinates to be used to position the legend. They can be specified by keyword or in any way which is accepted by `xy.coords`: See Details.
`legend`	a character vector or language object (a name, call or `expression` vector) of length ≥ 1 to appear in the legend.
`fill`	if specified, this argument will cause boxes filled with the specified colors (or shaded in the specified colors) to appear beside the legend text.
`col`	the color of points or lines appearing in the legend.
`lty, lwd`	the line types and widths for lines appearing in the legend. One of these two *must* be specified for line drawing.
`pch`	the plotting symbols appearing in the legend, either as vector of 1-character strings, or one (multi character) string. *Must* be specified for symbol drawing.
`angle`	angle of shading lines.
`density`	the density of shading lines, if numeric and positive. If `NULL` or negative or `NA` color filling is assumed.
`bty`	the type of box to be drawn around the legend. The allowed values are `"o"` (the default) and `"n"`.
`bg`	the background color for the legend box. (Note that this is only used if `bty != "n"`.)
`box.lty, box.lwd`	the line type and width for the legend box.
`pt.bg`	the background color for the `points`, corresponding to its argument `bg`.
`cex`	character expansion factor **relative** to current `par("cex")`.
`pt.cex`	expansion factor(s) for the points.
`pt.lwd`	line width for the points, defaults to the one for lines, or if that is not set, to `par("lwd")`.
`xjust`	how the legend is to be justified relative to the legend x location. A value of 0 means left justified, 0.5 means centered and 1 means right justified.
`yjust`	the same as `xjust` for the legend y location.
`x.intersp`	character interspacing factor for horizontal (x) spacing.
`y.intersp`	the same for vertical (y) line distances.
`adj`	numeric of length 1 or 2; the string adjustment for legend text. Useful for y-adjustment when `labels` are plotmath expressions.
`text.width`	the width of the legend text in x (`"user"`) coordinates. (Should be positive even for a reversed x axis.) Defaults to the proper value computed by `strwidth(legend)`.
`text.col`	the color used for the legend text.

`merge`	logical; if `TRUE`, "merge" points and lines but not filled boxes. Defaults to `TRUE` if there are points and lines.
`trace`	logical; if `TRUE`, shows how `legend` does all its magical computations.
`plot`	logical. If `FALSE`, nothing is plotted but the sizes are returned.
`ncol`	the number of columns in which to set the legend items (default is 1, a vertical legend).
`horiz`	logical; if `TRUE`, set the legend horizontally rather than vertically (specifying `horiz` overrides the `ncol` specification).
`title`	a text value giving a title to be placed at the top of the legend.
`inset`	inset distance(s) from the margins as a fraction of the plot region when legend is placed by keyword.

Details

Arguments `x, y, legend` are interpreted in a non-standard way to allow the coordinates to be specified *via* one or two arguments. If `legend` is missing and `y` is not numeric, it is assumed that the second argument is intended to be `legend` and that the first argument specifies the coordinates.

The coordinates can be specified in any way which is accepted by `xy.coords`. If this gives the coordinates of one point, it is used as the top-left coordinate of the rectangle containing the legend. If it gives the coordinates of two points, these specify opposite corners of the rectangle (either pair of corners, in any order).

The location may also be specified by setting `x` to a single keyword from the list `"bottomright"`, `"bottom"`, `"bottomleft"`, `"left"`, `"topleft"`, `"top"`, `"topright"`, `"right"` and `"center"`. This places the legend on the inside of the plot frame at the given location. Partial argument matching is used. The optional `inset` argument specifies how far the legend is inset from the plot margins. If a single value is given, it is used for both margins; if two values are given, the first is used for `x`- distance, the second for `y`-distance.

"Attribute" arguments such as `col`, `pch`, `lty`, etc, are recycled if necessary. `merge` is not.

Points are drawn *after* lines in order that they can cover the line with their background color `pt.bg`, if applicable.

See the examples for how to right-justify labels.

Value

A list with list components

`rect`	a list with components
	`w, h` positive numbers giving **w**idth and **h**eight of the legend's box.
	`left, top` x and y coordinates of upper left corner of the box.
`text`	a list with components
	`x, y` numeric vectors of length `length(legend)`, giving the x and y coordinates of the legend's text(s).

returned invisibly.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

`plot`, `barplot` which uses `legend()`, and `text` for more examples of math expressions.

Examples

```
## Run the example in '?matplot' or the following:
leg.txt <- c("Setosa     Petals", "Setosa     Sepals",
             "Versicolor Petals", "Versicolor Sepals")
y.leg <- c(4.5, 3, 2.1, 1.4, .7)
cexv  <- c(1.2, 1, 4/5, 2/3, 1/2)
matplot(c(1,8), c(0,4.5), type = "n", xlab = "Length", ylab = "Width",
        main = "Petal and Sepal Dimensions in Iris Blossoms")
for (i in seq(cexv)) {
  text  (1, y.leg[i]-.1, paste("cex=",formatC(cexv[i])), cex=.8, adj = 0)
  legend(3, y.leg[i], leg.txt, pch = "sSvV", col = c(1, 3), cex = cexv[i])
}

## 'merge = TRUE' for merging lines & points:
x <- seq(-pi, pi, len = 65)
plot(x, sin(x), type = "l", ylim = c(-1.2, 1.8), col = 3, lty = 2)
points(x, cos(x), pch = 3, col = 4)
lines(x, tan(x), type = "b", lty = 1, pch = 4, col = 6)
title("legend(..., lty = c(2, -1, 1), pch = c(-1,3,4), merge = TRUE)",
      cex.main = 1.1)
legend(-1, 1.9, c("sin", "cos", "tan"), col = c(3,4,6),
       text.col = "green4", lty = c(2, -1, 1), pch = c(-1, 3, 4),
       merge = TRUE, bg = 'gray90')

## right-justifying a set of labels: thanks to Uwe Ligges
x <- 1:5; y1 <- 1/x; y2 <- 2/x
plot(rep(x, 2), c(y1, y2), type="n", xlab="x", ylab="y")
lines(x, y1); lines(x, y2, lty=2)
temp <- legend("topright", legend = c(" ", " "),
               text.width = strwidth("1,000,000"),
               lty = 1:2, xjust = 1, yjust = 1,
               title = "Line Types")
text(temp$rect$left + temp$rect$w, temp$text$y,
     c("1,000", "1,000,000"), pos=2)

##--- log scaled Examples ------------------------------
leg.txt <- c("a one", "a two")

par(mfrow = c(2,2))
for(ll in c("","x","y","xy")) {
  plot(2:10, log=ll, main=paste("log = '",ll,"'", sep=""))
  abline(1,1)
  lines(2:3,3:4, col=2) #
  points(2,2, col=3)    #
  rect(2,3,3,2, col=4)
  text(c(3,3),2:3, c("rect(2,3,3,2, col=4)",
                     "text(c(3,3),2:3,\"c(rect(...)\")"), adj = c(0,.3))
  legend(list(x=2,y=8), legend = leg.txt, col=2:3, pch=1:2,
         lty=1, merge=TRUE)#, trace=TRUE)
}
par(mfrow=c(1,1))
```

```
##-- Math expressions:  ------------------------------
x <- seq(-pi, pi, len = 65)
plot(x, sin(x), type="l", col = 2, xlab = expression(phi),
     ylab = expression(f(phi)))
abline(h=-1:1, v=pi/2*(-6:6), col="gray90")
lines(x, cos(x), col = 3, lty = 2)
ex.cs1 <- expression(plain(sin) * phi,  paste("cos", phi))# 2 ways
utils::str(legend(-3, .9, ex.cs1, lty=1:2, plot=FALSE,
           adj = c(0, .6)))# adj y !
legend(-3, .9, ex.cs1, lty=1:2, col=2:3,    adj = c(0, .6))

x <- rexp(100, rate = .5)
hist(x, main = "Mean and Median of a Skewed Distribution")
abline(v = mean(x),   col=2, lty=2, lwd=2)
abline(v = median(x), col=3, lty=3, lwd=2)
ex12 <- expression(bar(x) == sum(over(x[i], n), i==1, n),
                   hat(x) == median(x[i], i==1,n))
utils::str(legend(4.1, 30, ex12, col = 2:3, lty=2:3, lwd=2))

## 'Filled' boxes -- for more, see example(plotfactor)
op <- par(bg="white") # to get an opaque box for the legend
plot(cut(weight, 3) ~ group, data = PlantGrowth, col = NULL,
     density = 16*(1:3))
par(op)

## Using 'ncol' :
x <- 0:64/64
matplot(x, outer(x, 1:7, function(x, k) sin(k * pi * x)),
        type = "o", col = 1:7, ylim = c(-1, 1.5), pch = "*")
op <- par(bg="antiquewhite1")
legend(0, 1.5, paste("sin(", 1:7, "pi * x)"), col=1:7, lty=1:7, pch = "*",
       ncol = 4, cex = 0.8)
legend(.8,1.2, paste("sin(", 1:7, "pi * x)"), col=1:7, lty=1:7,
       pch = "*", cex = 0.8)
legend(0, -.1, paste("sin(", 1:4, "pi * x)"), col=1:4, lty=1:4,
       ncol = 2, cex = 0.8)
legend(0, -.4, paste("sin(", 5:7, "pi * x)"), col=4:6,  pch=24,
       ncol = 2, cex = 1.5, lwd = 2, pt.bg = "pink", pt.cex = 1:3)
par(op)

## point covering line :
y <- sin(3*pi*x)
plot(x, y, type="l", col="blue", main = "points with bg & legend(*, pt.bg)")
points(x, y, pch=21, bg="white")
legend(.4,1, "sin(c x)", pch=21, pt.bg="white", lty=1, col = "blue")

## legends with titles at different locations
plot(x, y, type='n')
legend("bottomright", "(x,y)", pch=1, title="bottomright")
legend("bottom", "(x,y)", pch=1, title="bottom")
legend("bottomleft", "(x,y)", pch=1, title="bottomleft")
legend("left", "(x,y)", pch=1, title="left")
legend("topleft", "(x,y)", pch=1, title="topleft, inset = .05", inset = .05)
legend("top", "(x,y)", pch=1, title="top")
legend("topright", "(x,y)", pch=1, title="topright, inset = .02", inset = .02)
legend("right", "(x,y)", pch=1, title="right")
legend("center", "(x,y)", pch=1, title="center")
```

`lines` *Add Connected Line Segments to a Plot*

Description

A generic function taking coordinates given in various ways and joining the corresponding points with line segments.

Usage

```
lines(x, ...)

## Default S3 method:
lines(x, y = NULL, type = "l", ...)
```

Arguments

`x, y`	coordinate vectors of points to join.
`type`	character indicating the type of plotting; actually any of the `type`s as in `plot.default`.
`...`	Further graphical parameters (see `par`) may also be supplied as arguments, particularly, line type, `lty`, line width, `lwd`, color, `col` and for `type = "b"`, `pch`. Also the line characteristics `lend`, `ljoin` and `lmitre`.

Details

The coordinates can be passed to `lines` in a plotting structure (a list with `x` and `y` components), a time series, etc. See `xy.coords`.

The coordinates can contain `NA` values. If a point contains `NA` in either its `x` or `y` value, it is omitted from the plot, and lines are not drawn to or from such points. Thus missing values can be used to achieve breaks in lines.

For `type = "h"`, `col` can be a vector and will be recycled as needed.

`lwd` can be a vector: its first element will apply to lines but the whole vector to symbols (recycled as necessary).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`points`, particularly for `type %in% c("p","b","o")`, `plot`, and the workhorse function `plot.xy`.

`abline` for drawing (single) straight lines.

`par` for how to specify colors.

Examples

```
# draw a smooth line through a scatter plot
plot(cars, main="Stopping Distance versus Speed")
lines(lowess(cars))
```

`locator` *Graphical Input*

Description

Reads the position of the graphics cursor when the (first) mouse button is pressed.

Usage

```
locator(n = 512, type = "n", ...)
```

Arguments

`n`	the maximum number of points to locate. Valid values start at 1.
`type`	One of `"n"`, `"p"`, `"l"` or `"o"`. If `"p"` or `"o"` the points are plotted; if `"l"` or `"o"` they are joined by lines.
`...`	additional graphics parameters used if `type != "n"` for plotting the locations.

Details

`locator` is only supported on screen devices such as `X11`, `windows` and `quartz`. On other devices the call will do nothing.

Unless the process is terminated prematurely by the user (see below) at most `n` positions are determined.

For the usual `X11` device the identification process is terminated by pressing any mouse button other than the first. For the `quartz` device the process is terminated by pressing the `ESC` key.

The current graphics parameters apply just as if `plot.default` has been called with the same value of `type`. The plotting of the points and lines is subject to clipping, but locations outside the current clipping rectangle will be returned.

On most devices which support `locator`, successful selection of a point is indicated by a bell sound unless `options(locatorBell=FALSE)` has been set.

If the window is resized or hidden and then exposed before the input process has terminated, any lines or points drawn by `locator` will disappear. These will reappear once the input process has terminated and the window is resized or hidden and exposed again. This is because the points and lines drawn by `locator` are not recorded in the device's display list until the input process has terminated.

Value

A list containing `x` and `y` components which are the coordinates of the identified points in the user coordinate system, i.e., the one specified by `par("usr")`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

identify

---

matplot *Plot Columns of Matrices*

Description

Plot the columns of one matrix against the columns of another.

Usage

```
matplot(x, y, type = "p", lty = 1:5, lwd = 1, pch = NULL,
        col = 1:6, cex = NULL, bg = NA,
        xlab = NULL, ylab = NULL, xlim = NULL, ylim = NULL,
        ..., add = FALSE, verbose = getOption("verbose"))

matpoints(x, y, type = "p", lty = 1:5, lwd = 1, pch = NULL,
          col = 1:6, ...)

matlines (x, y, type = "l", lty = 1:5, lwd = 1, pch = NULL,
          col = 1:6, ...)
```

Arguments

x,y	vectors or matrices of data for plotting. The number of rows should match. If one of them are missing, the other is taken as y and an x vector of 1:n is used. Missing values (NAs) are allowed.
type	character string (length 1 vector) or vector of 1-character strings indicating the type of plot for each column of y, see plot for all possible types. The first character of type defines the first plot, the second character the second, etc. Characters in type are cycled through; e.g., "pl" alternately plots points and lines.
lty,lwd	vector of line types and widths. The first element is for the first column, the second element for the second column, etc., even if lines are not plotted for all columns. Line types will be used cyclically until all plots are drawn.
pch	character string or vector of 1-characters or integers for plotting characters, see points. The first character is the plotting-character for the first plot, the second for the second, etc. The default is the digits (1 through 9, 0) then the letters.
col	vector of colors. Colors are used cyclically.
cex	vector of character expansion sizes, used cyclically. This works as a multiple of par("cex"). NULL is equivalent to 1.0.
bg	vector of ("fill") colors for the open plot symbols given by pch=21:25 as in points. The default NA corresponds to the one of the underlying function plot.xy.

xlab, ylab	titles for x and y axes, as in plot.
xlim, ylim	ranges of x and y axes, as in plot.
...	Graphical parameters (see par) and any further arguments of plot, typically plot.default, may also be supplied as arguments to this function. Hence, the high-level graphics control arguments described under par and the arguments to title may be supplied to this function.
add	logical. If TRUE, plots are added to current one, using points and lines.
verbose	logical. If TRUE, write one line of what is done.

Details

Points involving missing values are not plotted.

The first column of x is plotted against the first column of y, the second column of x against the second column of y, etc. If one matrix has fewer columns, plotting will cycle back through the columns again. (In particular, either x or y may be a vector, against which all columns of the other argument will be plotted.)

The first element of col, cex, lty, lwd is used to plot the axes as well as the first line.

Because plotting symbols are drawn with lines and because these functions may be changing the line style, you should probably specify lty=1 when using plotting symbols.

Side Effects

Function matplot generates a new plot; matpoints and matlines add to the current one.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

plot, points, lines, matrix, par.

Examples

```
matplot((-4:5)^2, main = "Quadratic") # almost identical to plot(*)
sines <- outer(1:20, 1:4, function(x, y) sin(x / 20 * pi * y))
matplot(sines, pch = 1:4, type = "o", col = rainbow(ncol(sines)))
matplot(sines, type = "b", pch = 21:23, col = 2:5, bg = 2:5,
        main = "matplot(...., pch = 21:23, bg = 2:5)")

x <- 0:50/50
matplot(x, outer(x, 1:8, function(x, k) sin(k*pi * x)),
        ylim = c(-2,2), type = "plobcsSh",
        main= "matplot(,type = \"plobcsSh\" )")
## pch & type =  vector of 1-chars :
matplot(x, outer(x, 1:4, function(x, k) sin(k*pi * x)),
        pch = letters[1:4], type = c("b","p","o"))

table(iris$Species) # is data.frame with 'Species' factor
iS <- iris$Species == "setosa"
iV <- iris$Species == "versicolor"
op <- par(bg = "bisque")
```

```
matplot(c(1, 8), c(0, 4.5), type= "n", xlab = "Length", ylab = "Width",
        main = "Petal and Sepal Dimensions in Iris Blossoms")
matpoints(iris[iS,c(1,3)], iris[iS,c(2,4)], pch = "sS", col = c(2,4))
matpoints(iris[iV,c(1,3)], iris[iV,c(2,4)], pch = "vV", col = c(2,4))
legend(1, 4, c("    Setosa Petals", "    Setosa Sepals",
               "Versicolor Petals", "Versicolor Sepals"),
       pch = "sSvV", col = rep(c(2,4), 2))

nam.var <- colnames(iris)[-5]
nam.spec <- as.character(iris[1+50*0:2, "Species"])
iris.S <- array(NA, dim = c(50,4,3), dimnames = list(NULL, nam.var, nam.spec))
for(i in 1:3) iris.S[,,i] <- data.matrix(iris[1:50+50*(i-1), -5])

matplot(iris.S[,"Petal.Length",], iris.S[,"Petal.Width",], pch="SCV",
        col = rainbow(3, start = .8, end = .1),
        sub = paste(c("S", "C", "V"), dimnames(iris.S)[[3]],
                    sep = "=", collapse= ",  "),
        main = "Fisher's Iris Data")
par(op)
```

`mosaicplot` *Mosaic Plots*

Description

Plots a mosaic on the current graphics device.

Usage

```
mosaicplot(x, ...)

## Default S3 method:
mosaicplot(x, main = deparse(substitute(x)),
           sub = NULL, xlab = NULL, ylab = NULL,
           sort = NULL, off = NULL, dir = NULL,
           color = NULL, shade = FALSE, margin = NULL,
           cex.axis = 0.66, las = par("las"),
           type = c("pearson", "deviance", "FT"), ...)

## S3 method for class 'formula':
mosaicplot(formula, data = NULL, ...,
           main = deparse(substitute(data)), subset,
           na.action = stats::na.omit)
```

Arguments

`x`	a contingency table in array form, with optional category labels specified in the `dimnames(x)` attribute. The table is best created by the `table()` command.
`main`	character string for the mosaic title.
`sub`	character string for the mosaic sub-title (at bottom).
`xlab,ylab`	x- and y-axis labels used for the plot; by default, the first and second element of `names(dimnames(X))` (i.e., the name of the first and second variable in `X`).

sort	vector ordering of the variables, containing a permutation of the integers 1:length(dim(x)) (the default).
off	vector of offsets to determine percentage spacing at each level of the mosaic (appropriate values are between 0 and 20, and the default is 20 times the number of splits for 2-dimensional tables, and 10 otherwise. Rescaled to maximally 50, and recycled if necessary.
dir	vector of split directions ("v" for vertical and "h" for horizontal) for each level of the mosaic, one direction for each dimension of the contingency table. The default consists of alternating directions, beginning with a vertical split.
color	logical or (recycling) vector of colors for color shading, used only when shade is FALSE, or NULL (default). By default, grey boxes are drawn. color=TRUE uses a gamma-corrected grey palette. color=FALSE gives empty boxes with no shading.
shade	a logical indicating whether to produce extended mosaic plots, or a numeric vector of at most 5 distinct positive numbers giving the absolute values of the cut points for the residuals. By default, shade is FALSE, and simple mosaics are created. Using shade = TRUE cuts absolute values at 2 and 4.
margin	a list of vectors with the marginal totals to be fit in the log-linear model. By default, an independence model is fitted. See loglin for further information.
cex.axis	The magnification to be used for axis annotation, as a multiple of par("cex").
las	numeric; the style of axis labels, see par.
type	a character string indicating the type of residual to be represented. Must be one of "pearson" (giving components of Pearson's χ^2), "deviance" (giving components of the likelihood ratio χ^2), or "FT" for the Freeman-Tukey residuals. The value of this argument can be abbreviated.
formula	a formula, such as y ~ x.
data	a data frame (or list), or a contingency table from which the variables in formula should be taken.
...	further arguments to be passed to or from methods.
subset	an optional vector specifying a subset of observations in the data frame to be used for plotting.
na.action	a function which indicates what should happen when the data contains variables to be cross-tabulated, and these variables contain NAs. The default is to omit cases which have an NA in any variable. Since the tabulation will omit all cases containing missing values, this will only be useful if the na.action function replaces missing values.

Details

This is a generic function. It currently has a default method (mosaicplot.default) and a formula interface (mosaicplot.formula).

Extended mosaic displays visualize standardized residuals of a loglinear model for the table by color and outline of the mosaic's tiles. (Standardized residuals are often referred to a standard normal distribution.) Negative residuals are drawn in shaded of red and with broken outlines; positive ones are drawn in blue with solid outlines.

For the formula method, if data is an object inheriting from classes "table" or "ftable", or an array with more than 2 dimensions, it is taken as a contingency table, and hence all entries should

be nonnegative. In this case, the left-hand side of `formula` should be empty, and the variables on the right-hand side should be taken from the names of the dimnames attribute of the contingency table. A marginal table of these variables is computed, and a mosaic of this table is produced.

Otherwise, `data` should be a data frame or matrix, list or environment containing the variables to be cross-tabulated. In this case, after possibly selecting a subset of the data as specified by the `subset` argument, a contingency table is computed from the variables given in `formula`, and a mosaic is produced from this.

See Emerson (1998) for more information and a case study with television viewer data from Nielsen Media Research.

Missing values are not supported except via an `na.action` function when `data` contains variables to be cross-tabulated.

A more flexible and extensible implementation of mosaic plots written in the grid graphics system is provided in the function `mosaic` in the contributed package **vcd** (Meyer, Zeileis and Hornik, 2005).

Author(s)

S-PLUS original by John Emerson ⟨emerson@stat.yale.edu⟩. Originally modified and enhanced for R by Kurt Hornik.

References

Hartigan, J.A., and Kleiner, B. (1984) A mosaic of television ratings. *The American Statistician*, **38**, 32–35.

Emerson, J. W. (1998) Mosaic displays in S-PLUS: A general implementation and a case study. *Statistical Computing and Graphics Newsletter (ASA)*, **9**, 1, 17–23.

Friendly, M. (1994) Mosaic displays for multi-way contingency tables. *Journal of the American Statistical Association*, **89**, 190–200.

Meyer, D., Zeileis, A., and Hornik, K. (2005) The strucplot framework: Visualizing multi-way contingency tables with vcd. *Report 22*, Department of Statistics and Mathematics, Wirtschaftsuniversität Wien, Research Report Series. http://epub.wu-wien.ac.at/dyn/openURL?id=oai:epub.wu-wien.ac.at:epub-wu-01_8a1

The home page of Michael Friendly (http://www.math.yorku.ca/SCS/friendly.html) provides information on various aspects of graphical methods for analyzing categorical data, including mosaic plots.

See Also

`assocplot`, `loglin`.

Examples

```
mosaicplot(Titanic, main = "Survival on the Titanic", color = TRUE)
## Formula interface for tabulated data:
mosaicplot(~ Sex + Age + Survived, data = Titanic, color = TRUE)

mosaicplot(HairEyeColor, shade = TRUE)
## Independence model of hair and eye color and sex.  Indicates that
## there are more blue eyed blonde females than expected in the case
## of independence and too few brown eyed blonde females.
## The corresponding model is:
fm <- loglin(HairEyeColor, list(1, 2, 3))
```

```
pchisq(fm$pearson, fm$df, lower.tail = FALSE)

mosaicplot(HairEyeColor, shade = TRUE, margin = list(1:2, 3))
## Model of joint independence of sex from hair and eye color.  Males
## are underrepresented among people with brown hair and eyes, and are
## overrepresented among people with brown hair and blue eyes.
## The corresponding model is:
fm <- loglin(HairEyeColor, list(1:2, 3))
pchisq(fm$pearson, fm$df, lower.tail = FALSE)

## Formula interface for raw data: visualize crosstabulation of numbers
## of gears and carburettors in Motor Trend car data.
mosaicplot(~ gear + carb, data = mtcars, color = TRUE, las = 1)
# color recycling
mosaicplot(~ gear + carb, data = mtcars, color = 2:3, las = 1)
```

`mtext` *Write Text into the Margins of a Plot*

Description

Text is written in one of the four margins of the current figure region or one of the outer margins of the device region.

Usage

```
mtext(text, side = 3, line = 0, outer = FALSE, at = NA,
      adj = NA, padj = NA, cex = NA, col = NA, font = NA, ...)
```

Arguments

`text`	one or more character strings or language objects (names, calls or expressions).
`side`	on which side of the plot (1=bottom, 2=left, 3=top, 4=right).
`line`	on which MARgin line, starting at 0 counting outwards.
`outer`	use outer margins if available.
`at`	give location in user-coordinates. If `length(at)==0` (the default), the location will be determined by `adj`.
`adj`	adjustment for each string in reading direction. For strings parallel to the axes, `adj = 0` means left or bottom alignment, and `adj = 1` means right or top alignment. If `adj` is not a finite value (the default), the value of `par("las")` determines the adjustment. For strings plotted parallel to the axis the default is to centre the string.
`padj`	adjustment for each string perpendicular to the reading direction (which is controlled by `adj`). For strings parallel to the axes, `padj = 0` means right or top alignment, and `padj = 1` means left or bottom alignment. If `padj` is not a finite value (the default), the value of `par("las")` determines the adjustment. For strings plotted perpendicular to the axis the default is to centre the string.

`cex`	character expansion factor. `NULL` and `NA` are equivalent to `1.0`. This is an absolute measure, not scaled by `par("cex")` or by setting `par("mfrow")` or `par("mfcol")`. Can be a vector.
`col`	color to use. Can be a vector. `NA` values (the default) mean use `par("col")`.
`font`	font for text. Can be a vector. `NA` values (the default) mean use `par("font")`.
`...`	Further graphical parameters (see `par`), including `family`, `las` and `xpd`. (This defaults to the figure region unless `outer = TRUE`, otherwise the device region. It can only be increased.)

Details

The "user coordinates" in the outer margins always range from zero to one, and are not affected by the user coordinates in the figure region(s) — R is differing here from other implementations of S.

All of the named arguments can be vectors, and recycling will take place to plot as many strings as the longest of the vector arguments.

Note that a vector `adj` has a different meaning from `text`. `adj = 0.5` will centre the string, but for `outer=TRUE` on the device region rather than the plot region.

Parameter `las` will determine the orientation of the string(s). For strings plotted perpendicular to the axis the default justification is to place the end of the string nearest the axis on the specified line. (Note that this differs from S, which uses `srt` if `at` is supplied and `las` if it is not.)

Note that if the text is to be plotted perpendicular to the axis, `adj` determines the justification of the string *and* the position along the axis unless `at` is specified.

Side Effects

The given text is written onto the current plot.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`title`, `text`, `plot`, `par`; `plotmath` for details on mathematical annotation.

Examples

```
plot(1:10, (-4:5)^2, main="Parabola Points", xlab="xlab")
mtext("10 of them")
for(s in 1:4)
    mtext(paste("mtext(..., line= -1, {side, col, font} = ",s,
          ", cex = ", (1+s)/2, ")"), line = -1,
          side=s, col=s, font=s, cex= (1+s)/2)
mtext("mtext(..., line= -2)", line = -2)
mtext("mtext(..., line= -2, adj = 0)", line = -2, adj =0)
##--- log axis :
plot(1:10, exp(1:10), log='y', main="log='y'", xlab="xlab")
for(s in 1:4) mtext(paste("mtext(...,side=",s,")"), side=s)
```

`pairs` *Scatterplot Matrices*

Description

A matrix of scatterplots is produced.

Usage

```
pairs(x, ...)

## S3 method for class 'formula':
pairs(formula, data = NULL, ..., subset,
      na.action = stats::na.pass)

## Default S3 method:
pairs(x, labels, panel = points, ...,
      lower.panel = panel, upper.panel = panel,
      diag.panel = NULL, text.panel = textPanel,
      label.pos = 0.5 + has.diag/3,
      cex.labels = NULL, font.labels = 1,
      row1attop = TRUE, gap = 1)
```

Arguments

`x`	the coordinates of points given as numeric columns of a matrix or dataframe. Logical and factor columns are converted to numeric in the same way that `data.matrix` does.
`formula`	a formula, such as `~ x + y + z`. Each term will give a separate variable in the pairs plot, so terms should be numeric vectors. (A response will be interpreted as another variable, but not treated specially, so it is confusing to use one.)
`data`	a data.frame (or list) from which the variables in `formula` should be taken.
`subset`	an optional vector specifying a subset of observations to be used for plotting.
`na.action`	a function which indicates what should happen when the data contain `NA`s. The default is to pass missing values on to the panel functions, but `na.action = na.omit` will cause cases with missing values in any of the variables to be omitted entirely.
`labels`	the names of the variables.
`panel`	`function(x,y,...)` which is used to plot the contents of each panel of the display.
`...`	arguments to be passed to or from methods. Also, graphical parameters can be given as can arguments to `plot` such as `main`. `par("oma")` will be set appropriately unless specified.
`lower.panel, upper.panel`	separate panel functions to be used below and above the diagonal respectively.
`diag.panel`	optional `function(x, ...)` to be applied on the diagonals.

`text.panel` optional `function(x, y, labels, cex, font, ...)` to be applied on the diagonals.

`label.pos` `y` position of labels in the text panel.

`cex.labels, font.labels` graphics parameters for the text panel.

`row1attop` logical. Should the layout be matrix-like with row 1 at the top, or graph-like with row 1 at the bottom?

`gap` Distance between subplots, in margin lines.

Details

The ijth scatterplot contains `x[,i]` plotted against `x[,j]`. The "scatterplot" can be customised by setting panel functions to appear as something completely different. The off-diagonal panel functions are passed the appropriate columns of `x` as `x` and `y`: the diagonal panel function (if any) is passed a single column, and the `text.panel` function is passed a single `(x, y)` location and the column name.

The graphical parameters `pch` and `col` can be used to specify a vector of plotting symbols and colors to be used in the plots.

The graphical parameter `oma` will be set by `pairs.default` unless supplied as an argument.

A panel function should not attempt to start a new plot, but just plot within a given coordinate system: thus `plot` and `boxplot` are not panel functions.

By default, missing values are passed to the panel functions and will often be ignored within a panel. However, for the formula method and `na.action = na.omit`, all cases which contain a missing values for any of the variables are omitted completely (including when the scales are selected).

Author(s)

Enhancements for R 1.0.0 contributed by Dr. Jens Oehlschlaegel-Akiyoshi and R-core members.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
pairs(iris[1:4], main = "Anderson's Iris Data -- 3 species",
      pch = 21, bg = c("red", "green3", "blue")[unclass(iris$Species)])

## formula method
pairs(~ Fertility + Education + Catholic, data = swiss,
      subset = Education < 20, main = "Swiss data, Education < 20")

pairs(USJudgeRatings)

## put histograms on the diagonal
panel.hist <- function(x, ...)
{
    usr <- par("usr"); on.exit(par(usr))
    par(usr = c(usr[1:2], 0, 1.5) )
    h <- hist(x, plot = FALSE)
```

```
    breaks <- h$breaks; nB <- length(breaks)
    y <- h$counts; y <- y/max(y)
    rect(breaks[-nB], 0, breaks[-1], y, col="cyan", ...)
}
pairs(USJudgeRatings[1:5], panel=panel.smooth,
      cex = 1.5, pch = 24, bg="light blue",
      diag.panel=panel.hist, cex.labels = 2, font.labels=2)

## put (absolute) correlations on the upper panels,
## with size proportional to the correlations.
panel.cor <- function(x, y, digits=2, prefix="", cex.cor)
{
    usr <- par("usr"); on.exit(par(usr))
    par(usr = c(0, 1, 0, 1))
    r <- abs(cor(x, y))
    txt <- format(c(r, 0.123456789), digits=digits)[1]
    txt <- paste(prefix, txt, sep="")
    if(missing(cex.cor)) cex <- 0.8/strwidth(txt)
    text(0.5, 0.5, txt, cex = cex * r)
}
pairs(USJudgeRatings, lower.panel=panel.smooth, upper.panel=panel.cor)
```

`panel.smooth` *Simple Panel Plot*

Description

An example of a simple useful `panel` function to be used as argument in e.g., `coplot` or `pairs`.

Usage

```
panel.smooth(x, y, col = par("col"), bg = NA, pch = par("pch"),
             cex = 1, col.smooth = "red", span = 2/3, iter = 3,
             ...)
```

Arguments

`x, y`	numeric vectors of the same length
`col, bg, pch, cex`	numeric or character codes for the color(s), point type and size of `points`; see also `par`.
`col.smooth`	color to be used by `lines` for drawing the smooths.
`span`	smoothing parameter `f` for `lowess`, see there.
`iter`	number of robustness iterations for `lowess`.
`...`	further arguments to `lines`.

See Also

`coplot` and `pairs` where `panel.smooth` is typically used; `lowess` which does the smoothing.

Examples

```
pairs(swiss, panel = panel.smooth, pch = ".")# emphasize the smooths
pairs(swiss, panel = panel.smooth, lwd = 2, cex= 1.5, col="blue")# hmm...
```

`par` *Set or Query Graphical Parameters*

Description

`par` can be used to set or query graphical parameters. Parameters can be set by specifying them as arguments to `par` in `tag = value` form, or by passing them as a list of tagged values.

Usage

```
par(..., no.readonly = FALSE)

<highlevel plot> (..., <tag> = <value>)
```

Arguments

`...`	arguments in `tag = value` form, or a list of tagged values. The tags must come from the graphical parameters described below.
`no.readonly`	logical; if `TRUE` and there are no other arguments, only parameters are returned which can be set by a subsequent `par()` call.

Details

Parameters are queried by giving one or more character vectors to `par`.

`par()` (no arguments) or `par(no.readonly=TRUE)` is used to get *all* the graphical parameters (as a named list). Their names are currently taken from the unexported variable `.Pars`.

R.O. indicates ***read-only arguments***: These may only be used in queries and cannot be set. (`"cin"`, `"cra"`, `"csi"`, `"cxy"` and `"din"` are always read-only, and `"gamma"` is on most devices.)

There are several parameters can only be set by a call to `par()`:

- `"ask"`,
- `"fig"`, `"fin"`,
- `"lheight"`,
- `"mai"`, `"mar"`, `"mex"`, `"mfcol"`, `"mfrow"`, `"mfg"`,
- `"new"`,
- `"oma"`, `"omd"`, `"omi"`,
- `"pin"`, `"plt"`, `"ps"`, `"pty"`,
- `"usr"`,
- `"xlog"`, `"ylog"`

The remaining parameters can also be set as arguments (often via `...`) to high-level plot functions such as `plot.default`, `plot.window`, `points`, `lines`, `abline`, `axis`, `title`, `text`, `mtext`, `segments`, `symbols`, `arrows`, `polygon`, `rect`, `box`, `contour`, `filled.contour` and `image`. Such settings will be active during the execution of the function, only. However, see the comments on `bg` and `cex`, which may be taken as arguments to certain plot functions rather than as graphical parameters.

The meaning of 'character size' is not well-defined: this is set up for the device taking `pointsize` into account but often not the actual font family in use. It can be considered to be an estimate of the size of character `M` in a proportionally-spaced font.

Value

When parameters are set, their former values are returned in an invisible named list. Such a list can be passed as an argument to `par` to restore the parameter values. Use `par(no.readonly = TRUE)` for the full list of parameters that can be restored.

When just one parameter is queried, the value of that parameter is returned as (atomic) vector. When two or more parameters are queried, their values are returned in a list, with the list names giving the parameters.

Note the inconsistency: setting one parameter returns a list, but querying one parameter returns a vector.

Graphical Parameters

`adj` The value of `adj` determines the way in which text strings are justified. A value of `0` produces left-justified text, `0.5` centered text and `1` right-justified text. (Any value in $[0, 1]$ is allowed, and on most devices values outside that interval will also work.) Note that the `adj` argument of `text` also allows `adj = c(x, y)` for different adjustment in x- and y- direction.

`ann` If set to `FALSE`, high-level plotting functions calling `plot.default` do not annotate the plots they produce with axis titles and overall titles. The default is to do annotation.

`ask` logical. If `TRUE` (and the R session is interactive) the user is asked for input, before a new figure is drawn. As this applies to the device, it also affects output by packages **grid** and **lattice**. It can be set even on non-screen devices.

`bg` The color to be used for the background of plots. When called from `par()` it also sets `new=FALSE`. A description of how colors are specified is given below. Note that some graphics functions such as `plot.default` and `points` have an *argument* of this name with a different meaning.

`bty` A character string which determined the type of `box` which is drawn about plots. If `bty` is one of `"o"`, `"l"`, `"7"`, `"c"`, `"u"`, or `"]"` the resulting box resembles the corresponding upper case letter. A value of `"n"` suppresses the box.

`cex` A numerical value giving the amount by which plotting text and symbols should be scaled relative to the default. Note that some graphics functions such as `plot.default` have an *argument* of this name which multiplies this graphical parameter.

`cex.axis` The magnification to be used for axis annotation relative to the current setting of `cex`. (Some functions such as `points` accept a vector of values which are recycled. Other uses will take just the first value if a vector of length greater than one is supplied.)

`cex.lab` The magnification to be used for x and y labels relative to the current setting of `cex`.

`cex.main` The magnification to be used for main titles relative to the current setting of `cex`.

`cex.sub` The magnification to be used for sub-titles relative to the current setting of `cex`.

`cin` ***R.O.***; character size `(width, height)` in inches.

col A specification for the default plotting color. A description of how colors are specified is given below. (Some functions such as `lines` accept a vector of values which are recycled. Other uses will take just the first value if a vector of length greater than one is supplied.)

col.axis The color to be used for axis annotation.

col.lab The color to be used for x and y labels.

col.main The color to be used for plot main titles.

col.sub The color to be used for plot sub-titles.

cra ***R.O.***; size of default character `(width, height)` in "rasters" (pixels).

crt A numerical value specifying (in degrees) how single characters should be rotated. It is unwise to expect values other than multiples of 90 to work. Compare with `srt` which does string rotation.

csi ***R.O.***; height of (default sized) characters in inches.

cxy ***R.O.***; size of default character `(width, height)` in user coordinate units. `par("cxy")` is `par("cin")/par("pin")` scaled to user coordinates. Note that `c(strwidth(ch), strwidth(ch))` for a given string `ch` is usually much more precise.

din ***R.O.***; the device dimensions, `(width,height)`, in inches.

err (*Unimplemented*; R is silent when points outside the plot region are *not* plotted.) The degree of error reporting desired.

family The name of a font family for drawing text. The maximum allowed length is 200 bytes. This name gets mapped by each graphics device to device-specific font descriptions. The default value is `""` which means that the default device fonts will be used. Standard values are `"serif"`, `"sans"`, `"mono"`, and `"symbol"` and the Hershey font families are also available. (Different devices may define others, and some devices will ignore this setting completely.) This can be specified inline for `text`.

fg The color to be used for the foreground of plots. This is the default color used for things like axes and boxes around plots. When called from `par()` this also sets parameter `col` to the same value.

A description of how colors are specified is given below.

fig A numerical vector of the form `c(x1, x2, y1, y2)` which gives the (NDC) coordinates of the figure region in the display region of the device. If you set this, unlike S, you start a new plot, so to add to an existing plot use `new=TRUE` as well.

fin The figure region dimensions, `(width,height)`, in inches. If you set this, unlike S, you start a new plot.

font An integer which specifies which font to use for text. If possible, device drivers arrange so that 1 corresponds to plain text, 2 to bold face, 3 to italic and 4 to bold italic. Also, font 5 is expected to be the symbol font, in Adobe symbol encoding. On some devices font families can be selected by `family` to choose different sets of 5 fonts.

font.axis The font to be used for axis annotation.

font.lab The font to be used for x and y labels.

font.main The font to be used for plot main titles.

font.sub The font to be used for plot sub-titles.

gamma the gamma correction, see argument `gamma` to `hsv`. This is only accepted if the current device has support for changing the gamma correction: currently only `windows` and `quartz` do. (`X11` has support for setting the gamma correction when opening the device, but not for changing it.)

lab A numerical vector of the form `c(x, y, len)` which modifies the default way that axes are annotated. The values of `x` and `y` give the (approximate) number of tickmarks on the x and y axes and `len` specifies the label length. The default is `c(5, 5, 7)`. Note that this only affects the way the parameters `xaxp` and `yaxp` are set when the user coordinate system is set up, and is not consulted when axes are drawn. `len` *is unimplemented* in R.

las numeric in {0,1,2,3}; the style of axis labels.

- **0:** always parallel to the axis [*default*],
- **1:** always horizontal,
- **2:** always perpendicular to the axis,
- **3:** always vertical.

Note that other string/character rotation (via argument `srt` to `par`) does *not* affect the axis labels.

lend The line end style. This can be specified as an integer or string:

- **0** and `"round"` mean rounded line caps [*default*];
- **1** and `"butt"` mean butt line caps;
- **2** and `"square"` mean square line caps.

lheight The line height multiplier. The height of a line of text (used to vertically space multi-line text) is found by multiplying the current font size both by the current character expansion and by the line height multiplier. Default value is 1.

ljoin The line join style. This can be specified as an integer or string:

- **0** and `"round"` mean rounded line joins [*default*];
- **1** and `"mitre"` mean mitred line joins;
- **2** and `"bevel"` mean bevelled line joins.

lmitre The line mitre limit. This controls when mitred line joins are automatically converted into bevelled line joins. The value must be larger than 1 and the default is 10. Not all devices will honour this setting.

lty The line type. Line types can either be specified as an integer (0=blank, 1=solid, 2=dashed, 3=dotted, 4=dotdash, 5=longdash, 6=twodash) or as one of the character strings `"blank"`, `"solid"`, `"dashed"`, `"dotted"`, `"dotdash"`, `"longdash"`, or `"twodash"`, where `"blank"` uses 'invisible lines' (i.e., doesn't draw them).

Alternatively, a string of up to 8 characters (from `c(1:9, "A":"F")`) may be given, giving the length of line segments which are alternatively drawn and skipped. See section 'Line Type Specification' below.

Some functions such as `lines` accept a vector of values which are recycled. Other uses will take just the first value if a vector of length greater than one is supplied.

lwd The line width, a *positive* number, defaulting to `1`. The interpretation is device-specific, and some devices do not implement line widths less than one. (Some functions such as `lines` accept a vector of values which are recycled. Other uses will take just the first value if a vector of length greater than one is supplied.)

mai A numerical vector of the form `c(bottom, left, top, right)` which gives the margin size specified in inches.

mar A numerical vector of the form `c(bottom, left, top, right)` which gives the number of lines of margin to be specified on the four sides of the plot. The default is `c(5, 4, 4, 2) + 0.1`.

mex `mex` is a character size expansion factor which is used to describe coordinates in the margins of plots. Note that this does not change the font size, rather specifies the size of font used to convert between `mar` and `mai`, and between `oma` and `omi`.

mfcol, mfrow A vector of the form `c(nr, nc)`. Subsequent figures will be drawn in an `nr`-by-`nc` array on the device by *columns* (`mfcol`), or *rows* (`mfrow`), respectively.

In a layout with exactly two rows and columns the base value of `"cex"` is reduced by a factor of 0.83: if there are three or more of either rows or columns, the reduction factor is 0.66.

Consider the alternatives, `layout` and `split.screen`.

mfg A numerical vector of the form `c(i, j)` where `i` and `j` indicate which figure in an array of figures is to be drawn next (if setting) or is being drawn (if enquiring). The array must already have been set by `mfcol` or `mfrow`.

For compatibility with S, the form `c(i, j, nr, nc)` is also accepted, when `nr` and `nc` should be the current number of rows and number of columns. Mismatches will be ignored, with a warning.

mgp The margin line (in `mex` units) for the axis title, axis labels and axis line. The default is `c(3, 1, 0)`.

mkh The height in inches of symbols to be drawn when the value of `pch` is an integer. *Completely ignored currently*.

new logical, defaulting to `FALSE`. If set to `TRUE`, the next high-level plotting command (actually `plot.new`) should *not clean* the frame before drawing "as if it was on a ***new*** device".

oma A vector of the form `c(bottom, left, top, right)` giving the size of the outer margins in lines of text.

omd A vector of the form `c(x1, x2, y1, y2)` giving the outer margin region in NDC (= normalized device coordinates), i.e., as fraction (in $[0, 1]$) of the device region.

omi A vector of the form `c(bottom, left, top, right)` giving the size of the outer margins in inches.

pch Either an integer specifying a symbol or a single character to be used as the default in plotting points. See `points` for possible values and their interpretation.

pin The current plot dimensions, `(width,height)`, in inches.

plt A vector of the form `c(x1, x2, y1, y2)` giving the coordinates of the plot region as fractions of the current figure region.

ps integer; the pointsize of text (but not symbols). Unlike the `pointsize` argument of most devices, this does not change the relationship between `mar` and `mai` (nor `oma` and `omi`), but it does scale `cin` and `csi`.

What is meant by 'pointsize' is device-specific, but most devices mean a multiple of 1bp, that is 1/72 of an inch.

pty A character specifying the type of plot region to be used; `"s"` generates a square plotting region and `"m"` generates the maximal plotting region.

smo (*Unimplemented*) a value which indicates how smooth circles and circular arcs should be.

srt The string rotation in degrees. See the comment about `crt`.

tck The length of tick marks as a fraction of the smaller of the width or height of the plotting region. If `tck >= 0.5` it is interpreted as a fraction of the relevant side, so if `tck = 1` grid lines are drawn. The default setting (`tck = NA`) is to use `tcl = -0.5` (see below).

tcl The length of tick marks as a fraction of the height of a line of text. The default value is `-0.5`; setting `tcl = NA` sets `tck = -0.01` which is S' default.

usr A vector of the form `c(x1, x2, y1, y2)` giving the extremes of the user coordinates of the plotting region. When a logarithmic scale is in use (i.e., `par("xlog")` is true, see below), then the x-limits will be `10 ^ par("usr")[1:2]`. Similarly for the y-axis.

xaxp A vector of the form `c(x1, x2, n)` giving the coordinates of the extreme tick marks and the number of intervals between tick-marks when `par("xlog")` is false. Otherwise, when *log* coordinates are active, the three values have a different meaning: For a small range, `n` is *negative*, and the ticks are as in the linear case, otherwise, `n` is in `1:3`, specifying a case number, and `x1` and `x2` are the lowest and highest power of 10 inside the user coordinates, `10 ^ par("usr")[1:2]`. (The `"usr"` coordinates are log10-transformed here!)

n=1 will produce tick marks at 10^j for integer j,

n=2 gives marks $k10^j$ with $k \in \{1, 5\}$,

n=3 gives marks $k10^j$ with $k \in \{1, 2, 5\}$.

See `axTicks()` for a pure R implementation of this.

This parameter is reset when a user coordinate system is set up, for example by starting a new page or by calling `plot.window` or setting `par("usr")`: `n` is taken from `par("lab")`. It affects the default behaviour of subsequent calls to `axis` for sides 1 or 3.

xaxs The style of axis interval calculation to be used for the x-axis. Possible values are `"r"`, `"i"`, `"e"`, `"s"`, `"d"`. The styles are generally controlled by the range of data or `xlim`, if given. Style `"r"` (regular) first extends the data range by 4 percent and then finds an axis with pretty labels that fits within the range. Style `"i"` (internal) just finds an axis with pretty labels that fits within the original data range. Style `"s"` (standard) finds an axis with pretty labels within which the original data range fits. Style `"e"` (extended) is like style `"s"`, except that it is also ensures that there is room for plotting symbols within the bounding box. Style `"d"` (direct) specifies that the current axis should be used on subsequent plots. (*Only `"r"` and `"i"` styles are currently implemented*)

xaxt A character which specifies the x axis type. Specifying `"n"` suppresses plotting of the axis. The standard value is `"s"`: for compatibility with S values `"l"` and `"t"` are accepted but are equivalent to `"s"`: any value other than `"n"` implies plotting.

xlog A logical value (see `log` in `plot.default`). If `TRUE`, a logarithmic scale is in use (e.g., after `plot(*, log = "x")`). For a new device, it defaults to `FALSE`, i.e., linear scale.

xpd A logical value or `NA`. If `FALSE`, all plotting is clipped to the plot region, if `TRUE`, all plotting is clipped to the figure region, and if `NA`, all plotting is clipped to the device region.

yaxp A vector of the form `c(y1, y2, n)` giving the coordinates of the extreme tick marks and the number of intervals between tick-marks unless for log coordinates, see `xaxp` above.

yaxs The style of axis interval calculation to be used for the y-axis. See `xaxs` above.

yaxt A character which specifies the y axis type. Specifying `"n"` suppresses plotting.

ylog A logical value; see `xlog` above.

Color Specification

Colors can be specified in several different ways. The simplest way is with a character string giving the color name (e.g., `"red"`). A list of the possible colors can be obtained with the function `colors`. Alternatively, colors can be specified directly in terms of their RGB components with a string of the form `"#RRGGBB"` where each of the pairs `RR`, `GG`, `BB` consist of two hexadecimal digits giving a value in the range `00` to `FF`. Colors can also be specified by giving an index into a small table of colors, the `palette`. This provides compatibility with S. Index `0` corresponds to the background color.

Additionally, `"transparent"` or (integer) `NA` is *transparent*, useful for filled areas (such as the background!), and just invisible for things like lines or text.

The functions `rgb`, `hsv`, `hcl`, `gray` and `rainbow` provide additional ways of generating colors.

Line Type Specification

Line types can either be specified by giving an index into a small built-in table of line types (1 = solid, 2 = dashed, etc, see `lty` above) or directly as the lengths of on/off stretches of line. This is done with a string of an even number (up to eight) of characters, namely *non-zero* (hexadecimal) digits which give the lengths in consecutive positions in the string. For example, the string `"33"` specifies three units on followed by three off and `"3313"` specifies three units on followed by three off followed by one on and finally three off. The 'units' here are (on most devices) proportional to `lwd`, and with `lwd = 1` are in pixels or points.

The five standard dash-dot line types (`lty = 2:6`) correspond to `c("44", "13", "1343", "73", "2262")`.

Note that `NA` is not a valid value for `lty`.

Prior to R 2.4.0, zero hex digits were accepted and handled in a device-dependent way, e.g. `X11()` mapped them to one, `pdf()` regarded zero as terminating the string and `windows()` gave a zero-length stretch.

Note

The effect of restoring all the (settable) graphics parameters as in the examples is hard to predict if the device has been resized. Several of them are attempting to set the same things in different ways, and those last in the alphabet will win. In particular, the settings of `mai`, `mar`, `pin`, `plt` and `pty` interact, as do the outer margin settings, the figure layout and figure region size.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

`plot.default` for some high-level plotting parameters; `colors`; `options` for other setup parameters; graphic devices `x11`, `postscript` and setting up device regions by `layout` and `split.screen`.

Examples

```
op <- par(mfrow = c(2, 2), # 2 x 2 pictures on one plot
          pty = "s")       # square plotting region,
                           # independent of device size

## At end of plotting, reset to previous settings:
par(op)

## Alternatively,
op <- par(no.readonly = TRUE) # the whole list of settable par's.
## do lots of plotting and par(.) calls, then reset:
par(op)

par("ylog") # FALSE
plot(1 : 12, log = "y")
par("ylog") # TRUE

plot(1:2, xaxs = "i") # 'inner axis' w/o extra space
```

```
par(c("usr", "xaxp"))

( nr.prof <-
  c(prof.pilots=16,lawyers=11,farmers=10,salesmen=9,physicians=9,
    mechanics=6,policemen=6,managers=6,engineers=5,teachers=4,
    housewives=3,students=3,armed.forces=1))
par(las = 3)
barplot(rbind(nr.prof)) # R 0.63.2: shows alignment problem
par(las = 0)# reset to default

## 'fg' use:
plot(1:12, type = "b", main="'fg' : axes, ticks and box in gray",
     fg = gray(0.7), bty="7" , sub=R.version.string)

ex <- function() {
   old.par <- par(no.readonly = TRUE) # all par settings which
                                      # could be changed.
   on.exit(par(old.par))
   ## ...
   ## ... do lots of par() settings and plots
   ## ...
   invisible() #-- now,  par(old.par)  will be executed
}
ex()
```

`persp` *Perspective Plots*

Description

This function draws perspective plots of surfaces over the x–y plane. `persp` is a generic function.

Usage

```
persp(x, ...)

## Default S3 method:
persp(x = seq(0, 1, len = nrow(z)), y = seq(0, 1, len = ncol(z)),
      z, xlim = range(x), ylim = range(y),
      zlim = range(z, na.rm = TRUE),
      xlab = NULL, ylab = NULL, zlab = NULL, main = NULL, sub = NULL,
      theta = 0, phi = 15, r = sqrt(3), d = 1, scale = TRUE, expand = 1,
      col = "white", border = NULL, ltheta = -135, lphi = 0, shade = NA,
      box = TRUE, axes = TRUE, nticks = 5, ticktype = "simple", ...)
```

Arguments

`x, y`	locations of grid lines at which the values in `z` are measured. These must be in ascending order. By default, equally spaced values from 0 to 1 are used. If `x` is a `list`, its components `x$x` and `x$y` are used for `x` and `y`, respectively.
`z`	a matrix containing the values to be plotted (`NA`s are allowed). Note that `x` can be used instead of `z` for convenience.

`xlim, ylim, zlim`
: x-, y- and z-limits. The plot is produced so that the rectangular volume defined by these limits is visible.

`xlab, ylab, zlab`
: titles for the axes. N.B. These must be character strings; expressions are not accepted. Numbers will be coerced to character strings.

`main, sub`
: main and sub title, as for `title`.

`theta, phi`
: angles defining the viewing direction. `theta` gives the azimuthal direction and `phi` the colatitude.

`r`
: the distance of the eyepoint from the centre of the plotting box.

`d`
: a value which can be used to vary the strength of the perspective transformation. Values of `d` greater than 1 will lessen the perspective effect and values less and 1 will exaggerate it.

`scale`
: before viewing the x, y and z coordinates of the points defining the surface are transformed to the interval [0,1]. If `scale` is `TRUE` the x, y and z coordinates are transformed separately. If `scale` is `FALSE` the coordinates are scaled so that aspect ratios are retained. This is useful for rendering things like DEM information.

`expand`
: a expansion factor applied to the `z` coordinates. Often used with `0 < expand < 1` to shrink the plotting box in the `z` direction.

`col`
: the color(s) of the surface facets. Transparent colours are ignored. This is recycled to the $(nx-1)(ny-1)$ facets.

`border`
: the color of the line drawn around the surface facets. The default, `NULL`, corresponds to `par("fg")`. A value of `NA` will disable the drawing of borders: this is sometimes useful when the surface is shaded.

`ltheta, lphi`
: if finite values are specified for `ltheta` and `lphi`, the surface is shaded as though it was being illuminated from the direction specified by azimuth `ltheta` and colatitude `lphi`.

`shade`
: the shade at a surface facet is computed as `((1+d)/2)^shade`, where `d` is the dot product of a unit vector normal to the facet and a unit vector in the direction of a light source. Values of `shade` close to one yield shading similar to a point light source model and values close to zero produce no shading. Values in the range 0.5 to 0.75 provide an approximation to daylight illumination.

`box`
: should the bounding box for the surface be displayed. The default is `TRUE`.

`axes`
: should ticks and labels be added to the box. The default is `TRUE`. If `box` is `FALSE` then no ticks or labels are drawn.

`ticktype`
: character: `"simple"` draws just an arrow parallel to the axis to indicate direction of increase; `"detailed"` draws normal ticks as per 2D plots.

`nticks`
: the (approximate) number of tick marks to draw on the axes. Has no effect if `ticktype` is `"simple"`.

`...`
: additional graphical parameters (see `par`).

Details

The plots are produced by first transforming the coordinates to the interval [0,1]. The surface is then viewed by looking at the origin from a direction defined by `theta` and `phi`. If `theta` and `phi` are both zero the viewing direction is directly down the negative y axis. Changing `theta` will vary the azimuth and changing `phi` the colatitude.

There is a hook called `"persp"` (see `setHook`) called after the plot is completed, which is used in the testing code to annotate the plot page. The hook function(s) are called with no argument.

Notice that `persp` interprets the `z` matrix as a table of `f(x[i], y[j])` values, so that the x axis corresponds to row number and the y axis to column number, with column 1 at the bottom, so that with the standard rotation angles, the top left corner of the matrix is displayed at the left hand side, closest to the user.

Value

`persp()` returns the *viewing transformation matrix*, say `VT`, a 4×4 matrix suitable for projecting 3D coordinates (x, y, z) into the 2D plane using homogenous 4D coordinates (x, y, z, t). It can be used to superimpose additional graphical elements on the 3D plot, by `lines()` or `points()`, using the simple function `trans3d()`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`contour` and `image`; `trans3d`.

Examples

```
## More examples in  demo(persp) !!
##                   -----------

# (1) The Obligatory Mathematical surface.
#     Rotated sinc function.

x <- seq(-10, 10, length= 30)
y <- x
f <- function(x,y) { r <- sqrt(x^2+y^2); 10 * sin(r)/r }
z <- outer(x, y, f)
z[is.na(z)] <- 1
op <- par(bg = "white")
persp(x, y, z, theta = 30, phi = 30, expand = 0.5, col = "lightblue")
persp(x, y, z, theta = 30, phi = 30, expand = 0.5, col = "lightblue",
      ltheta = 120, shade = 0.75, ticktype = "detailed",
      xlab = "X", ylab = "Y", zlab = "Sinc( r )"
) -> res
round(res, 3)

# (2) Add to existing persp plot - using trans3d() :

xE <- c(-10,10); xy <- expand.grid(xE, xE)
points(trans3d(xy[,1], xy[,2], 6, pm = res), col = 2, pch =16)
lines (trans3d(x, y=10, z= 6 + sin(x), pm = res), col = 3)

phi <- seq(0, 2*pi, len = 201)
r1 <- 7.725 # radius of 2nd maximum
xr <- r1 * cos(phi)
yr <- r1 * sin(phi)
lines(trans3d(xr,yr, f(xr,yr), res), col = "pink", lwd = 2)
## (no hidden lines)
```

```
# (3) Visualizing a simple DEM model

z <- 2 * volcano        # Exaggerate the relief
x <- 10 * (1:nrow(z))   # 10 meter spacing (S to N)
y <- 10 * (1:ncol(z))   # 10 meter spacing (E to W)
## Don't draw the grid lines :  border = NA
par(bg = "slategray")
persp(x, y, z, theta = 135, phi = 30, col = "green3", scale = FALSE,
      ltheta = -120, shade = 0.75, border = NA, box = FALSE)
par(op)
```

`pie` *Pie Charts*

Description

Draw a pie chart.

Usage

```
pie(x, labels = names(x), edges = 200, radius = 0.8,
    clockwise = FALSE, init.angle = if(clockwise) 90 else 0,
    density = NULL, angle = 45, col = NULL, border = NULL,
    lty = NULL, main = NULL, ...)
```

Arguments

`x`	a vector of non-negative numerical quantities. The values in `x` are displayed as the areas of pie slices.
`labels`	one or more expressions or character strings giving names for the slices. For empty or `NA` (after coercion to character) labels, no label and pointing line is drawn.
`edges`	the circular outline of the pie is approximated by a polygon with this many edges.
`radius`	the pie is drawn centered in a square box whose sides range from -1 to 1. If the character strings labeling the slices are long it may be necessary to use a smaller radius.
`clockwise`	logical indicating if slices are drawn clockwise or counter clockwise (i.e., mathematically positive direction), the latter is default.
`init.angle`	number specifying the *starting angle* (in degrees) for the slices. Defaults to 0 (i.e., '3 o'clock') unless `clockwise` is true where `init.angle` defaults to 90 (degrees), (i.e., '12 o'clock').
`density`	the density of shading lines, in lines per inch. The default value of `NULL` means that no shading lines are drawn. Non-positive values of `density` also inhibit the drawing of shading lines.
`angle`	the slope of shading lines, given as an angle in degrees (counter-clockwise).
`col`	a vector of colors to be used in filling or shading the slices. If missing a set of 6 pastel colours is used, unless `density` is specified when `par("fg")` is used.

`border, lty`	(possibly vectors) arguments passed to `polygon` which draws each slice.
`main`	an overall title for the plot.
`...`	graphical parameters can be given as arguments to `pie`. They will affect the main title and labels only.

Note

Pie charts are a very bad way of displaying information. The eye is good at judging linear measures and bad at judging relative areas. A bar chart or dot chart is a preferable way of displaying this type of data.

Cleveland (1985), page 264: "Data that can be shown by pie charts always can be shown by a dot chart. This means that judgements of position along a common scale can be made instead of the less accurate angle judgements." This statement is based on the empirical investigations of Cleveland and McGill as well as investigations by perceptual psychologists.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Cleveland, W. S. (1985) *The elements of graphing data*. Wadsworth: Monterey, CA, USA.

See Also

`dotchart`.

Examples

```
pie(rep(1, 24), col = rainbow(24), radius = 0.9)

pie.sales <- c(0.12, 0.3, 0.26, 0.16, 0.04, 0.12)
names(pie.sales) <- c("Blueberry", "Cherry",
    "Apple", "Boston Cream", "Other", "Vanilla Cream")
pie(pie.sales) # default colours
pie(pie.sales,
    col = c("purple", "violetred1", "green3", "cornsilk", "cyan", "white"))
pie(pie.sales, col = gray(seq(0.4,1.0,length=6)))
pie(pie.sales, density = 10, angle = 15 + 10 * 1:6)
pie(pie.sales, clockwise=TRUE, main="pie(*, clockwise=TRUE)")
segments(0,0, 0,1, col= "red", lwd = 2)
text(0,1, "init.angle = 90", col= "red")

n <- 200
pie(rep(1,n), labels="", col=rainbow(n), border=NA,
    main = "pie(*, labels=\"\", col=rainbow(n), border=NA,..")
```

`plot` *Generic X-Y Plotting*

Description

Generic function for plotting of R objects. For more details about the graphical parameter arguments, see `par`.

Usage

```
plot(x, y, ...)
```

Arguments

x	the coordinates of points in the plot. Alternatively, a single plotting structure, function or *any* R *object with a* `plot` *method* can be provided.
y	the y coordinates of points in the plot, *optional* if `x` is an appropriate structure.
...	Arguments to be passed to methods, such as graphical parameters (see `par`). Many methods will accept the following arguments:
type	what type of plot should be drawn. Possible types are • `"p"` for **p**oints, • `"l"` for **l**ines, • `"b"` for **b**oth, • `"c"` for the lines part alone of `"b"`, • `"o"` for both "**o**verplotted", • `"h"` for "**h**istogram" like (or "high-density") vertical lines, • `"s"` for stair **s**teps, • `"S"` for other **s**teps, see *Details* below, • `"n"` for no plotting. All other `type`s give a warning or an error; using, e.g., `type = "punkte"` being equivalent to `type = "p"` for S compatibility.
main	an overall title for the plot: see `title`.
sub	a sub title for the plot: see `title`.
xlab	a title for the x axis: see `title`.
ylab	a title for the y axis: see `title`.
asp	the y/x aspect ratio, see `plot.window`.

Details

For simple scatter plots, `plot.default` will be used. However, there are `plot` methods for many R objects, including `function`s, `data.frame`s, `density` objects, etc. Use `methods(plot)` and the documentation for these.

The two step types differ in their x-y preference: Going from $(x1, y1)$ to $(x2, y2)$ with $x1 < x2$, `type = "s"` moves first horizontal, then vertical, whereas `type = "S"` moves the other way around.

See Also

`plot.default`, `plot.formula` and other methods; `points`, `lines`, `par`.

Examples

```
plot(cars)
lines(lowess(cars))

plot(sin, -pi, 2*pi)

## Discrete Distribution Plot:
```

```
plot(table(rpois(100,5)), type = "h", col = "red", lwd=10,
     main="rpois(100,lambda=5)")

## Simple quantiles/ECDF, see ecdf() {library(stats)} for a better one:
plot(x <- sort(rnorm(47)), type = "s", main = "plot(x, type = \"s\")")
points(x, cex = .5, col = "dark red")
```

`plot.data.frame` *Plot Method for Data Frames*

Description

`plot.data.frame`, a method for the `plot` generic. It is designed for a quick look at numeric data frames.

Usage

```
## S3 method for class 'data.frame':
plot(x, ...)
```

Arguments

`x`	object of class `data.frame`.
`...`	further arguments to `stripchart`, `plot.default` or `pairs`.

Details

This is intended for data frames with *numeric* columns. For more than two columns it first calls `data.matrix` to convert the data frame to a numeric matrix and then calls `pairs` to produce a scatterplot matrix). This can fail and may well be inappropriate: for example numerical conversion of dates will lose their special meaning and a warning will be given.

For a two-column data frame it plots the second column against the first by the most appropriate method for the first column.

For a single numeric column it uses `stripchart`, and for other single-column data frames tries to find a plot method for the single column.

See Also

`data.frame`

Examples

```
plot(OrchardSprays[1], method="jitter")
plot(OrchardSprays[c(4,1)])
plot(OrchardSprays)

plot(iris)
plot(iris[5:4])
plot(women)
```

`plot.default` *The Default Scatterplot Function*

Description

Draw a scatter plot with "decorations" such as axes and titles in the active graphics window.

Usage

```
## Default S3 method:
plot(x, y = NULL, type = "p",  xlim = NULL, ylim = NULL,
     log = "", main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
     ann = par("ann"), axes = TRUE, frame.plot = axes,
     panel.first = NULL, panel.last = NULL, asp = NA, ...)
```

Arguments

`x, y`	the `x` and `y` arguments provide the x and y coordinates for the plot. Any reasonable way of defining the coordinates is acceptable. See the function `xy.coords` for details.
`type`	1-character string giving the type of plot desired. The following values are possible, for details, see `plot`: `"p"` for points, `"l"` for lines, `"o"` for overplotted points and lines, `"b"`, `"c"`) for (empty if `"c"`) points joined by lines, `"s"` and `"S"` for stair steps and `"h"` for histogram-like vertical lines. Finally, `"n"` does not produce any points or lines.
`xlim`	the x limits (x1, x2) of the plot. Note that `x1 > x2` is allowed and leads to a "reversed axis".
`ylim`	the y limits of the plot.
`log`	a character string which contains `"x"` if the x axis is to be logarithmic, `"y"` if the y axis is to be logarithmic and `"xy"` or `"yx"` if both axes are to be logarithmic.
`main`	a main title for the plot, see also `title`.
`sub`	a sub title for the plot.
`xlab`	a label for the x axis, defaults to a description of `x`.
`ylab`	a label for the y axis, defaults to a description of `y`.
`ann`	a logical value indicating whether the default annotation (title and x and y axis labels) should appear on the plot.
`axes`	a logical value indicating whether both axes should be drawn on the plot. Use graphical parameter `"xaxt"` or `"yaxt"` to suppress just one of the axes.
`frame.plot`	a logical indicating whether a box should be drawn around the plot.
`panel.first`	an expression to be evaluated after the plot axes are set up but before any plotting takes place. This can be useful for drawing background grids or scatterplot smooths.
`panel.last`	an expression to be evaluated after plotting has taken place.
`asp`	the y/x aspect ratio, see `plot.window`.
`...`	other graphical parameters (see `par` and section 'Details' below).

Details

Commonly used graphical parameters are:

col The colors for lines and points. Multiple colors can be specified so that each point can be given its own color. If there are fewer colors than points they are recycled in the standard fashion. Lines will all be plotted in the first colour specified.

bg a vector of background colors for open plot symbols, see points. Note: this is **not** the same setting as par("bg").

pch a vector of plotting characters or symbols: see points.

cex a numerical vector giving the amount by which plotting text and symbols should be scaled relative to the default. This works as a multiple of par("cex"). NULL and NA are equivalent to 1.0.

lty the line type, see par.

cex.main, col.lab, font.sub, etc settings for main- and sub-title and axis annotation, see title and par.

lwd the line width, see par.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Cleveland, W. S. (1985) *The Elements of Graphing Data.* Monterey, CA: Wadsworth.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

plot, plot.window, xy.coords.

Examples

```
Speed <- cars$speed
Distance <- cars$dist
plot(Speed, Distance, panel.first = grid(8,8),
     pch = 0, cex = 1.2, col = "blue")
plot(Speed, Distance,
     panel.first = lines(lowess(Speed, Distance), lty = "dashed"),
     pch = 0, cex = 1.2, col = "blue")

## Show the different plot types
x <- 0:12
y <- sin(pi/5 * x)
op <- par(mfrow = c(3,3), mar = .1+ c(2,2,3,1))
for (tp in c("p","l","b",  "c","o","h",  "s","S","n")) {
   plot(y ~ x, type = tp,
        main = paste("plot(*, type = \"",tp,"\")",sep=""))
   if(tp == "S") {
      lines(x,y, type = "s", col = "red", lty = 2)
      mtext("lines(*, type = \"s\", ...)", col = "red", cex=.8)
   }
}
par(op)
```

```
##--- Log-Log Plot  with  custom axes
lx <- seq(1,5, length=41)
yl <- expression(e^{-frac(1,2) * {log[10](x)}^2})
y <- exp(-.5*lx^2)
op <- par(mfrow=c(2,1), mar=par("mar")+c(0,1,0,0))
plot(10^lx, y, log="xy", type="l", col="purple",
     main="Log-Log plot", ylab=yl, xlab="x")
plot(10^lx, y, log="xy", type="o", pch='.', col="forestgreen",
     main="Log-Log plot with custom axes", ylab=yl, xlab="x",
     axes = FALSE, frame.plot = TRUE)
axis(1, at = my.at <- 10^(1:5), labels = formatC(my.at, format="fg"))
at.y <- 10^(-5:-1)
axis(2, at = at.y, labels = formatC(at.y, format="fg"), col.axis="red")
par(op)
```

`plot.design` *Plot Univariate Effects of a 'Design' or Model*

Description

Plot univariate effects of one ore more `factor`s, typically for a designed experiment as analyzed by `aov()`. Further, in S this a method of the `plot` generic function for `design` objects.

Usage

```
plot.design(x, y = NULL, fun = mean, data = NULL, ...,
            ylim = NULL, xlab = "Factors", ylab = NULL,
            main = NULL, ask = NULL, xaxt = par("xaxt"),
            axes = TRUE, xtick = FALSE)
```

Arguments

`x`	either a data frame containing the design factors and optionally the response, or a `formula` or `terms` object.
`y`	the response, if not given in x.
`fun`	a function (or name of one) to be applied to each subset. It must return one number for a numeric (vector) input.
`data`	data frame containing the variables referenced by `x` when that is formula like.
`...`	graphical arguments such as `col`, see `par`.
`ylim`	range of y values, as in `plot.default`.
`xlab`	x axis label, see `title`.
`ylab`	y axis label with a "smart" default.
`main`	main title, see `title`.
`ask`	logical indicating if the user should be asked before a new page is started – in the case of multiple y's.
`xaxt`	character giving the type of x axis.
`axes`	logical indicating if axes should be drawn.
`xtick`	logical indicating if "ticks" (one per factor) should be drawn on the x axis.

Details

The supplied function will be called once for each level of each factor in the design and the plot will show these summary values. The levels of a particular factor are shown along a vertical line, and the overall value of `fun()` for the response is drawn as a horizontal line.

This is a new R implementation which will not be completely compatible to the earlier S implementations. This is not a bug but might still change.

Note

A big effort was taken to make this closely compatible to the S version. However, `col` (and `fg`) specification has different effects.

Author(s)

Roberto Frisullo and Martin Maechler

References

Chambers, J. M. and Hastie, T. J. eds (1992) *Statistical Models in S*. Chapman & Hall, London, **the** *white book*, pp. 546–7 (and 163–4).

Freeny, A. E. and Landwehr, J. M. (1990) Displays for data from large designed experiments; Computer Science and Statistics: Proc. 22nd SympİnterfaceInterface, 117–126, Springer Verlag.

See Also

`interaction.plot` for a "standard graphic" of designed experiments.

Examples

```
plot.design(warpbreaks)# automatic for data frame with one numeric var.

Form <- breaks ~ wool + tension
summary(fm1 <- aov(Form, data = warpbreaks))
plot.design(       Form, data = warpbreaks, col = 2)# same as above

## More than one y :
utils::str(esoph)
plot.design(esoph) ## two plots; if interactive you are "ask"ed

## or rather, compare mean and median:
op <- par(mfcol = 1:2)
plot.design(ncases/ncontrols ~ ., data = esoph, ylim = c(0, 0.8))
plot.design(ncases/ncontrols ~ ., data = esoph, ylim = c(0, 0.8),
            fun = median)
par(op)
```

plot.factor *Plotting Factor Variables*

Description

This functions implements a "scatterplot" method for `factor` arguments of the *generic* `plot` function. Actually, `boxplot` is used when `y` is numeric and a `spineplot` when `y` is a factor. For a single factor `x` (i.e., with `y` missing) a simple `barplot` is produced.

Usage

```
## S3 method for class 'factor':
plot(x, y, legend.text = NULL, ...)
```

Arguments

`x, y`	numeric or factor. `y` may be missing.
`legend.text`	character vector for annotation of y axis, defaults to `levels(y)`. Can be used instead of `yaxlabels` (for backward compatibility).
`...`	Further arguments to `plot`, see also `par`.

See Also

`plot.default`, `plot.formula`, `barplot`, `boxplot`, `spineplot`.

Examples

```
plot(PlantGrowth)                                   # -> plot.data.frame
plot(weight ~ group, data = PlantGrowth)            # numeric vector ~ factor
plot(cut(weight, 2) ~ group, data = PlantGrowth) # factor ~ factor
## passing "..." to spineplot() eventually:
plot(cut(weight, 3) ~ group, data = PlantGrowth,
     col = hcl(c(0, 120, 240), 50, 70))

plot(PlantGrowth$group, axes=FALSE, main="no axes")# extremly silly
```

plot.formula *Formula Notation for Scatterplots*

Description

Specify a scatterplot or add points or lines via a formula.

Usage

```
## S3 method for class 'formula':
plot(formula, data = parent.frame(), ..., subset,
             ylab = varnames[response], ask = TRUE)

## S3 method for class 'formula':
points(formula, data = parent.frame(), ..., subset)

## S3 method for class 'formula':
lines(formula, data = parent.frame(), ..., subset)
```

Arguments

formula	a formula, such as y ~ x.
data	a data.frame (or list) from which the variables in formula should be taken.
...	Arguments to be passed to or from other methods. horizontal = TRUE is also accepted.
subset	an optional vector specifying a subset of observations to be used in the fitting process.
ylab	the y label of the plot(s).
ask	logical, see par.

Details

Both the terms in the formula and the ... arguments are evaluated in data enclosed in parent.frame() if data is a list or a data frame. The terms of the formula and those arguments in ... that are of the same length as data are subjected to the subsetting specified in subset. If the formula in plot.formula contains more than one non-response term, a series of plots of y against each term is given. A plot against the running index can be specified as plot(y ~ 1).

Missing values are not considered in these methods, and in particular cases with missing values are not removed.

If y is an object (i.e. has a class attribute) then plot.formula looks for a plot method for that class first. Otherwise, the class of x will determine the type of the plot. For factors this will be a parallel boxplot, and argument horizontal = TRUE can be used (see boxplot).

Value

These functions are invoked for their side effect of drawing in the active graphics device.

See Also

plot.default, points, lines, plot.factor.

Examples

```
op <- par(mfrow=c(2,1))
plot(Ozone ~ Wind, data = airquality, pch=as.character(Month))
plot(Ozone ~ Wind, data = airquality, pch=as.character(Month),
     subset = Month != 7)
par(op)
```

`plot.histogram` *Plot Histograms*

Description

These are methods for objects of class `"histogram"`, typically produced by `hist`.

Usage

```
## S3 method for class 'histogram':
plot(x, freq = equidist, density = NULL, angle = 45,
               col = NULL, border = par("fg"), lty = NULL,
               main = paste("Histogram of",
                            paste(x$xname, collapse="\n")),
               sub = NULL, xlab = x$xname, ylab,
               xlim = range(x$breaks), ylim = NULL,
               axes = TRUE, labels = FALSE, add = FALSE, ...)

## S3 method for class 'histogram':
lines(x, ...)
```

Arguments

`x`	a `histogram` object, or a list with components `density`, `mid`, etc, see `hist` for information about the components of `x`.
`freq`	logical; if `TRUE`, the histogram graphic is to present a representation of frequencies, i.e, `x$counts`; if `FALSE`, *relative* frequencies ("probabilities"), i.e., `x$density`, are plotted. The default is true for equidistant `breaks` and false otherwise.
`col`	a colour to be used to fill the bars. The default of `NULL` yields unfilled bars.
`border`	the color of the border around the bars.
`angle, density`	select shading of bars by lines: see `rect`.
`lty`	the line type used for the bars, see also `lines`.
`main, sub, xlab, ylab`	these arguments to `title` have useful defaults here.
`xlim, ylim`	the range of x and y values with sensible defaults.
`axes`	logical, indicating if axes should be drawn.
`labels`	logical or character. Additionally draw labels on top of bars, if not `FALSE`; if `TRUE`, draw the counts or rounded densities; if `labels` is a `character`, draw itself.
`add`	logical. If `TRUE`, only the bars are added to the current plot. This is what `lines.histogram(*)` does.
`...`	further graphical parameters to `title` and `axis`.

Details

`lines.histogram(*)` is the same as `plot.histogram(*, add = TRUE)`.

See Also

hist, stem, density.

Examples

```
(wwt <- hist(women$weight, nc= 7, plot = FALSE))
plot(wwt, labels = TRUE) # default main & xlab using wwt$xname
plot(wwt, border = "dark blue", col = "light blue",
     main = "Histogram of 15 women's weights", xlab = "weight [pounds]")

## Fake "lines" example, using non-default labels:
w2 <- wwt; w2$counts <- w2$counts - 1
lines(w2, col = "Midnight Blue", labels = ifelse(w2$counts, "> 1", "1"))
```

---

plot.table *Plot Methods for 'table' Objects*

Description

This is a method of the generic plot function for (contingency) table objects. Whereas for two- and more dimensional tables, a mosaicplot is drawn, one-dimensional ones are plotted "bar like".

Usage

```
## S3 method for class 'table':
plot(x, type = "h", ylim = c(0, max(x)), lwd = 2,
     xlab = NULL, ylab = NULL, frame.plot = is.num, ...)
```

Arguments

x	a table (like) object.
type	plotting type.
ylim	range of y-axis.
lwd	line width for bars when type = "h" is used in the 1D case.
xlab, ylab	x- and y-axis labels.
frame.plot	logical indicating if a frame (box) should be drawn in the 1D case. Defaults to true when x has dimnames coerceable to numbers.
...	further graphical arguments, see plot.default.

Details

The current implementation (R 1.2) is somewhat experimental and will be improved and extended.

See Also

plot.factor, the plot method for factors.

Examples

```
## 1-d tables
(Poiss.tab <- table(N = rpois(200, lam= 5)))
plot(Poiss.tab, main = "plot(table(rpois(200, lam=5)))")

plot(table(state.division))

## 4-D :
plot(Titanic, main ="plot(Titanic, main= *)")
```

---

plot.window *Set up World Coordinates for Graphics Window*

Description

This function sets up the world coordinate system for a graphics window. It is called by higher level functions such as `plot.default` (*after* `plot.new`).

Usage

```
plot.window(xlim, ylim, log = "", asp = NA, ...)
```

Arguments

`xlim, ylim`	numeric vectors of length 2, giving the x and y coordinates ranges.
`log`	character; indicating which axes should be in log scale.
`asp`	numeric, giving the **asp**ect ratio y/x.
`...`	further graphical parameters as in `par`. The relevant ones are `xaxs`, `yaxs` and `lab`.

Details

Note that if `asp` is a finite positive value then the window is set up so that one data unit in the x direction is equal in length to `asp` $\times$ one data unit in the y direction.

The special case `asp == 1` produces plots where distances between points are represented accurately on screen. Values with `asp > 1` can be used to produce more accurate maps when using latitude and longitude.

To reverse an axis, use `xlim` or `ylim` of the form `c(hi, lo)`.

The function attempts to produce a plausible set of scales if one or both of `xlim` and `ylim` is of length one or the two values given are identical, but it is better to avoid that case.

Usually, one should rather use the higher level functions such as `plot`, `hist`, `image`, ..., instead and refer to their help pages for explanation of the arguments.

A side-effect of the call is to set up the `usr`, `xaxp` and `yaxp` graphical parameters. (It is for the latter two that `lab` is used.)

See Also

`xy.coords`, `plot.xy`, `plot.default`.

Examples

```
##--- An example for the use of 'asp' :
require(stats)  # normally loaded
loc <- cmdscale(eurodist)
rx <- range(x <- loc[,1])
ry <- range(y <- -loc[,2])
plot(x, y, type="n", asp=1, xlab="", ylab="")
abline(h = pretty(rx, 10), v = pretty(ry, 10), col = "lightgray")
text(x, y, labels(eurodist), cex=0.8)
```

---

plot.xy *Basic Internal Plot Function*

Description

This is *the* internal function that does the basic plotting of points and lines. Usually, one should rather use the higher level functions instead and refer to their help pages for explanation of the arguments.

Usage

```
plot.xy(xy, type, pch = par("pch"), lty = par("lty"),
        col = par("col"), bg = NA,
        cex = 1, lwd = par("lwd"), ...)
```

Arguments

xy	A four-element list as results from xy.coords.
type	1 character code: see plot.default. NULL is accepted as a synonym for "p".
pch	character or integer code for kind of points, see points.default.
lty	line type code, see lines.
col	color code or name, see colors, palette. Here NULL means colour 0.
bg	background ("fill") color for the open plot symbols 21:25: see points.default.
cex	character expansion.
lwd	line width, also used for (non-filled) plot symbols, see lines and points.
...	further graphical parameters such as xpd, lend, ljoin and lmitre.

Details

The arguments pch, col, bg, cex, lwd may be vectors and may be recycled, depending on type: see points and lines for specifics. In particular note that lwd is treated as a vector for points and as a single (first) value for lines.

See Also

plot, plot.default, points, lines.

Examples

```
points.default # to see how it calls "plot.xy(xy.coords(x, y), ...)"
```

`points` *Add Points to a Plot*

Description

`points` is a generic function to draw a sequence of points at the specified coordinates. The specified character(s) are plotted, centered at the coordinates.

Usage

```
points(x, ...)

## Default S3 method:
points(x, y = NULL, type = "p", ...)
```

Arguments

`x, y`	coordinate vectors of points to plot.
`type`	character indicating the type of plotting; actually any of the `type`s as in `plot.default`.
`...`	Further graphical parameters may also be supplied as arguments. See Details.

Details

The coordinates can be passed in a plotting structure (a list with `x` and `y` components), a two-column matrix, a time series, ... See `xy.coords`.

Graphical parameters commonly used are

`pch` plotting "character", i.e., symbol to use. This can either be a single character or an integer code for one of a set of graphics symbols. The full set of S symbols is available with `pch=0:18`, see the last picture from `example(points)`, i.e., the examples below.

In addition, there is a special set of R plotting symbols which can be obtained with `pch=19:25` and `21:25` can be colored and filled with different colors:

- `pch=19`: solid circle,
- `pch=20`: bullet (smaller circle),
- `pch=21`: circle,
- `pch=22`: square,
- `pch=23`: diamond,
- `pch=24`: triangle point-up,
- `pch=25`: triangle point down.

Values `pch=26:32` are currently unused, and `pch=32:255` give the text symbol in a single-byte locale. In a multi-byte locale such as UTF-8, numeric values of `pch` greater than or equal to 32 specify a Unicode code point (except for the symbol font as selected by `par(font = 5)`).

If `pch` is an integer or character `NA` or an empty character string, the point is omitted from the plot.

Value `pch="."` is handled specially. It is a rectangle of side 0.01 inch (scaled by `cex`). In addition, if `cex = 1` (the default), each side is at least one pixel (1/72 inch on the `pdf`, `postscript` and `xfig` devices).

col color code or name, see par.

bg background ("fill") color for the open plot symbols given by pch=21:25.

cex character (or symbol) expansion: a numerical vector. This works as a multiple of par("cex").

lwd line width for drawing symbols see par.

Others less commonly used are lty and lwd for types such as "b" and "l".

Graphical parameters pch, col, bg, cex and lwd can be vectors (which will be recycled as needed) giving a value for each point plotted. If lines are to be plotted (e.g. for type = "b" the first element of lwd is used.

Points whose x, y, pch, col or cex value is NA are omitted from the plot.

Note

What is meant by 'a single character' is locale-dependent.

The encoding may not have symbols for some or all of the characters in pch=128:255

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

plot, lines, and the underlying workhorse function plot.xy.

Examples

```
plot(-4:4, -4:4, type = "n")# setting up coord. system
points(rnorm(200), rnorm(200), col = "red")
points(rnorm(100)/2, rnorm(100)/2, col = "blue", cex = 1.5)

op <- par(bg = "light blue")
x <- seq(0,2*pi, len=51)
## something "between type='b' and type='o'":
plot(x, sin(x), type="o", pch=21, bg=par("bg"), col = "blue", cex=.6,
 main='plot(..., type="o", pch=21, bg=par("bg"))')
par(op)

##-------- Showing all the extra & some char graphics symbols ------------
Pex <- 3 ## good for both .Device=="postscript" and "x11"
ipch <- 0:35; np <- length(ipch); k <- floor(sqrt(np)); dd <- c(-1,1)/2
rx <- dd + range(ix <- ipch %/% k)
ry <- dd + range(iy <- 3 + (k-1)- ipch %% k)
pch <- as.list(ipch)
pch[26+ 1:10] <- as.list(c("*",".", "o","O","0","+","-","|","%","#"))
plot(rx, ry, type="n", axes = FALSE, xlab = "", ylab = "",
     main = paste("plot symbols :  points (...  pch = *, cex =", Pex,")"))
abline(v = ix, h = iy, col = "lightgray", lty = "dotted")
for(i in 1:np) {
  pc <- pch[[i]]
  points(ix[i], iy[i], pch = pc, col = "red", bg = "yellow", cex = Pex)
  ## red symbols with a yellow interior (where available)
  text(ix[i] - .3, iy[i], pc, col = "brown", cex = 1.2)
}
```

`polygon` *Polygon Drawing*

Description

`polygon` draws the polygons whose vertices are given in `x` and `y`.

Usage

```
polygon(x, y = NULL, density = NULL, angle = 45,
        border = NULL, col = NA, lty = par("lty"), ...)
```

Arguments

`x,y`	vectors containing the coordinates of the vertices of the polygon.
`density`	the density of shading lines, in lines per inch. The default value of `NULL` means that no shading lines are drawn. A zero value of `density` means no shading lines whereas negative values (and `NA`) suppress shading (and so allow color filling).
`angle`	the slope of shading lines, given as an angle in degrees (counter-clockwise).
`col`	the color for filling the polygon. The default, `NA`, is to leave polygons unfilled, unless `density` is specified. (For back-compatibility, `NULL` is equivalent to `NA`.)
`border`	the color to draw the border. The default, `NULL`, means to use `par("fg")`. Use `border = NA` to omit borders. For compatibility with S, `border` can also be logical, in which case `FALSE` is equivalent to `NA` (borders omitted) and `TRUE` is equivalent to `NULL` (use the foreground colour),
`lty`	the line type to be used, as in `par`.
`...`	graphical parameters such as `xpd`, `lend`, `ljoin` and `lmitre` can be given as arguments.

Details

The coordinates can be passed in a plotting structure (a list with `x` and `y` components), a two-column matrix, ... See `xy.coords`.

It is assumed that the polygon is to be closed by joining the last point to the first point.

The coordinates can contain missing values. The behaviour is similar to that of `lines`, except that instead of breaking a line into several lines, `NA` values break the polygon into several complete polygons (including closing the last point to the first point). See the examples below.

When multiple polygons are produced, the values of `density`, `angle`, `col`, `border`, and `lty` are recycled in the usual manner.

Bugs

The present shading algorithm can produce incorrect results for self-intersecting polygons.

Author(s)

The code implementing polygon shading was donated by Kevin Buhr ⟨buhr@stat.wisc.edu⟩.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

`segments` for even more flexibility, `lines`, `rect`, `box`, `abline`.

`par` for how to specify colors.

Examples

```
x <- c(1:9,8:1)
y <- c(1,2*(5:3),2,-1,17,9,8,2:9)
op <- par(mfcol=c(3,1))
for(xpd in c(FALSE,TRUE,NA)) {
  plot(1:10, main = paste("xpd =", xpd))
  box("figure", col = "pink", lwd=3)
  polygon(x,y, xpd=xpd, col="orange", lty=2, lwd=2, border="red")
}
par(op)

n <- 100
xx <- c(0:n, n:0)
yy <- c(c(0,cumsum(rnorm(n))), rev(c(0,cumsum(rnorm(n)))))
plot   (xx, yy, type="n", xlab="Time", ylab="Distance")
polygon(xx, yy, col="gray", border = "red")
title("Distance Between Brownian Motions")

# Multiple polygons from NA values
# and recycling of col, border, and lty
op <- par(mfrow=c(2,1))
plot(c(1,9), 1:2, type="n")
polygon(1:9, c(2,1,2,1,1,2,1,2,1),
        col=c("red", "blue"),
        border=c("green", "yellow"),
        lwd=3, lty=c("dashed", "solid"))
plot(c(1,9), 1:2, type="n")
polygon(1:9, c(2,1,2,1,NA,2,1,2,1),
        col=c("red", "blue"),
        border=c("green", "yellow"),
        lwd=3, lty=c("dashed", "solid"))
par(op)

# Line-shaded polygons
plot(c(1,9), 1:2, type="n")
polygon(1:9, c(2,1,2,1,NA,2,1,2,1),
        density=c(10, 20), angle=c(-45, 45))
```

rect *Draw One or More Rectangles*

Description

rect draws a rectangle (or sequence of rectangles) with the given coordinates, fill and border colors.

Usage

```
rect(xleft, ybottom, xright, ytop, density = NULL, angle = 45,
     col = NA, border = NULL, lty = par("lty"), lwd = par("lwd"),
     ...)
```

Arguments

xleft	a vector (or scalar) of left x positions.
ybottom	a vector (or scalar) of bottom y positions.
xright	a vector (or scalar) of right x positions.
ytop	a vector (or scalar) of top y positions.
density	the density of shading lines, in lines per inch. The default value of NULL means that no shading lines are drawn. A zero value of density means no shading lines whereas negative values (and NA) suppress shading (and so allow color filling).
angle	angle (in degrees) of the shading lines.
col	color(s) to fill or shade the rectangle(s) with. The default NA (or also NULL) means do not fill, i.e., draw transparent rectangles, unless density is specified.
border	color for rectangle border(s). The default means par("fg"). Use border = NA to omit borders. Can also be FALSE to suppress the border, or TRUE in which case col is used.
lty	line type for borders and shading; defaults to "solid".
lwd	line width for borders and shading.
...	graphical parameters such as xpd, lend, ljoin and lmitre can be given as arguments.

Details

The positions supplied, i.e., xleft, ..., are relative to the current plotting region. If the x-axis goes from 100 to 200 then xleft must be larger than 100 and xright must be less than 200.

It is a primitive function used in hist, barplot, legend, etc.

See Also

box for the "standard" box around the plot; polygon and segments for flexible line drawing.

par for how to specify colors.

Examples

```
## set up the plot region:
op <- par(bg = "thistle")
plot(c(100, 250), c(300, 450), type = "n", xlab="", ylab="",
     main = "2 x 11 rectangles; 'rect(100+i,300+i,  150+i,380+i)'")
i <- 4*(0:10)
## draw rectangles with bottom left (100, 300)+i
## and top right (150, 380)+i
rect(100+i, 300+i, 150+i, 380+i, col=rainbow(11, start=.7,end=.1))
rect(240-i, 320+i, 250-i, 410+i, col=heat.colors(11), lwd=i/5)
## Background alternating  ( transparent / "bg" ) :
j <- 10*(0:5)
rect(125+j, 360+j,   141+j, 405+j/2, col = c(NA,0),
     border = "gold", lwd = 2)
rect(125+j, 296+j/2, 141+j, 331+j/5, col = c(NA,"midnightblue"))
mtext("+  2 x 6 rect(*, col = c(NA,0)) and  col = c(NA,\"m..blue\"))")

## an example showing colouring and shading
plot(c(100, 200), c(300, 450), type= "n", xlab="", ylab="")
rect(100, 300, 125, 350) # transparent
rect(100, 400, 125, 450, col="green", border="blue") # coloured
rect(115, 375, 150, 425, col=par("bg"), border="transparent")
rect(150, 300, 175, 350, density=10, border="red")
rect(150, 400, 175, 450, density=30, col="blue",
     angle=-30, border="transparent")

legend(180, 450, legend=1:4, fill=c(NA, "green", par("fg"), "blue"),
       density=c(NA, NA, 10, 30), angle=c(NA, NA, 30, -30))

par(op)
```

---

rug *Add a Rug to a Plot*

Description

Adds a *rug* representation (1-d plot) of the data to the plot.

Usage

```
rug(x, ticksize = 0.03, side = 1, lwd = 0.5, col = par("fg"),
    quiet = getOption("warn") < 0, ...)
```

Arguments

x	A numeric vector
ticksize	The length of the ticks making up the 'rug'. Positive lengths give inwards ticks.
side	On which side of the plot box the rug will be plotted. Normally 1 (bottom) or 3 (top).
lwd	The line width of the ticks.
col	The colour the ticks are plotted in.
quiet	logical indicating if there should be a warning about clipped values.

`...` further arguments, passed to `axis`, such as `line` or `pos` for specifying the location of the rug.

Details

Because of the way `rug` is implemented, only values of `x` that fall within the plot region are included. There will be a warning if any finite values are omitted, but non-finite values are omitted silently.

Because of the way colours are done the axis itself is redrawn in the same colour, `lty` and `lwd` as the ticks. You can replot the box if you don't like this feature.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

See Also

`jitter` which you may want for ties in `x`.

Examples

```
require(stats)# both 'density' and its default method
with(faithful, {
    plot(density(eruptions, bw = 0.15))
    rug(eruptions)
    rug(jitter(eruptions, amount = 0.01), side = 3, col = "light blue")
})
```

---

screen *Creating and Controlling Multiple Screens on a Single Device*

Description

`split.screen` defines a number of regions within the current device which can, to some extent, be treated as separate graphics devices. It is useful for generating multiple plots on a single device. Screens can themselves be split, allowing for quite complex arrangements of plots.

`screen` is used to select which screen to draw in.

`erase.screen` is used to clear a single screen, which it does by filling with the background colour.

`close.screen` removes the specified screen definition(s).

Usage

```
split.screen(figs, screen, erase = TRUE)
screen(n = , new = TRUE)
erase.screen(n = )
close.screen(n, all.screens = FALSE)
```

Arguments

figs	A two-element vector describing the number of rows and the number of columns in a screen matrix *or* a matrix with 4 columns. If a matrix, then each row describes a screen with values for the left, right, bottom, and top of the screen (in that order) in NDC units, that is 0 at the lower left corner of the device surface, and 1 at the upper right corner.
screen	A number giving the screen to be split. It defaults to the current screen if there is one, otherwise the whole device region.
erase	logical: should be selected screen be cleared?
n	A number indicating which screen to prepare for drawing (screen), erase (erase.screen), or close (close.screen). (close.screen will accept a vector of screen numbers.)
new	A logical value indicating whether the screen should be erased as part of the preparation for drawing in the screen.
all.screens	A logical value indicating whether all of the screens should be closed.

Details

The first call to split.screen places R into split-screen mode. The other split-screen functions only work within this mode. While in this mode, certain other commands should be avoided (see the Warnings section below). Split-screen mode is exited by the command close.screen(all = TRUE).

If the current screen is closed, close.screen sets the current screen to be the next larger screen number if there is one, otherwise to the first available screen.

Value

split.screen returns a vector of screen numbers for the newly-created screens. With no arguments, split.screen returns a vector of valid screen numbers.

screen invisibly returns the number of the selected screen. With no arguments, screen returns the number of the current screen.

close.screen returns a vector of valid screen numbers.

screen, erase.screen, and close.screen all return FALSE if R is not in split-screen mode.

Warnings

The recommended way to use these functions is to completely draw a plot and all additions (i.e. points and lines) to the base plot, prior to selecting and plotting on another screen. The behavior associated with returning to a screen to add to an existing plot is unpredictable and may result in problems that are not readily visible.

These functions are totally incompatible with the other mechanisms for arranging plots on a device: par(mfrow), par(mfcol) and layout().

The functions are also incompatible with some plotting functions, such as coplot, which make use of these other mechanisms.

erase.screen will appear not to work if the background colour is transparent (as it is by default on most devices).

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

par, layout, Devices, dev.*

Examples

```
if (interactive()) {
par(bg = "white")           # default is likely to be transparent
split.screen(c(2,1))        # split display into two screens
split.screen(c(1,3), screen = 2) # now split the bottom half into 3
screen(1) # prepare screen 1 for output
plot(10:1)
screen(4) # prepare screen 4 for output
plot(10:1)
close.screen(all = TRUE)    # exit split-screen mode

split.screen(c(2,1))        # split display into two screens
split.screen(c(1,2),2)      # split bottom half in two
plot(1:10)                  # screen 3 is active, draw plot
erase.screen()              # forgot label, erase and redraw
plot(1:10, ylab= "ylab 3")
screen(1)                   # prepare screen 1 for output
plot(1:10)
screen(4)                   # prepare screen 4 for output
plot(1:10, ylab="ylab 4")
screen(1, FALSE)            # return to screen 1, but do not clear
plot(10:1, axes=FALSE, lty=2, ylab="")  # overlay second plot
axis(4)                     # add tic marks to right-hand axis
title("Plot 1")
close.screen(all = TRUE)    # exit split-screen mode
}
```

---

segments *Add Line Segments to a Plot*

Description

Draw line segments between pairs of points.

Usage

```
segments(x0, y0, x1, y1,
         col = par("fg"), lty = par("lty"), lwd = par("lwd"),
         ...)
```

Arguments

`x0, y0`	coordinates of points **from** which to draw.
`x1, y1`	coordinates of points **to** which to draw.
`col, lty, lwd`	usual graphical parameters as in `par`, possibly vectors. `NA` values in `col` cause the segment to be omitted.
`...`	further graphical parameters (from `par`), such as `xpd` and the line characteristics `lend`, `ljoin` and `lmitre`.

Details

For each `i`, a line segment is drawn between the point `(x0[i], y0[i])` and the point `(x1[i], y1[i])`.

The graphical parameters `col`, `lty` and `lwd` can be vectors of length greater than one and will be recycled if necessary.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`arrows`, `polygon` for slightly easier and less flexible line drawing, and `lines` for the usual polygons.

Examples

```
x <- runif(12); y <- rnorm(12)
i <- order(x,y); x <- x[i]; y <- y[i]
plot(x, y, main="arrows(.) and segments(.)")
## draw arrows from point to point :
s <- seq(length(x)-1)# one shorter than data
arrows(x[s], y[s], x[s+1], y[s+1], col= 1:3)
s <- s[-length(s)]
segments(x[s], y[s], x[s+2], y[s+2], col= 'pink')
```

`spineplot` *Spine Plots and Spinograms*

Description

Spine plots are a special cases of mosaic plots, and can be seen as a generalization of stacked (or highlighted) bar plots. Analogously, spinograms are an extension of histograms.

Usage

```
spineplot(x, ...)

## Default S3 method:
spineplot(x, y = NULL,
          breaks = NULL, tol.ylab = 0.05, off = NULL,
          col = NULL, main = "", xlab = NULL, ylab = NULL,
          xaxlabels = NULL, yaxlabels = NULL,
          xlim = NULL, ylim = c(0, 1), ...)

## S3 method for class 'formula':
spineplot(formula, data = list(),
          breaks = NULL, tol.ylab = 0.05, off = NULL,
          col = NULL, main = "", xlab = NULL, ylab = NULL,
          xaxlabels = NULL, yaxlabels = NULL,
          xlim = NULL, ylim = c(0, 1), ...,
          subset = NULL)
```

Arguments

x	an object, the default method expects either a single variable (interpreted to be the explanatory variable) or a 2-way table. See details.
y	a "factor" interpreted to be the dependent variable
formula	a "formula" of type y ~ x with a single dependent "factor" and a single explanatory variable.
data	an optional data frame.
breaks	if the explantory variable is numeric, this controls how it is discretized. breaks is passed to hist and can be a list of arguments.
tol.ylab	convenience tolerance parameter for y-axis annotation. If the distance between two labels drops under this threshold, they are plotted equidistantly.
off	vertical offset between the bars (in per cent). It is fixed to 0 for spinograms and defaults to 2 for spine plots.
col	a vector of fill colors of the same length as levels(y). The default is to call gray.colors.
main, xlab, ylab	character strings for annotation
xaxlabels, yaxlabels	character vectors for annotation of x and y axis. Default to levels(y) and levels(x), respectively for the spine plot. For xaxlabels in the spinogram, the breaks are used.
xlim, ylim	the range of x and y values with sensible defaults.
...	additional arguments passed to rect.
subset	an optional vector specifying a subset of observations to be used for plotting.

Details

spineplot creates either a spinogram or a spine plot. It can be called via spineplot(x, y) or spineplot(y ~ x) where y is interpreted to be the dependent variable (and has to be categorical) and x the explanatory variable. x can be either categorical (then a spine plot is created)

or numerical (then a spinogram is plotted). Additionally, `spineplot` can also be called with only a single argument which then has to be a 2-way table, interpreted to correspond to `table(x, y)`.

Both, spine plots and spinograms, are essentially mosaic plots with special formatting of spacing and shading. Conceptually, they plot $P(y|x)$ against $P(x)$. For the spine plot (where both x and y are categorical), both quantities are approximated by the corresponding empirical relative frequencies. For the spinogram (where x is numerical), x is first discretized (by caling `hist` with `breaks` argument) and then empirical relative frequencies are taken.

Thus, spine plots can also be seen as a generalization of stacked bar plots where not the heights but the widths of the bars corresponds to the relative frequencies of `x`. The heights of the bars then correspond to the conditional relative frequencies of `y` in every `x` group. Analogously, spinograms extend stacked histograms.

Value

The table visualized is returned invisibly.

Author(s)

Achim Zeileis ⟨Achim.Zeileis@R-project.org⟩

References

Friendly, M. (1994), Mosaic displays for multi-way contingency tables. *Journal of the American Statistical Association*, **89**, 190–200.

Hartigan, J.A., and Kleiner, B. (1984), A mosaic of television ratings. *The American Statistician*, **38**, 32–35.

Hofmann, H., Theus, M. (2005), *Interactive graphics for visualizing conditional distributions*, Unpublished Manuscript.

Hummel, J. (1996), Linked bar charts: Analysing categorical data graphically. *Computational Statistics*, **11**, 23–33.

See Also

`mosaicplot`, `hist`, `cdplot`

Examples

```
## treatment and improvement of patients with rheumatoid arthritis
treatment <- factor(rep(c(1, 2), c(43, 41)), levels = c(1, 2),
                    labels = c("placebo", "treated"))
improved <- factor(rep(c(1, 2, 3, 1, 2, 3), c(29, 7, 7, 13, 7, 21)),
                   levels = c(1, 2, 3), labels = c("none", "some", "marked"))

## (dependence on a categorical variable)
(spineplot(improved ~ treatment))

## applications and admissions by department at UC Berkeley
## (two-way tables)
(spineplot(margin.table(UCBAdmissions, c(3, 2)),
           main = "Applications at UCB"))
(spineplot(margin.table(UCBAdmissions, c(3, 1)),
           main = "Admissions at UCB"))
```

```
## NASA space shuttle o-ring failures
fail <- factor(c(2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 2, 1, 2, 1,
                 1, 1, 1, 2, 1, 1, 1, 1, 1),
               levels = c(1, 2), labels = c("no", "yes"))
temperature <- c(53, 57, 58, 63, 66, 67, 67, 67, 68, 69, 70, 70,
                 70, 70, 72, 73, 75, 75, 76, 76, 78, 79, 81)

## (dependence on a numerical variable)
(spineplot(fail ~ temperature))
(spineplot(fail ~ temperature, breaks = 3))
(spineplot(fail ~ temperature, breaks = quantile(temperature)))
```

`stars` *Star (Spider/Radar) Plots and Segment Diagrams*

Description

Draw star plots or segment diagrams of a multivariate data set. With one single location, also draws "spider" (or "radar") plots.

Usage

```
stars(x, full = TRUE, scale = TRUE, radius = TRUE,
      labels = dimnames(x)[[1]], locations = NULL,
      nrow = NULL, ncol = NULL, len = 1,
      key.loc = NULL, key.labels = dimnames(x)[[2]], key.xpd = TRUE,
      xlim = NULL, ylim = NULL, flip.labels = NULL,
      draw.segments = FALSE, col.segments = 1:n.seg, col.stars = NA,
      axes = FALSE, frame.plot = axes,
      main = NULL, sub = NULL, xlab = "", ylab = "",
      cex = 0.8, lwd = 0.25, lty = par("lty"), xpd = FALSE,
      mar = pmin(par("mar"),
                 1.1+ c(2*axes+ (xlab != ""),
                 2*axes+ (ylab != ""), 1,0)),
      add = FALSE, plot = TRUE, ...)
```

Arguments

`x`	matrix or data frame of data. One star or segment plot will be produced for each row of `x`. Missing values (`NA`) are allowed, but they are treated as if they were 0 (after scaling, if relevant).
`full`	logical flag: if `TRUE`, the segment plots will occupy a full circle. Otherwise, they occupy the (upper) semicircle only.
`scale`	logical flag: if `TRUE`, the columns of the data matrix are scaled independently so that the maximum value in each column is 1 and the minimum is 0. If `FALSE`, the presumption is that the data have been scaled by some other algorithm to the range $[0, 1]$.
`radius`	logical flag: in `TRUE`, the radii corresponding to each variable in the data will be drawn.
`labels`	vector of character strings for labeling the plots. Unlike the S function `stars`, no attempt is made to construct labels if `labels = NULL`.

locations	Either two column matrix with the x and y coordinates used to place each of the segment plots; or numeric of length 2 when all plots should be superimposed (for a "spider plot"). By default, locations = NULL, the segment plots will be placed in a rectangular grid.
nrow, ncol	integers giving the number of rows and columns to use when locations is NULL. By default, nrow == ncol, a square layout will be used.
len	scale factor for the length of radii or segments.
key.loc	vector with x and y coordinates of the unit key.
key.labels	vector of character strings for labeling the segments of the unit key. If omitted, the second component of dimnames(x) is used, if available.
key.xpd	clipping switch for the unit key (drawing and labeling), see par("xpd").
xlim	vector with the range of x coordinates to plot.
ylim	vector with the range of y coordinates to plot.
flip.labels	logical indicating if the label locations should flip up and down from diagram to diagram. Defaults to a somewhat smart heuristic.
draw.segments	logical. If TRUE draw a segment diagram.
col.segments	color vector (integer or character, see par), each specifying a color for one of the segments (variables). Ignored if draw.segments = FALSE.
col.stars	color vector (integer or character, see par), each specifying a color for one of the stars (cases). Ignored if draw.segments = TRUE.
axes	logical flag: if TRUE axes are added to the plot.
frame.plot	logical flag: if TRUE, the plot region is framed.
main	a main title for the plot.
sub	a sub title for the plot.
xlab	a label for the x axis.
ylab	a label for the y axis.
cex	character expansion factor for the labels.
lwd	line width used for drawing.
lty	line type used for drawing.
xpd	logical or NA indicating if clipping should be done, see par(xpd = .).
mar	argument to par(mar = *), typically choosing smaller margings than by default.
...	further arguments, passed to the first call of plot(), see plot.default and to box() if frame.plot is true.
add	logical, if TRUE *add* stars to current plot.
plot	logical, if FALSE, nothing is plotted.

Details

Missing values are treated as 0.

Each star plot or segment diagram represents one row of the input x. Variables (columns) start on the right and wind counterclockwise around the circle. The size of the (scaled) column is shown by the distance from the center to the point on the star or the radius of the segment representing the variable.

Only one page of output is produced.

Note

This code started life as spatial star plots by David A. Andrews. See http://www.udallas.edu:8080/~andrews/software/software.html.

Prior to 1.4.1, scaling only shifted the maximum to 1, although documented as here.

Author(s)

Thomas S. Dye

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
stars(mtcars[, 1:7], key.loc = c(14, 2),
      main = "Motor Trend Cars : stars(*, full = F)", full = FALSE)
stars(mtcars[, 1:7], key.loc = c(14, 1.5),
      main = "Motor Trend Cars : full stars()",flip.labels=FALSE)

## 'Spider' or 'Radar' plot:
stars(mtcars[, 1:7], locations = c(0,0), radius = FALSE,
      key.loc=c(0,0), main="Motor Trend Cars", lty = 2)

## Segment Diagrams:
palette(rainbow(12, s = 0.6, v = 0.75))
stars(mtcars[, 1:7], len = 0.8, key.loc = c(12, 1.5),
      main = "Motor Trend Cars", draw.segments = TRUE)
stars(mtcars[, 1:7], len = 0.6, key.loc = c(1.5, 0),
      main = "Motor Trend Cars", draw.segments = TRUE,
      frame.plot=TRUE, nrow = 4, cex = .7)

## scale linearly (not affinely) to [0, 1]
USJudge <- apply(USJudgeRatings, 2, function(x) x/max(x))
Jnam <- row.names(USJudgeRatings)
Snam <- abbreviate(substring(Jnam,1,regexpr("[,.]",Jnam) - 1), 7)
stars(USJudge, labels = Jnam, scale = FALSE,
      key.loc = c(13, 1.5), main = "Judge not ...", len = 0.8)
stars(USJudge, labels = Snam, scale = FALSE,
      key.loc = c(13, 1.5), radius = FALSE)

loc <- stars(USJudge, labels = NULL, scale = FALSE,
             radius = FALSE, frame.plot = TRUE,
             key.loc = c(13, 1.5), main = "Judge not ...", len = 1.2)
text(loc, Snam, col = "blue", cex = 0.8, xpd = TRUE)

## 'Segments':
stars(USJudge, draw.segments = TRUE, scale = FALSE, key.loc = c(13,1.5))

## 'Spider':
stars(USJudgeRatings, locations=c(0,0), scale=FALSE,radius = FALSE,
      col.stars=1:10, key.loc = c(0,0), main="US Judges rated")
## 'Radar-Segments'
stars(USJudgeRatings[1:10,], locations = 0:1, scale=FALSE,
      draw.segments = TRUE, col.segments=0, col.stars=1:10,key.loc= 0:1,
```

```
       main="US Judges 1-10 ")
palette("default")
stars(cbind(1:16,10*(16:1)),draw.segments=TRUE,
      main = "A Joke -- do *not* use symbols on 2D data!")
```

`stem` *Stem-and-Leaf Plots*

Description

`stem` produces a stem-and-leaf plot of the values in `x`. The parameter `scale` can be used to expand the scale of the plot. A value of `scale=2` will cause the plot to be roughly twice as long as the default.

Usage

```
stem(x, scale = 1, width = 80, atom = 1e-08)
```

Arguments

`x`	a numeric vector.
`scale`	This controls the plot length.
`width`	The desired width of plot.
`atom`	a tolerance.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
stem(islands)
stem(log10(islands))
```

`stripchart` *1-D Scatter Plots*

Description

`stripchart` produces one dimensional scatter plots (or dot plots) of the given data. These plots are a good alternative to `boxplot`s when sample sizes are small.

Usage

```
stripchart(x, method = "overplot", jitter = 0.1, offset = 1/3,
           vertical = FALSE, group.names, add = FALSE,
           at = NULL, xlim = NULL, ylim = NULL,
           main = "", ylab = "", xlab = "",
           log = "", pch = 0, col = par("fg"), cex = par("cex"))
```

Arguments

x	the data from which the plots are to be produced. The data can be specified as a single numeric vector, or as list of numeric vectors, each corresponding to a component plot. Alternatively a symbolic specification of the form `x ~ g` can be given, indicating the observations in the vector `x` are to be grouped according to the levels of the factor `g`. `NA`s are allowed in the data.
method	the method to be used to separate coincident points. The default method `"overplot"` causes such points to be overplotted, but it is also possible to specify `"jitter"` to jitter the points, or `"stack"` have coincident points stacked. The last method only makes sense for very granular data.
jitter	when `method="jitter"` is used, `jitter` gives the amount of jittering applied.
offset	when stacking is used, points are stacked this many line-heights (symbol widths) apart.
vertical	when vertical is `TRUE` the plots are drawn vertically rather than the default horizontal.
group.names	group labels which will be printed alongside (or underneath) each plot.
add	logical, if true *add* the chart to the current plot.
at	numeric vector giving the locations where the charts should be drawn, particularly when `add = TRUE`; defaults to `1:n` where `n` is the number of boxes.
main, ylab, xlab	labels: see `title`.
xlim, ylim	plot limits: see `plot.window`.
log, pch, col, cex	Graphical parameters: see `par`.

Details

Extensive examples of the use of this kind of plot can be found in Box, Hunter and Hunter or Seber and Wild.

Examples

```
x <- rnorm(50)
xr <- round(x, 1)
stripchart(x) ; m <- mean(par("usr")[1:2])
text(m, 1.04, "stripchart(x, \"overplot\")")
stripchart(xr, method = "stack", add = TRUE, at = 1.2)
text(m, 1.35, "stripchart(round(x,1), \"stack\")")
stripchart(xr, method = "jitter", add = TRUE, at = 0.7)
text(m, 0.85, "stripchart(round(x,1), \"jitter\")")

with(OrchardSprays,
     stripchart(decrease ~ treatment,
                main = "stripchart(Orchardsprays)", ylab = "decrease",
                vertical = TRUE, log = "y"))

with(OrchardSprays,
     stripchart(decrease ~ treatment, at = c(1:8)^2,
                main = "stripchart(Orchardsprays)", ylab = "decrease",
                vertical = TRUE, log = "y"))
```

strwidth *Plotting Dimensions of Character Strings and Math Expressions*

Description

These functions compute the width or height, respectively, of the given strings or mathematical expressions `s[i]` on the current plotting device in *user* coordinates, *inches* or as fraction of the figure width `par("fin")`.

Usage

```
strwidth(s, units = "user", cex = NULL)
strheight(s, units = "user", cex = NULL)
```

Arguments

`s`	character vector or expressions whose string widths in plotting units are to be determined. An attempt is made to coerce other vectors to character, and other language objects (names and calls) to expressions.
`units`	character indicating in which units `s` is measured; should be one of `"user"`, `"inches"`, `"figure"`; partial matching is performed.
`cex`	numeric **c**haracter **ex**pansion factor; multiplied by `par("cex")` yields the final character size; the default `NULL` is equivalent to `1`.

Value

Numeric vector with the same length as `s`, giving the width or height for each `s[i]`. `NA` strings are given width and height 0 (as they are not plotted).

See Also

`text`, `nchar`

Examples

```
str.ex <- c("W","w","I",".","WwI.")
op <- par(pty='s'); plot(1:100,1:100, type="n")
sw <- strwidth(str.ex); sw
all.equal(sum(sw[1:4]), sw[5])#- since the last string contains the others

sw.i <- strwidth(str.ex, "inches"); 25.4 * sw.i # width in [mm]
unique(sw / sw.i)
# constant factor: 1 value
mean(sw.i / strwidth(str.ex, "fig")) / par('fin')[1] # = 1: are the same

## See how letters fall in classes -- depending on graphics device and font!
all.lett <- c(letters, LETTERS)
shL <- strheight(all.lett, units = "inches") * 72 # 'big points'
table(shL) # all have same heights ...
mean(shL)/par("cin")[2] # around 0.6

(swL <- strwidth(all.lett, units="inches") * 72) # 'big points'
```

```
split(all.lett, factor(round(swL, 2)))

sumex <- expression(sum(x[i], i=1,n), e^{i * pi} == -1)
strwidth(sumex)
strheight(sumex)

par(op)#- reset to previous setting
```

`sunflowerplot` *Produce a Sunflower Scatter Plot*

Description

Multiple points are plotted as "sunflowers" with multiple leaves ("petals") such that overplotting is visualized instead of accidental and invisible.

Usage

```
sunflowerplot(x, y = NULL, number, log = "", digits = 6,
              xlab = NULL, ylab = NULL, xlim = NULL, ylim = NULL,
              add = FALSE, rotate = FALSE,
              pch = 16, cex = 0.8, cex.fact = 1.5,
              col = par("col"), bg = NA, size = 1/8, seg.col = 2,
              seg.lwd = 1.5, ...)
```

Arguments

`x`	numeric vector of `x`-coordinates of length `n`, say, or another valid plotting structure, as for `plot.default`, see also `xy.coords`.
`y`	numeric vector of `y`-coordinates of length `n`.
`number`	integer vector of length `n`. `number[i]` = number of replicates for `(x[i],y[i])`, may be 0. Default: compute the exact multiplicity of the points `x[],y[]`.
`log`	character indicating log coordinate scale, see `plot.default`.
`digits`	when `number` is computed (i.e., not specified), `x` and `y` are rounded to `digits` significant digits before multiplicities are computed.
`xlab,ylab`	character label for x-, or y-axis, respectively.
`xlim,ylim`	`numeric(2)` limiting the extents of the x-, or y-axis.
`add`	logical; should the plot be added on a previous one ? Default is `FALSE`.
`rotate`	logical; if `TRUE`, randomly rotate the sunflowers (preventing artefacts).
`pch`	plotting character to be used for points (`number[i]==1`) and center of sunflowers.
`cex`	numeric; character size expansion of center points (s. `pch`).
`cex.fact`	numeric *shrinking* factor to be used for the center points *when there are flower leaves*, i.e., `cex / cex.fact` is used for these.
`col, bg`	colors for the plot symbols, passed to `plot.default`.
`size`	of sunflower leaves in inches, 1[in] := 2.54[cm]. Default: 1/8, approximately 3.2mm.

`seg.col`	color to be used for the **seg**ments which make the sunflowers leaves, see `par(col=)`; `col = "gold"` reminds of real sunflowers.
`seg.lwd`	numeric; the line width for the leaves' segments.
`...`	further arguments to `plot` [if `add=FALSE`].

Details

For `number[i]==1`, a (slightly enlarged) usual plotting symbol (`pch`) is drawn. For `number[i] > 1`, a small plotting symbol is drawn and `number[i]` equi-angular "rays" emanate from it.

If `rotate=TRUE` and `number[i] >= 2`, a random direction is chosen (instead of the y-axis) for the first ray. The goal is to `jitter` the orientations of the sunflowers in order to prevent artefactual visual impressions.

Value

A list with three components of same length,

`x`	x coordinates
`y`	y coordinates
`number`	number

Side Effects

A scatter plot is drawn with "sunflowers" as symbols.

Author(s)

Andreas Ruckstuhl, Werner Stahel, Martin Maechler, Tim Hesterberg, 1989–1993. Port to R by Martin Maechler ⟨maechler@stat.math.ethz.ch⟩.

References

Chambers, J. M., Cleveland, W. S., Kleiner, B. and Tukey, P. A. (1983) *Graphical Methods for Data Analysis.* Wadsworth.

Schilling, M. F. and Watkins, A. E. (1994) A suggestion for sunflower plots. *The American Statistician*, **48**, 303–305.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

`density`

Examples

```
## 'number' is computed automatically:
sunflowerplot(iris[, 3:4])
## Imitating  Chambers et al., p.109, closely:
sunflowerplot(iris[, 3:4],cex=.2, cex.f=1, size=.035, seg.lwd=.8)

sunflowerplot(x=sort(2*round(rnorm(100))), y= round(rnorm(100),0),
              main = "Sunflower Plot of Rounded N(0,1)")
```

```
## A 'point process' {explicit 'number' argument}:
sunflowerplot(rnorm(100),rnorm(100), number=rpois(n=100,lambda=2),
              rotate=TRUE, main="Sunflower plot", col = "blue4")
```

`symbols` *Draw Symbols (Circles, Squares, Stars, Thermometers, Boxplots) on a Plot*

Description

This function draws symbols on a plot. One of six symbols; *circles*, *squares*, *rectangles*, *stars*, *thermometers*, and *boxplots*, can be plotted at a specified set of x and y coordinates. Specific aspects of the symbols, such as relative size, can be customized by additional parameters.

Usage

```
symbols(x, y = NULL, circles, squares, rectangles, stars,
        thermometers, boxplots, inches = TRUE, add = FALSE,
        fg = par("col"), bg = NA, xlab = NULL, ylab = NULL, main = NULL,
        xlim = NULL, ylim = NULL, ...)
```

Arguments

`x, y`	the x and y co-ordinates for the centres of the symbols. They can be specified in any way which is accepted by `xy.coords`.
`circles`	a vector giving the radii of the circles.
`squares`	a vector giving the length of the sides of the squares.
`rectangles`	a matrix with two columns. The first column gives widths and the second the heights of rectangles.
`stars`	a matrix with three or more columns giving the lengths of the rays from the center of the stars. `NA` values are replaced by zeroes.
`thermometers`	a matrix with three or four columns. The first two columns give the width and height of the thermometer symbols. If there are three columns, the third is taken as a proportion: the thermometers are filled (using colour `fg`) from their base to this proportion of their height. If there are four columns, the third and fourth columns are taken as proportions and the thermometers are filled between these two proportions of their heights. The part of the box not filled in `fg` will be filled in the background colour (default transparent) given by `bg`.
`boxplots`	a matrix with five columns. The first two columns give the width and height of the boxes, the next two columns give the lengths of the lower and upper whiskers and the fifth the proportion (with a warning if not in [0,1]) of the way up the box that the median line is drawn.
`inches`	`TRUE`, `FALSE` or a positive number. See Details.
`add`	if `add` is `TRUE`, the symbols are added to an existing plot, otherwise a new plot is created.
`fg`	colour(s) the symbols are to be drawn in.
`bg`	if specified, the symbols are filled with colour(s), the vector `bg` being recycled to the number of symbols. The default is to leave the symbols unfilled.

xlab	the x label of the plot if add is not true. Defaults to the deparsed expression used for x.
ylab	the y label of the plot. Unused if add = TRUE.
main	a main title for the plot. Unused if add = TRUE.
xlim	numeric vector of length 2 giving the x limits for the plot. Unused if add = TRUE.
ylim	numeric vector of length 2 giving the y limits for the plot. Unused if add = TRUE.
...	graphics parameters can also be passed to this function, as can the plot aspect ratio asp (see plot.window).

Details

Observations which have missing coordinates or missing size parameters are not plotted. The exception to this is *stars*. In that case, the length of any ray which is NA is reset to zero.

Argument inches controls the sizes of the symbols. If TRUE (the default), the symbols are scaled so that the largest dimension of any symbol is one inch. If a positive number is given the symbols are scaled to make largest dimension this size in inches (so TRUE and 1 are equivalent). If inches is FALSE, the units are taken to be those of the appropriate axes. (For circles, squares and stars the units of the x axis are used. For boxplots, the lengths of the whiskers are regarded as dimensions alongside width and height when scaling by inches, and are otherwise interpreted in the units of the y axis.)

Circles of radius zero are plotted at radius one pixel (which is device-dependent).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

W. S. Cleveland (1985) *The Elements of Graphing Data.* Monterey, California: Wadsworth.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

stars for drawing *stars* with a bit more flexibility. If are about doing "bubble plots" by symbols(*,circles=*), you should *really* consider using sunflowerplot instead.

Examples

```
x <- 1:10
y <- sort(10*runif(10))
z <- runif(10)
z3 <- cbind(z, 2*runif(10), runif(10))
symbols(x, y, thermometers=cbind(.5, 1, z), inches=.5, fg = 1:10)
symbols(x, y, thermometers = z3, inches=FALSE)
text(x,y, apply(format(round(z3, dig=2)), 1, paste, collapse = ","),
     adj = c(-.2,0), cex = .75, col = "purple", xpd=NA)

## Note that  example(trees)  shows more sensible plots!
N <- nrow(trees)
with(trees, {
## Girth is diameter in inches
symbols(Height, Volume, circles=Girth/24, inches=FALSE,
```

```
        main="Trees' Girth")# xlab and ylab automatically
## Colours too:
palette(rainbow(N, end = 0.9))
symbols(Height, Volume, circles=Girth/16, inches=FALSE, bg = 1:N,
        fg="gray30", main="symbols(*, circles=Girth/16, bg = 1:N)")
palette("default")
})
```

`text` *Add Text to a Plot*

Description

`text` draws the strings given in the vector `labels` at the coordinates given by `x` and `y`. `y` may be missing since `xy.coords(x,y)` is used for construction of the coordinates.

Usage

```
text(x, ...)

## Default S3 method:
text (x, y = NULL, labels = seq(along = x), adj = NULL,
     pos = NULL, offset = 0.5, vfont = NULL,
     cex = 1, col = NULL, font = NULL, ...)
```

Arguments

`x, y`	numeric vectors of coordinates where the text `labels` should be written. If the length of `x` and `y` differs, the shorter one is recycled.
`labels`	one or more character strings or expressions specifying the *text* to be written. An attempt is made to coerce other vectors (and factors) to character, and other language objects (names and calls) to expressions. If `labels` is longer than `x` and `y`, the coordinates are recycled to the length of `labels`.
`adj`	one or two values in $[0, 1]$ which specify the x (and optionally y) adjustment of the labels. On most devices values outside that interval will also work.
`pos`	a position specifier for the text. If specified this overrides any `adj` value given. Values of `1`, `2`, `3` and `4`, respectively indicate positions below, to the left of, above and to the right of the specified coordinates.
`offset`	when `pos` is specified, this value gives the offset of the label from the specified coordinate in fractions of a character width.
`vfont`	if a character vector of length 2 is specified, then Hershey vector fonts are used. The first element of the vector selects a typeface and the second element selects a style.
`cex`	numeric **c**haracter **ex**pansion factor; multiplied by `par("cex")` yields the final character size. `NULL` and `NA` are equivalent to `1.0`.
`col, font`	the color and font to be used, possibly vectors. These default to the values of the global graphical parameters in `par()`.
`...`	further graphical parameters (from `par`), such as `family` and `xpd`.

Details

`labels` must be of type `character` or `expression` (or be coercible to such a type). In the latter case, quite a bit of mathematical notation is available such as sub- and superscripts, greek letters, fractions, etc.

`adj` allows *adj*ustment of the text with respect to `(x,y)`. Values of 0, 0.5, and 1 specify left/bottom, middle and right/top, respectively. The default is for centered text, i.e., `adj = c(0.5, 0.5)`. Accurate vertical centering needs character metric information on individual characters, which is only available on some devices.

The `pos` and `offset` arguments can be used in conjunction with values returned by `identify` to recreate an interactively labelled plot.

Text can be rotated by using graphical parameters `srt` (see `par`); this rotates about the centre set by `adj`.

Graphical parameters `col`, `cex` and `font` can be vectors and will then be applied cyclically to the `labels` (and extra values will be ignored). `NA` values of `font` are replaced by `par("font")`.

Labels whose `x`, `y`, `labels`, `cex` or `col` value is `NA` are omitted from the plot.

Euro symbol

The Euro symbol was introduced relatively recently, and may not be available in older fonts. In recent versions of Adobe symbol fonts it is character 160, so `text(x, y, "\xA0", font = 5)` will work. People using Western European locales on Unix-alikes can probably select ISO-8895-15 (Latin-9) which has the Euro as character 165: this can also be used for `postscript` and `pdf`.

The Euro should be rendered correctly by `X11` in UTF-8 locales, but the corresponding single-byte encoding in `postscript` and `pdf` will be to be selected as `ISOLatin9.enc`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

`mtext`, `title`, `Hershey` for details on Hershey vector fonts, `plotmath` for details and more examples on mathematical annotation.

Examples

```
plot(-1:1,-1:1, type = "n", xlab = "Re", ylab = "Im")
K <- 16; text(exp(1i * 2 * pi * (1:K) / K), col = 2)

## The following two examples use latin1 characters: these may not
## appear correctly (or be omitted entirely).
plot(1:10, 1:10, main = "text(...) examples\n~~~~~~~~~~~~~~~",
     sub = "R is GNU ©, but not ® ...")
mtext("«Latin-1 accented chars»: éè øØ å<Å æ<Æ", side=3)
points(c(6,2), c(2,1), pch = 3, cex = 4, col = "red")
text(6, 2, "the text is CENTERED around (x,y) = (6,2) by default",
     cex = .8)
text(2, 1, "or Left/Bottom - JUSTIFIED at (2,1) by 'adj = c(0,0)'",
     adj = c(0,0))
```

```
text(4, 9, expression(hat(beta) == (X^t * X)^{-1} * X^t * y))
text(4, 8.4, "expression(hat(beta) == (X^t * X)^{-1} * X^t * y)", cex = .75)
text(4, 7, expression(bar(x) == sum(frac(x[i], n), i==1, n)))

## Two more latin1 examples
text(5,10.2,
     "Le français, c'est façile: Règles, Liberté, Egalité, Fraternité...")
text(5,9.8, "Jetz no chli züritüütsch: (noch ein bißchen Zürcher deutsch)")
```

`title` *Plot Annotation*

Description

This function can be used to add labels to a plot. Its first four principal arguments can also be used as arguments in most high-level plotting functions. They must be of type `character` or `expression`. In the latter case, quite a bit of mathematical notation is available such as sub- and superscripts, greek letters, fractions, etc.

Usage

```
title(main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
      line = NA, outer = FALSE, ...)
```

Arguments

`main`	The main title (on top) using font and size (character expansion) `par("font.main")` and color `par("col.main")`.
`sub`	Sub-title (at bottom) using font and size `par("font.sub")` and color `par("col.sub")`.
`xlab`	X axis label using font and character expansion `par("font.lab")` and color `par("col.lab")`.
`ylab`	Y axis label, same font attributes as `xlab`.
`line`	specifying a value for `line` overrides the default placement of labels, and places them this many lines outwards from the plot edge.
`outer`	a logical value. If `TRUE`, the titles are placed in the outer margins of the plot.
`...`	further graphical parameters from `par`. Use e.g., `col.main` or `cex.sub` instead of just `col` or `cex`. `xpd` can be used to set the clipping region: this defaults to the figure region unless `outer = TRUE`, otherwise the device region and can only be increased. `mgp` controls the default placing of the axis titles.

Details

The labels passed to `title` can be a character vector or a language object (names, calls or expressions), or a list containing the string to be plotted, and a selection of the optional modifying graphical parameters `cex=`, `col=` and `font=`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`mtext`, `text`; `plotmath` for details on mathematical annotation.

Examples

```
plot(cars, main = "") # here, could use main directly
title(main = "Stopping Distance versus Speed")

plot(cars, main = "")
title(main = list("Stopping Distance versus Speed", cex=1.5,
                  col="red", font=3))

## Specifying "..." :
plot(1, col.axis = "sky blue", col.lab = "thistle")
title("Main Title", sub = "sub title",
      cex.main = 2,   font.main= 4, col.main= "blue",
      cex.sub = 0.75, font.sub = 3, col.sub = "red")

x <- seq(-4, 4, len = 101)
y <- cbind(sin(x), cos(x))
matplot(x, y, type = "l", xaxt = "n",
        main = expression(paste(plain(sin) * phi, "  and  ",
                                plain(cos) * phi)),
        ylab = expression("sin" * phi, "cos" * phi), # only 1st is taken
        xlab = expression(paste("Phase Angle ", phi)),
        col.main = "blue")
axis(1, at = c(-pi, -pi/2, 0, pi/2, pi),
     labels = expression(-pi, -pi/2, 0, pi/2, pi))
abline(h = 0, v = pi/2 * c(-1,1), lty = 2, lwd = .1, col = "gray70")
```

`units` *Graphical Units*

Description

`xinch` and `yinch` convert the specified number of inches given as their arguments into the correct units for plotting with graphics functions. Usually, this only makes sense when normal coordinates are used, i.e., *no* `log` scale (see the `log` argument to `par`).

`xyinch` does the same for a pair of numbers `xy`, simultaneously.

Usage

```
xinch(x = 1, warn.log = TRUE)
yinch(y = 1, warn.log = TRUE)
xyinch(xy = 1, warn.log = TRUE)
```

Arguments

`x,y`	numeric vector
`xy`	numeric of length 1 or 2.
`warn.log`	logical; if `TRUE`, a warning is printed in case of active log scale.

Examples

```
all(c(xinch(),yinch()) == xyinch()) # TRUE
xyinch()
xyinch #- to see that is really   delta{"usr"} / "pin"

## plot labels offset 0.12 inches to the right
## of plotted symbols in a plot
with(mtcars, {
    plot(mpg, disp, pch=19, main= "Motor Trend Cars")
    text(mpg + xinch(0.12), disp, row.names(mtcars),
         adj = 0, cex = .7, col = 'blue')
    })
```

Chapter 5

The `grid` package

`grid-package` *The Grid Graphics Package*

Description

A rewrite of the graphics layout capabilities, plus some support for interaction.

Details

This package contains a graphics system which supplements old-style S graphics.

Further information is available in the following vignettes:

`displaylist`	Display Lists in grid (source, pdf)
`frame`	Frames and packing grobs (source, pdf)
`grid`	Introduction to grid (source, pdf)
`grobs`	Working with grid grobs (source, pdf)
`interactive`	Editing grid Graphics (source, pdf)
`locndimn`	Locations versus Dimensions (source, pdf)
`moveline`	Demonstrating move-to and line-to (source, pdf)
`nonfinite`	Non-finite values (source, pdf)
`plotexample`	Writing grid Code (source, pdf)
`rotated`	Rotated Viewports (source, pdf)
`saveload`	Persistent representations (source, pdf)
`sharing`	Modifying multiple grobs simultaneously (source, pdf)
`viewports`	Working with viewports (source, pdf)

For a complete list of functions with individual help pages, use `library(help="grid")`.

Author(s)

Paul Murrell ⟨paul@stat.auckland.ac.nz⟩

Maintainer: R Core Team ⟨R-core@r-project.org⟩

References

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

absolute.size *Absolute Size of a Grob*

Description

This function converts a unit object into absolute units. Absolute units are unaffected, but non-absolute units are converted into "null" units.

Usage

```
absolute.size(unit)
```

Arguments

unit	An object of class "unit".

Details

Absolute units are things like "inches", "cm", and "lines". Non-absolute units are "npc" and "native".

This function is designed to be used in widthDetails and heightDetails methods.

Value

An object of class "unit".

Author(s)

Paul Murrell

See Also

widthDetails and heightDetails methods.

arrow *Describe arrows to add to a line.*

Description

Produces a description of what arrows to add to a line. The result can be passed to a function that draws a line, e.g., grid.lines.

Usage

```
arrow(angle = 30, length = unit(0.25, "inches"),
      ends = "last", type = "open")
```

Arguments

angle	The angle of the arrow head in degrees (smaller numbers produce narrower, pointier arrows). Essentially describes the width of the arrow head.
length	A unit specifying the length of the arrow head (from tip to base).
ends	One of "last", "first", or "both", indicating which ends of the line to draw arrow heads.
type	One of "open" or "closed" indicating whether the arrow head should be a closed triangle.

Examples

```
arrow()
```

---

convertNative *Convert a Unit Object to Native units*

Description

This function is deprecated in grid version 0.8 and will be made defunct in grid version 1.9

You should use the convertUnit() function or one of its close allies instead.

This function returns a numeric vector containing the specified x or y locations or dimensions, converted to "user" or "data" units, relative to the current viewport.

Usage

```
convertNative(unit, dimension="x", type="location")
```

Arguments

unit	A unit object.
dimension	Either "x" or "y".
type	Either "location" or "dimension".

Value

A numeric vector.

WARNING

If you draw objects based on output from these conversion functions, then resize your device, the objects will be drawn incorrectly – the base R display list will not recalculate these conversions. This means that you can only rely on the results of these calculations if the size of your device is fixed.

Author(s)

Paul Murrell

See Also

grid.convert, unit

Examples

```
grid.newpage()
pushViewport(viewport(width=unit(.5, "npc"),
                      height=unit(.5, "npc")))
grid.rect()
w <- convertNative(unit(1, "inches"))
h <- convertNative(unit(1, "inches"), "y")
# This rectangle starts off life as 1in square, but if you
# resize the device it will no longer be 1in square
grid.rect(width=unit(w, "native"), height=unit(h, "native"),
          gp=gpar(col="red"))
popViewport(1)

# How to use grid.convert(), etc instead
convertNative(unit(1, "inches")) ==
  convertX(unit(1, "inches"), "native", valueOnly=TRUE)
convertNative(unit(1, "inches"), "y", "dimension") ==
  convertHeight(unit(1, "inches"), "native", valueOnly=TRUE)
```

---

dataViewport *Create a Viewport with Scales based on Data*

Description

This is a convenience function for producing a viewport with x- and/or y-scales based on numeric values passed to the function.

Usage

```
dataViewport(xData = NULL, yData = NULL, xscale = NULL,
             yscale = NULL, extension = 0.05, ...)
```

Arguments

xData	A numeric vector of data.
yData	A numeric vector of data.
xscale	A numeric vector (length 2).
yscale	A numeric vector (length 2).
extension	A numeric. If length greater than 1, then first value is used to extend the xscale and second value is used to extend the yscale.
...	All other arguments will be passed to a call to the viewport() function.

Details

If xscale is not specified then the values in x are used to generate an x-scale based on the range of x, extended by the proportion specified in extension. Similarly for the y-scale.

Value

A grid viewport object.

Author(s)

Paul Murrell

See Also

`viewport` and `plotViewport`.

---

drawDetails *Customising grid Drawing*

Description

These generic hook functions are called whenever a grid grob is drawn. They provide an opportunity for customising the drawing of a new class derived from grob (or gTree).

Usage

```
drawDetails(x, recording)
draw.details(x, recording)
preDrawDetails(x)
postDrawDetails(x)
```

Arguments

`x`	A grid grob.
`recording`	A logical value indicating whether a grob is being added to the display list or redrawn from the display list.

Details

These functions are called by the `grid.draw` methods for grobs and gTrees.

`preDrawDetails` is called first during the drawing of a grob. This is where any additional viewports should be pushed (see, for example, `grid:::preDrawDetails.frame`). Note that the default behaviour for grobs is to push any viewports in the `vp` slot, and for gTrees is to also push and up any viewports in the `childrenvp` slot so there is typically nothing to do here.

`drawDetails` is called next and is where any additional calculations and graphical output should occur (see, for example, `grid:::drawDetails.xaxis`. Note that the default behaviour for gTrees is to draw all grobs in the `children` slot so there is typically nothing to do here.

`postDrawDetails` is called last and should reverse anything done in `preDrawDetails` (i.e., pop or up any viewports that were pushed; again, see, for example, `grid:::postDrawDetails.frame`). Note that the default behaviour for grobs is to pop any viewports that were pushed so there is typically nothing to do here.

Note that `preDrawDetails` and `postDrawDetails` are also called in the calculation of `"grobwidth"` and `"grobheight"` units.

Value

None of these functions are expected to return a value.

Author(s)

Paul Murrell

See Also

`grid.draw`

`editDetails` *Customising grid Editing*

Description

This generic hook function is called whenever a grid grob is edited via `grid.edit` or `editGrob`. This provides an opportunity for customising the editing of a new class derived from grob (or gTree).

Usage

```
editDetails(x, specs)
```

Arguments

`x`	A grid grob.
`specs`	A list of named elements. The names indicate the grob slots to modify and the values are the new values for the slots.

Details

This function is called by `grid.edit` and `editGrob`. A method should be written for classes derived from grob or gTree if a change in a slot has an effect on other slots in the grob or children of a gTree (e.g., see `grid:::editDetails.xaxis`).

Note that the slot already has the new value.

Value

The function MUST return the modified grob.

Author(s)

Paul Murrell

See Also

`grid.edit`

gEdit *Create and Apply Edit Objects*

Description

The functions gEdit and gEditList create objects representing an edit operation (essentially a list of arguments to editGrob).

The functions applyEdit and applyEdits apply one or more edit operations to a graphical object.

These functions are most useful for developers creating new graphical functions and objects.

Usage

```
gEdit(...)
gEditList(...)
applyEdit(x, edit)
applyEdits(x, edits)
```

Arguments

...	one or more arguments to the editGrob function (for gEdit) or one or more "gEdit" objects (for gEditList).
x	a grob (grid graphical object).
edit	a "gEdit" object.
edits	either a "gEdit" object or a "gEditList" object.

Value

gEdit returns an object of class "gEdit".

gEditList returns an object of class "gEditList".

applyEdit and applyEditList return the modified grob.

Author(s)

Paul Murrell

See Also

grob editGrob

Examples

```
grid.rect(gp=gpar(col="red"))
# same thing, but more verbose
grid.draw(applyEdit(rectGrob(), gEdit(gp=gpar(col="red"))))
```

getNames *List the names of grobs on the display list*

Description

Returns a character vector containing the names of all top-level grobs on the display list.

Usage

```
getNames()
```

Value

A character vector.

Author(s)

Paul Murrell

Examples

```
grid.grill()
getNames()
```

gpar *Handling Grid Graphical Parameters*

Description

`gpar()` should be used to create a set of graphical parameter settings. It returns an object of class `"gpar"`. This is basically a list of name-value pairs.

`get.gpar()` can be used to query the current graphical parameter settings.

Usage

```
gpar(...)
get.gpar(names = NULL)
```

Arguments

`...`	Any number of named arguments.
`names`	A character vector of valid graphical parameter names.

Details

All grid viewports and (predefined) graphical objects have a slot called `gp`, which contains a `"gpar"` object. When a viewport is pushed onto the viewport stack and when a graphical object is drawn, the settings in the `"gpar"` object are enforced. In this way, the graphical output is modified by the `gp` settings until the graphical object has finished drawing, or until the viewport is popped off the viewport stack, or until some other viewport or graphical object is pushed or begins drawing.

Valid parameter names are:

col	Colour for lines and borders.
fill	Colour for filling rectangles, polygons, ...
alpha	Alpha channel for transparency
lty	Line type
lwd	Line width
lex	Multiplier applied to line width
lineend	Line end style (round, butt, square)
linejoin	Line join style (round, mitre, bevel)
linemitre	Line mitre limit (number greater than 1)
fontsize	The size of text (in points)
cex	Multiplier applied to fontsize
fontfamily	The font family
fontface	The font face (bold, italic, ...)
lineheight	The height of a line as a multiple of the size of text
font	Font face (alias for fontface; for backward compatibility)

The `alpha` setting is combined with the alpha channel for individual colours by multiplying (with both alpha settings normalised to the range 0 to 1).

The size of text is `fontsize*cex`. The size of a line is `fontsize*cex*lineheight`.

The `cex` setting is cumulative; if a viewport is pushed with a `cex` of 0.5 then another viewport is pushed with a `cex` of 0.5, the effective `cex` is 0.25.

The `alpha` and `lex` settings are also cumulative.

Changes to the `fontfamily` may be ignored by some devices, but is supported by PostScript, PDF, X11, Windows, and Quartz. The `fontfamily` may be used to specify one of the Hershey Font families (e.g., `HersheySerif`) and this specification will be honoured on all devices.

The specification of `fontface` can be an integer or a string. If an integer, then it follows the R base graphics standard: 1 = plain, 2 = bold, 3 = italic, 4 = bold italic. If a string, then valid values are: `"plain"`, `"bold"`, `"italic"`, `"oblique"`, and `"bold.italic"`. For the special case of the HersheySerif font family, `"cyrillic"`, `"cyrillic.oblique"`, and `"EUC"` are also available.

Specifying the value `NULL` for a parameter is the same as not specifying any value for that parameter, except for `col` and `fill`, where `NULL` indicates not to draw a border or not to fill an area (respectively).

All parameter values can be vectors of multiple values. (This will not always make sense – for example, viewports will only take notice of the first parameter value.)

The `gamma` parameter is deprecated.

`get.gpar()` returns all current graphical parameter settings.

Value

An object of class `"gpar"`.

Author(s)

Paul Murrell

See Also

`Hershey`.

Examples

```
gp <- get.gpar()
utils::str(gp)
## These *do* nothing but produce a "gpar" object:
gpar(col = "red")
gpar(col = "blue", lty = "solid", lwd = 3, fontsize = 16)
get.gpar(c("col", "lty"))
grid.newpage()
vp <- viewport(w = .8, h = .8, gp = gpar(col="blue"))
grid.draw(gTree(children=gList(rectGrob(gp = gpar(col="red")),
                     textGrob(paste("The rect is its own colour (red)",
                                    "but this text is the colour",
                                    "set by the gTree (green)",
                                    sep = "\n"))),
      gp = gpar(col="green"), vp = vp))
grid.text("This text is the colour set by the viewport (blue)",
          y = 1, just = c("center", "bottom"),
          gp = gpar(fontsize=20), vp = vp)
grid.newpage()
## example with multiple values for a parameter
pushViewport(viewport())
grid.points(1:10/11, 1:10/11, gp = gpar(col=1:10))
popViewport()
```

---

gPath *Concatenate Grob Names*

Description

This function can be used to generate a grob path for use in `grid.edit` and friends.

A grob path is a list of nested grob names.

Usage

```
gPath(...)
```

Arguments

`...`	Character values which are grob names.

Details

Grob names must only be unique amongst grobs which share the same parent in a gTree.

This function can be used to generate a specification for a grob that includes the grob's parent's name (and the name of its parent and so on).

For interactive use, it is possible to directly specify a path, but it is strongly recommended that this function is used otherwise in case the path separator is changed in future versions of grid.

Value

A `gPath` object.

See Also

grob, editGrob, addGrob, removeGrob, getGrob, setGrob

Examples

```
gPath("g1", "g2")
```

Grid *Grid Graphics*

Description

General information about the grid graphics package.

Details

Grid graphics provides an alternative to the standard R graphics. The user is able to define arbitrary rectangular regions (called *viewports*) on the graphics device and define a number of coordinate systems for each region. Drawing can be specified to occur in any viewport using any of the available coordinate systems.

Grid graphics and standard R graphics do not mix!

Type `library(help = grid)` to see a list of (public) Grid graphics functions.

Author(s)

Paul Murrell

See Also

viewport, grid.layout, and unit.

Examples

```
## Diagram of a simple layout
grid.show.layout(grid.layout(4,2,
                     heights=unit(rep(1, 4),
                                  c("lines", "lines", "lines", "null")),
                     widths=unit(c(1, 1), "inches")))
## Diagram of a sample viewport
grid.show.viewport(viewport(x=0.6, y=0.6,
                            w=unit(1, "inches"), h=unit(1, "inches")))
## A flash plotting example
grid.multipanel(vp=viewport(0.5, 0.5, 0.8, 0.8))
```

`Grid Viewports` *Create a Grid Viewport*

Description

These functions create viewports, which describe rectangular regions on a graphics device and define a number of coordinate systems within those regions.

Usage

```
viewport(x = unit(0.5, "npc"), y = unit(0.5, "npc"),
         width = unit(1, "npc"), height = unit(1, "npc"),
         default.units = "npc", just = "centre",
         gp = gpar(), clip = "inherit",
         xscale = c(0, 1), yscale = c(0, 1),
         angle = 0,
         layout = NULL,
         layout.pos.row = NULL, layout.pos.col = NULL,
         name = NULL)
vpList(...)
vpStack(...)
vpTree(parent, children)
```

Arguments

`x`	A numeric vector or unit object specifying x-location.
`y`	A numeric vector or unit object specifying y-location.
`width`	A numeric vector or unit object specifying width.
`height`	A numeric vector or unit object specifying height.
`default.units`	A string indicating the default units to use if `x`, `y`, `width`, or `height` are only given as numeric vectors.
`just`	A string or numeric vector specifying the justification of the viewport relative to its (x, y) location. If there are two values, the first value specifes horizontal justification and the second value specifies vertical justification. Possible string values are: `"left"`, `"right"`, `"centre"`, `"center"`, `"bottom"`, and `"top"`. For numeric values, 0 means left alignment and 1 means right alignment.
`gp`	An object of class `gpar`, typically the output from a call to the function `gpar`. This is basically a list of graphical parameter settings.
`clip`	One of `"on"`, `"inherit"`, or `"off"`, indicating whether to clip to the extent of this viewport, inherit the clipping region from the parent viewport, or turn clipping off altogether. For back-compatibility, a logical value of `TRUE` corresponds to `"on"` and `FALSE` corresponds to `"inherit"`.
`xscale`	A numeric vector of length two indicating the minimum and maximum on the x-scale.
`yscale`	A numeric vector of length two indicating the minimum and maximum on the y-scale.

angle	A numeric value indicating the angle of rotation of the viewport. Positive values indicate the amount of rotation, in degrees, anitclockwise from the positive x-axis.
layout	A Grid layout object which splits the viewport into subregions.
layout.pos.row	A numeric vector giving the rows occupied by this viewport in its parent's layout.
layout.pos.col	A numeric vector giving the columns occupied by this viewport in its parent's layout.
name	A character value to uniquely identify the viewport once it has been pushed onto the viewport tree.
...	Any number of grid viewport objects.
parent	A grid viewport object.
children	A vpList object.

Details

The location and size of a viewport are relative to the coordinate systems defined by the viewport's parent (either a graphical device or another viewport). The location and size can be specified in a very flexible way by specifying them with unit objects. When specifying the location of a viewport, specifying both `layout.pos.row` and `layout.pos.col` as `NULL` indicates that the viewport ignores its parent's layout and specifies its own location and size (via its `locn`). If only one of `layout.pos.row` and `layout.pos.col` is `NULL`, this means occupy ALL of the appropriate row(s)/column(s). For example, `layout.pos.row = 1` and `layout.pos.col = NULL` means occupy all of row 1. Specifying non-`NULL` values for both `layout.pos.row` and `layout.pos.col` means occupy the intersection of the appropriate rows and columns. If a vector of length two is specified for `layout.pos.row` or `layout.pos.col`, this indicates a range of rows or columns to occupy. For example, `layout.pos.row = c(1, 3)` and `layout.pos.col = c(2, 4)` means occupy cells in the intersection of rows 1, 2, and 3, and columns, 2, 3, and 4.

Clipping obeys only the most recent viewport clip setting. For example, if you clip to viewport1, then clip to viewport2, the clipping region is determined wholly by viewport2, the size and shape of viewport1 is irrelevant (until viewport2 is popped of course).

If a viewport is rotated (because of its own `angle` setting or because it is within another viewport which is rotated) then the `clip` flag is ignored.

Viewport names need not be unique. When pushed, viewports sharing the same parent must have unique names, which means that if you push a viewport with the same name as an existing viewport, the existing viewport will be replaced in the viewport tree. A viewport name can be any string, but grid uses the reserved name `"ROOT"` for the top-level viewport. Also, when specifying a viewport name in `downViewport` and `seekViewport`, it is possible to provide a viewport path, which consists of several names concatenated using the separator (currently `::`). Consequently, it is not advisable to use this separator in viewport names.

The viewports in a `vpList` are pushed in parallel. The viewports in a `vpStack` are pushed in series. When a `vpTree` is pushed, the parent is pushed first, then the children are pushed in parallel.

Value

An R object of class `viewport`.

Author(s)

Paul Murrell

See Also

Grid, pushViewport, popViewport, downViewport, seekViewport, upViewport, unit, grid.layout, grid.show.layout.

Examples

```
# Diagram of a sample viewport
grid.show.viewport(viewport(x=0.6, y=0.6,
                   w=unit(1, "inches"), h=unit(1, "inches")))
# Demonstrate viewport clipping
clip.demo <- function(i, j, clip1, clip2) {
  pushViewport(viewport(layout.pos.col=i,
                        layout.pos.row=j))
  pushViewport(viewport(width=0.6, height=0.6, clip=clip1))
  grid.rect(gp=gpar(fill="white"))
  grid.circle(r=0.55, gp=gpar(col="red", fill="pink"))
  popViewport()
  pushViewport(viewport(width=0.6, height=0.6, clip=clip2))
  grid.polygon(x=c(0.5, 1.1, 0.6, 1.1, 0.5, -0.1, 0.4, -0.1),
               y=c(0.6, 1.1, 0.5, -0.1, 0.4, -0.1, 0.5, 1.1),
               gp=gpar(col="blue", fill="light blue"))
  popViewport(2)
}

grid.newpage()
grid.rect(gp=gpar(fill="grey"))
pushViewport(viewport(layout=grid.layout(2, 2)))
clip.demo(1, 1, FALSE, FALSE)
clip.demo(1, 2, TRUE, FALSE)
clip.demo(2, 1, FALSE, TRUE)
clip.demo(2, 2, TRUE, TRUE)
popViewport()
# Demonstrate turning clipping off
grid.newpage()
pushViewport(viewport(w=.5, h=.5, clip="on"))
grid.rect()
grid.circle(r=.6, gp=gpar(lwd=10))
pushViewport(viewport(clip="inherit"))
grid.circle(r=.6, gp=gpar(lwd=5, col="grey"))
pushViewport(viewport(clip="off"))
grid.circle(r=.6)
popViewport(3)
# Demonstrate vpList, vpStack, and vpTree
grid.newpage()
tree <- vpTree(viewport(w=0.8, h=0.8, name="A"),
               vpList(vpStack(viewport(x=0.1, y=0.1, w=0.5, h=0.5,
                                       just=c("left", "bottom"), name="B"),
                              viewport(x=0.1, y=0.1, w=0.5, h=0.5,
                                       just=c("left", "bottom"), name="C"),
                              viewport(x=0.1, y=0.1, w=0.5, h=0.5,
                                       just=c("left", "bottom"), name="D")),
                      viewport(x=0.5, w=0.4, h=0.9,
```

```
                                    just="left", name="E")))
pushViewport(tree)
for (i in LETTERS[1:5]) {
  seekViewport(i)
  grid.rect()
  grid.text(current.vpTree(FALSE),
            x=unit(1, "mm"), y=unit(1, "npc") - unit(1, "mm"),
            just=c("left", "top"),
            gp=gpar(fontsize=8))
}
```

`grid.add` *Add a Grid Graphical Object*

Description

Add a grob to a gTree or a descendant of a gTree.

Usage

```
grid.add(gPath, child, strict = FALSE, grep = FALSE,
         global = FALSE, allDevices = FALSE, redraw = TRUE)

addGrob(gTree, child, gPath = NULL, strict = FALSE, grep = FALSE,
        global = FALSE)

setChildren(x, children)
```

Arguments

`gTree, x`	A gTree object.
`gPath`	A gPath object. For `grid.add` this specifyies a gTree on the display list. For `addGrob` this specifies a descendant of the specified gTree.
`child`	A grob object.
`children`	A gList object.
`strict`	A boolean indicating whether the gPath must be matched exactly.
`grep`	A boolean indicating whether the `gPath` should be treated as a regular expression. Values are recycled across elements of the `gPath` (e.g., `c(TRUE, FALSE)` means that every odd element of the `gPath` will be treated as a regular expression).
`global`	A boolean indicating whether the function should affect just the first match of the `gPath`, or whether all matches should be affected.
`allDevices`	A boolean indicating whether all open devices should be searched for matches, or just the current device. NOT YET IMPLEMENTED.
`redraw`	A logical value to indicate whether to redraw the grob.

Details

addGrob copies the specified grob and returns a modified grob.

grid.add destructively modifies a grob on the display list. If redraw is TRUE it then redraws everything to reflect the change.

setChildren is a basic function for setting all children of a gTree at once (instead of repeated calls to addGrob).

Value

addGrob returns a grob object; grid.add returns NULL.

Author(s)

Paul Murrell

See Also

grob, getGrob, addGrob, removeGrob.

---

grid.arrows *Draw Arrows*

Description

Functions to create and draw arrows at either end of a line, or at either end of a line.to, lines, or segments grob.

These functions have been deprecated in favour of arrow arguments to the line-related primitives.

Usage

```
grid.arrows(x = c(0.25, 0.75), y = 0.5, default.units = "npc",
            grob = NULL,
            angle = 30, length = unit(0.25, "inches"),
            ends = "last", type = "open", name = NULL,
            gp = gpar(), draw = TRUE, vp = NULL)

arrowsGrob(x = c(0.25, 0.75), y = 0.5, default.units = "npc",
            grob = NULL,
            angle = 30, length = unit(0.25, "inches"),
            ends = "last", type = "open", name = NULL,
            gp = gpar(), vp = NULL)
```

Arguments

x	A numeric vector or unit object specifying x-values.
y	A numeric vector or unit object specifying y-values.
default.units	A string indicating the default units to use if x or y are only given as numeric vectors.
grob	A grob to add arrows to; currently can only be a line.to, lines, or segments grob.

angle	A numeric specifying (half) the width of the arrow head (in degrees).
length	A unit object specifying the length of the arrow head.
ends	One of "first", "last", or "both", indicating which end of the line to add arrow heads.
type	Either "open" or "closed" to indicate the type of arrow head.
name	A character identifier.
gp	An object of class gpar, typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
draw	A logical value indicating whether graphics output should be produced.
vp	A Grid viewport object (or NULL).

Details

Both functions create an arrows grob (a graphical object describing arrows), but only grid.arrows() draws the arrows (and then only if draw is TRUE).

If the grob argument is specified, this overrides any x and/or y arguments.

Value

An arrows grob. grid.arrows() returns the value invisibly.

Author(s)

Paul Murrell

See Also

Grid, viewport, grid.line.to, grid.lines, grid.segments

Examples

```
## Not run:
## to avoid lots of deprecation warnings
pushViewport(viewport(layout=grid.layout(2, 4)))
pushViewport(viewport(layout.pos.col=1,
                      layout.pos.row=1))
grid.rect(gp=gpar(col="grey"))
grid.arrows()
popViewport()
pushViewport(viewport(layout.pos.col=2,
                      layout.pos.row=1))
grid.rect(gp=gpar(col="grey"))
grid.arrows(angle=15, type="closed")
popViewport()
pushViewport(viewport(layout.pos.col=3,
                      layout.pos.row=1))
grid.rect(gp=gpar(col="grey"))
grid.arrows(angle=5, length=unit(0.1, "npc"),
            type="closed", gp=gpar(fill="white"))
popViewport()
pushViewport(viewport(layout.pos.col=4,
                      layout.pos.row=1))
grid.rect(gp=gpar(col="grey"))
```

```
grid.arrows(x=unit(0:80/100, "npc"),
            y=unit(1 - (0:80/100)^2, "npc"))
popViewport()
pushViewport(viewport(layout.pos.col=1,
                      layout.pos.row=2))
grid.rect(gp=gpar(col="grey"))
grid.arrows(ends="both")
popViewport()
pushViewport(viewport(layout.pos.col=2,
                      layout.pos.row=2))
grid.rect(gp=gpar(col="grey"))
# Recycling arguments
grid.arrows(x=unit(1:10/11, "npc"), y=unit(1:3/4, "npc"))
popViewport()
pushViewport(viewport(layout.pos.col=3,
                      layout.pos.row=2))
grid.rect(gp=gpar(col="grey"))
# Drawing arrows on a segments grob
gs <- segmentsGrob(x0=unit(1:4/5, "npc"),
                   x1=unit(1:4/5, "npc"))
grid.arrows(grob=gs, length=unit(0.1, "npc"),
            type="closed", gp=gpar(fill="white"))
popViewport()
pushViewport(viewport(layout.pos.col=4,
                      layout.pos.row=2))
grid.rect(gp=gpar(col="grey"))
# Arrows on a lines grob
# Name these because going to grid.edit them later
gl <- linesGrob(name="curve", x=unit(0:80/100, "npc"),
                y=unit((0:80/100)^2, "npc"))
grid.arrows(name="arrowOnLine", grob=gl, angle=15, type="closed",
            gp=gpar(fill="black"))
popViewport()
pushViewport(viewport(layout.pos.col=1,
                      layout.pos.row=2))
grid.move.to(x=0.5, y=0.8)
popViewport()
pushViewport(viewport(layout.pos.col=4,
                      layout.pos.row=1))
# Arrows on a line.to grob
glt <- lineToGrob(x=0.5, y=0.2, gp=gpar(lwd=3))
grid.arrows(grob=glt, ends="first", gp=gpar(lwd=3))
popViewport(2)
grid.edit(gPath("arrowOnLine", "curve"), y=unit((0:80/100)^3, "npc"))
## End(Not run)
```

`grid.circle` *Draw a Circle*

Description

Functions to create and draw a circle.

Usage

```
grid.circle(x=0.5, y=0.5, r=0.5, default.units="npc", name=NULL,
            gp=gpar(), draw=TRUE, vp=NULL)
circleGrob(x=0.5, y=0.5, r=0.5, default.units="npc", name=NULL,
            gp=gpar(), vp=NULL)
```

Arguments

x	A numeric vector or unit object specifying x-locations.
y	A numeric vector or unit object specifying y-locations.
r	A numeric vector or unit object specifying radii.
default.units	A string indicating the default units to use if x, y, width, or height are only given as numeric vectors.
name	A character identifier.
gp	An object of class gpar, typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
draw	A logical value indicating whether graphics output should be produced.
vp	A Grid viewport object (or NULL).

Details

Both functions create a circle grob (a graphical object describing a circle), but only `grid.circle()` draws the circle (and then only if `draw` is `TRUE`).

The radius may be given in any units; if the units are *relative* (e.g., `"npc"` or `"native"`) then the radius will be different depending on whether it is interpreted as a width or as a height. In such cases, the smaller of these two values will be the result. To see the effect, type `grid.circle()` and adjust the size of the window.

Value

A circle grob. `grid.circle()` returns the value invisibly.

Warning

Negative values for the radius are silently converted to their absolute value.

Author(s)

Paul Murrell

See Also

Grid, `viewport`

grid.clip *Set the Clipping Region*

Description

These functions set the clipping region within the current viewport *without* altering the current coordinate system.

Usage

```
grid.clip(...)
clipGrob(x = unit(0.5, "npc"), y = unit(0.5, "npc"),
         width = unit(1, "npc"), height = unit(1, "npc"),
         just = "centre", hjust = NULL, vjust = NULL,
         default.units = "npc", name = NULL, vp = NULL)
```

Arguments

x	A numeric vector or unit object specifying x-location.
y	A numeric vector or unit object specifying y-location.
width	A numeric vector or unit object specifying width.
height	A numeric vector or unit object specifying height.
just	The justification of the clip rectangle relative to its (x, y) location. If there are two values, the first value specifes horizontal justification and the second value specifies vertical justification. Possible string values are: "left", "right", "centre", "center", "bottom", and "top". For numeric values, 0 means left alignment and 1 means right alignment.
hjust	A numeric vector specifying horizontal justification. If specified, overrides the just setting.
vjust	A numeric vector specifying vertical justification. If specified, overrides the just setting.
default.units	A string indicating the default units to use if x, y, width, or height are only given as numeric vectors.
name	A character identifier.
vp	A Grid viewport object (or NULL).
...	Arguments passed to clipGrob.

Details

Both functions create a clip rectangle (a graphical object describing a clip rectangle), but only grid.clip enforces the clipping.

Pushing or popping a viewport *always* overrides the clip region set by a clip grob, regardless of whether that viewport explicitly enforces a clipping region.

Value

clipGrob returns a clip grob.

Author(s)

Paul Murrell

See Also

Grid, `viewport`

Examples

```
# draw across entire viewport, but clipped
grid.clip(x = 0.3, width = 0.1)
grid.lines(gp=gpar(col="green", lwd=5))
# draw across entire viewport, but clipped (in different place)
grid.clip(x = 0.7, width = 0.1)
grid.lines(gp=gpar(col="red", lwd=5))
# Viewport sets new clip region
pushViewport(viewport(width=0.5, height=0.5, clip=TRUE))
grid.lines(gp=gpar(col="grey", lwd=3))
# Return to original viewport;  get
# clip region from previous grid.clip()
# (NOT from previous viewport clip region)
popViewport()
grid.lines(gp=gpar(col="black"))
```

`grid.collection` *Create a Coherent Group of Grid Graphical Objects*

Description

This function is deprecated; please use `gTree`.

This function creates a graphical object which contains several other graphical objects. When it is drawn, it draws all of its children.

It may be convenient to name the elements of the collection.

Usage

```
grid.collection(..., gp=gpar(), draw=TRUE, vp=NULL)
```

Arguments

`...`	Zero or more objects of class `"grob"`.
`gp`	An object of class `gpar`, typically the output from a call to the function `gpar`. This is basically a list of graphical parameter settings.
`draw`	A logical value to indicate whether to produce graphical output.
`vp`	A Grid viewport object (or NULL).

Value

A collection grob.

Author(s)

Paul Murrell

See Also

grid.grob.

---

grid.convert *Convert Between Different grid Coordinate Systems*

Description

These functions take a unit object and convert it to an equivalent unit object in a different coordinate system.

Usage

```
convertX(x, unitTo, valueOnly = FALSE)
convertY(x, unitTo, valueOnly = FALSE)
convertWidth(x, unitTo, valueOnly = FALSE)
convertHeight(x, unitTo, valueOnly = FALSE)
convertUnit(x, unitTo,
            axisFrom = "x", typeFrom = "location",
            axisTo = axisFrom, typeTo = typeFrom,
            valueOnly = FALSE)
grid.convertX(x, unitTo, valueOnly = FALSE)
grid.convertY(x, unitTo, valueOnly = FALSE)
grid.convertWidth(x, unitTo, valueOnly = FALSE)
grid.convertHeight(x, unitTo, valueOnly = FALSE)
grid.convert(x, unitTo,
            axisFrom = "x", typeFrom = "location",
            axisTo = axisFrom, typeTo = typeFrom,
            valueOnly = FALSE)
```

Arguments

x	A unit object.
unitTo	The coordinate system to convert the unit to. See the unit function for valid coordinate systems.
axisFrom	Either "x" or "y" to indicate whether the unit object represents a value in the x- or y-direction.
typeFrom	Either "location" or "dimension" to indicate whether the unit object represents a location or a length.
axisTo	Same as axisFrom, but applies to the unit object that is to be created.
typeTo	Same as typeFrom, but applies to the unit object that is to be created.
valueOnly	A logical indicating. If TRUE then the function does not return a unit object, but rather only the converted numeric values.

Details

The `convertUnit` function allows for general-purpose conversions. The other four functions are just more convenient front-ends to it for the most common conversions.

The conversions occur within the current viewport.

It is not currently possible to convert to all valid coordinate systems (e.g., "strwidth" or "grobwidth"). I'm not sure if all of these are impossible, they just seem implausible at this stage.

In normal usage of grid, these functions should not be necessary. If you want to express a location or dimension in inches rather than user coordinates then you should simply do something like `unit(1, "inches")` rather than something like `unit(0.134, "native")`.

In some cases, however, it is necessary for the user to perform calculations on a unit value and this function becomes necessary. In such cases, please take note of the warning below.

The grid.* versions are just previous incarnations which have been deprecated.

Value

A unit object in the specified coordinate system (unless `valueOnly` is `TRUE` in which case the returned value is a numeric).

Warning

The conversion is only valid for the current device size. If the device is resized then at least some conversions will become invalid. For example, suppose that I create a unit object as follows: `oneinch <- convertUnit(unit(1, "inches"), "native"`. Now if I resize the device, the unit object in oneinch no longer corresponds to a physical length of 1 inch.

Author(s)

Paul Murrell

See Also

`unit`

Examples

```
## A tautology
convertX(unit(1, "inches"), "inches")
## The physical units
convertX(unit(2.54, "cm"), "inches")
convertX(unit(25.4, "mm"), "inches")
convertX(unit(72.27, "points"), "inches")
convertX(unit(1/12*72.27, "picas"), "inches")
convertX(unit(72, "bigpts"), "inches")
convertX(unit(1157/1238*72.27, "dida"), "inches")
convertX(unit(1/12*1157/1238*72.27, "cicero"), "inches")
convertX(unit(65536*72.27, "scaledpts"), "inches")
convertX(unit(1/2.54, "inches"), "cm")
convertX(unit(1/25.4, "inches"), "mm")
convertX(unit(1/72.27, "inches"), "points")
convertX(unit(1/(1/12*72.27), "inches"), "picas")
convertX(unit(1/72, "inches"), "bigpts")
convertX(unit(1/(1157/1238*72.27), "inches"), "dida")
convertX(unit(1/(1/12*1157/1238*72.27), "inches"), "cicero")
```

```
convertX(unit(1/(65536*72.27), "inches"), "scaledpts")

pushViewport(viewport(width=unit(1, "inches"),
                      height=unit(2, "inches"),
                      xscale=c(0, 1),
                      yscale=c(1, 3)))
  ## Location versus dimension
  convertY(unit(2, "native"), "inches")
  convertHeight(unit(2, "native"), "inches")
  ## From "x" to "y" (the conversion is via "inches")
  convertUnit(unit(1, "native"), "native",
              axisFrom="x", axisTo="y")
  ## Convert several values at once
  convertX(unit(c(0.5, 2.54), c("npc", "cm")),
               c("inches", "native"))
popViewport()
## Convert a complex unit
convertX(unit(1, "strwidth", "Hello"), "native")
```

`grid.copy` *Make a Copy of a Grid Graphical Object*

Description

This function is redundant and will disappear in future versions.

Usage

```
grid.copy(grob)
```

Arguments

`grob`	A grob object.

Value

A copy of the grob object.

Author(s)

Paul Murrell

See Also

`grid.grob`.

grid.curve *Draw a Curve Between Locations*

Description

These functions create and draw a curve from one location to another.

Usage

```
grid.curve(...)
curveGrob(x1, y1, x2, y2, default.units = "npc",
          curvature = 1, angle = 90, ncp = 1, shape = 0.5,
          square = TRUE, squareShape = 1,
          inflect = FALSE, arrow = NULL, debug = FALSE,
          name = NULL, gp = gpar(), vp = NULL)
arcCurvature(theta)
```

Arguments

x1	A numeric vector or unit object specifying the x-location of the start point.
y1	A numeric vector or unit object specifying the y-location of the start point.
x2	A numeric vector or unit object specifying the x-location of the end point.
y2	A numeric vector or unit object specifying the y-location of the end point.
default.units	A string indicating the default units to use if x1, y1, x2 or y2 are only given as numeric values.
curvature	A numeric value giving the amount of curvature. Negative values produce left-hand curves, positive values produce right-hand curves, and zero produces a straight line.
angle	A numeric value between 0 and 180, giving an amount to skew the control points of the curve. Values less than 90 skew the curve towards the start point and values greater than 90 skew the curve towards the end point.
ncp	The number of control points used to draw the curve. More control points creates a smoother curve.
shape	A numeric vector of values between -1 and 1, which control the shape of the curve relative to its control points. See grid.xspline for more details.
square	A logical value that controls whether control points for the curve are created city-block fashion or obliquely. When ncp is 1 and angle is 90, this is typically TRUE, otherwise this should probably be set to FALSE (see Examples below).
squareShape	A shape value to control the behaviour of the curve relative to any additional control point that is inserted if square is TRUE.
inflect	A logical value specifying whether the curve should be cut in half and inverted (see Examples below).
arrow	A list describing arrow heads to place at either end of the curve, as produced by the arrow function.
debug	A logical value indicating whether debugging information should be drawn.

name	A character identifier.
gp	An object of class gpar, typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
vp	A Grid viewport object (or NULL).
...	Arguments to be passed to curveGrob.
theta	An angle (in degrees).

Details

Both functions create a curve grob (a graphical object describing an curve), but only grid.curve draws the curve.

The arcCurvature function can be used to calculate a curvature such that control points are generated on an arc corresponding to angle theta. This is typically used in conjunction with a large ncp to produce a curve corresponding to the desired arc.

Value

A grob object.

See Also

Grid, viewport, grid.xspline, arrow

Examples

```
curveTest <- function(i, j, ...) {
  pushViewport(viewport(layout.pos.col=j, layout.pos.row=i))
  do.call("grid.curve", c(list(x1=.25, y1=.25, x2=.75, y2=.75),
          if (is.null(...)) NULL else list(...)))
  grid.text(sub("list", "", deparse(substitute(list(...)))),
            y=unit(1, "npc"))
  popViewport()
}
# grid.newpage()
pushViewport(plotViewport(c(0, 0, 1, 0),
                          layout=grid.layout(2, 1, heights=c(2, 1))))
pushViewport(viewport(layout.pos.row=1,
                      layout=grid.layout(3, 3, respect=TRUE)))
curveTest(1, 1, NULL)
curveTest(1, 2, inflect=TRUE)
curveTest(1, 3, angle=135)
curveTest(2, 1, arrow=arrow())
curveTest(2, 2, ncp=8)
curveTest(2, 3, shape=0)
curveTest(3, 1, curvature=-1)
curveTest(3, 2, square=FALSE)
curveTest(3, 3, debug=TRUE)
popViewport()
pushViewport(viewport(layout.pos.row=2,
                      layout=grid.layout(3, 3)))
curveTest(1, 1, NULL)
curveTest(1, 2, inflect=TRUE)
curveTest(1, 3, angle=135)
curveTest(2, 1, arrow=arrow())
```

```
curveTest(2, 2, ncp=8)
curveTest(2, 3, shape=0)
curveTest(3, 1, curvature=-1)
curveTest(3, 2, square=FALSE)
curveTest(3, 3, debug=TRUE)
popViewport(2)
```

`grid.display.list` *Control the Grid Display List*

Description

Turn the Grid display list on or off.

Usage

```
grid.display.list(on=TRUE)
engine.display.list(on=TRUE)
```

Arguments

`on`	A logical value to indicate whether the display list should be on or off.

Details

All drawing and viewport-setting operations are (by default) recorded in the Grid display list. This allows redrawing to occur following an editing operation.

This display list could get very large so it may be useful to turn it off in some cases; this will of course disable redrawing.

All graphics output is also recorded on the main display list of the R graphics engine (by default). This supports redrawing following a device resize and allows copying between devices.

Turning off this display list means that grid will redraw from its own display list for device resizes and copies. This will be slower than using the graphics engine display list.

Value

None.

WARNING

Turning the display list on causes the display list to be erased!

Turning off both the grid display list and the graphics engine display list will result in no redrawing whatsoever.

Author(s)

Paul Murrell

grid.draw *Draw a grid grob*

Description

Produces graphical output from a graphical object.

Usage

```
grid.draw(x, recording=TRUE)
```

Arguments

x	An object of class "grob" or NULL.
recording	A logical value to indicate whether the drawing operation should be recorded on the Grid display list.

Details

This is a generic function with methods for grob and gTree objects.

The grob and gTree methods automatically push any viewports in a vp slot and automatically apply any gpar settings in a gp slot. In addition, the gTree method pushes and ups any viewports in a childrenvp slot and automatically calls grid.draw for any grobs in a children slot.

The methods for grob and gTree call the generic hook functions preDrawDetails, drawDetails, and postDrawDetails to allow classes derived from grob or gTree to perform additional viewport pushing/popping and produce additional output beyond the default behaviour for grobs and gTrees.

Value

None.

Author(s)

Paul Murrell

See Also

grob.

Examples

```
grid.newpage()
## Create a graphical object, but don't draw it
l <- linesGrob()
## Draw it
grid.draw(l)
```

`grid.edit` *Edit the Description of a Grid Graphical Object*

Description

Changes the value of one of the slots of a grob and redraws the grob.

Usage

```
grid.edit(gPath, ..., strict = FALSE, grep = FALSE, global = FALSE,
          allDevices = FALSE, redraw = TRUE)
editGrob(grob, gPath = NULL, ..., strict = FALSE, grep = FALSE,
         global = FALSE)
```

Arguments

`grob`	A grob object.
`...`	Zero or more named arguments specifying new slot values.
`gPath`	A gPath object. For `grid.edit` this specifyies a grob on the display list. For `editGrob` this specifies a descendant of the specified grob.
`strict`	A boolean indicating whether the gPath must be matched exactly.
`grep`	A boolean indicating whether the `gPath` should be treated as a regular expression. Values are recycled across elements of the `gPath` (e.g., `c(TRUE, FALSE)` means that every odd element of the `gPath` will be treated as a regular expression).
`global`	A boolean indicating whether the function should affect just the first match of the `gPath`, or whether all matches should be affected.
`allDevices`	A boolean indicating whether all open devices should be searched for matches, or just the current device. NOT YET IMPLEMENTED.
`redraw`	A logical value to indicate whether to redraw the grob.

Details

`editGrob` copies the specified grob and returns a modified grob.

`grid.edit` destructively modifies a grob on the display list. If `redraw` is `TRUE` it then redraws everything to reflect the change.

Both functions call `editDetails` to allow a grob to perform custom actions and `validDetails` to check that the modified grob is still coherent.

Value

`editGrob` returns a grob object; `grid.edit` returns `NULL`.

Author(s)

Paul Murrell

See Also

`grob`, `getGrob`, `addGrob`, `removeGrob`.

Examples

```
grid.newpage()
grid.xaxis(name = "xa", vp = viewport(width=.5, height=.5))
grid.edit("xa", gp = gpar(col="red"))
# won't work because no ticks (at is NULL)
try(grid.edit(gPath("xa", "ticks"), gp = gpar(col="green")))
grid.edit("xa", at = 1:4/5)
# Now it should work
try(grid.edit(gPath("xa", "ticks"), gp = gpar(col="green")))
```

---

grid.frame *Create a Frame for Packing Objects*

Description

These functions, together with `grid.pack`, `grid.place`, `packGrob`, and `placeGrob` are part of a GUI-builder-like interface to constructing graphical images. The idea is that you create a frame with this function then use `grid.pack` or whatever to pack/place objects into the frame.

Usage

```
grid.frame(layout=NULL, name=NULL, gp=gpar(), vp=NULL, draw=TRUE)
frameGrob(layout=NULL, name=NULL, gp=gpar(), vp=NULL)
```

Arguments

`layout`	A Grid layout, or NULL. This can be used to initialise the frame with a number of rows and columns, with initial widths and heights, etc.
`name`	A character identifier.
`vp`	An object of class `viewport`, or NULL.
`gp`	An object of class `gpar`; typically the output from a call to the function `gpar`.
`draw`	Should the frame be drawn.

Details

Both functions create a frame grob (a graphical object describing a frame), but only `grid.frame()` draws the frame (and then only if `draw` is `TRUE`). Nothing will actually be drawn, but it will put the frame on the display list, which means that the output will be dynamically updated as objects are packed into the frame. Possibly useful for debugging.

Value

A frame grob. `grid.frame()` returns the value invisibly.

Author(s)

Paul Murrell

See Also

`grid.pack`

Examples

```
grid.newpage()
grid.frame(name="gf", draw=TRUE)
grid.pack("gf", rectGrob(gp=gpar(fill="grey")), width=unit(1, "null"))
grid.pack("gf", textGrob("hi there"), side="right")
```

---

grid.get *Get a Grid Graphical Object*

Description

Retrieve a grob or a descendant of a grob.

Usage

```
grid.get(gPath, strict = FALSE, grep = FALSE, global = FALSE,
         allDevices = FALSE)

getGrob(gTree, gPath, strict = FALSE, grep = FALSE, global = FALSE)
```

Arguments

gTree	A gTree object.
gPath	A gPath object. For grid.get this specifyies a grob on the display list. For getGrob this specifies a descendant of the specified gTree.
strict	A boolean indicating whether the gPath must be matched exactly.
grep	A boolean indicating whether the gPath should be treated as a regular expression. Values are recycled across elements of the gPath (e.g., c(TRUE, FALSE) means that every odd element of the gPath will be treated as a regular expression).
global	A boolean indicating whether the function should affect just the first match of the gPath, or whether all matches should be affected.
allDevices	A boolean indicating whether all open devices should be searched for matches, or just the current device. NOT YET IMPLEMENTED.

Value

A grob object.

Author(s)

Paul Murrell

See Also

grob, getGrob, addGrob, removeGrob.

Examples

```
grid.xaxis(name="xa")
grid.get("xa")
grid.get(gPath("xa", "ticks"))

grid.draw(gTree(name="gt", children=gList(xaxisGrob(name="axis"))))
grid.get(gPath("gt", "axis", "ticks"))
```

---

grid.grab *Grab the current grid output*

Description

Creates a gTree object from the current grid display list or from a scene generated by user-specified code.

Usage

```
grid.grab(warn = 2, wrap = FALSE, ...)
grid.grabExpr(expr, warn = 2, wrap = FALSE, ...)
```

Arguments

expr	An expression to be evaluated. Typically, some calls to grid drawing functions.
warn	An integer specifying the amount of warnings to emit. 0 means no warnings, 1 means warn when it is certain that the grab will not faithfully represent the original scene. 2 means warn if there's any possibility that the grab will not faithfully represent the original scene.
wrap	A logical indicating how the output should be captured. If TRUE, each non-grob element on the display list is captured by wrapping it in a grob.
...	arguments passed to gTree, for example, a name and/or class for the gTree that is created.

Details

There are four ways to capture grid output as a gTree.

There are two functions for capturing output: use grid.grab to capture an existing drawing and grid.grabExpr to capture the output from an expression (without drawing anything).

For each of these functions, the output can be captured in two ways. One way tries to be clever and make a gTree with a childrenvp slot containing all viewports on the display list (including those that are popped) and every grob on the display list as a child of the new gTree; each child has a vpPath in the vp slot so that it is drawn in the appropriate viewport. In other words, the gTree contains all elements on the display list, but in a slightly altered form.

The other way, wrap=TRUE, is to create a grob for every element on the display list (and make all of those grobs children of the gTree).

The first approach creates a more compact and elegant gTree, which is more flexible to work with, but is not guaranteed to faithfully replicate all possible grid output. The second approach is more brute force, and harder to work with, but should always faithfully replicate the original output.

Value

A gTree object.

See Also

gTree

Examples

```
pushViewport(viewport(w=.5, h=.5))
grid.rect()
grid.points(runif(10), runif(10))
popViewport()
grab <- grid.grab()
grid.newpage()
grid.draw(grab)
```

---

grid.grill *Draw a Grill*

Description

This function draws a grill within a Grid viewport.

Usage

```
grid.grill(h = unit(seq(0.25, 0.75, 0.25), "npc"),
           v = unit(seq(0.25, 0.75, 0.25), "npc"),
           default.units = "npc", gp=gpar(col = "grey"), vp = NULL)
```

Arguments

h	A numeric vector or unit object indicating the horizontal location of the vertical grill lines.
v	A numeric vector or unit object indicating the vertical location of the horizontal grill lines.
default.units	A string indicating the default units to use if h or v are only given as numeric vectors.
gp	An object of class gpar, typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
vp	A Grid viewport object.

Value

None.

Author(s)

Paul Murrell

See Also

Grid, viewport.

---

grid.grob *Create a Grid Graphical Object*

Description

These functions create grid graphical objects.

Usage

```
grid.grob(list.struct, cl = NULL, draw = TRUE)
grob(..., name = NULL, gp = NULL, vp = NULL, cl = NULL)
gTree(..., name = NULL, gp = NULL, vp = NULL, children = NULL,
      childrenvp = NULL, cl = NULL)
childNames(gTree)
gList(...)
```

Arguments

...	For grob and gTree, the named slots describing important features of the graphical object. For gList, a series of grob objects.
list.struct	A list (preferably with each element named).
name	A character identifier for the grob. Used to find the grob on the display list and/or as a child of another grob.
children	A gList object.
childrenvp	A viewport object (or NULL).
gp	A gpar object, typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
vp	A viewport object (or NULL).
cl	A string giving the class attribute for the list.struct
draw	A logical value to indicate whether to produce graphical output.
gTree	A gTree object.

Details

These functions can be used to create a basic grob, gTree, or gList object, or a new class derived from one of these.

A grid graphical object (grob) is a description of a graphical item. These basic classes provide default behaviour for validating, drawing, and modifying graphical objects. Both call the function validDetails to check that the object returned is coherent.

A gTree can have other grobs as children; when a gTree is drawn, it draws all of its children. Before drawing its children, a gTree pushes its childrenvp slot and then navigates back up (calls upViewport) so that the children can specify their location within the childrenvp via a vpPath.

Grob names need not be unique in general, but all children of a gTree must have different names. A grob name can be any string, though it is not advisable to use the gPath separator (currently ::) in grob names.

The function `childNames` returns the names of the grobs which are children of a gTree.

All grid primitives (`grid.lines`, `grid.rect`, ...) and some higher-level grid components (e.g., `grid.xaxis` and `grid.yaxis`) are derived from these classes.

`grid.grob` is deprecated.

Value

A grob object.

Author(s)

Paul Murrell

See Also

`grid.draw`, `grid.edit`, `grid.get`.

Examples

`grid.layout` *Create a Grid Layout*

Description

This function returns a Grid layout, which describes a subdivision of a rectangular region.

Usage

```
grid.layout(nrow = 1, ncol = 1,
        widths = unit(rep(1, ncol), "null"),
        heights = unit(rep(1, nrow), "null"),
        default.units = "null", respect = FALSE,
        just="centre")
```

Arguments

`nrow`	An integer describing the number of rows in the layout.
`ncol`	An integer describing the number of columns in the layout.
`widths`	A numeric vector or unit object describing the widths of the columns in the layout.
`heights`	A numeric vector or unit object describing the heights of the rows in the layout.
`default.units`	A string indicating the default units to use if `widths` or `heights` are only given as numeric vectors.
`respect`	A logical value or a numeric matrix. If a logical, this indicates whether row heights and column widths should respect each other. If a matrix, non-zero values indicate that the corresponding row and column should be respected (see examples below).

`just` A string vector indicating how the layout should be justified if it is not the same size as its parent viewport. If there are two values, the first value specifes horizontal justification and the second value specifies vertical justification. Possible values are: `"left"`, `"right"`, `"centre"`, `"center"`, `"bottom"`, and `"top"`. NOTE that in this context, `"left"`, for example, means align the left edge of the left-most layout column with the left edge of the parent viewport.

Details

The unit objects given for the `widths` and `heights` of a layout may use a special `units` that only has meaning for layouts. This is the `"null"` unit, which indicates what relative fraction of the available width/height the column/row occupies. See the reference for a better description of relative widths and heights in layouts.

Value

A Grid layout object.

WARNING

This function must NOT be confused with the base R graphics function `layout`. In particular, do not use `layout` in combination with Grid graphics. The documentation for `layout` may provide some useful information and this function should behave identically in comparable situations. The `grid.layout` function has *added* the ability to specify a broader range of units for row heights and column widths, and allows for nested layouts (see `viewport`).

Author(s)

Paul Murrell

References

Murrell, P. R. (1999), Layouts: A Mechanism for Arranging Plots on a Page, *Journal of Computational and Graphical Statistics*, **8**, 121–134.

See Also

Grid, `grid.show.layout`, `viewport`, `layout`

Examples

```
## A variety of layouts (some a bit mid-bending ...)
layout.torture()
## Demonstration of layout justification
grid.newpage()
testlay <- function(just="centre") {
  pushViewport(viewport(layout=grid.layout(1, 1, widths=unit(1, "inches"),
                           height=unit(0.25, "npc"),
                           just=just)))
  pushViewport(viewport(layout.pos.col=1, layout.pos.row=1))
  grid.rect()
  grid.text(paste(just, collapse="-"))
  popViewport(2)
}
testlay()
```

```
testlay(c("left", "top"))
testlay(c("right", "top"))
testlay(c("right", "bottom"))
testlay(c("left", "bottom"))
testlay(c("left"))
testlay(c("right"))
testlay(c("bottom"))
testlay(c("top"))
```

`grid.lines` *Draw Lines in a Grid Viewport*

Description

These functions create and draw a series of lines.

Usage

```
grid.lines(x = unit(c(0, 1), "npc"),
           y = unit(c(0, 1), "npc"),
           default.units = "npc",
           arrow = NULL, name = NULL,
           gp=gpar(), draw = TRUE, vp = NULL)
linesGrob(x = unit(c(0, 1), "npc"),
           y = unit(c(0, 1), "npc"),
           default.units = "npc",
           arrow = NULL, name = NULL,
           gp=gpar(), vp = NULL)
grid.polyline(...)
polylineGrob(x = unit(c(0, 1), "npc"),
             y = unit(c(0, 1), "npc"),
             id=NULL, id.lengths=NULL,
             default.units = "npc",
             arrow = NULL, name = NULL,
             gp=gpar(), vp = NULL)
```

Arguments

`x`	A numeric vector or unit object specifying x-values.
`y`	A numeric vector or unit object specifying y-values.
`default.units`	A string indicating the default units to use if `x` or `y` are only given as numeric vectors.
`arrow`	A list describing arrow heads to place at either end of the line, as produced by the `arrow` function.
`name`	A character identifier.
`gp`	An object of class `gpar`, typically the output from a call to the function `gpar`. This is basically a list of graphical parameter settings.
`draw`	A logical value indicating whether graphics output should be produced.
`vp`	A Grid viewport object (or NULL).

id	A numeric vector used to separate locations in x and y into multiple lines. All locations with the same id belong to the same line.
id.lengths	A numeric vector used to separate locations in x and y into multiple lines. Specifies consecutive blocks of locations which make up separate lines.
...	Arguments passed to polylineGrob.

Details

The first two functions create a lines grob (a graphical object describing lines), and grid.lines draws the lines (if draw is TRUE).

The second two functions create or draw a polyline grob, which is just like a lines grob, except that there can be multiple distinct lines drawn.

Value

A lines grob or a polyline grob. grid.lines returns a lines grob invisibly.

Author(s)

Paul Murrell

See Also

Grid, viewport, arrow

Examples

```
grid.lines()
# Using id (NOTE: locations are not in consecutive blocks)
grid.newpage()
grid.polyline(x=c((0:4)/10, rep(.5, 5), (10:6)/10, rep(.5, 5)),
              y=c(rep(.5, 5), (10:6/10), rep(.5, 5), (0:4)/10),
              id=rep(1:5, 4),
              gp=gpar(col=1:5, lwd=3))
# Using id.lengths
grid.newpage()
grid.polyline(x=outer(c(0, .5, 1, .5), 5:1/5),
              y=outer(c(.5, 1, .5, 0), 5:1/5),
              id.lengths=rep(4, 5),
              gp=gpar(col=1:5, lwd=3))
```

---

grid.locator *Capture a Mouse Click*

Description

Allows the user to click the mouse once within the current graphics device and returns the location of the mouse click within the current viewport, in the specified coordinate system.

Usage

```
grid.locator(unit = "native")
```

Arguments

unit	The coordinate system in which to return the location of the mouse click. See the unit function for valid coordinate systems.

Details

This function is modal (like the graphics package function locator) so the command line and graphics drawing is blocked until the use has clicked the mouse in the current device.

Value

A unit object representing the location of the mouse click within the current viewport, in the specified coordinate system.

If the user did not click mouse button 1, the function (invisibly) returns NULL.

Author(s)

Paul Murrell

See Also

viewport, unit, locator in package graphics, and for an application see trellis.focus and trellis.identify in package lattice.

Examples

```
if (interactive()) {
  ## Need to write a more sophisticated unit as.character method
  unittrim <- function(unit) {
    sub("^([0-9]+|[0-9]+[.][0-9])[0-9]*", "\\1", as.character(unit))
  }
  do.click <- function(unit) {
    click.locn <- grid.locator(unit)
    grid.segments(unit.c(click.locn$x, unit(0, "npc")),
                  unit.c(unit(0, "npc"), click.locn$y),
                  click.locn$x, click.locn$y,
                  gp=gpar(lty="dashed", col="grey"))
    grid.points(click.locn$x, click.locn$y, pch=16, size=unit(1, "mm"))
    clickx <- unittrim(click.locn$x)
    clicky <- unittrim(click.locn$y)
    grid.text(paste("(", clickx, ", ", clicky, ")", sep=""),
              click.locn$x + unit(2, "mm"), click.locn$y,
              just="left")
  }
  do.click("inches")
  pushViewport(viewport(width=0.5, height=0.5,
                        xscale=c(0, 100), yscale=c(0, 10)))
  grid.rect()
  grid.xaxis()
  grid.yaxis()
  do.click("native")
  popViewport()
}
```

grid.move.to *Move or Draw to a Specified Position*

Description

Grid has the notion of a current location. These functions sets that location.

Usage

```
grid.move.to(x = 0, y = 0, default.units = "npc", name = NULL,
             draw = TRUE, vp = NULL)

moveToGrob(x = 0, y = 0, default.units = "npc", name = NULL, vp = NULL)

grid.line.to(x = 1, y = 1, default.units = "npc",
             arrow = NULL, name = NULL,
             gp = gpar(), draw = TRUE, vp = NULL)

lineToGrob(x = 1, y = 1, default.units = "npc", arrow = NULL,
           name = NULL, gp = gpar(), vp = NULL)
```

Arguments

x	A numeric value or a unit object specifying an x-value.
y	A numeric value or a unit object specifying a y-value.
default.units	A string indicating the default units to use if x or y are only given as numeric values.
arrow	A list describing arrow heads to place at either end of the line, as produced by the arrow function.
name	A character identifier.
draw	A logical value indicating whether graphics output should be produced.
gp	An object of class gpar, typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
vp	A Grid viewport object (or NULL).

Details

Both functions create a move.to/line.to grob (a graphical object describing a move-to/line-to), but only grid.move.to/line.to() draws the move.to/line.to (and then only if draw is TRUE).

Value

A move.to/line.to grob. grid.move.to/line.to() returns the value invisibly.

Author(s)

Paul Murrell

See Also

Grid, viewport, arrow

Examples

```
grid.newpage()
grid.move.to(0.5, 0.5)
grid.line.to(1, 1)
grid.line.to(0.5, 0)
pushViewport(viewport(x=0, y=0, w=0.25, h=0.25, just=c("left", "bottom")))
grid.rect()
grid.grill()
grid.line.to(0.5, 0.5)
popViewport()
```

---

grid.newpage *Move to a New Page on a Grid Device*

Description

This function erases the current device or moves to a new page.

Usage

```
grid.newpage(recording = TRUE)
```

Arguments

recording	A logical value to indicate whether the new-page operation should be saved onto the Grid display list.

Details

There is a hook called "grid.newpage" (see setHook) which is used in the testing code to annotate the new page. The hook function(s) are called with no argument. (If the value is a character string, get is called on it from within the **grid** namespace.)

Value

None.

Author(s)

Paul Murrell

See Also

Grid

grid.pack *Pack an Object within a Frame*

Description

This functions, together with `grid.frame` and `frameGrob` are part of a GUI-builder-like interface to constructing graphical images. The idea is that you create a frame with `grid.frame` or `frameGrob` then use this functions to pack objects into the frame.

Usage

```
grid.pack(gPath, grob, redraw = TRUE, side = NULL,
          row = NULL, row.before = NULL, row.after = NULL,
          col = NULL, col.before = NULL, col.after = NULL,
          width = NULL, height = NULL,
          force.width = FALSE, force.height = FALSE, border = NULL,
          dynamic = FALSE)

packGrob(frame, grob, side = NULL,
          row = NULL, row.before = NULL, row.after = NULL,
          col = NULL, col.before = NULL, col.after = NULL,
          width = NULL, height = NULL,
          force.width = FALSE, force.height = FALSE, border = NULL,
          dynamic = FALSE)
```

Arguments

`gPath`	A gPath object, which specifies a frame on the display list.
`frame`	An object of class `frame`, typically the output from a call to `grid.frame`.
`grob`	An object of class `grob`. The object to be packed.
`redraw`	A boolean indicating whether the output should be updated.
`side`	One of `"left"`, `"top"`, `"right"`, `"bottom"` to indicate which side to pack the object on.
`row`	Which row to add the object to. Must be between 1 and the-number-of-rows-currently-in-the-frame + 1, or `NULL` in which case the object occupies all rows.
`row.before`	Add the object to a new row just before this row.
`row.after`	Add the object to a new row just after this row.
`col`	Which col to add the object to. Must be between 1 and the-number-of-cols-currently-in-the-frame + 1, or `NULL` in which case the object occupies all cols.
`col.before`	Add the object to a new col just before this col.
`col.after`	Add the object to a new col just after this col.
`width`	Specifies the width of the column that the object is added to (rather than allowing the width to be taken from the object).
`height`	Specifies the height of the row that the object is added to (rather than allowing the height to be taken from the object).
`force.width`	A logical value indicating whether the width of the column that the grob is being packed into should be EITHER the width specified in the call to `grid.pack` OR the maximum of that width and the pre-existing width.

`force.height` A logical value indicating whether the height of the column that the grob is being packed into should be EITHER the height specified in the call to `grid.pack` OR the maximum of that height and the pre-existing height.

`border` A `unit` object of length 4 indicating the borders around the object.

`dynamic` If the width/height is taken from the grob being packed, this boolean flag indicates whether the grobwidth/height unit refers directly to the grob, or uses a gPath to the grob. In the latter case, changes to the grob will trigger a recalculation of the width/height.

Details

`packGrob` modifies the given frame grob and returns the modified frame grob.

`grid.pack` destructively modifies a frame grob on the display list (and redraws the display list if `redraw` is `TRUE`).

These are (meant to be) very flexible functions. There are many different ways to specify where the new object is to be added relative to the objects already in the frame. The function checks that the specification is not self-contradictory.

NOTE that the width/height of the row/col that the object is added to is taken from the object itself unless the `width`/`height` is specified.

Value

`packGrob` returns a frame grob, but `grid.pack` returns `NULL`.

Author(s)

Paul Murrell

See Also

`grid.frame`, `grid.place`, `grid.edit`, and `gPath`.

---

grid.place *Place an Object within a Frame*

Description

These functions provide a simpler (and faster) alternative to the `grid.pack()` and `packGrob` functions. They can be used to place objects within the existing rows and columns of a frame layout. They do not provide the ability to add new rows and columns nor do they affect the heights and widths of the rows and columns.

Usage

```
grid.place(gPath, grob, row = 1, col = 1, redraw = TRUE)
placeGrob(frame, grob, row = NULL, col = NULL)
```

Arguments

gPath	A gPath object, which specifies a frame on the display list.
frame	An object of class frame, typically the output from a call to grid.frame.
grob	An object of class grob. The object to be placed.
row	Which row to add the object to. Must be between 1 and the-number-of-rows-currently-in-the-frame.
col	Which col to add the object to. Must be between 1 and the-number-of-cols-currently-in-the-frame.
redraw	A boolean indicating whether the output should be updated.

Details

placeGrob modifies the given frame grob and returns the modified frame grob.

grid.place destructively modifies a frame grob on the display list (and redraws the display list if redraw is TRUE).

Value

placeGrob returns a frame grob, but grid.place returns NULL.

Author(s)

Paul Murrell

See Also

grid.frame, grid.pack, grid.edit, and gPath.

---

grid.plot.and.legend

A Simple Plot and Legend Demo

Description

This function is just a wrapper for a simple demonstration of how a basic plot and legend can be drawn from scratch using grid.

Usage

```
grid.plot.and.legend()
```

Author(s)

Paul Murrell

Examples

```
grid.plot.and.legend()
```

`grid.points` *Draw Data Symbols*

Description

These functions create and draw data symbols.

Usage

```
grid.points(x = stats::runif(10),
            y = stats::runif(10),
            pch = 1, size = unit(1, "char"),
            default.units = "native", name = NULL,
            gp=gpar(), draw = TRUE, vp = NULL)
pointsGrob(x = stats::runif(10),
            y = stats::runif(10),
            pch = 1, size = unit(1, "char"),
            default.units = "native", name = NULL,
            gp=gpar(), vp = NULL)
```

Arguments

`x`	A numeric vector or unit object specifying x-values.
`y`	A numeric vector or unit object specifying y-values.
`pch`	A numeric or character vector indicating what sort of plotting symbol to use.
`size`	A unit object specifying the size of the plotting symbols.
`default.units`	A string indicating the default units to use if `x` or `y` are only given as numeric vectors.
`name`	A character identifier.
`gp`	An object of class `gpar`, typically the output from a call to the function `gpar`. This is basically a list of graphical parameter settings.
`draw`	A logical value indicating whether graphics output should be produced.
`vp`	A Grid viewport object (or NULL).

Details

Both functions create a points grob (a graphical object describing points), but only `grid.points` draws the points (and then only if `draw` is `TRUE`).

Value

A points grob. `grid.points` returns the value invisibly.

Author(s)

Paul Murrell

See Also

Grid, `viewport`

grid.polygon *Draw a Polygon*

Description

These functions create and draw a polygon. The final point will automatically be connected to the initial point.

Usage

```
grid.polygon(x=c(0, 0.5, 1, 0.5), y=c(0.5, 1, 0.5, 0),
             id=NULL, id.lengths=NULL,
             default.units="npc", name=NULL,
             gp=gpar(), draw=TRUE, vp=NULL)
polygonGrob(x=c(0, 0.5, 1, 0.5), y=c(0.5, 1, 0.5, 0),
             id=NULL, id.lengths=NULL,
             default.units="npc", name=NULL,
             gp=gpar(), vp=NULL)
```

Arguments

x	A numeric vector or unit object specifying x-locations.
y	A numeric vector or unit object specifying y-locations.
id	A numeric vector used to separate locations in x and y into multiple polygons. All locations with the same id belong to the same polygon.
id.lengths	A numeric vector used to separate locations in x and y into multiple polygons. Specifies consecutive blocks of locations which make up separate polygons.
default.units	A string indicating the default units to use if x, y, width, or height are only given as numeric vectors.
name	A character identifier.
gp	An object of class gpar, typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
draw	A logical value indicating whether graphics output should be produced.
vp	A Grid viewport object (or NULL).

Details

Both functions create a polygon grob (a graphical object describing a polygon), but only grid.polygon draws the polygon (and then only if draw is TRUE).

Value

A grob object.

Author(s)

Paul Murrell

See Also

Grid, viewport

Examples

```
grid.polygon()
# Using id (NOTE: locations are not in consecutive blocks)
grid.newpage()
grid.polygon(x=c((0:4)/10, rep(.5, 5), (10:6)/10, rep(.5, 5)),
             y=c(rep(.5, 5), (10:6/10), rep(.5, 5), (0:4)/10),
             id=rep(1:5, 4),
             gp=gpar(fill=1:5))
# Using id.lengths
grid.newpage()
grid.polygon(x=outer(c(0, .5, 1, .5), 5:1/5),
             y=outer(c(.5, 1, .5, 0), 5:1/5),
             id.lengths=rep(4, 5),
             gp=gpar(fill=1:5))
```

---

grid.pretty *Generate a Sensible Set of Breakpoints*

Description

Produces a pretty set of breakpoints within the range given.

Usage

```
grid.pretty(range)
```

Arguments

range	A numeric vector

Value

A numeric vector of breakpoints.

Author(s)

Paul Murrell

grid.prompt *Prompt before new page*

Description

This function can be used to control whether the user is prompted before starting a new page of output.

Usage

```
grid.prompt(ask)
```

Arguments

ask	a logical value. If TRUE, the user is prompted before a new page of output is started.

Value

The current prompt setting *before* any new setting is applied.

Author(s)

Paul Murrell

See Also

grid.newpage

grid.record *Encapsulate calculations and drawing*

Description

Evaluates an expression that includes both calculatons and drawing that depends on the calculations so that both the calculations and the drawing will be rerun when the scene is redrawn (e.g., device resize or editing).

Intended *only* for expert use.

Usage

```
recordGrob(expr, list, name=NULL, gp=NULL, vp=NULL)
grid.record(expr, list, name=NULL, gp=NULL, vp=NULL)
```

Arguments

expr	object of mode expression or call or an "unevaluated expression".
list	a list defining the environment in which expr is to be evaluated.
name	A character identifier.
gp	An object of class gpar, typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
vp	A Grid viewport object (or NULL).

Details

A grob is created of special class "recordedGrob" (and drawn, in the case of grid.record). The drawDetails method for this class evaluates the expression with the list as the evaluation environment (and the grid Namespace as the parent of that environment).

Note

This function *must* be used instead of the function recordGraphics; all of the dire warnings about using recordGraphics responsibly also apply here.

Author(s)

Paul Murrell

See Also

recordGraphics

Examples

```
grid.record({
              w <- convertWidth(unit(1, "inches"), "npc")
                      grid.rect(width=w)
            },
            list())
```

---

grid.rect *Draw rectangles*

Description

These functions create and draw rectangles.

Usage

```
grid.rect(x = unit(0.5, "npc"), y = unit(0.5, "npc"),
          width = unit(1, "npc"), height = unit(1, "npc"),
          just = "centre", hjust = NULL, vjust = NULL,
          default.units = "npc", name = NULL,
          gp=gpar(), draw = TRUE, vp = NULL)
rectGrob(x = unit(0.5, "npc"), y = unit(0.5, "npc"),
         width = unit(1, "npc"), height = unit(1, "npc"),
```

```
          just = "centre", hjust = NULL, vjust = NULL,
          default.units = "npc", name = NULL,
          gp=gpar(), vp = NULL)
```

Arguments

x	A numeric vector or unit object specifying x-location.
y	A numeric vector or unit object specifying y-location.
width	A numeric vector or unit object specifying width.
height	A numeric vector or unit object specifying height.
just	The justification of the rectangle relative to its (x, y) location. If there are two values, the first value specifes horizontal justification and the second value specifies vertical justification. Possible string values are: "left", "right", "centre", "center", "bottom", and "top". For numeric values, 0 means left alignment and 1 means right alignment.
hjust	A numeric vector specifying horizontal justification. If specified, overrides the just setting.
vjust	A numeric vector specifying vertical justification. If specified, overrides the just setting.
default.units	A string indicating the default units to use if x, y, width, or height are only given as numeric vectors.
name	A character identifier.
gp	An object of class gpar, typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
draw	A logical value indicating whether graphics output should be produced.
vp	A Grid viewport object (or NULL).

Details

Both functions create a rect grob (a graphical object describing rectangles), but only grid.rect draws the rectangles (and then only if draw is TRUE).

Value

A rect grob. grid.rect returns the value invisibly.

Author(s)

Paul Murrell

See Also

Grid, viewport

grid.refresh *Refresh the current grid scene*

Description

Replays the current grid display list.

Usage

```
grid.refresh()
```

Author(s)

Paul Murrell

grid.remove *Remove a Grid Graphical Object*

Description

Remove a grob from a gTree or a descendant of a gTree.

Usage

```
grid.remove(gPath, warn = TRUE, strict  =  FALSE, grep = FALSE,
            global = FALSE, allDevices = FALSE, redraw = TRUE)

removeGrob(gTree, gPath, strict = FALSE, grep = FALSE,
           global = FALSE, warn = TRUE)
```

Arguments

gTree	A gTree object.
gPath	A gPath object. For `grid.remove` this specifyies a gTree on the display list. For `removeGrob` this specifies a descendant of the specified gTree.
strict	A boolean indicating whether the gPath must be matched exactly.
grep	A boolean indicating whether the `gPath` should be treated as a regular expression. Values are recycled across elements of the `gPath` (e.g., `c(TRUE, FALSE)` means that every odd element of the `gPath` will be treated as a regular expression).
global	A boolean indicating whether the function should affect just the first match of the `gPath`, or whether all matches should be affected.
allDevices	A boolean indicating whether all open devices should be searched for matches, or just the current device. NOT YET IMPLEMENTED.
warn	A logical to indicate whether failing to find the specified grob should trigger an error.
redraw	A logical value to indicate whether to redraw the grob.

Details

`removeGrob` copies the specified grob and returns a modified grob.

`grid.remove` destructively modifies a grob on the display list. If `redraw` is `TRUE` it then redraws everything to reflect the change.

Value

`removeGrob` returns a grob object; `grid.remove` returns `NULL`.

Author(s)

Paul Murrell

See Also

`grob`, `getGrob`, `removeGrob`, `removeGrob`.

`grid.segments` *Draw Line Segments*

Description

These functions create and draw line segments.

Usage

```
grid.segments(x0 = unit(0, "npc"), y0 = unit(0, "npc"),
              x1 = unit(1, "npc"), y1 = unit(1, "npc"),
              default.units = "npc",
              arrow = NULL,
              name = NULL, gp = gpar(), draw = TRUE, vp = NULL)
segmentsGrob(x0 = unit(0, "npc"), y0 = unit(0, "npc"),
              x1 = unit(1, "npc"), y1 = unit(1, "npc"),
              default.units = "npc",
              arrow = NULL, name = NULL, gp = gpar(), vp = NULL)
```

Arguments

`x0`	Numeric indicating the starting x-values of the line segments.
`y0`	Numeric indicating the starting y-values of the line segments.
`x1`	Numeric indicating the stopping x-values of the line segments.
`y1`	Numeric indicating the stopping y-values of the line segments.
`default.units`	A string.
`arrow`	A list describing arrow heads to place at either end of the line segments, as produced by the `arrow` function.
`name`	A character identifier.
`gp`	An object of class `gpar`.
`draw`	A logical value indicating whether graphics output should be produced.
`vp`	A Grid viewport object (or NULL)

Details

Both functions create a segments grob (a graphical object describing segments), but only `grid.segments` draws the segments (and then only if `draw` is `TRUE`).

Value

A segments grob. `grid.segments` returns the value invisibly.

Author(s)

Paul Murrell

See Also

Grid, `viewport`, `arrow`

`grid.set` *Set a Grid Graphical Object*

Description

Replace a grob or a descendant of a grob.

Usage

```
grid.set(gPath, newGrob, strict = FALSE, grep = FALSE,
         redraw = TRUE)

setGrob(gTree, gPath, newGrob, strict = FALSE, grep = FALSE)
```

Arguments

`gTree`	A gTree object.
`gPath`	A gPath object. For `grid.set` this specifyies a grob on the display list. For `setGrob` this specifies a descendant of the specified gTree.
`newGrob`	A grob object.
`strict`	A boolean indicating whether the gPath must be matched exactly.
`grep`	A boolean indicating whether the `gPath` should be treated as a regular expression. Values are recycled across elements of the `gPath` (e.g., `c(TRUE, FALSE)` means that every odd element of the `gPath` will be treated as a regular expression).
`redraw`	A logical value to indicate whether to redraw the grob.

Details

`setGrob` copies the specified grob and returns a modified grob.

`grid.set` destructively replaces a grob on the display list. If `redraw` is `TRUE` it then redraws everything to reflect the change.

These functions should not normally be called by the user.

Value

`setGrob` returns a grob object; `grid.set` returns `NULL`.

Author(s)

Paul Murrell

See Also

`grid.grob`.

`grid.show.layout` *Draw a Diagram of a Grid Layout*

Description

This function uses Grid graphics to draw a diagram of a Grid layout.

Usage

```
grid.show.layout(l, newpage=TRUE, bg = "light grey",
                 cell.border = "blue", cell.fill = "light blue",
                 cell.label = TRUE, label.col = "blue",
                 unit.col = "red", vp = NULL)
```

Arguments

`l`	A Grid layout object.
`newpage`	A logical value indicating whether to move on to a new page before drawing the diagram.
`bg`	The colour used for the background.
`cell.border`	The colour used to draw the borders of the cells in the layout.
`cell.fill`	The colour used to fill the cells in the layout.
`cell.label`	A logical indicating whether the layout cells should be labelled.
`label.col`	The colour used for layout cell labels.
`unit.col`	The colour used for labelling the widths/heights of columns/rows.
`vp`	A Grid viewport object (or NULL).

Details

A viewport is created within `vp` to provide a margin for annotation, and the layout is drawn within that new viewport. The margin is filled with light grey, the new viewport is filled with white and framed with a black border, and the layout regions are filled with light blue and framed with a blue border. The diagram is annotated with the widths and heights (including units) of the columns and rows of the layout using red text. (All colours are defaults and may be customised via function arguments.)

Value

None.

Author(s)

Paul Murrell

See Also

Grid, viewport, grid.layout

Examples

```
## Diagram of a simple layout
grid.show.layout(grid.layout(4,2,
                     heights=unit(rep(1, 4),
                                  c("lines", "lines", "lines", "null")),
                     widths=unit(c(1, 1), "inches")))
```

---

grid.show.viewport *Draw a Diagram of a Grid Viewport*

Description

This function uses Grid graphics to draw a diagram of a Grid viewport.

Usage

```
grid.show.viewport(v, parent.layout = NULL, newpage = TRUE,
                   border.fill="light grey",
                   vp.col="blue", vp.fill="light blue",
                   scale.col="red",
                   vp = NULL)
```

Arguments

v	A Grid viewport object.
parent.layout	A grid layout object. If this is not NULL and the viewport given in v has its location specified relative to the layout, then the diagram shows the layout and which cells v occupies within the layout.
newpage	A logical value to indicate whether to move to a new page before drawing the diagram.
border.fill	Colour to fill the border margin.
vp.col	Colour for the border of the viewport region.
vp.fill	Colour to fill the viewport region.
scale.col	Colour to draw the viewport axes.
vp	A Grid viewport object (or NULL).

Details

A viewport is created within `vp` to provide a margin for annotation, and the diagram is drawn within that new viewport. By default, the margin is filled with light grey, the new viewport is filled with white and framed with a black border, and the viewport region is filled with light blue and framed with a blue border. The diagram is annotated with the width and height (including units) of the viewport, the (x, y) location of the viewport, and the x- and y-scales of the viewport, using red lines and text.

Value

None.

Author(s)

Paul Murrell

See Also

Grid, `viewport`

Examples

```
## Diagram of a sample viewport
grid.show.viewport(viewport(x=0.6, y=0.6,
                            w=unit(1, "inches"), h=unit(1, "inches")))
grid.show.viewport(viewport(layout.pos.row=2, layout.pos.col=2:3),
                   grid.layout(3, 4))
```

---

grid.text *Draw Text*

Description

These functions create and draw text.

Usage

```
grid.text(label, x = unit(0.5, "npc"), y = unit(0.5, "npc"),
          just = "centre", hjust = NULL, vjust = NULL, rot = 0,
          check.overlap = FALSE, default.units = "npc",
          name = NULL, gp = gpar(), draw = TRUE, vp = NULL)

textGrob(label, x = unit(0.5, "npc"), y = unit(0.5, "npc"),
          just = "centre", hjust = NULL, vjust = NULL, rot = 0,
          check.overlap = FALSE, default.units = "npc",
          name = NULL, gp = gpar(), vp = NULL)
```

Arguments

label	A vector of strings or expressions to draw.
x	A numeric vector or unit object specifying x-values.
y	A numeric vector or unit object specifying y-values.
just	The justification of the text relative to its (x, y) location. If there are two values, the first value specifes horizontal justification and the second value specifies vertical justification. Possible string values are: "left", "right", "centre", "center", "bottom", and "top". For numeric values, 0 means left alignment and 1 means right alignment.
hjust	A numeric vector specifying horizontal justification. If specified, overrides the just setting.
vjust	A numeric vector specifying vertical justification. If specified, overrides the just setting.
rot	The angle to rotate the text.
check.overlap	A logical value to indicate whether to check for and omit overlapping text.
default.units	A string indicating the default units to use if x or y are only given as numeric vectors.
name	A character identifier.
gp	An object of class gpar, typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
draw	A logical value indicating whether graphics output should be produced.
vp	A Grid viewport object (or NULL).

Details

Both functions create a text grob (a graphical object describing text), but only grid.text draws the text (and then only if draw is TRUE).

If the label argument is an expression, the output is formatted as a mathematical annotation, as for base graphics text.

Value

A text grob. grid.text returns the value invisibly.

Author(s)

Paul Murrell

See Also

Grid, viewport

Examples

```
grid.newpage()
x <- stats::runif(20)
y <- stats::runif(20)
rot <- stats::runif(20, 0, 360)
grid.text("SOMETHING NICE AND BIG", x=x, y=y, rot=rot,
          gp=gpar(fontsize=20, col="grey"))
grid.text("SOMETHING NICE AND BIG", x=x, y=y, rot=rot,
          gp=gpar(fontsize=20), check=TRUE)
grid.newpage()
draw.text <- function(just, i, j) {
  grid.text("ABCD", x=x[j], y=y[i], just=just)
  grid.text(deparse(substitute(just)), x=x[j], y=y[i] + unit(2, "lines"),
            gp=gpar(col="grey", fontsize=8))
}
x <- unit(1:4/5, "npc")
y <- unit(1:4/5, "npc")
grid.grill(h=y, v=x, gp=gpar(col="grey"))
draw.text(c("bottom"), 1, 1)
draw.text(c("left", "bottom"), 2, 1)
draw.text(c("right", "bottom"), 3, 1)
draw.text(c("centre", "bottom"), 4, 1)
draw.text(c("centre"), 1, 2)
draw.text(c("left", "centre"), 2, 2)
draw.text(c("right", "centre"), 3, 2)
draw.text(c("centre", "centre"), 4, 2)
draw.text(c("top"), 1, 3)
draw.text(c("left", "top"), 2, 3)
draw.text(c("right", "top"), 3, 3)
draw.text(c("centre", "top"), 4, 3)
draw.text(c(), 1, 4)
draw.text(c("left"), 2, 4)
draw.text(c("right"), 3, 4)
draw.text(c("centre"), 4, 4)
```

---

grid.xaxis *Draw an X-Axis*

Description

These functions create and draw an x-axis.

Usage

```
grid.xaxis(at = NULL, label = TRUE, main = TRUE,
           edits = NULL, name = NULL,
           gp = gpar(), draw = TRUE, vp = NULL)

xaxisGrob(at = NULL, label = TRUE, main = TRUE,
          edits = NULL, name = NULL,
          gp = gpar(), vp = NULL)
```

Arguments

at	A numeric vector of x-value locations for the tick marks.
label	A logical value indicating whether to draw the labels on the tick marks, or an expression or string vector which specify the labels to use. If not logical, must be the same length as the at argument.
main	A logical value indicating whether to draw the axis at the bottom (TRUE) or at the top (FALSE) of the viewport.
edits	A gEdit or gEditList containing edit operations to apply (to the children of the axis) when the axis is first created and during redrawing whenever at is NULL.
name	A character identifier.
gp	An object of class gpar, typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
draw	A logical value indicating whether graphics output should be produced.
vp	A Grid viewport obect (or NULL).

Details

Both functions create an xaxis grob (a graphical object describing an xaxis), but only grid.xaxis draws the xaxis (and then only if draw is TRUE).

Value

An xaxis grob. grid.xaxis returns the value invisibly.

Children

If the at slot of an xaxis grob is not NULL then the xaxis will have the following children:

major representing the line at the base of the tick marks.

ticks representing the tick marks.

labels representing the tick labels.

If the at slot is NULL then there are no children and ticks are drawn based on the current viewport scale.

Author(s)

Paul Murrell

See Also

Grid, viewport, grid.yaxis

grid.xspline *Draw an Xspline*

Description

These functions create and draw an xspline, a curve drawn relative to control points.

Usage

```
grid.xspline(...)
xsplineGrob(x = c(0, 0.5, 1, 0.5), y = c(0.5, 1, 0.5, 0),
            id = NULL, id.lengths = NULL,
            default.units = "npc",
            shape = 0, open = TRUE, arrow = NULL, repEnds = TRUE,
            name = NULL, gp = gpar(), vp = NULL)
```

Arguments

x	A numeric vector or unit object specifying x-locations of spline control points.
y	A numeric vector or unit object specifying y-locations of spline control points.
id	A numeric vector used to separate locations in x and y into multiple xsplines. All locations with the same id belong to the same xspline.
id.lengths	A numeric vector used to separate locations in x and y into multiple xspline. Specifies consecutive blocks of locations which make up separate xsplines.
default.units	A string indicating the default units to use if x or y are only given as numeric vectors.
shape	A numeric vector of values between -1 and 1, which control the shape of the spline relative to the control points.
open	A logical value indicating whether the spline is a line or a closed shape.
arrow	A list describing arrow heads to place at either end of the xspline, as produced by the arrow function.
repEnds	A logical value indicating whether the first and last control points should be replicated for drawing the curve (see Details below).
name	A character identifier.
gp	An object of class gpar, typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
vp	A Grid viewport object (or NULL).
...	Arguments to be passed to xsplineGrob.

Details

Both functions create an xspline grob (a graphical object describing an xspline), but only grid.xspline draws the xspline.

An xspline is a line drawn relative to control points. For each control point, the line may pass through (interpolate) the control point or it may only approach (approximate) the control point; the behaviour is determined by a shape parameter for each control point.

If the shape parameter is greater than zero, the spline approximates the control points (and is very similar to a cubic B-spline when the shape is 1). If the shape parameter is less than zero, the spline interpolates the control points (and is very similar to a Catmull-Rom spline when the shape is -1). If the shape parameter is 0, the spline forms a sharp corner at that control point.

For open xsplines, the start and end control points must have a shape of 0 (and non-zero values are silently converted to zero without warning).

For open xsplines, by default the start and end control points are actually replicated before the curve is drawn. A curve is drawn between (interpolating or approximating) the second and third of each set of four control points, so this default behaviour ensures that the resulting curve starts at the first control point you have specified and ends at the last control point. The default behaviour can be turned off via the `repEnds` argument, in which case the curve that is drawn starts (approximately) at the second control point and ends (approximately) at thefirst and second-to-last control point.

The `repEnds` argument is ignored for closed xsplines.

Missing values are not allowed for `x` and `y` (i.e., it is not valid for a control point to be missing).

For closed xsplines, a curve is automatically drawn between the final control point and the initial control point.

Value

A grob object.

References

Blanc, C. and Schlick, C. (1995), "X-splines : A Spline Model Designed for the End User", in *Proceedings of SIGGRAPH 95*.

See Also

Grid, `viewport`,
`code{viewport}`

Examples

```
x <- c(0.25, 0.25, 0.75, 0.75)
y <- c(0.25, 0.75, 0.75, 0.25)

xsplineTest <- function(s, i, j, open) {
  pushViewport(viewport(layout.pos.col=j, layout.pos.row=i))
  grid.points(x, y, default.units="npc", pch=16, size=unit(2, "mm"))
  grid.xspline(x, y, shape=s, open=open, gp=gpar(fill="grey"))
  grid.text(s, gp=gpar(col="grey"),
            x=unit(x, "npc") + unit(c(-1, -1, 1, 1), "mm"),
            y=unit(y, "npc") + unit(c(-1, 1, 1, -1), "mm"),
            hjust=c(1, 1, 0, 0),
            vjust=c(1, 0, 0, 1))
  popViewport()
}

pushViewport(viewport(width=.5, x=0, just="left",
                      layout=grid.layout(3, 3, respect=TRUE)))
pushViewport(viewport(layout.pos.row=1))
grid.text("Open Splines", y=1, just="bottom")
popViewport()
```

```
xsplineTest(c(0, -1, -1, 0), 1, 1, TRUE)
xsplineTest(c(0, -1,  0, 0), 1, 2, TRUE)
xsplineTest(c(0, -1,  1, 0), 1, 3, TRUE)
xsplineTest(c(0,  0, -1, 0), 2, 1, TRUE)
xsplineTest(c(0,  0,  0, 0), 2, 2, TRUE)
xsplineTest(c(0,  0,  1, 0), 2, 3, TRUE)
xsplineTest(c(0,  1, -1, 0), 3, 1, TRUE)
xsplineTest(c(0,  1,  0, 0), 3, 2, TRUE)
xsplineTest(c(0,  1,  1, 0), 3, 3, TRUE)
popViewport()
pushViewport(viewport(width=.5, x=1, just="right",
                      layout=grid.layout(3, 3, respect=TRUE)))
pushViewport(viewport(layout.pos.row=1))
grid.text("Closed Splines", y=1, just="bottom")
popViewport()
xsplineTest(c(-1, -1, -1, -1), 1, 1, FALSE)
xsplineTest(c(-1, -1,  0, -1), 1, 2, FALSE)
xsplineTest(c(-1, -1,  1, -1), 1, 3, FALSE)
xsplineTest(c( 0,  0, -1,  0), 2, 1, FALSE)
xsplineTest(c( 0,  0,  0,  0), 2, 2, FALSE)
xsplineTest(c( 0,  0,  1,  0), 2, 3, FALSE)
xsplineTest(c( 1,  1, -1,  1), 3, 1, FALSE)
xsplineTest(c( 1,  1,  0,  1), 3, 2, FALSE)
xsplineTest(c( 1,  1,  1,  1), 3, 3, FALSE)
popViewport()
```

`grid.yaxis` *Draw a Y-Axis*

Description

These functions create and draw a y-axis.

Usage

```
grid.yaxis(at = NULL, label = TRUE, main = TRUE,
           edits = NULL, name = NULL,
           gp = gpar(), draw = TRUE, vp = NULL)

yaxisGrob(at = NULL, label = TRUE, main = TRUE,
          edits = NULL, name = NULL,
          gp = gpar(), vp = NULL)
```

Arguments

`at`	A numeric vector of y-value locations for the tick marks.
`label`	A logical value indicating whether to draw the labels on the tick marks, or an expression or string vector which specify the labels to use. If not logical, must be the same length as the `at` argument.
`main`	A logical value indicating whether to draw the axis at the left (`TRUE`) or at the right (`FALSE`) of the viewport.
`edits`	A gEdit or gEditList containing edit operations to apply (to the children of the axis) when the axis is first created and during redrawing whenever `at` is `NULL`.

name	A character identifier.
gp	An object of class gpar, typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
draw	A logical value indicating whether graphics output should be produced.
vp	A Grid viewport obect (or NULL).

Details

Both functions create a yaxis grob (a graphical object describing a yaxis), but only grid.yaxis draws the yaxis (and then only if draw is TRUE).

Value

A yaxis grob. grid.yaxis returns the value invisibly.

Children

If the at slot of an xaxis grob is not NULL then the xaxis will have the following children:

major representing the line at the base of the tick marks.

ticks representing the tick marks.

labels representing the tick labels.

If the at slot is NULL then there are no children and ticks are drawn based on the current viewport scale.

Author(s)

Paul Murrell

See Also

Grid, viewport, grid.xaxis

---

grobName *Generate a Name for a Grob*

Description

This function generates a unique (within-session) name for a grob, based on the grob's class.

Usage

```
grobName(grob = NULL, prefix = "GRID")
```

Arguments

grob	A grob object or NULL.
prefix	The prefix part of the name.

Value

A character string of the form `prefix.class(grob).index`

Author(s)

Paul Murrell

`grobWidth` *Create a Unit Describing the Width of a Grob*

Description

These functions create a unit object describing the width or height of a grob. They are generic.

Usage

```
grobWidth(x)
grobHeight(x)
```

Arguments

`x`	A grob object.

Value

A unit object.

Author(s)

Paul Murrell

See Also

`unit` and `stringWidth`

`grobX` *Create a Unit Describing a Grob Boundary Location*

Description

These functions create a unit object describing a location somewhere on the boundary of a grob. They are generic.

Usage

```
grobX(x, theta)
grobY(x, theta)
```

Arguments

x	A grob, or gList, or gTree, or gPath.
theta	An angle indicating where the location is on the grob boundary. Can be one of "east", "north", "west", or "south", which correspond to angles 0, 90, 180, and 270, respectively.

Details

The angle is anti-clockwise with zero corresponding to a line with an origin centred between the extreme points of the shape, and pointing at 3 o'clock.

If the grob describes a single shape, the boundary value should correspond to the exact edge of the shape.

If the grob describes multiple shapes, in most cases, the boundary value will correspond to the edge of a bounding box around all of the shapes. The exception to this is a polygon grob describing multiple polygons; in that case, the edge corresponds to a convex hull around all points of all polygons described by the grob.

Value

A unit object.

Author(s)

Paul Murrell

See Also

unit and grobWidth

---

plotViewport *Create a Viewport with a Standard Plot Layout*

Description

This is a convenience function for producing a viewport with the common S-style plot layout – i.e., a central plot region surrounded by margins given in terms of a number of lines of text.

Usage

```
plotViewport(margins=c(5.1, 4.1, 4.1, 2.1), ...)
```

Arguments

margins	A numeric vector interpreted in the same way as par(mar) in base graphics.
...	All other arguments will be passed to a call to the viewport() function.

Value

A grid viewport object.

Author(s)

Paul Murrell

See Also

viewport and dataViewport.

---

pop.viewport *Pop a Viewport off the Grid Viewport Stack*

Description

Grid maintains a viewport stack — a list of nested drawing contexts.

This function makes the parent of the specified viewport the new default viewport.

Usage

```
pop.viewport(n=1, recording=TRUE)
```

Arguments

n	An integer giving the number of viewports to pop. Defaults to 1.
recording	A logical value to indicate whether the set-viewport operation should be recorded on the Grid display list.

Value

None.

Warning

This function has been deprecated. Please use popViewport instead.

Author(s)

Paul Murrell

See Also

push.viewport.

`push.viewport` *Push a Viewport onto the Grid Viewport Stack*

Description

Grid maintains a viewport stack — a list of nested drawing contexts.

This function makes the specified viewport the default viewport and makes its parent the previous default viewport (i.e., nests the specified context within the previous default context).

Usage

```
push.viewport(..., recording=TRUE)
```

Arguments

`...`	One or more objects of class `"viewport"`, or `NULL`.
`recording`	A logical value to indicate whether the set-viewport operation should be recorded on the Grid display list.

Value

None.

Warning

This function has been deprecated. Please use `pushViewport` instead.

Author(s)

Paul Murrell

See Also

`pop.viewport`.

`Querying the Viewport Tree`
Get the Current Grid Viewport (Tree)

Description

`current.viewport()` returns the viewport that Grid is going to draw into.

`current.vpTree` returns the entire Grid viewport tree.

`current.vpPath` returns the viewport path to the current viewport.

`current.transform` returns the transformation matrix for the current viewport.

Usage

```
current.viewport(vp=NULL)
current.vpTree(all=TRUE)
current.vpPath()
current.transform()
```

Arguments

`vp`	A Grid viewport object. Use of this argument has been deprecated.
`all`	A logical value indicating whether the entire viewport tree should be returned.

Details

If `all` is `FALSE` then `current.vpTree` only returns the subtree below the current viewport.

Value

A Grid viewport object from `current.viewport` or `current.vpTree`.

`current.transform` returns a 4x4 transformation matrix.

The viewport path returned by `current.vpPath` is `NULL` if the current viewport is the `ROOT` viewport

Author(s)

Paul Murrell

See Also

`viewport`

Examples

```
grid.newpage()
pushViewport(viewport(width=0.8, height=0.8, name="A"))
pushViewport(viewport(x=0.1, width=0.3, height=0.6,
  just="left", name="B"))
upViewport(1)
pushViewport(viewport(x=0.5, width=0.4, height=0.8,
  just="left", name="C"))
pushViewport(viewport(width=0.8, height=0.8, name="D"))
current.vpPath()
upViewport(1)
current.vpPath()
current.vpTree()
current.viewport()
current.vpTree(all=FALSE)
popViewport(0)
```

stringWidth *Create a Unit Describing the Width of a String*

Description

These functions create a unit object describing the width or height of a string.

Usage

```
stringWidth(string)
stringHeight(string)
```

Arguments

string	A character vector.

Value

A unit object.

Author(s)

Paul Murrell

See Also

unit and grobWidth

unit *Function to Create a Unit Object*

Description

This function creates a unit object — a vector of unit values. A unit value is typically just a single numeric value with an associated unit.

Usage

```
unit(x, units, data=NULL)
```

Arguments

x	A numeric vector.
units	A character vector specifying the units for the corresponding numeric values.
data	This argument is used to supply extra information for special unit types.

Details

Unit objects allow the user to specify locations and dimensions in a large number of different coordinate systems. All drawing occurs relative to a viewport and the `units` specifies what coordinate system to use within that viewport.

Possible `units` (coordinate systems) are:

`"npc"` Normalised Parent Coordinates (the default). The origin of the viewport is (0, 0) and the viewport has a width and height of 1 unit. For example, (0.5, 0.5) is the centre of the viewport.

`"cm"` Centimetres.

`"inches"` Inches. 1 in = 2.54 cm.

`"mm"` Millimetres. 10 mm = 1 cm.

`"points"` Points. 72.27 pt = 1 in.

`"picas"` Picas. 1 pc = 12 pt.

`"bigpts"` Big Points. 72 bp = 1 in.

`"dida"` Dida. 1157 dd = 1238 pt.

`"cicero"` Cicero. 1 cc = 12 dd.

`"scaledpts"` Scaled Points. 65536 sp = 1 pt.

`"lines"` Lines of text. Locations and dimensions are in terms of multiples of the default text size of the viewport (as specified by the viewport's `fontsize` and `lineheight`).

`"char"` Multiples of nominal font height of the viewport (as specified by the viewport's `fontsize`).

`"native"` Locations and dimensions are relative to the viewport's `xscale` and `yscale`.

`"snpc"` Square Normalised Parent Coordinates. Same as Normalised Parent Coordinates, except gives the same answer for horizontal and vertical locations/dimensions. It uses the *lesser* of npc-width and npc-height. This is useful for making things which are a proportion of the viewport, but have to be square (or have a fixed aspect ratio).

`"strwidth"` Multiples of the width of the string specified in the `data` argument. The font size is determined by the pointsize of the viewport.

`"strheight"` Multiples of the height of the string specified in the `data` argument. The font size is determined by the pointsize of the viewport.

`"grobwidth"` Multiples of the width of the grob specified in the `data` argument.

`"grobheight"` Multiples of the height of the grob specified in the `data` argument.

A special `units` value of `"null"` is also allowed, but only makes sense when used in specifying widths of columns or heights of rows in grid layouts (see `grid.layout`).

The `data` argument must be a list when the `unit.length()` is greater than 1. For example, `unit(rep(1, 3), c("npc", "strwidth", "inches"), data=list(NULL, "my string", NULL))`.

It is possible to subset unit objects in the normal way (e.g., `unit(1:5, "npc")[2:4]`), but a special function `unit.c` is provided for combining unit objects.

Certain arithmetic and summary operations are defined for unit objects. In particular, it is possible to add and subtract unit objects (e.g., `unit(1, "npc") - unit(1, "inches")`), and to specify the minimum or maximum of a list of unit objects (e.g., `min(unit(0.5, "npc"), unit(1, "inches"))`).

Value

An object of class "unit".

WARNING

There is a special function unit.c for concatenating several unit objects.

The c function will not give the right answer.

There used to be "mylines", "mychar", "mystrwidth", "mystrheight" units. These will still be accepted, but work exactly the same as "lines", "char", "strwidth", "strheight".

Author(s)

Paul Murrell

See Also

unit.c

Examples

```
unit(1, "npc")
unit(1:3/4, "npc")
unit(1:3/4, "npc") + unit(1, "inches")
min(unit(0.5, "npc"), unit(1, "inches"))
unit.c(unit(0.5, "npc"), unit(2, "inches") + unit(1:3/4, "npc"),
       unit(1, "strwidth", "hi there"))
```

---

unit.c *Combine Unit Objects*

Description

This function produces a new unit object by combining the unit objects specified as arguments.

Usage

```
unit.c(...)
```

Arguments

...	An arbitrary number of unit objects.

Value

An object of class unit.

Author(s)

Paul Murrell

See Also

unit.

unit.length *Length of a Unit Object*

Description

The length of a unit object is defined as the number of unit values in the unit object.

This function has been deprecated in favour of a unit method for the generic `length` function.

Usage

```
unit.length(unit)
```

Arguments

`unit`	A unit object.

Value

An integer value.

Author(s)

Paul Murrell

See Also

`unit`

Examples

```
length(unit(1:3, "npc"))
length(unit(1:3, "npc") + unit(1, "inches"))
length(max(unit(1:3, "npc") + unit(1, "inches")))
length(max(unit(1:3, "npc") + unit(1, "strwidth", "a"))*4)
length(unit(1:3, "npc") + unit(1, "strwidth", "a")*4)
```

unit.pmin *Parallel Unit Minima and Maxima*

Description

Returns a unit object whose i'th value is the minimum (or maximum) of the i'th values of the arguments.

Usage

```
unit.pmin(...)
unit.pmax(...)
```

Arguments

...	One or more unit objects.

Details

The length of the result is the maximum of the lengths of the arguments; shorter arguments are recycled in the usual manner.

Value

A unit object.

Author(s)

Paul Murrell

Examples

```
max(unit(1:3, "cm"), unit(0.5, "npc"))
unit.pmax(unit(1:3, "cm"), unit(0.5, "npc"))
```

---

unit.rep *Replicate Elements of Unit Objects*

Description

Replicates the units according to the values given in `times` and `length.out`.

This function has been deprecated in favour of a unit method for the generic `rep` function.

Usage

```
unit.rep(x, ...)
```

Arguments

x	An object of class `"unit"`.
...	arguments to be passed to `rep` such as `times` and `length.out`.

Value

An object of class `"unit"`.

Author(s)

Paul Murrell

See Also

`rep`

Examples

```
rep(unit(1:3, "npc"), 3)
rep(unit(1:3, "npc"), 1:3)
rep(unit(1:3, "npc") + unit(1, "inches"), 3)
rep(max(unit(1:3, "npc") + unit(1, "inches")), 3)
rep(max(unit(1:3, "npc") + unit(1, "strwidth", "a"))*4, 3)
rep(unit(1:3, "npc") + unit(1, "strwidth", "a")*4, 3)
```

---

validDetails *Customising grid grob Validation*

Description

This generic hook function is called whenever a grid grob is created or edited via grob, gTree, grid.edit or editGrob. This provides an opportunity for customising the validation of a new class derived from grob (or gTree).

Usage

```
validDetails(x)
```

Arguments

x	A grid grob.

Details

This function is called by grob, gTree, grid.edit and editGrob. A method should be written for classes derived from grob or gTree to validate the values of slots specific to the new class. (e.g., see grid:::validDetails.axis).

Note that the standard slots for grobs and gTrees are automatically validated (e.g., vp, gp slots for grobs and, in addition, children, and childrenvp slots for gTrees) so only slots specific to a new class need to be addressed.

Value

The function MUST return the validated grob.

Author(s)

Paul Murrell

See Also

grid.edit

vpPath *Concatenate Viewport Names*

Description

This function can be used to generate a viewport path for use in `downViewport` or `seekViewport`.

A viewport path is a list of nested viewport names.

Usage

```
vpPath(...)
```

Arguments

`...`	Character values which are viewport names.

Details

Viewport names must only be unique amongst viewports which share the same parent in the viewport tree.

This function can be used to generate a specification for a viewport that includes the viewport's parent's name (and the name of its parent and so on).

For interactive use, it is possible to directly specify a path, but it is strongly recommended that this function is used otherwise in case the path separator is changed in future versions of grid.

Value

A `vpPath` object.

See Also

`viewport`, `pushViewport`, `popViewport`, `downViewport`, `seekViewport`, `upViewport`

Examples

```
vpPath("vp1", "vp2")
```

widthDetails *Width and Height of a grid grob*

Description

These generic functions are used to determine the size of grid grobs.

Usage

```
widthDetails(x)
heightDetails(x)
```

Arguments

x	A grid grob.

Details

These functions are called in the calculation of "grobwidth" and "grobheight" units. Methods should be written for classes derived from grob or gTree where the size of the grob can be determined (see, for example grid:::widthDetails.frame).

Value

A unit object.

Author(s)

Paul Murrell

See Also

absolute.size.

Working with Viewports *Maintaining and Navigating the Grid Viewport Tree*

Description

Grid maintains a tree of viewports — nested drawing contexts.

These functions provide ways to add or remove viewports and to navigate amongst viewports in the tree.

Usage

```
pushViewport(..., recording=TRUE)
popViewport(n, recording=TRUE)
downViewport(name, strict=FALSE, recording=TRUE)
seekViewport(name, recording=TRUE)
upViewport(n, recording=TRUE)
```

Arguments

`...`	One or more objects of class `"viewport"`.
`n`	An integer value indicating how many viewports to pop or navigate up. The special value `0` indicates to pop or navigate viewports right up to the root viewport.
`name`	A character value to identify a viewport in the tree.
`strict`	A boolean indicating whether the vpPath must be matched exactly.
`recording`	A logical value to indicate whether the viewport operation should be recorded on the Grid display list.

Details

Objects created by the `viewport()` function are only descriptions of a drawing context. A viewport object must be pushed onto the viewport tree before it has any effect on drawing.

The viewport tree always has a single root viewport (created by the system) which corresponds to the entire device (and default graphical parameter settings). Viewports may be added to the tree using `pushViewport()` and removed from the tree using `popViewport()`.

There is only ever one current viewport, which is the current position within the viewport tree. All drawing and viewport operations are relative to the current viewport. When a viewport is pushed it becomes the current viewport. When a viewport is popped, the parent viewport becomes the current viewport. Use `upViewport` to navigate to the parent of the current viewport, without removing the current viewport from the viewport tree. Use `downViewport` to navigate to a viewport further down the viewport tree and `seekViewport` to navigate to a viewport anywhere else in the tree.

If a viewport is pushed and it has the same `name` as a viewport at the same level in the tree, then it replaces the existing viewport in the tree.

Value

`downViewport` returns the number of viewports it went down.

This can be useful for returning to your starting point by doing something like `depth <- downViewport()` then `upViewport(depth)`.

Author(s)

Paul Murrell

See Also

`viewport` and `vpPath`.

Examples

```
# push the same viewport several times
grid.newpage()
vp <- viewport(width=0.5, height=0.5)
pushViewport(vp)
grid.rect(gp=gpar(col="blue"))
grid.text("Quarter of the device",
  y=unit(1, "npc") - unit(1, "lines"), gp=gpar(col="blue"))
pushViewport(vp)
grid.rect(gp=gpar(col="red"))
grid.text("Quarter of the parent viewport",
```

```
  y=unit(1, "npc") - unit(1, "lines"), gp=gpar(col="red"))
popViewport(2)
# push several viewports then navigate amongst them
grid.newpage()
grid.rect(gp=gpar(col="grey"))
grid.text("Top-level viewport",
  y=unit(1, "npc") - unit(1, "lines"), gp=gpar(col="grey"))
if (interactive()) Sys.sleep(1.0)
pushViewport(viewport(width=0.8, height=0.7, name="A"))
grid.rect(gp=gpar(col="blue"))
grid.text("1. Push Viewport A",
  y=unit(1, "npc") - unit(1, "lines"), gp=gpar(col="blue"))
if (interactive()) Sys.sleep(1.0)
pushViewport(viewport(x=0.1, width=0.3, height=0.6,
  just="left", name="B"))
grid.rect(gp=gpar(col="red"))
grid.text("2. Push Viewport B (in A)",
  y=unit(1, "npc") - unit(1, "lines"), gp=gpar(col="red"))
if (interactive()) Sys.sleep(1.0)
upViewport(1)
grid.text("3. Up from B to A",
  y=unit(1, "npc") - unit(2, "lines"), gp=gpar(col="blue"))
if (interactive()) Sys.sleep(1.0)
pushViewport(viewport(x=0.5, width=0.4, height=0.8,
  just="left", name="C"))
grid.rect(gp=gpar(col="green"))
grid.text("4. Push Viewport C (in A)",
  y=unit(1, "npc") - unit(1, "lines"), gp=gpar(col="green"))
if (interactive()) Sys.sleep(1.0)
pushViewport(viewport(width=0.8, height=0.6, name="D"))
grid.rect()
grid.text("5. Push Viewport D (in C)",
  y=unit(1, "npc") - unit(1, "lines"))
if (interactive()) Sys.sleep(1.0)
upViewport(0)
grid.text("6. Up from D to top-level",
  y=unit(1, "npc") - unit(2, "lines"), gp=gpar(col="grey"))
if (interactive()) Sys.sleep(1.0)
downViewport("D")
grid.text("7. Down from top-level to D",
  y=unit(1, "npc") - unit(2, "lines"))
if (interactive()) Sys.sleep(1.0)
seekViewport("B")
grid.text("8. Seek from D to B",
  y=unit(1, "npc") - unit(2, "lines"), gp=gpar(col="red"))
pushViewport(viewport(width=0.9, height=0.5, name="A"))
grid.rect()
grid.text("9. Push Viewport A (in B)",
  y=unit(1, "npc") - unit(1, "lines"))
if (interactive()) Sys.sleep(1.0)
seekViewport("A")
grid.text("10. Seek from B to A (in ROOT)",
  y=unit(1, "npc") - unit(3, "lines"), gp=gpar(col="blue"))
if (interactive()) Sys.sleep(1.0)
seekViewport(vpPath("B", "A"))
grid.text("11. Seek from\nA (in ROOT)\nto A (in B)")
popViewport(0)
```

`xDetails` *Boundary of a grid grob*

Description

These generic functions are used to determine a location on the boundary of a grid grob.

Usage

```
xDetails(x, theta)
yDetails(x, theta)
```

Arguments

`x`	A grid grob.
`theta`	A numeric angle, in degrees, measured anti-clockwise from the 3 o'clock *or* one of the following character strings: `"north"`, `"east"`, `"west"`, `"south"`.

Details

The location on the grob boundary is determined by taking a line from the centre of the grob at the angle `theta` and intersecting it with the convex hull of the grob (for the basic grob primitives, the centre is determined as half way between the minimum and maximum values in x and y directions).

These functions are called in the calculation of `"grobx"` and `"groby"` units as produced by the `grobX` and `grobY` functions. Methods should be written for classes derived from grob or gTree where the boundary of the grob can be determined.

Value

A unit object.

Author(s)

Paul Murrell

See Also

`grobX`, `grobY`.

Chapter 6

The `methods` package

`methods-package` *Formal Methods and Classes*

Description

Formally defined methods and classes for R objects, plus other programming tools, as described in the references.

Details

This package provides the "S4" or "S version 4" approach to methods and classes in a functional language.

For a complete list of functions and classes, use `library(help="methods")`.

Author(s)

R Development Core Team

Maintainer: R Core Team ⟨R-core@r-project.org⟩

References

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the methods package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details see `?Methods` and the links from that documentation.

.BasicFunsList *List of Builtin and Special Functions*

Description

A named list providing instructions for turning builtin and special functions into generic functions.

Functions in R that are defined as `.Primitive(<name>)` are not suitable for formal methods, because they lack the basic reflectance property. You can't find the argument list for these functions by examining the function object itself.

Future versions of R may fix this by attaching a formal argument list to the corresponding function. While generally the names of arguments are not checked by the internal code implementing the function, the number of arguments frequently is.

In any case, some definition of a formal argument list is needed if users are to define methods for these functions. In particular, if methods are to be merged from multiple packages, the different sets of methods need to agree on the formal arguments.

In the absence of reflectance, this list provides the relevant information via a dummy function associated with each of the known specials for which methods are allowed.

At the same, the list flags those specials for which methods are meaningless (e.g., `for`) or just a very bad idea (e.g., `.Primitive`).

A generic function created via `setMethod`, for example, for one of these special functions will have the argument list from `.BasicFunsList`. If no entry exists, the argument list `(x, ...)` is assumed.

as *Force an Object to Belong to a Class*

Description

These functions manage the relations that allow coercing an object to a given class.

Usage

```
as(object, Class, strict=TRUE, ext)

as(object, Class) <- value

setAs(from, to, def, replace, where = topenv(parent.frame()))
```

Arguments

`object`	any R object.
`Class`	the name of the class to which `object` should be coerced.

`strict` logical flag. If `TRUE`, the returned object must be strictly from the target class (unless that class is a virtual class, in which case the object will be from the closest actual class (often the original object, if that class extends the virtual class directly).

If `strict = FALSE`, any simple extension of the target class will be returned, without further change. A simple extension is, roughly, one that just adds slots to an existing class.

`value` The value to use to modify `object` (see the discussion below). You should supply an object with class `Class`; some coercion is done, but you're unwise to rely on it.

`from, to` The classes between which `def` performs coercion.

(In the case of the `coerce` function these are objects from the classes, not the names of the classes, but you're not expected to call `coerce` directly.)

`def` function of one argument. It will get an object from class `from` and had better return an object of class `to`. (If you want to save `setAs` a little work, make the name of the argument `from`, but don't worry about it, `setAs` will do the conversion.)

`replace` if supplied, the function to use as a replacement method.

`where` the position or environment in which to store the resulting method for `coerce`.

`ext` the optional object defining how `Class` is extended by the class of the object (as returned by `possibleExtends`). This argument is used internally (to provide essential information for non-public classes), but you are unlikely to want to use it directly.

Summary of Functions

`as`: Returns the version of this object coerced to be the given `Class`.

If the corresponding `is(object, Class)` relation is true, it will be used. In particular, if the relation has a coerce method, the method will be invoked on `object`. However, if the object's class extends `Class` in a simple way (e.g, by including the superclass in the definition, then the actual coercion will be done only if `strict` is `TRUE` (non-strict coercion, is used in passing objects to methods).

Coerce methods are pre-defined for basic classes (including all the types of vectors, functions and a few others). See `showMethods(coerce)` for a list of these.

Beyond these two sources of methods, further methods are defined by calls to the `setAs` function.

`coerce`: Coerce `from` to be of the same class as `to`.

Not a function you should usually call explicitly. The function `setAs` creates methods for `coerce` for the `as` function to use.

`setAs`: The function supplied as the third argument is to be called to implement `as(x, to)` when `x` has class `from`. Need we add that the function should return a suitable object with class `to`.

How Functions 'as' and 'setAs' Work

The function `as` contrives to turn `object` into an object with class `Class`. In doing so, it uses information about classes and methods, but in a somewhat special way. Keep in mind that objects from one class can turn into objects from another class either automatically or by an explicit call to the `as` function. Automatic conversion is special, and comes from the designer of one class of objects asserting that this class extends another class (see `setClass` and `setIs`).

Because inheritance is a powerful assertion, it should be used sparingly (otherwise your computations may produce unexpected, and perhaps incorrect, results). But objects can also be converted explicitly, by calling `as`, and that conversion is designed to use any inheritance information, as well as explicit methods.

As a first step in conversion, the `as` function determines whether `is(object, Class)` is `TRUE`. This can be the case either because the class definition of `object` includes `Class` as a "super class" (directly or indirectly), or because a call to `setIs` established the relationship.

Either way, the inheritance relation defines a method to coerce `object` to `Class`. In the most common case, the method is just to extract from `object` the slots needed for `Class`, but it's also possible to specify a method explicitly in a `setIs` call.

So, if inheritance applies, the `as` function calls the appropriate method. If inheritance does not apply, and `coerceFlag` is `FALSE`, `NULL` is returned.

By default, `coerceFlag` is `TRUE`. In this case the `as` function goes on to look for a method for the function `coerce` for the signature `c(from = class(object), to = Class)`.

Method selection is used in the `as` function in two special ways. First, inheritance is applied for the argument `from` but not for the argument `to` (if you think about it, you'll probably agree that you wouldn't want the result to be from some class other than the `Class` specified). Second, the function tries to use inheritance information to convert the object indirectly, by first converting it to an inherited class. It does this by examining the classes that the `from` class extends, to see if any of them has an explicit conversion method. Suppose class `"by"` does: Then the `as` function implicitly computes `as(as(object, "by"), Class)`.

With this explanation as background, the function `setAs` does a fairly obvious computation: It constructs and sets a method for the function `coerce` with signature `c(from, to)`, using the `def` argument to define the body of the method. The function supplied as `def` can have one argument (interpreted as an object to be coerced) or two arguments (the `from` object and the `to` class). Either way, `setAs` constructs a function of two arguments, with the second defaulting to the name of the `to` class. The method will be called from `as` with the object as the only argument: The default for the second argument is provided so the method can know the intended `to` class.

The function `coerce` exists almost entirely as a repository for such methods, to be selected as described above by the `as` function. In fact, it would usually be a bad idea to call `coerce` directly, since then you would get inheritance on the `to` argument; as mentioned, this is not likely to be what you want.

The Function 'as' Used in Replacements

When `as` appears on the left of an assignment, the intuitive meaning is "Replace the part of `object` that was inherited from `Class` by the `value` on the right of the assignment."

This usually has a straightforward interpretation, but you can control explicitly what happens, and sometimes you should to avoid possible corruption of objects.

When `object` inherits from `Class` in the usual way, by including the slots of `Class`, the default `as` method is to set the corresponding slots in `object` to those in `value`.

The default computation may be reasonable, but usually only if all *other* slots in `object` are unrelated to the slots being changed. Often, however, this is not the case. The class of `object` may have extended `Class` with a new slot whose value depends on the inherited slots. In this case, you may want to define a method for replacing the inherited information that recomputes all the dependent information. Or, you may just want to prohibit replacing the inherited information directly .

The way to control such replacements is through the `replace` argument to function `setIs`. This argument is a method that function `as` calls when used for replacement. It can do whatever you

like, including calling `stop` if you want to prohibit replacements. It should return a modified object with the same class as the `object` argument to `as`.

In R, you can also explicitly supply a replacement method, even in the case that inheritance does not apply, through the `replace` argument to `setAs`. It works essentially the same way, but in this case by constructing a method for `"coerce<-"`. (Replace methods for coercion without inheritance are not in the original description and so may not be compatible with S-Plus, at least not yet.)

When inheritance does apply, coerce and replace methods can be specified through either `setIs` or `setAs`; the effect is essentially the same.

Basic Coercion Methods

Methods are pre-defined for coercing any object to one of the basic datatypes. For example, `as(x, "numeric")` uses the existing `as.numeric` function. These built-in methods can be listed by `showMethods("coerce")`.

References

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the **methods** package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details see `?Methods` and the links from that documentation.

See Also

If you think of using `try(as(x, cl))`, consider `canCoerce(x, cl)` instead.

Examples

```
## using the definition of class "track" from Classes



setAs("track", "numeric", function(from) from@y)

t1 <- new("track", x=1:20, y=(1:20)^2)

as(t1, "numeric")

## The next example shows:
##  1. A virtual class to define setAs for several classes at once.
##  2. as() using inherited information

setClass("ca", representation(a = "character", id = "numeric"))

setClass("cb", representation(b = "character", id = "numeric"))

setClass("id")
setIs("ca", "id")
setIs("cb", "id")
```

```
setAs("id", "numeric", function(from) from@id)

CA <- new("ca", a = "A", id = 1)
CB <- new("cb", b = "B", id = 2)

setAs("cb", "ca", function(from, to )new(to, a=from@b, id = from@id))

as(CB, "numeric")
```

`BasicClasses` *Classes Corresponding to Basic Data Types*

Description

Formal classes exist corresponding to the basic R data types, allowing these types to be used in method signatures, as slots in class definitions, and to be extended by new classes.

Usage

```
### The following are all basic vector classes.
### They can appear as class names in method signatures,
### in calls to as(), is(), and new().
"character"
"complex"
"double"
"expression"
"integer"
"list"
"logical"
"numeric"
"single"
"raw"

### the class
"vector"
### is a virtual class, extended by all the above

### The following are additional basic classes
"NULL"     #  NULL objects
"function" #  function objects, including primitives
"externalptr" # raw external pointers for use in C code

"ANY"  # virtual classes used by the methods package itself
"VIRTUAL"
"missing"
```

Objects from the Classes

Objects can be created by calls of the form `new(Class, ...)`, where `Class` is the quoted class name, and the remaining arguments if any are objects to be interpreted as vectors of this class. Multiple arguments will be concatenated.

The class `"expression"` is slightly odd, in that the ... arguments will *not* be evaluated; therefore, don't enclose them in a call to `quote()`.

Extends

Class `"vector"`, directly.

Methods

coerce Methods are defined to coerce arbitrary objects to these classes, by calling the corresponding basic function, for example, `as(x, "numeric")` calls `as.numeric(x)`.

`callNextMethod` *Call an Inherited Method*

Description

A call to `callNextMethod` can only appear inside a method definition. It then results in a call to the first inherited method after the current method, with the arguments to the current method passed down to the next method. The value of that method call is the value of `callNextMethod`.

Usage

```
callNextMethod(...)
```

Arguments

`...`	Optionally, the arguments to the function in its next call (but note that the dispatch is as in the detailed description below; the arguments have no effect on selecting the next method.) If no arguments are included in the call to `callNextMethod`, the effect is to call the method with the current arguments. See the detailed description for what this really means. Calling with no arguments is often the natural way to use `callNextMethod`; see the examples.

Details

The "next" method (i.e., the first inherited method) is defined to be that method which *would* have been called if the current method did not exist. This is more-or-less literally what happens: The current method (to be precise, the method with signature given by the `defined` slot of the method from which `callNextMethod` is called) is deleted from a copy of the methods for the current generic, and `selectMethod` is called to find the next method (the result is cached in a special object, so the search only typically happens once per session per combination of argument classes).

Note that the preceding definition means that the next method is defined uniquely when `setMethod` inserts the method containing the `callNextMethod` call, given the definitions of

the classes in the signature. The choice does not depend on the path that gets us to that method (for example, through inheritance or from another `callNextMethod` call). This definition was not enforced in versions of R prior to 2.3.0, where the method was selected based on the target signature, and so could vary depending on the actual arguments.

It is also legal, and often useful, for the method called by `callNextMethod` to itself have a call to `callNextMethod`. This generally works as you would expect, but for completeness be aware that it is possible to have ambiguous inheritance in the S structure, in the sense that the same two classes can appear as superclasses *in the opposite order* in two other class definitions. In this case the effect of a nested instance of `callNextMethod` is not well defined. Such inconsistent class hierarchies are both rare and nearly always the result of bad design, but they are possible, and currently undetected.

The statement that the method is called with the current arguments is more precisely as follows. Arguments that were missing in the current call are still missing (remember that `"missing"` is a valid class in a method signature). For a formal argument, say `x`, that appears in the original call, there is a corresponding argument in the next method call equivalent to "`x = x`". In effect, this means that the next method sees the same actual arguments, but arguments are evaluated only once.

Value

The value returned by the selected method.

References

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the **methods** package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details see `?Methods` and the links from that documentation.

See Also

Methods for the general behavior of method dispatch

Examples

```
## some class definitions with simple inheritance
setClass("B0" , representation(b0 = "numeric"))

setClass("B1", representation(b1 = "character"), contains = "B0")

setClass("B2", representation(b2 = "logical"), contains = "B1")

## and a rather silly function to illustrate callNextMethod

f <- function(x) class(x)

setMethod("f", "B0", function(x) c(x@b0^2, callNextMethod()))
setMethod("f", "B1", function(x) c(paste(x@b1,":"), callNextMethod()))
setMethod("f", "B2", function(x) c(x@b2, callNextMethod()))

b1 <- new("B1", b0 = 2, b1 = "Testing")
```

```
b2 <- new("B2", b2 = FALSE, b1 = "More testing", b0 = 10)

f(b2)

f(b1)
```

---

canCoerce *Can an Object be Coerced to a Certain S4 Class?*

Description

Test if an object can be coerced to a given S4 class. Maybe useful inside `if()` to ensure that calling `as(object, Class)` will find a method.

Usage

```
canCoerce(object, Class)
```

Arguments

`object`	any R object, typically of a formal S4 class.
`Class`	an S4 class (see `isClass`

.

Value

a scalar logical, `TRUE` if there is a `coerce` method (as defined by `setAs`, e.g.) for the signature `(from = class(object), to = Class)`.

See Also

`as`, `setAs`, `selectMethod`, `setClass`,

Examples

```
m <- matrix(pi, 2,3)
canCoerce(m, "numeric") # TRUE
canCoerce(m, "array")   # TRUE
```

cbind2 *Combine two Objects by Columns or Rows*

Description

Combine two "matrix-like" R objects by columns (cbind2) or rows (rbind2). These are (S4) generic functions with default methods.

Usage

```
cbind2(x, y)
rbind2(x, y)
```

Arguments

x	any R object, typically "matrix-like".
y	any R object, typically similar to x, or missing completely.

Details

The main use of cbind2 (rbind2) is to be called by cbind() (rbind()) **if** these are activated. This allows cbind (rbind) to "work" for formally classed (aka 'S4') objects by providing S4 methods for these objects. Currently, a call
methods:::bind_activation(TRUE)
is needed to install a "cbind2-calling" version of cbind (into the base namespace) and the same for rbind.
methods:::bind_activation(FALSE) reverts to the previous internal version of cbind which does not build on cbind2, see the examples.

Value

A matrix (or matrix like object) combining the columns (or rows) of x and y.

Methods

x = "ANY", y = "ANY" the default method using R's internal code.

x = "ANY", y = "missing" the default method for one argument using R's internal code.

See Also

cbind, rbind.

Examples

```
cbind2(1:3, 4)
m <- matrix(3:8, 2,3, dimnames=list(c("a","b"), LETTERS[1:3]))
cbind2(1:2, m) # keeps dimnames from m

### Note: Use the following activation if you want cbind() to work
### ----  on S4 objects -- be careful otherwise!

methods:::bind_activation(on = TRUE)
```

```
trace("cbind2")
cbind(a=1:3)# no call to cbind2()
cbind(a=1:3, four=4, 7:9)# calling cbind2() twice
untrace("cbind2")



## The following fails currently,
## since cbind() works recursively from the tail:
try( cbind(m, a=1, b=3) )

## turn off the `special cbind()' :
methods:::bind_activation(FALSE)
```

`Classes` *Class Definitions*

Description

Class definitions are objects that contain the formal definition of a class of R objects, usually referred to as an S4 class, to distinguish them from the informal S3 classes. This document gives an overview of S4 classes; for details of the class representation objects themselves, see `class?classRepresentation` (`classRepresentation-class`).

Details

When a class is defined, an object is stored that contains the information about that class, including:

Slots: The data contained in an object from an S4 class is defined by the *slots* in the class definition.

Each slot in an object is a component of the object; like components (that is, elements) of a list, these may be extracted and set, using the function `slot()` or more often the operator `"@"`. However, they differ from list components in important ways. First, slots can only be referred to by name, not by position, and there is no partial matching of names as with list elements.

All the objects from a particular class have the same set of slot names; specifically, the slot names that are contained in the class definition. Each slot in each object always has the same class; again, this is defined by the overall class definition. The phrase "has the same class" means that the class of the object in the slot must be the same as the class specified in the definition, or some class that extends the one in the definition.

One class name is special, `.Data`. This stands for the "data part" of the object. Any class that contains one of the basic data types in R, has implicitly a corresponding `.Data` slot of that type, allowing computations to extract or replace the data part while leaving other slots unchanged. The `.Data` slot also determines the type of the object; if `x` has a `.Data` slot, the type of the slot is the type of the object (that is, the value of `typeof(x)`. Otherwise the type of the object is `"S4"`. Extending a basic type this way allows objects to use old-style code for the corresponding type as well as S4 methods. Any basic type can be used for `.Data`, with the exception of a few that do not behave like ordinary objects; namely, `"NULL"`, environments, and external pointers.

Classes exist for which there are no actual objects, the *virtual* classes, in fact a very important programming tool. They are used to group together ordinary classes that want to share some

programming behavior, without necessarily restricting how the behavior is implemented. Virtual class definitions may if you want include slots (to provide some common behavior without fully defining the object—see traceable-class for an example).

A simple and useful form of virtual class is the *class union*, a virtual class that is defined in a call to `setClassUnion` by listing one or more of subclasses (classes that extend the class union). Class unions can include as subclasses basic data types (whose definition is otherwise sealed).

Superclasses: The definition of a class includes the *superclasses* —the classes that this class extends. A class `Fancy`, say, extends a class `Simple` if an object from the `Fancy` class has all the capabilities of the `Simple` class (and probably some more as well). In particular, and very usefully, any method defined to work for a `Simple` object can be applied to a `Fancy` object as well.

This relationship is expressed equivalently by saying that `Simple` is a superclass of `Fancy`, or that `Fancy` is a subclass of `Simple`.

The direct superclasses of a class are those superclasses explicitly defined. Direct superclasses can be defined in three ways. Most commonly, the superclasses are listed in the `contains=` argument in the call to `setClass` that creates the subclass. In this case the subclass will contain all the slots of the superclass, and the relation between the class is called *simple*, as it in fact is. Superclasses can also be defined explicitly by a call to `setIs`; in this case, the relation requires methods to be specified to go from subclass to superclass. Thirdly, a class union is a superclass of all the members of the union. In this case too the relation is simple, but notice that the relation is defined when the superclass is created, not when the subclass is created as with the `contains=` mechanism.

The definition of a superclass will also potentially contain its own direct superclasses. These are considered (and shown) as superclasses at distance 2 from the original class; their direct superclasses are at distance 3, and so on. All these are legitimate superclasses for purposes such as method selection.

When superclasses are defined by including the names of superclasses in the `contains=` argument to `setClass`, an object from the class will have all the slots defined for its own class *and* all the slots defined for all its superclasses as well.

The information about the relation between a class and a particular superclass is encoded as an object of class `"SClassExtension"` (see SClassExtension-class). A list of such objects for the superclasses (and sometimes for the subclasses) is included in the metadata object defining the class. If you need to compute with these objects (for example, to compare the distances), call the function `extends` with argument `fullInfo=TRUE`.

Objects: The objects from a class, typically created by a call to `new` or by assigning another object from the class, are defined by the *prototype* object for the class and by additional arguments in the call to `new`, which are passed to a method for that class for the function `initialize`.

Each class definition contains a prototype object for the class. This must have values for all the slots defined by the class definition. By default, these are the prototypes of all the slot classes, if those are not virtual classes. However, the definition of the class can specify any valid object for any of the slots.

There are a number of "basic" classes, corresponding to the ordinary kinds of data occurring in R. For example, `"numeric"` is a class corresponding to numeric vectors. There are also basic classes corresponding to objects in the language, such as `"function"` and `"call"`, and for specialized objects, such as `"environment"` These classes are predefined and can then be used as slots or as superclasses for any other class definitions. The prototypes for the vector classes are vectors of length 0 of the corresponding type. Notice that basic classes are unusual in that the prototype object is from the class itself.

There are also a few basic virtual classes, the most important being `"vector"`, grouping together all the vector classes; and `"language"`, grouping together all the types of objects

making up the R language.

References

The functions in this package emulate the facility for classes and methods described in *Programming with Data* (John M. Chambers, Springer, 1998). See this book for further details and examples.

See Also

Methods, setClass, is, as, new, slot

---

classRepresentation-class
Class Objects

Description

These are the objects that hold the definition of classes of objects. They are constructed and stored as meta-data by calls to the function setClass. Don't manipulate them directly, except perhaps to look at individual slots.

Details

Class definitions are stored as metadata in various packages. Additional metadata supplies information on inheritance (the result of calls to setIs). Inheritance information implied by the class definition itself (because the class contains one or more other classes) is also constructed automatically.

When a class is to be used in an R session, this information is assembled to complete the class definition. The completion is a second object of class "classRepresentation", cached for the session or until something happens to change the information. A call to getClass returns the completed definition of a class; a call to getClassDef returns the stored definition (uncompleted).

In particular, completion fills in the upward- and downward-pointing inheritance information for the class, in slots contains and subclasses respectively. It's in principle important to note that this information can depend on which packages are installed, since these may define additional subclasses or superclasses.

Slots

slots: A named list of the slots in this class; the elements of the list are the classes to which the slots must belong (or extend), and the names of the list gives the corresponding slot names.

contains: A named list of the classes this class "contains"; the elements of the list are objects of SClassExtension-class. The list may be only the direct extensions or all the currently known extensions (see the details).

virtual: Logical flag, set to TRUE if this is a virtual class.

prototype: The object that represents the standard prototype for this class; i.e., the data and slots returned by a call to new for this class with no special arguments. Don't mess with the prototype object directly.

validity: Optionally, a function to be used to test the validity of objects from this class. See validObject.

access: Access control information. Not currently used.

className: The character string name of the class.

package: The character string name of the package to which the class belongs. Nearly always the package on which the metadata for the class is stored, but in operations such as constructing inheritance information, the internal package name rules.

subclasses: A named list of the classes known to extend this class'; the elements of the list are objects of `SClassExtension-class`. The list is currently only filled in when completing the class definition (see the details).

versionKey: Object of class `"externalptr"`; eventually will perhaps hold some versioning information, but not currently used.

sealed: Object of class `"logical"`; is this class sealed? If so, no modifications are allowed.

See Also

See function `setClass` to supply the information in the class definition. See Classes for a more basic discussion of class information.

`Documentation` *Using and Creating On-line Documentation for Classes and Methods*

Description

Special documentation can be supplied to describe the classes and methods that are created by the software in the methods package. Techniques to access this documentation and to create it in R help files are described here.

Getting documentation on classes and methods

You can ask for on-line help for class definitions, for specific methods for a generic function, and for general discussion of methods for a generic function. These requests use the `?` operator (see `help` for a general description of the operator). Of course, you are at the mercy of the implementer as to whether there *is* any documentation on the corresponding topics.

Documentation on a class uses the argument `class` on the left of the `?`, and the name of the class on the right; for example,

```
class ?  genericFunction
```

to ask for documentation on the class `"genericFunction"`.

When you want documentation for the methods defined for a particular function, you can ask either for a general discussion of the methods or for documentation of a particular method (that is, the method that would be selected for a particular set of actual arguments).

Overall methods documentation is requested by calling the `?` operator with `methods` as the left-side argument and the name of the function as the right-side argument. For example,

```
methods ?  initialize
```

asks for documentation on the methods for the `initialize` function.

Asking for documentation on a particular method is done by giving a function call expression as the right-hand argument to the `"?"` operator. There are two forms, depending on whether you prefer to give the class names for the arguments or expressions that you intend to use in the actual call.

If you planned to evaluate a function call, say `myFun(x, sqrt(wt))` and wanted to find out something about the method that would be used for this call, put the call on the right of the `"?"` operator:

```
?myFun(x, sqrt(wt))
```

A method will be selected, as it would be for the call itself, and documentation for that method will be requested. If `myFun` is not a generic function, ordinary documentation for the function will be requested.

If you know the actual classes for which you would like method documentation, you can supply these explicitly in place of the argument expressions. In the example above, if you want method documentation for the first argument having class `"maybeNumber"` and the second `"logical"`, call the `"?"` operator, this time with a left-side argument `method`, and with a function call on the right using the class names as arguments:

```
method ?  myFun("maybeNumber", "logical")
```

Once again, a method will be selected, this time corresponding to the specified classes, and method documentation will be requested. This version only works with generic functions.

The two forms each have advantages. The version with actual arguments doesn't require you to figure out (or guess at) the classes of the arguments. On the other hand, evaluating the arguments may take some time, depending on the example. The version with class names does require you to pick classes, but it's otherwise unambiguous. It has a subtler advantage, in that the classes supplied may be virtual classes, in which case no actual argument will have specifically this class. The class `"maybeNumber"`, for example, might be a class union (see the example for `setClassUnion`).

In either form, methods will be selected as they would be in actual computation, including use of inheritance and group generic functions. See `selectMethod` for the details, since it is the function used to find the appropriate method.

Writing Documentation for Methods

The on-line documentation for methods and classes uses some extensions to the R documentation format to implement the requests for class and method documentation described above. See the document *Writing R Extensions* for the available markup commands (you should have consulted this document already if you are at the stage of documenting your software).

In addition to the specific markup commands to be described, you can create an initial, overall file with a skeleton of documentation for the methods defined for a particular generic function:

```
promptMethods("myFun")
```

will create a file, 'myFun-methods.Rd' with a skeleton of documentation for the methods defined for function `myFun`. The output from `promptMethods` is suitable if you want to describe all or most of the methods for the function in one file, separate from the documentation of the generic function itself. Once the file has been filled in and moved to the 'man' subdirectory of your source package, requests for methods documentation will use that file, both for specific methods documentation as described above, and for overall documentation requested by

```
methods ?  myFun
```

You are not required to use `promptMethods`, and if you do, you may not want to use the entire file created:

- If you want to document the methods in the file containing the documentation for the generic function itself, you can cut-and-paste to move the `\alias` lines and the `Methods` section from the file created by `promptMethods` to the existing file.

- On the other hand, if these are auxiliary methods, and you only want to document the added or modified software, you should strip out all but the relevant `\alias` lines for the methods of interest, and remove all but the corresponding `\item` entries in the `Methods` section. Note that in this case you will usually remove the first `\alias` line as well, since that is the marker for general methods documentation on this function (in the example, `\alias{myfun-methods}`).

If you simply want to direct documentation for one or more methods to a particular R documentation file, insert the appropriate alias.

`environment-class` *Class "environment"*

Description

A formal class for R environments.

Objects from the Class

Objects can be created by calls of the form `new("environment", ...)`. The arguments in `...`, if any, should be named and will be assigned to the newly created environment.

Methods

coerce `signature(from = "ANY", to = "environment")`: calls `as.environment`.

initialize `signature(object = "environment")`: Implements the assignments in the new environment. Note that the `object` argument is ignored; a new environment is *always* created, since environments are not protected by copying.

See Also

`new.env`

`fixPre1.8` *Fix Objects Saved from R Versions Previous to 1.8*

Description

Beginning with R version 1.8.0, the class of an object contains the identification of the package in which the class is defined. The function `fixPre1.8` fixes and re-assigns objects missing that information (typically because they were loaded from a file saved with a previous version of R.)

Usage

```
fixPre1.8(names, where)
```

Arguments

names	Character vector of the names of all the objects to be fixed and re-assigned.
where	The environment from which to look for the objects, and for class definitions. Defaults to the top environment of the call to fixPre1.8, the global environment if the function is used interactively.

Details

The named object will be saved where it was found. Its class attribute will be changed to the full form required by R 1.8; otherwise, the contents of the object should be unchanged.

Objects will be fixed and re-assigned only if all the following conditions hold:

1. The named object exists.
2. It is from a defined class (not a basic datatype which has no actual class attribute).
3. The object appears to be from an earlier version of R.
4. The class is currently defined.
5. The object is consistent with the current class definition.

If any condition except the second fails, a warning message is generated.

Note that fixPre1.8 currently fixes *only* the change in class attributes. In particular, it will not fix binary versions of packages installed with earlier versions of R if these use incompatible features. Such packages must be re-installed from source, which is the wise approach always when major version changes occur in R.

Value

The names of all the objects that were in fact re-assigned.

---

genericFunction-class

Generic Function Objects

Description

Generic functions (objects from or extending class genericFunction) are extended function objects, containing information used in creating and dispatching methods for this function. They also identify the package associated with the function and its methods.

Objects from the Class

Generic functions are created and assigned by setGeneric or setGroupGeneric and, indirectly, by setMethod.

As you might expect setGeneric and setGroupGeneric create objects of class "genericFunction" and "groupGenericFunction" respectively.

Slots

.Data: Object of class "function", the function definition of the generic, usually created automatically as a call to standardGeneric.

generic: Object of class "character", the name of the generic function.

package: Object of class "character", the name of the package to which the function definition belongs (and *not* necessarily where the generic function is stored). If the package is not specified explicitly in the call to setGeneric, it is usually the package on which the corresponding non-generic function exists.

group: Object of class "list", the group or groups to which this generic function belongs. Empty by default.

valueClass: Object of class "character"; if not an empty character vector, identifies one or more classes. It is asserted that all methods for this function return objects from these class (or from classes that extend them).

signature: Object of class "character", the vector of formal argument names that can appear in the signature of methods for this generic function. By default, it is all the formal arguments, except for .... Order matters for efficiency: the most commonly used arguments in specifying methods should come first.

default: Object of class "OptionalMethods", the default method for this function. Generated automatically and used to initialize method dispatch.

skeleton: Object of class "call", a slot used internally in method dispatch. Don't expect to use it directly.

Extends

Class "function", from data part.
Class "OptionalMethods", by class "function".
Class "PossibleMethod", by class "function".

Methods

Generic function objects are used in the creation and dispatch of formal methods; information from the object is used to create methods list objects and to merge or update the existing methods for this generic.

GenericFunctions *Tools for Managing Generic Functions*

Description

The functions documented here manage collections of methods associated with a generic function, as well as providing information about the generic functions themselves.

Usage

```
isGeneric(f, where, fdef, getName = FALSE)
isGroup(f, where, fdef)
removeGeneric(f, where)

dumpMethod(f, signature, file, where, def)
```

```
findFunction(f, generic = TRUE, where = topenv(parent.frame()))
dumpMethods(f, file, signature, methods, where)
signature(...)

removeMethods(f, where = topenv(parent.frame()), all = TRUE)
setReplaceMethod(f, ..., where = topenv(parent.frame()))

getGenerics(where, searchForm = FALSE)
allGenerics(where, searchForm = FALSE)
callGeneric(...)
```

Arguments

f	The character string naming the function.
where	The environment, namespace, or search-list position from which to search for objects. By default, start at the top-level environment of the calling function, typically the global environment (i.e., use the search list), or the namespace of a package from which the call came. It is important to supply this argument when calling any of these functions indirectly. With package namespaces, the default is likely to be wrong in such calls.
signature	The class signature of the relevant method. A signature is a named or unnamed vector of character strings. If named, the names must be formal argument names for the generic function. If `signature` is unnamed, the default is to use the first `length(signature)` formal arguments of the function.
file	The file on which to dump method definitions.
def	The function object defining the method; if omitted, the current method definition corresponding to the signature.
...	Named or unnamed arguments to form a signature.
generic	In testing or finding functions, should generic functions be included. Supply as `FALSE` to get only non-generic functions.
fdef	Optional, the generic function definition. Usually omitted in calls to `isGeneric`
getName	If `TRUE`, `isGeneric` returns the name of the generic. By default, it returns `TRUE`.
methods	The methods object containing the methods to be dumped. By default, the methods defined for this generic (optionally on the specified `where` location).
all	in `removeMethods`, logical indicating if all (default) or only the first method found should be removed.
searchForm	In `getGenerics`, if `TRUE`, the `package` slot of the returned result is in the form used by `search()`, otherwise as the simple package name (e.g, `"package:base"` vs `"base"`).

Summary of Functions

`isGeneric`: Is there a function named `f`, and if so, is it a generic?

The `getName` argument allows a function to find the name from a function definition. If it is `TRUE` then the name of the generic is returned, or `FALSE` if this is not a generic function definition.

The behavior of `isGeneric` and `getGeneric` for primitive functions is slightly different. These functions don't exist as formal function objects (for efficiency and historical reasons),

regardless of whether methods have been defined for them. A call to `isGeneric` tells you whether methods have been defined for this primitive function, anywhere in the current search list, or in the specified position `where`. In contrast, a call to `getGeneric` will return what the generic for that function would be, even if no methods have been currently defined for it.

removeGeneric, removeMethods: Remove all the methods for the generic function of this name. In addition, `removeGeneric` removes the function itself; `removeMethods` restores the non-generic function which was the default method. If there was no default method, `removeMethods` leaves a generic function with no methods.

standardGeneric: Dispatches a method from the current function call for the generic function `f`. It is an error to call `standardGeneric` anywhere except in the body of the corresponding generic function.

Note that `standardGeneric` is a primitive function in the **base** package for efficiency reasons, but rather documented here where it belongs naturally.

dumpMethod: Dump the method for this generic function and signature.

findFunction: return a list of either the positions on the search list, or the current top-level environment, on which a function object for `name` exists. The returned value is *always* a list, use the first element to access the first visible version of the function. See the example.

NOTE: Use this rather than `find` with `mode="function"`, which is not as meaningful, and has a few subtle bugs from its use of regular expressions. Also, `findFunction` works correctly in the code for a package when attaching the package via a call to `library`.

dumpMethods: Dump all the methods for this generic.

signature: Returns a named list of classes to be matched to arguments of a generic function.

getGenerics: Returns the names of the generic functions that have methods defined on `where`; this argument can be an environment or an index into the search list. By default, the whole search list is used.

The methods definitions are stored with package qualifiers; for example, methods for function `"initialize"` might refer to two different functions of that name, on different packages. The package names corresponding to the method list object are contained in the slot `package` of the returned object. The form of the returned name can be plain (e.g., `"base"`), or in the form used in the search list (`"package:base"`) according to the value of `searchForm`

callGeneric: In the body of a method, this function will make a call to the current generic function. If no arguments are passed to `callGeneric`, the arguments to the current call are passed down; otherwise, the arguments are interpreted as in a call to the generic function.

Details

setGeneric: If there is already a non-generic function of this name, it will be used to define the generic unless `def` is supplied, and the current function will become the default method for the generic.

If `def` is supplied, this defines the generic function, and no default method will exist (often a good feature, if the function should only be available for a meaningful subset of all objects).

Arguments `group` and `valueClass` are retained for consistency with S-Plus, but are currently not used.

isGeneric: If the `fdef` argument is supplied, take this as the definition of the generic, and test whether it is really a generic, with `f` as the name of the generic. (This argument is not available in S-Plus.)

removeGeneric: If `where` supplied, just remove the version on this element of the search list; otherwise, removes the first version encountered.

standardGeneric: Generic functions should usually have a call to `standardGeneric` as their entire body. They can, however, do any other computations as well.

The usual `setGeneric` (directly or through calling `setMethod`) creates a function with a call to `standardGeneric`.

dumpMethod: The resulting source file will recreate the method.

findFunction: If `generic` is `FALSE`, ignore generic functions.

dumpMethods: If `signature` is supplied only the methods matching this initial signature are dumped. (This feature is not found in S-Plus: don't use it if you want compatibility.)

signature: The advantage of using `signature` is to provide a check on which arguments you meant, as well as clearer documentation in your method specification. In addition, `signature` checks that each of the elements is a single character string.

removeMethods: Returns `TRUE` if `f` was a generic function, `FALSE` (silently) otherwise.

If there is a default method, the function will be re-assigned as a simple function with this definition. Otherwise, the generic function remains but with no methods (so any call to it will generate an error). In either case, a following call to `setMethod` will consistently re-establish the same generic function as before.

References

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the methods package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details see `?Methods` and the links from that documentation.

See Also

`getMethod` (also for `selectMethod`), `setGeneric`, `setClass`, `showMethods`

Examples

```
## get the function "myFun" -- throw an error if 0 or > 1 versions visible:
findFuncStrict <- function(fName) {
  allF <- findFunction(fName)
  if(length(allF) == 0)
    stop("No versions of ",fName," visible")
  else if(length(allF) > 1)
    stop(fName," is ambiguous: ", length(allF), " versions")
  else
    get(fName, allF[[1]])
}

try(findFuncStrict("myFun"))# Error: no version
lm <- function(x) x+1
try(findFuncStrict("lm"))#    Error: 2 versions
findFuncStrict("findFuncStrict")# just 1 version
rm(lm)
```

`getClass` *Get Class Definition*

Description

Get the definition of a class.

Usage

```
getClass(Class, .Force = FALSE, where)
getClassDef(Class, where, package)
```

Arguments

`Class`	the character-string name of the class.
`.Force`	if `TRUE`, return `NULL` if the class is undefined; otherwise, an undefined class results in an error.
`where`	environment from which to begin the search for the definition; by default, start at the top-level (global) environment and proceed through the search list.
`package`	the name of the package asserted to hold the definition. Supplied instead of `where`, with the distinction that the package need not be currently attached.

Details

A call to `getClass` returns the complete definition of the class supplied as a string, including all slots, etc. in classes that this class extends. A call to `getClassDef` returns the definition of the class from the environment `where`, unadorned. It's usually `getClass` you want.

If you really want to know whether a class is formally defined, call `isClass`.

Value

The object defining the class. This is an object of class `"classRepEnvironment"`. However, *do not* deal with the contents of the object directly unless you are very sure you know what you're doing. Even then, it is nearly always better practice to use functions such as `setClass` and `setIs`. Messing up a class object will cause great confusion.

References

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the methods package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details see `?Methods` and the links from that documentation.

See Also

Classes, `setClass`, `isClass`.

Examples

```
getClass("numeric") ## a built in class

cld <- getClass("thisIsAnUndefinedClass", .Force = TRUE)
cld ## a NULL prototype
## If you are really curious:
str(cld)
## Whereas these generate errors:
try(getClass("thisIsAnUndefinedClass"))
try(getClassDef("thisIsAnUndefinedClass"))
```

---

getMethod *Get or Test for the Definition of a Method*

Description

The functions `getMethod` and `selectMethod` get the definition of a particular method; the functions `existsMethod` and `hasMethod` test for the existence of a method. In both cases the first function only gets direct definitions and the second uses inheritance. The function `findMethod` returns the package(s) in the search list (or in the packages specified by the `where` argument) that contain a method for this function and signature.

The other functions are support functions: see the details below.

Usage

```
getMethod(f, signature=character(), where, optional=FALSE, mlist, fdef)

findMethod(f, signature, where)

getMethods(f, where)

existsMethod(f, signature = character(), where)

hasMethod(f, signature=character(), where)

selectMethod(f, signature, optional = FALSE, useInherited = TRUE,
             mlist = (if (is.null(fdef)) NULL else
                      getMethodsForDispatch(f, fdef)),
             fdef = getGeneric(f, !optional))

MethodsListSelect(f, env, mlist, fEnv, finalDefault, evalArgs,
                  useInherited, fdef, resetAllowed)
```

Arguments

`f`	The character-string name of the generic function.
`signature`	the signature of classes to match to the arguments of `f`. See the details below. For `selectMethod`, the signature can optionally be an environment with classes assigned to the names of the corresponding arguments. Note: the names

correspond to the names of the classes, *not* to the objects supplied in a call to the generic function. (You are not likely to find this approach convenient, but it is used internally and is marginally more efficient.)

`where` The position or environment in which to look for the method(s): by default, anywhere in the current search list.

`optional` If the selection does not produce a unique result, an error is generated, unless this argument is `TRUE`. In that case, the value returned is either a `MethodsList` object, if more than one method matches this signature, or `NULL` if no method matches.

`mlist` Optionally, the list of methods in which to search. By default, the function finds the methods for the corresponding generic function. To restrict the search to a particular package or environment, e.g., supply this argument as `getMethodsMetaData(f,where)`. For `selectMethod`, see the discussion of argument `fdef`.

`fdef` Optionally, the generic function from which the method is to be retrieved. (Unlikely to be used, except internally in the package source.)

`env` The environment in which argument evaluations are done in `MethodsListSelect`. Currently must be supplied, but should usually be `sys.frame(sys.parent())` when calling the function explicitly for debugging purposes.

`fEnv, finalDefault, evalArgs, useInherited, resetAllowed`
Internal-use arguments for the function's environment, the method to use as the overall default, whether to evaluate arguments, which arguments should use inheritance, and whether the cached methods are allowed to be reset.

Details

The `signature` argument specifies classes, in an extended sense, corresponding to formal arguments of the generic function. As supplied, the argument may be a vector of strings identifying classes, and may be named or not. Names, if supplied, match the names of those formal arguments included in the signature of the generic. That signature is normally all the arguments except ... . However, generic functions can be specified with only a subset of the arguments permitted, or with the signature taking the arguments in a different order.

It's a good idea to name the arguments in the signature to avoid confusion, if you're dealing with a generic that does something special with its signature. In any case, the elements of the signature are matched to the formal signature by the same rules used in matching arguments in function calls (see `match.call`).

The strings in the signature may be class names, `"missing"` or `"ANY"`. See Methods for the meaning of these in method selection. Arguments not supplied in the signature implicitly correspond to class `"ANY"`; in particular, giving an empty signature means to look for the default method.

A call to `getMethod` returns the method for a particular function and signature. As with other `get` functions, argument `where` controls where the function looks (by default anywhere in the search list) and argument `optional` controls whether the function returns `NULL` or generates an error if the method is not found. The search for the method makes no use of inheritance.

The function `selectMethod` also looks for a method given the function and signature, but makes full use of the method dispatch mechanism; i.e., inherited methods and group generics are taken into account just as they would be in dispatching a method for the corresponding signature, with the one exception that conditional inheritance is not used. Like `getMethod`, `selectMethod` returns `NULL` or generates an error if the method is not found, depending on the argument `optional`.

The functions existsMethod and hasMethod return TRUE or FALSE according to whether a method is found, the first corresponding to getMethod (no inheritance) and the second to selectMethod.

The function getMethods returns all the methods for a particular generic (in the form of a generic function with the methods information in its environment). The function is called from the evaluator to merge method information, and is not intended to be called directly. Note that it gets *all* the visible methods for the specified functions. If you want only the methods defined explicitly in a particular environment, use the function getMethodsMetaData instead.

The function MethodsListSelect performs a full search (including all inheritance and group generic information: see the Methods documentation page for details on how this works). The call returns a possibly revised methods list object, incorporating any method found as part of the allMethods slot.

Normally you won't call MethodsListSelect directly, but it is possible to use it for debugging purposes (only for distinctly advanced users!).

Note that the statement that MethodsListSelect corresponds to the selection done by the evaluator is a fact, not an assertion, in the sense that the evaluator code constructs and executes a call to MethodsListSelect when it does not already have a cached method for this generic function and signature. (The value returned is stored by the evaluator so that the search is not required next time.)

Value

The call to selectMethod or getMethod returns a MethodDefinition-class object, the selected method, if a unique selection exists. (This class extends function, so you can use the result directly as a function if that is what you want.) Otherwise an error is thrown if optional is FALSE. If optional is TRUE, the value returned is NULL if no method matched, or a MethodsList object if multiple methods matched.

The call to getMethods returns the MethodsList object containing all the methods requested. If there are none, NULL is returned: getMethods does not generate an error in this case.

References

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the **methods** package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details see ?Methods and the links from that documentation.

See Also

GenericFunctions

Examples

```
setGeneric("testFun", function(x)standardGeneric("testFun"))
setMethod("testFun", "numeric", function(x)x+1)
hasMethod("testFun", "numeric")
## Not run: [1] TRUE
hasMethod("testFun", "integer") #inherited
```

```
## Not run: [1] TRUE
existsMethod("testFun", "integer")
## Not run: [1] FALSE
hasMethod("testFun") # default method
## Not run: [1] FALSE
hasMethod("testFun", "ANY")
## Not run: [1] FALSE
```

`getPackageName` *The Name associated with a Given Package*

Description

The functions below produce the package associated with a particular environment or position on the search list, or of the package containing a particular function. They are primarily used to support computations that need to differentiate objects on multiple packages.

Usage

```
getPackageName(where)

packageSlot(object)
packageSlot(object) <- value
```

Arguments

`where`	the environment or position on the search list associated with the desired package.
`object`	object providing a character string name, plus the package in which this object is to be found.
`value`	the name of the package.

Details

Package names are normally installed during loading of the package, by the INSTALL script or by the `library` function. (Currently, the name is stored as the object `.packageName` but don't trust this for the future.)

Value

`packageName` return the character-string name of the package (without the extraneous `"package:"` found in the search list).

`packageSlot` returns or sets the package name slot (currently an attribute, not a formal slot, but this will likely change).

See Also

`search`

Examples

```
## both the following usually return "base"
getPackageName(length(search()))
```

`hasArg` *Look for an Argument in the Call*

Description

Returns `TRUE` if `name` corresponds to an argument in the call, either a formal argument to the function, or a component of `...`, and `FALSE` otherwise.

Usage

```
hasArg(name)
```

Arguments

`name`	The unquoted name of a potential argument.

Details

The expression `hasArg(x)`, for example, is similar to `!missing(x)`, with two exceptions. First, `hasArg` will look for an argument named `x` in the call if `x` is not a formal argument to the calling function, but `...` is. Second, `hasArg` never generates an error if given a name as an argument, whereas `missing(x)` generates an error if `x` is not a formal argument.

Value

Always `TRUE` or `FALSE` as described above.

See Also

`missing`

Examples

```
ftest <- function(x1, ...) c(hasArg(x1), hasArg(y2))

ftest(1) ## c(TRUE, FALSE)
ftest(1, 2)  ## c(TRUE, FALSE)
ftest(y2=2)   ## c(FALSE, TRUE)
ftest(y=2)    ## c(FALSE, FALSE) (no partial matching)
ftest(y2 = 2, x=1)  ## c(TRUE, TRUE) partial match x1
```

`initialize-methods` *Methods to Initialize New Objects from a Class*

Description

The arguments to function `new` to create an object from a particular class can be interpreted specially for that class, by the definition of a method for function `initialize` for the class. This documentation describes some existing methods, and also outlines how to write new ones.

Methods

.Object = "ANY" The default method for `initialize` takes either named or unnamed arguments. Argument names must be the names of slots in this class definition, and the corresponding arguments must be valid objects for the slot (that is, have the same class as specified for the slot, or some superclass of that class). If the object comes from a superclass, it is not coerced strictly, so normally it will retain its current class (specifically, `as(object, Class, strict = FALSE)`).

Unnamed arguments must be objects of this class, of one of its superclasses, or one of its subclasses (from the class, from a class this class extends, or from a class that extends this class). If the object is from a superclass, this normally defines some of the slots in the object. If the object is from a subclass, the new object is that argument, coerced to the current class.

Unnamed arguments are processed first, in the order they appear. Then named arguments are processed. Therefore, explicit values for slots always override any values inferred from superclass or subclass arguments.

.Object = "traceable" Objects of a class that extends `traceable` are used to implement debug tracing (see traceable-class and `trace`).

The `initialize` method for these classes takes special arguments `def, tracer, exit, at, print`. The first of these is the object to use as the original definition (e.g., a function). The others correspond to the arguments to `trace`.

.Object = "environment" The `initialize` method for environments takes a named list of objects to be used to initialize the environment.

.Object = "signature" This is a method for internal use only. It takes an optional `functionDef` argument to provide a generic function with a `signature` slot to define the argument names. See Methods for details.

Writing Initialization Methods

Initialization methods provide a general mechanism corresponding to generator functions in other languages.

The arguments to `initialize` are `.Object` and `...`. Nearly always, `initialize` is called from `new`, not directly. The `.Object` argument is then the prototype object from the class.

Two techniques are often appropriate for `initialize` methods: special argument names and `callNextMethod`.

You may want argument names that are more natural to your users than the (default) slot names. These will be the formal arguments to your method definition, in addition to `.Object` (always) and ... (optionally). For example, the method for class `"traceable"` documented above would be created by a call to `setMethod` of the form:

```
setMethod("initialize", "traceable",
  function(.Object, def, tracer, exit, at, print) ...
)
```

In this example, no other arguments are meaningful, and the resulting method will throw an error if other names are supplied.

When your new class extends another class, you may want to call the initialize method for this superclass (either a special method or the default). For example, suppose you want to define a method for your class, with special argument `x`, but you also want users to be able to set slots specifically. If you want `x` to override the slot information, the beginning of your method definition might look something like this:

```
function(.Object, x, ...) {
  Object <- callNextMethod(.Object, ...)
  if(!missing(x)) { # do something with x
```

You could also choose to have the inherited method override, by first interpreting `x`, and then calling the next method.

`is` *Is an Object from a Class*

Description

`is`: With two arguments, tests whether `object` can be treated as from `class2`.

With one argument, returns all the super-classes of this object's class.

`extends`: Does the first class extend the second class? Returns `maybe` if the extension includes a test.

`setIs`: Defines `class1` to be an extension of `class2`.

Usage

```
is(object, class2)

extends(class1, class2, maybe=TRUE, fullInfo = FALSE)

setIs(class1, class2, test=NULL, coerce=NULL, replace=NULL,
      by = character(), where = topenv(parent.frame()), classDef =,
      extensionObject = NULL, doComplete = TRUE)
```

Arguments

`object`	any R object.
`class1, class2`	the names of the classes between which `is` relations are to be defined.
`maybe, fullInfo`	In a call to `extends`, `maybe` is a flag to include/exclude conditional relations, and `fullInfo` is a flag, which if `TRUE` causes object(s) of class `classExtension` to be returned, rather than just the names of the classes or a logical value. See the details below.

`extensionObject`
: alternative to the `test, coerce, replace, by` arguments; an object from class `SClassExtension` describing the relation. (Used in internal calls.)

`doComplete`
: when `TRUE`, the class definitions will be augmented with indirect relations as well. (Used in internal calls.)

`test, coerce, replace`
: In a call to `setIs`, functions optionally supplied to test whether the relation is defined, to coerce the object to `class2`, and to alter the object so that `is(object, class2)` is identical to `value`.

`by`
: In a call to `setIs`, the name of an intermediary class. Coercion will proceed by first coercing to this class and from there to the target class. (The intermediate coercions have to be valid.)

`where`
: In a call to `setIs`, where to store the metadata defining the relationship. Default is the global environment.

`classDef`
: Optional class definition for `class` , required internally when `setIs` is called during the initial definition of the class by a call to `setClass`. *Don't* use this argument, unless you really know why you're doing so.

Details

`extends`: Given two class names, `extends` by default says whether the first class extends the second; that is, it does for class names what `is` does for an object and a class. Given one class name, it returns all the classes that class extends (the "superclasses" of that class), including the class itself. If the flag `fullInfo` is `TRUE`, the result is a list, each element of which is an object describing the relationship; otherwise, and by default, the value returned is only the names of the classes.

`setIs`: This function establishes an inheritance relation between two classes, by some means other than having one class contain the other. It should *not* be used for ordinary relationships: either include the second class in the `contains=` argument to `setClass` if the class is contained in the usual way, or consider `setClassUnion` to define a virtual class that is extended by several ordinary classes. A call to `setIs` makes sense, for example, if one class ought to be automatically convertible into a second class, but they have different representations, so that the conversion must be done by an explicit computation, not just be inheriting slots, for example. In this case, you will typically need to provide both a `coerce=` and `replace=` argument to `setIs`.

The `coerce`, `replace`, and `by` arguments behave as described for the `setAs` function. It's unlikely you would use the `by` argument directly, but it is used in defining cached information about classes. The value returned (invisibly) by `setIs` is the extension information, as a list.

The `coerce` argument is a function that turns a `class1` object into a `class2` object. The `replace` argument is a function of two arguments that modifies a `class1` object (the first argument) to replace the part of it that corresponds to `class2` (supplied as `value`, the second argument). It then returns the modified object as the value of the call. In other words, it acts as a replacement method to implement the expression `as(object, class2) <- value`.

The easiest way to think of the `coerce` and `replace` functions is by thinking of the case that `class1` contains `class2` in the usual sense, by including the slots of the second class. (To repeat, in this situation you would not call `setIs`, but the analogy shows what happens when you do.)

The `coerce` function in this case would just make a `class2` object by extracting the corresponding slots from the `class1` object. The `replace` function would replace in the

`class1` object the slots corresponding to `class2`, and return the modified object as its value.

The relationship can also be conditional, if a function is supplied as the `test` argument. This should be a function of one argument that returns `TRUE` or `FALSE` according to whether the object supplied satisfies the relation `is(object, class2)`. If you worry about such things, conditional relations between classes are slightly deprecated because they cannot be implemented as efficiently as ordinary relations and because they sometimes can lead to confusion (in thinking about what methods are dispatched for a particular function, for example). But they can correspond to useful distinctions, such as when two classes have the same representation, but only one of them obeys certain additional constraints.

Because only global environment information is saved, it rarely makes sense to give a value other than the default for argument `where`. One exception is `where=0`, which modifies the cached (i.e., session-scope) information about the class. Class completion computations use this version, but don't use it yourself unless you are quite sure you know what you're doing.

References

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the **methods** package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details see `?Methods` and the links from that documentation.

Examples

```
## a class definition (see setClass for the example)
setClass("trackCurve",
         representation("track", smooth = "numeric"))
## A class similar to "trackCurve", but with different structure
## allowing matrices for the "y" and "smooth" slots
setClass("trackMultiCurve",
         representation(x="numeric", y="matrix", smooth="matrix"),
         prototype = structure(list(), x=numeric(), y=matrix(0,0,0),

                               smooth= matrix(0,0,0)))
## Automatically convert an object from class "trackCurve" into
## "trackMultiCurve", by making the y, smooth slots into 1-column matrices
setIs("trackCurve",
      "trackMultiCurve",
      coerce = function(obj) {
        new("trackMultiCurve",
            x = obj@x,
            y = as.matrix(obj@y),
            curve = as.matrix(obj@smooth))
      },
      replace = function(obj, value) {
        obj@y <- as.matrix(value@y)
        obj@x <- value@x
        obj@smooth <- as.matrix(value@smooth)
        obj})

## Automatically convert the other way, but ONLY
```

```
## if the y data is one variable.
setIs("trackMultiCurve",
      "trackCurve",
      test = function(obj) {ncol(obj@y) == 1},
      coerce = function(obj) {
        new("trackCurve",
            x = slot(obj, "x"),
            y = as.numeric(obj@y),
            smooth = as.numeric(obj@smooth))
      },
      replace = function(obj, value) {
        obj@y <- matrix(value@y, ncol=1)
        obj@x <- value@x
        obj@smooth <- value@smooth
        obj})
```

`isSealedMethod` *Check for a Sealed Method or Class*

Description

These functions check for either a method or a class that has been "sealed" when it was defined, and which therefore cannot be re-defined.

Usage

```
isSealedMethod(f, signature, fdef, where)
isSealedClass(Class, where)
```

Arguments

`f`	The quoted name of the generic function.
`signature`	The class names in the method's signature, as they would be supplied to `setMethod`.
`fdef`	Optional, and usually omitted: the generic function definition for `f`.
`Class`	The quoted name of the class.
`where`	where to search for the method or class definition. By default, searches from the top environment of the call to `isSealedMethod` or `isSealedClass`, typically the global environment or the namespace of a package containing a call to one of the functions.

Details

In the R implementation of classes and methods, it is possible to seal the definition of either a class or a method. The basic classes (numeric and other types of vectors, matrix and array data) are sealed. So also are the methods for the primitive functions on those data types. The effect is that programmers cannot re-define the meaning of these basic data types and computations. More precisely, for primitive functions that depend on only one data argument, methods cannot be specified for basic classes. For functions (such as the arithmetic operators) that depend on two arguments, methods can be specified if *one* of those arguments is a basic class, but not if both are.

Programmers can seal other class and method definitions by using the `sealed` argument to `setClass` or `setMethod`.

Value

The functions return `FALSE` if the method or class is not sealed (including the case that it is not defined); `TRUE` if it is.

References

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the **methods** package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details see `?Methods` and the links from that documentation.

Examples

```
## these are both TRUE
isSealedMethod("+", c("numeric", "character"))
isSealedClass("matrix")

setClass("track",
           representation(x="numeric", y="numeric"))
## but this is FALSE
isSealedClass("track")
## and so is this
isSealedClass("A Name for an undefined Class")
## and so are these, because only one of the two arguments is basic
isSealedMethod("+", c("track", "numeric"))
isSealedMethod("+", c("numeric", "track"))
```

`language-class` *Classes to Represent Unevaluated Language Objects*

Description

The virtual class `"language"` and the specific classes that extend it represent unevaluated objects, as produced for example by the parser or by functions such as `quote`.

Usage

```
### each of these classes corresponds to an unevaluated object
### in the S language.  The class name can appear in method signatures,
### and in a few other contexts (such as some calls to as()).

"("
"<-"
"call"
"for"
"if"
```

```
"repeat"
"while"
"name"
"{"

### Each of the classes above extends the virtual class

"language"
```

Objects from the Class

`"language"` is a virtual class; no objects may be created from it.

Objects from the other classes can be generated by a call to `new(Class, ...)`, where `Class` is the quoted class name, and the ... arguments are either empty or a *single* object that is from this class (or an extension).

Methods

coerce `signature(from = "ANY", to = "call")`. A method exists for `as(object, "call")`, calling `as.call()`.

LinearMethodsList-class

Class "LinearMethodsList"

Description

A version of methods lists that has been "linearized" for producing summary information. The actual objects from class `"MethodsList"` used for method dispatch are defined recursively over the arguments involved.

Objects from the Class

The function `linearizeMlist` converts an ordinary methods list object into the linearized form.

Slots

`methods:` Object of class `"list"`, the method definitions.

`arguments:` Object of class `"list"`, the corresponding formal arguments.

`classes:` Object of class `"list"`, the corresponding classes in the signatures.

`fromClasses:` Object of class `"list"`

Future Note

The current version of `linearizeMlist` does not take advantage of the `MethodDefinition` class, and therefore does more work for less effect than it could. In particular, we may move to redefine both the function and the class to take advantage of the stored signatures. Don't write code depending precisely on the present form, although all the current information will be obtainable in the future.

See Also

Function `linearizeMlist` for the computation, and `MethodsList-class` for the original, recursive form.

`makeClassRepresentation`
Create a Class Definition

Description

Constructs a `classRepresentation-class` object to describe a particular class. Mostly a utility function, but you can call it to create a class definition without assigning it, as `setClass` would do.

Usage

```
makeClassRepresentation(name, slots=list(), superClasses=character(),
                        prototype=NULL, package, validity, access,
                        version, sealed, virtual=NA, where)
```

Arguments

`name`	character string name for the class
`slots`	named list of slot classes as would be supplied to `setClass`, but *without* the unnamed arguments for superClasses if any.
`superClasses`	what classes does this class extend
`prototype`	an object providing the default data for the class, e.g, the result of a call to `prototype`.
`package`	The character string name for the package in which the class will be stored; see `getPackageName`.
`validity`	Optional validity method. See `validObject`, and the discussion of validity methods in the reference.
`access`	Access information. Not currently used.
`version`	Optional version key for version control. Currently generated, but not used.
`sealed`	Is the class sealed? See `setClass`.
`virtual`	Is this known to be a virtual class?
`where`	The environment from which to look for class definitions needed (e.g., for slots or superclasses). See the discussion of this argument under GenericFunctions.

References

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the **methods** package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details see `?Methods` and the links from that documentation.

See Also

setClass

MethodDefinition-class
Classes to Represent Method Definitions

Description

These classes extend the basic class "function" when functions are to be stored and used as method definitions.

Details

Method definition objects are functions with additional information defining how the function is being used as a method. The target slot is the class signature for which the method will be dispatched, and the defined slot the signature for which the method was orignally specified (that is, the one that appeared in some call to setMethod).

Objects from the Class

The action of setting a method by a call to setMethod creates an object of this class. It's unwise to create them directly.

The class "SealedMethodDefinition" is created by a call to setMethod with argument sealed = TRUE. It has the same representation as "MethodDefinition".

Slots

.Data: Object of class "function"; the data part of the definition.

target: Object of class "signature"; the signature for which the method was wanted.

defined: Object of class "signature"; the signature for which a method was found. If the method was inherited, this will not be identical to target.

generic: Object of class "character"; the function for which the method was created.

Extends

Class "function", from data part.
Class "PossibleMethod", directly.
Class "OptionalMethods", by class "function".

See Also

class MethodsList-class for the objects defining sets of methods associated with a particular generic function. The individual method definitions stored in these objects are from class MethodDefinition, or an extension. MethodWithNext-class for an extension used by callNextMethod.

`Methods` *General Information on Methods*

Description

This documentation section covers some general topics on how methods work and how the **methods** package interacts with the rest of R. The information is usually not needed to get started with methods and classes, but may be helpful for moderately ambitious projects, or when something doesn't work as expected.

The section **How Methods Work** describes the underlying mechanism; **Dispatch and Method Selection** provides more details on how class definitions determine which methods are used. For additional information specifically about class definitions, see `?Classes`.

How Methods Work

A generic function is a function that has associated with it a collection of other functions (the methods), all of which agree in formal arguments with the generic.

Each R package will include methods metadata objects corresponding to each generic function for which methods have been defined in that package. When the package is loaded into an R session, the methods for each generic function are *cached*, that is, stored in the environment of the generic function along with the methods from previously loaded packages. This merged table of methods is used to dispatch or select methods from the generic, using class inheritance and possibly group generic functions to find an applicable method. See the **Dispatch** section below. The caching computations ensure that only one version of each generic function is visible globally; although different attached packages may contain a copy of the generic function, these are in fact identical.

The methods for a generic are stored according to the corresponding `signature` for which the method was defined, in a call to `setMethod`. The signature associates one class name with each of a subset of the formal arguments to the generic function. Which formal arguments are available, and the order in which they appear, are determined by the `"signature"` slot of the generic function. By default, the signature of the generic consists of all the formal arguments except ..., in the order they appear in the function definition.

Trailing arguments in the signature will be *inactive* if no method has yet been specified that included those arguments. Inactive arguments are not needed or used in labeling the cached methods. (The distinction does not change which methods are dispatched, but ignoring inactive arguments does improve the efficiency of dispatch. Thus, defining the generic signature to contain the most useful arguments first can help efficiency somewhat.)

All arguments in the signature of the generic function will be evaluated when the function is called, rather than using the traditional lazy evaluation rules of S. Therefore, it's important to *exclude* from the signature any arguments that need to be dealt with symbolically (such as the first argument to function `substitute`). Note that only actual arguments are evaluated, not default expressions. A missing argument enters into the method selection as class `"missing"` and non-missing arguments according to their actual class.

As of version 2.4.0 of R, the cached methods are stored in an environment object. The names used for assignment are a concatenation of the class names for the arguments in the active signature.

Dispatch and Method Selection

When a call to a generic function is evaluated, a method is selected corresponding to the classes of the actual arguments in the signature. First, the cached methods table is searched for a *direct* match;

that is, a method stored under the direct class names. The direct class is the value of `class(x)` for each non-missing argument, and class `"missing"` for each missing argument. If no method is found directly for the actual arguments in a call to a generic function, an attempt is made to match the available methods to the arguments by using *inheritance*.

Each class definition potentially includes the names of one or more classes that the new class contains. (These are sometimes called the *superclasses* of the new class.) The S language has an additional, explicit mechanism for defining superclasses, the `setIs` mechanism. Also, a call to `setClassUnion` makes the union class a superclass of each of the members of the union. All three mechanisms are treated equivalently for purposes of inheritance: they define the *direct* superclasses of a particular class.

The direct superclasses themselves may contain other classes. Putting all this information together produces the full list of superclasses for this class. The superclass list is included in the definition of the class that is cached during the R session. Each element of the list describes the nature of the relationship (see SClassExtension-class for details). Included in the element is a `distance` slot giving a numeric distance between the two classes. The distance currently is the path length for the relationship: `1` for direct superclasses (regardless of which mechanism defined them), then `2` for the direct superclasses of those classes, and so on. In addition, any class implicitly has class `"ANY"` as a superclass. The distance to `"ANY"` is treated as larger than the distance to any actual class. The special class `"missing"` corresponding to missing arguments has only `"ANY"` as a superclass, while `"ANY"` has no superclasses.

The information about superclasses is summarized when a class definition is printed.

When a method is to be selected by inheritance, a search is made in the table for all methods directly corresponding to a combination of either the direct class or one of its superclasses, for each argument in the active signature. For an example, suppose there is only one argument in the signature and that the class of the corresponding object was `"dgeMatrix"` (from the `Matrix` package on CRAN). This class has two direct superclasses and through these 4 additional superclasses. Method selection finds all the methods in the table of directly specified methods labeled by one of these classes, or by `"ANY"`.

When there are multiple arguments in the signature, each argument will generate a similar list of inherited classes. The possible matches are now all the combinations of classes from each argument (think of the function `outer` generating an array of all possible combinations). The search now finds all the methods matching any of this combination of classes. The computation of distances also has to combine distances for the individual arguments. There are many ways to combine the distances; the current implementation simply adds them. The result of the search is then a list of zero, one or more methods, and a parallel vector of distances between the target signature and the available methods.

If the list has more than one matching method, only those corresponding to the minimum distance are considered. There may still be multiple best methods. The dispatch software considers this an ambiguous case and warns the user (only on the first call for this selection). The method occurring first in the list of superclasses is selected. By the mechanism of producing the extension information, this orders the direct superclasses by the order they appeared in the original call to `setClass`, followed by classes specified in `setIs` calls, in the order those calls were evaluated, followed by classes specified in unions. Then the superclasses of those classes are appended (note that only the ordering of classes within a particular generation of superclasses counts, because only these will have the same distance). For further discussion of method selection, see the document `http://developer.r-project.org/howMethodsWork.pdf`.

All this detail about selection is less important than the realization that having ambiguous method selection usually means that you need to be more specific about intentions. It is likely that some consideration other than the ordering of superclasses in the class definition is more important in determining which method *should* be selected, and the preference may well be different for differ-

ent generic functions. Where ambiguities arise, the best approach is usually to provide a specific method for the subclass.

When the inherited method has been selected, the selection is cached in the generic function so that future calls with the same class will not require repeating the search. Cached non-direct selections are not themselves used in inheritance searches, since that could result in invalid selections.

Besides being initiated through calls to the generic function, method selection can be done explicitly by calling the function `selectMethod`.

References

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the **methods** package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference.

See Also

`setGeneric`, `setClass` and the document `http://developer.r-project.org/howMethodsWork.pdf`.

MethodsList-class *Class MethodsList, Representation of Methods for a Generic Function*

Description

Objects from this class are generated and revised by the definition of methods for a generic function.

Details

Suppose a function `f` has formal arguments `x` and `y`. The methods list object for that function has the object `as.name("x")` as its `argument` slot. An element of the methods named `"track"` is selected if the actual argument corresponding to `x` is an object of class `"track"`. If there is such an element, it can generally be either a function or another methods list object.

In the first case, the function defines the method to use for any call in which `x` is of class `"track"`. In the second case, the new methods list object defines the available methods depending on the remaining formal arguments, in this example, `y`.

Each method corresponds conceptually to a *signature*; that is a named list of classes, with names corresponding to some or all of the formal arguments. In the previous example, if selecting class `"track"` for `x`, finding that the selection was another methods list and then selecting class `"numeric"` for `y` would produce a method associated with the signature `x = "track", y = "numeric"`.

You can see the methods arranged by signature by calling the function `showMethods`, A methods list can be converted into an ordinary list with the methods arranged this way (in two different forms) by calling the functions `listFromMlist` and `linearizeMlist`.

Slots

argument: Object of class "name". The name of the argument being used for dispatch at this level.

methods: A named list of the methods (and method lists) defined *explicitly* for this argument. The names are the names of classes, and the corresponding element defines the method or methods to be used if the corresponding argument has that class. See the details below.

allMethods: A named list, contains all the directly defined methods from the methods slot, plus any inherited methods. Ignored when methods tables are used for dispatch (see Methods

Extends

Class "OptionalMethods", directly.

MethodWithNext-class

Class MethodWithNext

Description

Class of method definitions set up for callNextMethod

Objects from the Class

Objects from this class are generated as a side-effect of calls to callNextMethod.

Slots

.Data: Object of class "function"; the actual function definition.

nextMethod: Object of class "PossibleMethod" the method to use in response to a callNextMethod() call.

excluded: Object of class "list"; one or more signatures excluded in finding the next method.

target: Object of class "signature", from class "MethodDefinition"

defined: Object of class "signature", from class "MethodDefinition"

generic: Object of class "character"; the function for which the method was created.

Extends

Class "MethodDefinition", directly.
Class "function", from data part.
Class "PossibleMethod", by class "MethodDefinition".
Class "OptionalMethods", by class "MethodDefinition".

Methods

findNextMethod signature(method = "MethodWithNext"): used internally by method dispatch.

loadMethod signature(method = "MethodWithNext"): used internally by method dispatch.

show signature(object = "MethodWithNext")

See Also

callNextMethod, and MethodDefinition-class.

`new` *Generate an Object from a Class*

Description

Given the name or the definition of a class, plus optionally data to be included in the object, `new` returns an object from that class.

Usage

```
new(Class, ...)

initialize(.Object, ...)
```

Arguments

`Class`	Either the name of a class (the usual case) or the object describing the class (e.g., the value returned by `getClass`).
`...`	Data to include in the new object. Named arguments correspond to slots in the class definition. Unnamed arguments must be objects from classes that this class extends.
`.Object`	An object: see the Details section.

Details

The function `new` begins by copying the prototype object from the class definition. Then information is inserted according to the `...` arguments, if any. As of version 2.4 of R, the type of the prototype object, and therefore of all objects returned by `new()`, is `"S4"` except for classes that extend one of the basic types, where the prototype has that basic type. User functions that depend on `typeof(object)` should be careful to handle `"S4"` as a possible type.

The interpretation of the `...` arguments can be specialized to particular classes, if an appropriate method has been defined for the generic function `"initialize"`. The `new` function calls `initialize` with the object generated from the prototype as the `.Object` argument to `initialize`.

By default, unnamed arguments in the `...` are interpreted as objects from a superclass, and named arguments are interpreted as objects to be assigned into the correspondingly named slots. Thus, explicit slots override inherited information for the same slot, regardless of the order in which the arguments appear.

The `initialize` methods do not have to have `...` as their second argument (see the examples), and generally it is better design *not* to have `...` as a formal argument, if only a fixed set of arguments make sense.

For examples of `initialize` methods, see `initialize-methods` for existing methods for classes `"traceable"` and `"environment"`, among others.

Note that the basic vector classes, `"numeric"`, etc. are implicitly defined, so one can use `new` for these classes.

References

The functions in this package emulate the facility for classes and methods described in *Programming with Data* (John M. Chambers, Springer, 1998). See this book for further details and examples.

See Also

Classes

Examples

```
## using the definition of class "track" from Classes



## a new object with two slots specified
t1 <- new("track", x = seq(along=ydata), y = ydata)

# a new object including an object from a superclass, plus a slot
t2 <- new("trackCurve", t1, smooth = ysmooth)

### define a method for initialize, to ensure that new objects have
### equal-length x and y slots.

setMethod("initialize",
          "track",
          function(.Object, x = numeric(0), y = numeric(0)) {
            if(nargs() > 1) {
              if(length(x) != length(y))
                stop("specified x and y of different lengths")
              .Object@x <- x
              .Object@y <- y
            }
            .Object
          })

### the next example will cause an error (x will be numeric(0)),
### because we didn't build in defaults for x,
### although we could with a more elaborate method for initialize

try(new("track", y = sort(rnorm(10))))

## a better way to implement the previous initialize method.
## Why?  By using callNextMethod to call the default initialize method
## we don't inhibit classes that extend "track" from using the general
## form of the new() function.  In the previous version, they could only
## use x and y as arguments to new, unless they wrote their own
## intialize method.

setMethod("initialize", "track", function(.Object, ...) {
    .Object <- callNextMethod()
    if(length(.Object@x) != length(.Object@y))
     stop("specified x and y of different lengths")
    .Object
  })
```

ObjectsWithPackage-class *A Vector of Object Names, with associated Package Names*

Description

This class of objects is used to represent ordinary character string object names, extended with a `package` slot naming the package associated with each object.

Objects from the Class

The function `getGenerics` returns an object of this class.

Slots

`.Data`: Object of class `"character"`: the object names.

`package`: Object of class `"character"` the package names.

Extends

Class `"character"`, from data part.
Class `"vector"`, by class `"character"`.

See Also

`Methods` for general background.

promptClass *Generate a Shell for Documentation of a Formal Class*

Description

Assembles all relevant slot and method information for a class, with minimal markup for Rd processing; no QC facilities at present.

Usage

```
promptClass(clName, filename = NULL, type = "class",
            keywords = "classes", where = topenv(parent.frame()))
```

Arguments

`clName`	a character string naming the class to be documented.
`filename`	usually, a connection or a character string giving the name of the file to which the documentation shell should be written. The default corresponds to a file whose name is the topic name for the class documentation, followed by `".Rd"`. Can also be `NA` (see below).
`type`	the documentation type to be declared in the output file.
`keywords`	the keywords to include in the shell of the documentation. The keyword `"classes"` should be one of them.
`where`	where to look for the definition of the class and of methods that use it.

Details

The class definition is found on the search list. Using that definition, information about classes extended and slots is determined.

In addition, the currently available generics with methods for this class are found (using `getGenerics`). Note that these methods need not be in the same environment as the class definition; in particular, this part of the output may depend on which packages are currently in the search list.

As with other prompt-style functions, unless `filename` is `NA`, the documentation shell is written to a file, and a message about this is given. The file will need editing to give information about the *meaning* of the class. The output of `promptClass` can only contain information from the metadata about the formal definition and how it is used.

If `filename` is `NA`, a list-style representation of the documentation shell is created and returned. Writing the shell to a file amounts to `cat(unlist(x), file = filename, sep = "\n")`, where `x` is the list-style representation.

Value

If `filename` is `NA`, a list-style representation of the documentation shell. Otherwise, the name of the file written to is returned invisibly.

Author(s)

VJ Carey ⟨stvjc@channing.harvard.edu⟩ and John Chambers

References

The functions in this package emulate the facility for classes and methods described in *Programming with Data* (John M. Chambers, Springer, 1998). See this book for further details and examples.

See Also

`prompt` for documentation of functions, `promptMethods` for documentation of method definitions.

For processing of the edited documentation, either use `R CMD Rdconv`, or include the edited file in the 'man' subdirectory of a package.

Examples

```
## Not run:
> promptClass("track")
A shell of class documentation has been written to the
file "track-class.Rd".
## End(Not run)
```

`promptMethods` *Generate a Shell for Documentation of Formal Methods*

Description

Generates a shell of documentation for the methods of a generic function.

Usage

```
promptMethods(f, filename = NULL, methods)
```

Arguments

`f`	a character string naming the generic function whose methods are to be documented.
`filename`	usually, a connection or a character string giving the name of the file to which the documentation shell should be written. The default corresponds to the coded topic name for these methods (currently, `f` followed by `"-methods.Rd"`). Can also be `FALSE` or `NA` (see below).
`methods`	Optional methods list object giving the methods to be documented. By default, the first methods object for this generic is used (for example, if the current global environment has some methods for `f`, these would be documented). If this argument is supplied, it is likely to be `getMethods(f, where)`, with `where` some package containing methods for `f`.

Details

If `filename` is `FALSE`, the text created is returned, presumably to be inserted some other documentation file, such as the documentation of the generic function itself (see `prompt`).

If `filename` is `NA`, a list-style representation of the documentation shell is created and returned. Writing the shell to a file amounts to `cat(unlist(x), file = filename, sep = "\n")`, where `x` is the list-style representation.

Otherwise, the documentation shell is written to the file specified by `filename`.

Value

If `filename` is `FALSE`, the text generated; if `filename` is `NA`, a list-style representation of the documentation shell. Otherwise, the name of the file written to is returned invisibly.

References

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the **methods** package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details see `?Methods` and the links from that documentation.

See Also

prompt and promptClass

---

representation *Construct a Representation or a Prototype for a Class Definition*

Description

In calls to setClass, these two functions construct, respectively, the representation and prototype arguments. They do various checks and handle special cases. You're encouraged to use them when defining classes that, for example, extend other classes as a data part or have multiple superclasses, or that combine extending a class and slots.

Usage

```
representation(...)
prototype(...)
```

Arguments

...	The call to representation takes arguments that are single character strings. Unnamed arguments are classes that a newly defined class extends; named arguments name the explicit slots in the new class, and specify what class each slot should have. In the call to prototype, if an unnamed argument is supplied, it unconditionally forms the basis for the prototype object. Remaining arguments are taken to correspond to slots of this object. It is an error to supply more than one unnamed argument.

Details

The representation function applies tests for the validity of the arguments. Each must specify the name of a class.

The classes named don't have to exist when representation is called, but if they do, then the function will check for any duplicate slot names introduced by each of the inherited classes.

The arguments to prototype are usually named initial values for slots, plus an optional first argument that gives the object itself. The unnamed argument is typically useful if there is a data part to the definition (see the examples below).

Value

The value pf representation is just the list of arguments, after these have been checked for validity.

The value of prototype is the object to be used as the prototype. Slots will have been set consistently with the arguments, but the construction does *not* use the class definition to test validity of the contents (it hardly can, since the prototype object is usually supplied to create the definition).

References

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the **methods** package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details see `?Methods` and the links from that documentation.

See Also

`setClass`

Examples

```
## representation for a new class with a directly define slot "smooth"
## which should be a "numeric" object, and extending class "track"
representation("track", smooth ="numeric")


setClass("Character",representation("character"))
setClass("TypedCharacter",representation("Character",type="character"),
          prototype(character(0),type="plain"))
ttt <- new("TypedCharacter", "foo", type = "character")


setClass("num1", representation(comment = "character"),
         contains = "numeric",
         prototype = prototype(pi, comment = "Start with pi"))
```

`S4groupGeneric` *S4 Group Generic Functions*

Description

Methods can be defined for groups of functions known as *group generic functions*. These exist in both S3 (see S3groupGeneric) and S4 flavours, with different groups.

Methods are defined for the group of functions as a whole. A method defined for an individual member of the group takes precedence over a method defined for the group as a whole.

When package **methods** is attached there are objects visible with the names of the group generics: these functions should never be called directly (a suitable error message will result if they are).

Usage

```
## S4 group generics:
Arith(e1, e2)
Compare(e1, e2)
Ops(e1, e2)
Logic(e1, e2)
Math(x)
Math2(x, digits)
Summary(x, ..., na.rm = FALSE)
Complex(z)
```

Arguments

`x, z, e1, e2`	objects.
`digits`	number of digits to be used in `round` or `signif`.
`...`	further arguments passed to or from methods.
`na.rm`	logical: should missing values be removed?

Details

When package **methods** is attached (which it is by default), formal (S4) methods can be defined for the group generic functions (which are R objects which should never be called directly – a suitable error message will result if they are). There are also S3 groups `Math`, `Ops`, `Summary` and `Complex`, see `?S3groupGeneric`, with no corresponding R objects.

The functions belonging to the various groups are as follows:

`Arith` `"+", "-", "*", "^", "%%", "%/%", "/"`

`Compare` `"==", ">", "<", "!=", "<=", ">="`

`Logic` `"&", "|"`, but **not** `"!"` since that has only one argument. Note that this is contrary to Chambers(1998), on purpose.

`Ops` `"Arith", "Compare", "Logic"`

`Math` `"log", "sqrt", "log10", "cumprod", "abs", "acos", "acosh", "asin", "asinh", "atan", "atanh", "ceiling", "cos", "cosh", "cumsum", "exp", "floor", "gamma", "lgamma", "sin", "sinh", "tan", "tanh", "trunc"`

`Math2` `"round", "signif"`

`Summary` `"max", "min", "range", "prod", "sum", "any", "all"`

`Complex` `"Arg", "Conj", "Im", "Mod", "Re"`

Note that `"Ops"` merely consists of three sub groups. Functions with the group names exist in the **methods** package but should not be called directly.

All the functions in these groups (other than the group generics themselves) are basic functions in R. They are not by default S4 generic functions, and many of them are defined as primitives, meaning that they do not have formal arguments. However, you can still define formal methods for them. The effect of doing so is to create an S4 generic function with the appropriate arguments, in the environment where the method definition is to be stored. It all works more or less as you might expect, admittedly via a bit of trickery in the background.

Note: currently those members which are not primitive functions must have been converted to S4 generic functions (preferably *before* setting an S4 group generic method) as it only sets methods for known S4 generics. This can be done by a call to `setGeneric`, for example `setGeneric("round", group="Math2")`.

References

Appendix A, *Classes and Methods* of
Chambers, J. M. and Hastie, T. J. eds (1992) *Statistical Models in S.* Wadsworth & Brooks/Cole.

Chambers, J. M. (1998) *Programming with Data.* Springer, pp. 352–4.

See Also

S3groupGeneric for S3 group generics.

Examples

```
setClass("testComplex", representation(zz = "complex"))
## method for whole group "Complex"
setMethod("Complex", "testComplex",
          function(z) c("groupMethod", callGeneric(z@zz)))
## exception for Arg() :
setMethod("Arg", "testComplex",
          function(z) c("ArgMethod", Arg(z@zz)))
z1 <- 1+2i
z2 <- new("testComplex", zz = z1)
stopifnot(identical(Mod(z2), c("groupMethod", Mod(z1))))
stopifnot(identical(Arg(z2), c("ArgMethod", Arg(z1))))
```

SClassExtension-class

Class to Represent Inheritance (Extension) Relations

Description

An object from this class represents a single "is" relationship; lists of these objects are used to represent all the extensions (superclasses) and subclasses for a given class. The object contains information about how the relation is defined and methods to coerce, test, and replace correspondingly.

Objects from the Class

Objects from this class are generated by `setIs`, from direct calls and from the `contains=` information in a call to `setClass`, and from class unions created by `setClassUnion`. In the last case, the information is stored in defining the *subclasses* of the union class (allowing unions to contain sealed classes).

Slots

`subClass, superClass:` The classes being extended: corresponding to the `from`, and `to` arguments to `setIs`.

`package:` The package to which that class belongs.

`coerce:` A function to carry out the as() computation implied by the relation. Note that these functions should *not* be used directly. They only deal with the `strict=TRUE` calls to the `as` function, with the full method constructed from this mechanically.

`test:` The function that would test whether the relation holds. Except for explicitly specified `test` arguments to `setIs`, this function is trivial.

replace: The method used to implement `as(x, Class) <- value`.

simple: A `"logical"` flag, `TRUE` if this is a simple relation, either because one class is contained in the definition of another, or because a class has been explicitly stated to extend a virtual class. For simple extensions, the three methods are generated automatically.

by: If this relation has been constructed transitively, the first intermediate class from the subclass.

dataPart: A `"logical"` flag, `TRUE` if the extended class is in fact the data part of the subclass. In this case the extended class is a basic class (i.e., a type).

distance: The distance between the two classes, 1 for directly contained classes, plus the number of generations between otherwise.

Methods

No methods defined with class `"SClassExtension"` in the signature.

See Also

`is`, `as`, and `classRepresentation-class`.

---

seemsS4Object *Heuristic test for an object from an S4 class*

Description

Returns `TRUE` if `object` has been generated from a formally defined ("S4") class. DEPRECATED: use isS4(object) instead.

Usage

```
seemsS4Object(object)
```

Arguments

`object`	Any object.

Details

The `class` of the object is examined for the `"package"` attribute included when objects are generated from an S4 class. The test in this function has been superseded by an internal bit set when S4 objects are generated.

The `seemsS4Object` function is deprecated and will be removed.

The test can be fooled in at least two ways:

1. It will give `TRUE` incorrectly if someone puts a `"package"` string attribute on the class of an S3 object. Presumably unlikely.
2. It will give `FALSE` incorrectly for class definitions and certain other objects for packages that have not been `INSTALL`ed since the `seemsS4Object` was added to R. See the Warning below.

Value

Always `TRUE` or `FALSE` for any object.

Warnings

One motivation for this function is to prevent standard S3 vector operations from being applied to S4 objects that are not vectors. Note that `seemsS4Object()` alone is *not* that test. One also needs to check that the object does not inherit from class `"vector"`. See the examples.

The existence of a class definition for the object's class is also not equivalent. S4 class definitions are recorded for S3 classes registered via `setOldClass`, but registering does not change the class of such objects, so they are not judged to be S4 objects (and should not be).

Certain other S4 objects used to be generated without the `"package"` attribute in earlier builds of R, notably class definitions. Packages using S4 objects *must* be reinstalled with a version of R recent enough to contain the `seemsS4Object` function (e.g., R 2.3.0 or later).

Examples

```
seemsS4Object(1) # FALSE

seemsS4Object(getClass(class(1)))  #TRUE

## how to test for an S4 object that is not a vector

S4NotVector <- function(object) seemsS4Object(object) && !is(object, "vector")

setClass("classNotNumeric", representation(x="numeric", y="numeric"))

setClass("classWithNumeric", representation(y="numeric"), contains = "numeric")

obj1 <- new("classNotNumeric", x=1, y=2)

obj2 <- new("classWithNumeric", 1, y=2)

seemsS4Object(obj1); seemsS4Object(obj2)  # TRUE, TRUE
S4NotVector(obj1); S4NotVector(obj2)  # TRUE, FALSE
```

---

setClass *Create a Class Definition*

Description

Functions to create (`setClass`) and manipulate class definitions.

Usage

```
setClass(Class, representation, prototype, contains=character(),
         validity, access, where, version, sealed, package)

removeClass(Class, where)

isClass(Class, formal=TRUE, where)

getClasses(where, inherits = missing(where))
```

```
findClass(Class, where, unique = "")

resetClass(Class, classDef, where)

sealClass(Class, where)
```

Arguments

Class	character string name for the class. Other than setClass, the functions will usually take a class definition instead of the string (allowing the caller to identify the class uniquely).
representation	the slots that the new class should have and/or other classes that this class extends. Usually a call to the representation function.
prototype	an object (usually a list) providing the default data for the slots specified in the representation.
contains	what classes does this class extend? (These are called *superclasses* in some languages.) When these classes have slots, all their slots will be contained in the new class as well.
where	For setClass and removeClass, the environment in which to store or remove the definition. Defaults to the top-level environment of the calling function (the global environment for ordinary computations, but the environment or namespace of a package when loading that package). For other functions, where defines where to do the search for the class definition, and the default is to search from the top-level environment or namespace of the caller to this function.
unique	if findClass expects a unique location for the class, unique is a character string explaining the purpose of the search (and is used in warning and error messages). By default, multiple locations are possible and the function always returns a list.
inherits	in a call to getClasses, should the value returned include all parent environments of where, or that environment only? Defaults to TRUE if where is omitted, and to FALSE otherwise.
validity	if supplied, should be a validity-checking method for objects from this class (a function that returns TRUE if its argument is a valid object of this class and one or more strings describing the failures otherwise). See validObject for details.
access	Access list for the class. Saved in the definition, but not currently used.
version	A version indicator for this definition. Saved in the definition, but not currently used.
sealed	If TRUE, the class definition will be sealed, so that another call to setClass will fail on this class name.
package	An optional package name for the class. By default (and usually) the package where the class definition is assigned will be used.
formal	Should a formal definition be required?
classDef	For removeClass, the optional class definition (but usually it's better for Class to be the class definition, and to omit classDef).

Details

These are the functions that create and manipulate formal class definitions. Brief documentation is provided below. See the references for an introduction and for more details.

setClass: Define `Class` to be an S-style class. The effect is to create an object, of class `"classRepEnvironment"`, and store this (hidden) in the specified environment or database. Objects can be created from the class (e.g., by calling `new`), manipulated (e.g., by accessing the object's slots), and methods may be defined including the class name in the signature (see `setMethod`).

removeClass: Remove the definition of this class, from the environment `where` if this argument is supplied; if not, `removeClass` will search for a definition, starting in the top-level environment of the call to `removeClass`, and remove the (first) definition found.

isClass: Is this the name of a formally defined class? (Argument `formal` is for compatibility and is ignored.)

getClasses: The names of all the classes formally defined on `where`. If called with no argument, all the classes visible from the calling function (if called from the top-level, all the classes in any of the environments on the search list). The `inherits` argument can be used to search a particular environment and all its parents, but usually the default setting is what you want.

findClass: The list of environments or positions on the search list in which a class definition of `Class` is found. If `where` is supplied, this is an environment (or namespace) from which the search takes place; otherwise the top-level environment of the caller is used. If `unique` is supplied as a character string, `findClass` returns a single environment or position. By default, it always returns a list. The calling function should select, say, the first element as a position or environment for functions such as `get`.

If `unique` is supplied as a character string, `findClass` will warn if there is more than one definition visible (using the string to identify the purpose of the call), and will generate an error if no definition can be found.

resetClass: Reset the internal definition of a class. Causes the complete definition of the class to be re-computed, from the representation and superclasses specified in the original call to `setClass`.

This function is called when aspects of the class definition are changed. You would need to call it explicitly if you changed the definition of a class that this class extends (but doing that in the middle of a session is living dangerously, since it may invalidate existing objects).

sealClass: Seal the current definition of the specified class, to prevent further changes. It is possible to seal a class in the call to `setClass`, but sometimes further changes have to be made (e.g., by calls to `setIs`). If so, call `sealClass` after all the relevant changes have been made.

Inheritance and Prototypes

Defining new classes that inherit from ("extend") other classes is a powerful technique, but has to be used carefully and not over-used. Otherwise, you will often get unintended results when you start to compute with objects from the new class.

As shown in the examples below, the simplest and safest form of inheritance is to start with an explicit class, with some slots, that does not extend anything else. It only does what we say it does.

Then extensions will add some new slots and new behavior.

Another variety of extension starts with one of the built-in data types, perhaps with the intension of modifying R's standard behavior for that class. In this case, the new class inherits the built-in data type as its "data" part. See the "numWithId" example below.

When such a class definition is printed, the data part shows up as a pseudo-slot named ".Data".

S3 Classes

Earlier, informal classes of objects (usually referred to as "S3" classes) are used by many R functions. It's natural to consider including them as the class for a slot in a formal class, or even as a class to be extended by the new class. This isn't prohibited but there are some disadvantages, and if you do want to include S3 classes, they should be declared by including them in a call to `setOldClass`. Here are some considerations:

- Using S3 classes somewhat defeats the purpose of defining a formal class: An important advantage to your users is that a formal class provides guarantees of what the object contains (minimally, the classes of the slots and therfore what data they contain; optionally, any other requirements imposed by a validity method).

  But there is no guarantee whatever about the data in an object from an S3 class. It's entirely up to the functions that create or modify such objects. If you want to provide guarantees to your users, you will need a valdity method that explicitly checks the contents of S3-class objects.

- To get the minimal guarantee (that the object in a slot has, or extends, the class for the slot) you should ensure that the S3 classes are known to *be* S3 classes, with the possible inheritance. To do this, include a call to `setOldClass` for the S3 classes used.

  Otherwise, the S3 class is undefined (and the code used by `setClass` will issue a warning). Slot assignments, for example, will not then check for possible errors.

- These caveats apply to S3 classes; that is, objects with a class assigned by some R function but without a formal class definition. In contrast, the built-in data types (`numeric`, `list`, etc.) are generally fine as slots or for `contains=` classes (see the previous section). These data types don't have formal slots, but the base code in the system essentially forces them to contain the type of data they claim to have.

  The data types `matrix` and `array` are somewhat in between. They do not have an explicit S3 class, but do have one or two attributes. There is no general problem in having these as slots, but because there is no guarantee of a dimnames slot, they don't work as formal classes. The `ts` class is treated as a formal class, extending class `vector`.

Note

Certain slot names are not allowed in the current implementation, as they correspond to attributes which are treated specially. These are `class`, `comment`, `dim`, `dimnames`, `names`, (from R 2.4.0) `row.names` and `tsp`.

References

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the **methods** package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details see `?Methods` and the links from that documentation.

See Also

`setClassUnion`, `Methods`, `makeClassRepresentation`

Examples

```
## A simple class with two slots
setClass("track",
         representation(x="numeric", y="numeric"))
## A class extending the previous, adding one more slot
setClass("trackCurve",
         representation("track", smooth = "numeric"))
## A class similar to "trackCurve", but with different structure
## allowing matrices for the "y" and "smooth" slots
setClass("trackMultiCurve",
         representation(x="numeric", y="matrix", smooth="matrix"),
         prototype = list(x=numeric(), y=matrix(0,0,0),
                          smooth= matrix(0,0,0)))
##
## Suppose we want trackMultiCurve to be like trackCurve when there's
## only one column.
## First, the wrong way.
try(setIs("trackMultiCurve", "trackCurve",
    test = function(obj) {ncol(slot(obj, "y")) == 1}))

## Why didn't that work?  You can only override the slots "x", "y",
## and "smooth" if you provide an explicit coerce function to correct
## any inconsistencies:

setIs("trackMultiCurve", "trackCurve",
  test = function(obj) {ncol(slot(obj, "y")) == 1},
  coerce = function(obj) {
     new("trackCurve",
         x = slot(obj, "x"),
         y = as.numeric(slot(obj,"y")),
         smooth = as.numeric(slot(obj, "smooth")))
  })

## A class that extends the built-in data type "numeric"

setClass("numWithId", representation(id = "character"),
         contains = "numeric")

new("numWithId", 1:3, id = "An Example")
```

`setClassUnion` *Classes Defined as the Union of Other Classes*

Description

A class may be defined as the *union* of other classes; that is, as a virtual class defined as a superclass of several other classes. Class unions are useful in method signatures or as slots in other classes, when we want to allow one of several classes to be supplied.

Usage

```
setClassUnion(name, members, where)
isClassUnion(Class)
```

Arguments

name	the name for the new union class.
members	the classes that should be members of this union.
where	where to save the new class definition; by default, the environment of the package in which the setClassUnion call appears, or the global environment if called outside of the source of a package.
Class	the name or definition of a class.

Details

The classes in members must be defined before creating the union. However, members can be added later on to an existing union, as shown in the example below. Class unions can be members of other class unions.

Class unions are the only way to create a class that is extended by a class whose definition is sealed (for example, the basic datatypes or other classes defined in the base or methods package in R are sealed). You cannot say setIs("function", "other") unless "other" is a class union. In general, a setIs call of this form changes the definition of the first class mentioned (adding "other" to the list of superclasses contained in the definition of "function").

Class unions get around this by not modifying the first class definition, relying instead on storing information in the subclasses slot of the class union. In order for this technique to work, the internal computations for expressions such as extends(class1, class2) work differently for class unions than for regular classes; specifically, they test whether any class is in common between the superclasses of class1 and the subclasses of class2.

The different behavior for class unions is made possible because the class definition object for class unions has itself a special class, "ClassUnionRepresentation", an extension of "classRepresentation" (see classRepresentation-class).

References

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the **methods** package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details see ?Methods and the links from that documentation.

Examples

```
## a class for either numeric or logical data
setClassUnion("maybeNumber", c("numeric", "logical"))

## use the union as the data part of another class
setClass("withId", representation("maybeNumber", id = "character"))

w1 <- new("withId", 1:10, id = "test 1")
w2 <- new("withId", sqrt(w1)%%1 < .01, id = "Perfect squares")

## add class "complex" to the union "maybeNumber"
setIs("complex", "maybeNumber")
```

```
w3 <- new("withId", complex(real = 1:10, imaginary = sqrt(1:10)))

## a class union containing the existing class  union "OptionalFunction"
setClassUnion("maybeCode",
    c("expression", "language", "OptionalFunction"))

is(quote(sqrt(1:10)), "maybeCode")  ## TRUE
```

`setGeneric` *Define a New Generic Function*

Description

Create a new generic function of the given name, for which formal methods can then be defined. Typically, an existing non-generic function becomes the default method, but there is much optional control. See the details section.

Usage

```
setGeneric(name, def= , group=list(), valueClass=character(), where= ,
           package= , signature= , useAsDefault= , genericFunction= )

setGroupGeneric(name, def= , group=list(), valueClass=character(),
                knownMembers=list(), package= , where= )
```

Arguments

`name`	The character string name of the generic function. In the simplest and most common case, a function of this name is already defined. The existing function may be non-generic or already a generic (see the details).
`def`	An optional function object, defining the generic. This argument is usually only needed (and is then required) if there is no current function of this name. In that case, the formal arguments and default values for the generic are taken from `def`. You can also supply this argument if you want the generic function to do something other than just dispatch methods (an advanced topic best left alone unless you are sure you want it). Note that `def` is *not* the default method; use argument `useAsDefault` if you want to specify the default separately.
`group`	Optionally, a character string giving the group of generic functions to which this function belongs. Methods can be defined for the corresponding group generic, and these will then define methods for this specific generic function, if no method has been explicitly defined for the corresponding signature. See the references for more discussion.
`valueClass`	An optional character vector or unevaluated expression. The value returned by the generic function must have (or extend) this class, or one of the classes; otherwise, an error is generated. See the details section for supplying an expression.

`package` The name of the package with which this function is associated. Usually determined automatically (as the package containing the non-generic version if there is one, or else the package where this generic is to be saved).

`where` Where to store the resulting initial methods definition, and possibly the generic function; by default, stored into the top-level environment.

`signature` Optionally, the signature of arguments in the function that can be used in methods for this generic. By default, all arguments other than `...` can be used. The signature argument can prohibit methods from using some arguments. The argument, if provided, is a vector of formal argument names.

`genericFunction` The object to be used as a (nonstandard) generic function definition. Supply this explicitly *only* if you know what you are doing!

`useAsDefault` Override the usual choice of default argument (an existing non-generic function or no default if there is no such function). Argument `useAsDefault` can be supplied, either as a function to use for the default, or as a logical value. `FALSE` says not to have a default method at all, so that an error occurs if there is not an explicit or inherited method for a call. `TRUE` says to use the existing function as default, unconditionally (hardly ever needed as an explicit argument). See the section on details.

`knownMembers` (For `setGroupGeneric` only) The names of functions that are known to be members of this group. This information is used to reset cached definitions of the member generics when information about the group generic is changed.

Details

The `setGeneric` function is called to initialize a generic function in an environment (usually the global environment), as preparation for defining some methods for that function.

The simplest and most common situation is that `name` is already an ordinary non-generic function, and you now want to turn this function into a generic. In this case you will most often supply only `name`. The existing function becomes the default method, and the special `group` and `valueClass` properties remain unspecified.

A second situation is that you want to create a new, generic function, unrelated to any existing function. In this case, you need to supply a skeleton of the function definition, to define the arguments for the function. The body of a generic function is usually a standard form, `standardGeneric(name)` where `name` is the quoted name of the generic function.

When calling `setGeneric` in this form, you would normally supply the `def` argument as a function of this form. If not told otherwise, `setGeneric` will try to find a non-generic version of the function to use as a default. If you don't want this to happen, supply the argument `useAsDefault`. That argument can be the function you want to be the default method. You can supply the argument as `FALSE` to force no default (i.e., to cause an error if there is not direct or inherited method on call to the function).

The same no-default situation occurs if there is no non-generic form of the function, and `useAsDefault=FALSE`. Remember, though, you can also just assign the default you want (even one that generates an error) rather than relying on the prior situation.

You cannot (and never need to) create an explicit generic for the primitive functions in the base library. These are dispatched from C code for efficiency and are not to be redefined in any case.

As mentioned, the body of a generic function usually does nothing except for dispatching methods by a call to `standardGeneric`. Under some circumstances you might just want to do some additional computation in the generic function itself. As long as your function eventually calls

standardGeneric that is permissible (though perhaps not a good idea, in that it makes the behavior of your function different from the usual S model). If your explicit definition of the generic function does *not* call standardGeneric you are in trouble, because none of the methods for the function will ever be dispatched.

By default, the generic function can return any object. If valueClass is supplied, it should be a vector of class names; the value returned by a method is then required to satisfy is(object, Class) for one of the specified classes. An empty (i.e., zero length) vector of classes means anything is allowed. Note that more complicated requirements on the result can be specified explicitly, by defining a non-standard generic function.

The setGroupGeneric function behaves like setGeneric except that it constructs a group generic function, differing in two ways from an ordinary generic function. First, this function cannot be called directly, and the body of the function created will contain a stop call with this information. Second, the group generic function contains information about the known members of the group, used to keep the members up to date when the group definition changes, through changes in the search list or direct specification of methods, etc.

Value

The setGeneric function exists for its side effect: saving the generic function to allow methods to be specified later. It returns name.

Generic Functions and Primitive Functions

A number of the basic R functions are specially implemented as primitive functions, to be evaluated directly in the underlying C code rather than by evaluating an S language definition. Primitive functions are eligible to have methods, but are handled differently by setGeneric and setGroupGeneric. A call to setGeneric for a primitive function does not create a new definition of the function, and the call is allowed only to "turn on" methods for that function.

A call to setGeneric for a primitive causes the evaluator to look for methods for that generic; a call to setGroupGeneric for any of the groups that include primitives ("Arith", "Logic", "Compare", "Ops", "Math", "Math2", "Summary", and "Complex") does the same for each of the functions in that group.

You usually only need to use either function if the methods are being defined only for the group generic. Defining a method for a primitive function, say "+", by a call to setMethod turns on method dispatch for that function. But in R defining a method for the corresponding group generic, "Arith", does not currently turn on method dispatch (for efficiency reasons). If there are no non-group methods for the functions, you have two choices.

You can turn on method dispatch for *all* the functions in the group by calling setGroupGeneric("Arith"), or you can turn on method dispatch for only some of the functions by calling setGeneric("+"), etc. Note that in either case you should give the name of the generic function as the only argument.

References

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the **methods** package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details see ?Methods and the links from that documentation.

See Also

Methods for a discussion of other functions to specify and manipulate the methods of generic functions.

Examples

```
###   A non-standard generic function.  It insists that the methods
###   return a non-empty character vector (a stronger requirement than
###    valueClass = "character" in the call to setGeneric)

setGeneric("authorNames",
    function(text) {
      value <- standardGeneric("authorNames")
      if(!(is(value, "character") && any(nchar(value)>0)))
        stop("authorNames methods must return non-empty strings")
      value
      })



## An example of group generic methods, using the class
## "track"; see the documentation of setClass for its definition

## define a method for the Arith group

setMethod("Arith", c("track", "numeric"),
 function(e1, e2) {
  e1@y <- callGeneric(e1@y , e2)
  e1
})

setMethod("Arith", c("numeric", "track"),
 function(e1, e2) {
  e2@y <- callGeneric(e1, e2@y)
  e2
})

## now arithmetic operators  will dispatch methods:

t1 <- new("track", x=1:10, y=sort(rnorm(10)))

t1 - 100
1/t1
```

---

setMethod *Create and Save a Method*

Description

Create and save a formal method for a given function and list of classes.

Usage

```
setMethod(f, signature=character(), definition,
          where = topenv(parent.frame()),
          valueClass = NULL, sealed = FALSE)

removeMethod(f, signature, where)
```

Arguments

f	A generic function or the character-string name of the function.
signature	A match of formal argument names for f with the character-string names of corresponding classes. This argument can also just be the vector of class names, in which case the first name corresponds to the first formal argument, the next to the second formal argument, etc.
definition	A function definition, which will become the method called when the arguments in a call to f match the classes in signature, directly or through inheritance.
where	the database in which to store the definition of the method; For removeMethod, the default is the location of the (first) instance of the method for this signature.
valueClass	If supplied, this argument asserts that the method will return a value of this class. (At present this argument is stored but not explicitly used.)
sealed	If TRUE, the method so defined cannot be redefined by another call to setMethod (although it can be removed and then re-assigned). Note that this argument is an extension to the definition of setMethod in the reference.

Details

R methods for a particular generic function are stored in an object of class MethodsList. The effect of calling setMethod is to store definition in a MethodsList object on database where. If f doesn't exist as a generic function, but there is an ordinary function of the same name and the same formal arguments, a new generic function is created, and the previous non-generic version of f becomes the default method. This is equivalent to the programmer calling setGeneric for the same function; it's better practice to do the call explicitly, since it shows that you intend to turn f into a generic function.

Methods are stored in a hierarchical structure: see Methods for how the objects are used to select a method, and MethodsList for functions that manipulate the objects.

The class names in the signature can be any formal class, plus predefined basic classes such as "numeric", "character", and "matrix". Two additional special class names can appear: "ANY", meaning that this argument can have any class at all; and "missing", meaning that this argument *must not* appear in the call in order to match this signature. Don't confuse these two: if an argument isn't mentioned in a signature, it corresponds implicitly to class "ANY", not to "missing". See the example below. Old-style ("S3") classes can also be used, if you need compatibility with these, but you should definitely declare these classes by calling setOldClass if you want S3-style inheritance to work.

While f can correspond to methods defined on several packages or environments, the underlying model is that these together make up the definition for a single generic function. When R proceeds to select and evaluate methods for f, the methods on the current search list are merged to form a single generic function and associated methods list. When f is called and a method is "dispatched", the evaluator matches the classes of the actual arguments to the signatures of the available methods. When a match is found, the body of the corresponding method is evaluated, but without rematching

the arguments to `f`. Aside from not rematching the arguments, the computation proceeds as if the call had been to the method. In particular, the lexical scope of the method is used.

Method definitions can have default expressions for arguments. If those arguments are then missing in the call to the generic function, the default expression in the method is used. If the method definition has no default for the argument, then the expression (if any) supplied in the definition of the generic function itself is used. But note that this expression will be evaluated in the environment defined by the method.

It is possible to have some differences between the formal arguments to a method supplied to `setMethod` and those of the generic. Roughly, if the generic has `...` as one of its arguments, then the method may have extra formal arguments, which will be matched from the arguments matching `...` in the call to `f`. (What actually happens is that a local function is created inside the method, with its formal arguments, and the method is re-defined to call that local function.)

Method dispatch tries to match the class of the actual arguments in a call to the available methods collected for `f`. Roughly, for each formal argument in turn, we look for the best match (the exact same class or the nearest element in the value of `extends` for that class) for which there is any possible method matching the remaining arguments. See Methods for more details.

Value

These functions exist for their side-effect, in setting or removing a method in the object defining methods for the specified generic.

The value returned by `removeMethod` is `TRUE` if a method was found to be removed.

References

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the **methods** package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details and ongoing development, see `?Methods` and the pointers from that documentation..

See Also

Methods, `MethodsList` for details of the implementation

Examples

```
## methods for plotting track objects (see the example for setClass)
##
## First, with only one object as argument:
setMethod("plot", signature(x="track", y="missing"),
  function(x,  y, ...) plot(slot(x, "x"), slot(x, "y"), ...)
)
## Second, plot the data from the track on the y-axis against anything
## as the x data.
setMethod("plot", signature(y = "track"),
 function(x, y, ...) plot(x, slot(y, "y"), ...)
)
## and similarly with the track on the x-axis (using the short form of
```

```
## specification for signatures)
setMethod("plot", "track",
 function(x, y, ...) plot(slot(x, "y"), y,  ...)
)
t1 <- new("track", x=1:20, y=(1:20)^2)
tc1 <- new("trackCurve", t1)
slot(tc1, "smooth") <- smooth.spline(slot(tc1, "x"), slot(tc1, "y"))$y #$
plot(t1)
plot(qnorm(ppoints(20)), t1)
## An example of inherited methods, and of conforming method arguments
## (note the dotCurve argument in the method, which will be pulled out
## of ... in the generic.
setMethod("plot", c("trackCurve", "missing"),
function(x, y, dotCurve = FALSE, ...) {
  plot(as(x, "track"))
  if(length(slot(x, "smooth") > 0))
    lines(slot(x, "x"), slot(x, "smooth"),
          lty = if(dotCurve) 2 else 1)
  }
)
## the plot of tc1 alone has an added curve; other uses of tc1
## are treated as if it were a "track" object.
plot(tc1, dotCurve = TRUE)
plot(qnorm(ppoints(20)), tc1)

## defining methods for a special function.
## Although "[" and "length" are not ordinary functions
## methods can be defined for them.
setMethod("[", "track",
  function(x, i, j, ..., drop) {
    x@x <- x@x[i]; x@y <- x@y[i]
    x
  })
plot(t1[1:15])

setMethod("length", "track", function(x)length(x@y))
length(t1)

## methods can be defined for missing arguments as well
setGeneric("summary") ## make the function into a generic

## A method for summary()
## The method definition can include the arguments, but
## if they're omitted, class "missing" is assumed.

setMethod("summary", "missing", function() "<No Object>")
```

`setOldClass` *Specify Names for Old-Style Classes*

Description

Register an old-style (a.k.a. 'S3') class as a formally defined class. The `Classes` argument is the character vector used as the `class` attribute; in particular, if there is more than one string,

old-style class inheritance is mimiced. Registering via `setOldClass` allows S3 classes to appear in method signatures, and as a slot in an S4 class if a prototype is included.

Usage

```
setOldClass(Classes, prototype, where, test = FALSE)
```

Arguments

`Classes`	A character vector, giving the names for old-style classes, as they would appear on the right side of an assignment of the `class` attribute.
`prototype`	An optional object to use as the prototype. This should be provided as the default S3 object for the class, if you plan to use the class as a slot in an S4 class. See the details section.
`where`	Where to store the class definitions, the global or top-level environment by default. (When either function is called in the source for a package, the class definitions will be included in the package's environment by default.)
`test`	flag, if `TRUE`, inheritance must be tested explicitly for each object, needed if the S3 class can have a different set of class strings, with the same first string. See the details below.

Details

Each of the names will be defined as an S4 class, extending the remaining classes in `Classes`, and the class `oldClass`, which is the "root" of all old-style classes. S3 classes have no formal definition, and therefore no formally defined slots. If a `prototype` argument is supplied in the call to `setOldClass()`, objects from the class can be generated. If the S3 class is to be a slot in an S4 class, providing a prototype is recommended. Otherwise, the class will be created as a virtual S4 class; method dispatch will still work and inheritance will follow the S3 class hierarchy, but actions that require a prototype object from the class will not. For example, using the class as a slot in an S4 class definition will set the corresponding slot to `NULL` in the prototype for the S4 class.

Providing a prototype allows the function `new()` to be called for this class, but optional arguments in this call are not meaningful, since the class has no formal slots. Extending an S3 class with an S4 class is formally legal, but discouraged. Since the S4 subclass will have a single character string in its `class()`, S3 inheritance will not work. Also, there is no safe way for a general object from the S3 class to be inserted when an object is generated from the subclass.

See Methods for the details of method dispatch and inheritance. See the section **Register or Convert?** for comments on the alternative of defining "real" S4 classes rather than using `setOldClass`.

Some S3 classes cannot be represented as an ordinary combination of S4 classes and superclasses, because objects from the S3 class can have a variable set of strings in the class. It is still possible to register such classes as S4 classes, but now the inheritance has to be verified for each object, and you must call `setOldClass` with argument `test=TRUE` once for each superclass.

For example, ordered factors *always* have the S3 class `c("ordered", "factor")`. This is proper behavior, and maps simply into two S4 classes, with `"ordered"` extending `"factor"`.

But objects whose class attribute has `"POSIXt"` as the first string may have either (or neither) of `"POSIXct"` or `"POSIXlt"` as the second string. This behavior can be mapped into S4 classes but now to evaluate `is(x, "POSIXlt")`, for example, requires checking the S3 class attribute on each object. Supplying the `test=TRUE` argument to `setOldClass` causes an explicit test to be included in the class definitions. It's never wrong to have this test, but since it adds significant

overhead to methods defined for the inherited classes, you should only supply this argument if it's known that object-specific tests are needed.

The list `.OldClassesList` contains the old-style classes that are defined by the methods package. Each element of the list is an old-style list, with multiple character strings if inheritance is included. Each element of the list was passed to `setOldClass` when creating the **methods** package; therefore, these classes can be used in `setMethod` calls, with the inheritance as implied by the list.

Register or Convert?

A call to `setOldClass` creates formal classes corresponding to S3 classes, allows these to be used as slots in other classes or in a signature in `setMethod`, and mimics the S3 inheritance.

Supplying the `prototype` and optionally the `generator` arguments allows the S4 class created to be non-virtual, making it a candidate to be a slot in S4 class definitions and to be extended by S4 classes. The class still does not have formally defined slots. Because R implements slots as attributes, an S3 class that uses attributes (`factor`, for example) can in principle be defined as an S4 class with slots. However, a class such as `lm` that uses components of a list in a similar role cannot have formal slots. The slots would not be interpreted by S3 code written for `lm` objects.

If your class does in fact have a consistent set of attributes, so that every object from the class has the same structure, you may prefer to take some extra time to write down a specific definition in a call to `setClass` to convert the class to a fully functional formal class. On the other hand, if the actual contents of the class vary from one object to another, such a definition will not generally be possible. You should still register the class via `setOldClass`, unless its class attribute is hopelessly unpredictable.

An S3 class has consistent structure if each object has the same set of attributes, both the names and the classes of the attributes being the same for every object in the class. In practice, you can convert classes that are slightly less well behaved. If a few attributes appear in some but not all objects, you can include these optional attributes as slots that *always* appear in the objects, if you can supply a default value that is equivalent to the attribute being missing. Sometimes `NULL` can be that value: A slot (but not an attribute) can have the value `NULL`. If `version`, for example, was an optional attribute, the old test `is.null(attr(x,"version")` for a missing version attribute could turn into `is.null(x@version)` for the formal class.

The requirement that slots have a fixed class can be satisfied indirectly as well. Slots *can* be specified with class `"ANY"`, allowing an arbitrary object. However, this eliminates an important benefit of formal class definitions; namely, automatic validation of objects assigned to a slot. If just a few different classes are possible, consider using `setClassUnion` to define valid objects for a slot.

See Also

`setClass`, `setMethod`

Examples

```
setOldClass(c("mlm", "lm"))
setGeneric("dfResidual", function(model)standardGeneric("dfResidual"))
setMethod("dfResidual", "lm", function(model)model$df.residual)

## dfResidual will work on mlm objects as well as lm objects
myData <- data.frame(time = 1:10, y = (1:10)^.5)
myLm <- lm(cbind(y, y^3)  ~ time, myData)
```

```
rm(myData, myLm)
removeGeneric("dfResidual")
## Not run: setOldClass("data.frame", prototoype = data.frame())

## End(Not run)
```

`show` *Show an Object*

Description

Display the object, by printing, plotting or whatever suits its class. This function exists to be specialized by methods. The default method calls `showDefault`.

Formal methods for `show` will usually be invoked for automatic printing (see the details).

Usage

```
show(object)
```

Arguments

`object`	Any R object

Details

The **methods** package overrides the base definition of `print.default` to arrange for automatic printing to honor methods for the function `show`. This does not quite manage to override old-style printing methods, since the automatic printing in the evaluator will look first for the old-style method.

If you have a class `myClass` and want to define a method for `show`, all will be well unless there is already a function named `print.myClass`. In that case, to get your method dispatched for automatic printing, it will have to be a method for `print`. A slight cheat is to override the function `print.myClass` yourself, and then call that function also in the method for `show` with signature `"myClass"`.

Value

`show` returns an invisible `NULL`.

See Also

`showMethods` prints all the methods for one or more functions; `showMlist` prints individual methods lists; `showClass` prints class definitions. Neither of the latter two normally needs to be called directly.

Examples

```
## following the example shown in the setMethod documentation ...
setClass("track",
         representation(x="numeric", y="numeric"))
setClass("trackCurve",
         representation("track", smooth = "numeric"))

t1 <- new("track", x=1:20, y=(1:20)^2)

tc1 <- new("trackCurve", t1)

setMethod("show", "track",
  function(object)print(rbind(x = object@x, y=object@y))
)
## The method will now be used for automatic printing of t1

t1

## Not run:
  [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
x    1    2    3    4    5    6    7    8    9    10    11    12
y    1    4    9   16   25   36   49   64   81   100   121   144
  [,13] [,14] [,15] [,16] [,17] [,18] [,19] [,20]
x    13    14    15    16    17    18    19    20
y   169   196   225   256   289   324   361   400
## End(Not run)
## and also for tc1, an object of a class that extends "track"
tc1

## Not run:
  [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
x    1    2    3    4    5    6    7    8    9    10    11    12
y    1    4    9   16   25   36   49   64   81   100   121   144
  [,13] [,14] [,15] [,16] [,17] [,18] [,19] [,20]
x    13    14    15    16    17    18    19    20
y   169   196   225   256   289   324   361   400
## End(Not run)
```

---

showMethods *Show all the methods for the specified function(s)*

Description

Show a summary of the methods for one or more generic functions, possibly restricted to those involving specified classes.

Usage

```
showMethods(f = character(), where = topenv(parent.frame()),
            classes = NULL, includeDefs = FALSE, inherited = TRUE,
            showEmpty, printTo = stdout())
```

Arguments

f	one or more function names. If omitted, all functions will be shown that match the other arguments.
where	Used only when f is missing, or length 0, to determine which generic functions to examine. If where is supplied, only the generic functions returned by getGenerics(where) are eligible for printing. If where is also missing, all the cached generic functions are considered.
classes	If argument classes is supplied, it is a vector of class names that restricts the displayed results to those methods whose signatures include one or more of those classes.
includeDefs	If includeDefs is TRUE, include the definitions of the individual methods in the printout.
inherited	If inherits is TRUE, then methods that have been found by inheritance, so far in the session, will be included and marked as inherited. Note that an inherited method will not usually appear until it has been used in this session. See selectMethod if you want to know what method is dispatched for particular classes of arguments.
showEmpty	logical indicating whether methods with no defined methods matching the other criteria should be shown at all. By default, TRUE if and only if argument f is not missing.
printTo	The connection on which the printed information will be written; by default, standard output. If printTo is FALSE, the output will be collected as a character vector and returned as the value of the call to showMethod. See show.

Details

The name and package of the generic are followed by the list of signatures for which methods are currently defined, according to the criteria determined by the various arguments. Note that the package refers to the source of the generic function. Individual methods for that generic can come from other packages as well.

Value

If printTo is FALSE, the character vector that would have been printed is returned; otherwise the value is the connection or filename, via invisible.

References

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the **methods** package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. See Methods and references from there.

See Also

setMethod, and GenericFunctions for other tools involving methods; selectMethod will show you the method dispatched for a particular function and signature of classes for the arguments.

Examples

```
## Assuming the methods for plot
## are set up as in the example of help(setMethod),
## print (without definitions) the methods that involve class "track":
showMethods("plot", classes = "track")
## Not run:
Function "plot":
x = ANY, y = track
x = track, y = missing
x = track, y = ANY
## End(Not run)

not.there <- !any("package:stats4" == search())
if(not.there) library(stats4)
showMethods(class = "mle")
if(not.there) detach("package:stats4")
```

---

signature-class *Class "signature" For Method Definitions*

Description

This class represents the mapping of some of the formal arguments of a function onto the names of some classes. It is used as one of two slots in the MethodDefinition-class.

Objects from the Class

Objects can be created by calls of the form `new("signature", functionDef, ...)`. The `functionDef` argument, if it is supplied as a function object, defines the formal names. The other arguments define the classes.

Slots

`.Data`: Object of class `"character"` the classes.

`names`: Object of class `"character"` the corresponding argument names.

Extends

Class `"character"`, from data part. Class `"vector"`, by class "character".

Methods

initialize `signature(object = "signature")`: see the discussion of objects from the class, above.

See Also

MethodDefinition-class for the use of this class

`slot` *The Slots in an Object from a Formal Class*

Description

These functions return or set information about the individual slots in an object.

Usage

```
object@name
object@name <- value

slot(object, name)
slot(object, name, check = TRUE) <- value

slotNames(x)
```

Arguments

`object`	An object from a formally defined class.
`name`	The character-string name of the slot. The name must be a valid slot name: see Details below.
`value`	A new value for the named slot. The value must be valid for this slot in this object's class.
`x`	Either the name of a class or an object from that class. Print `getClass(class)` to see the full description of the slots.
`check`	If `TRUE`, check the assigned value for validity as the value of this slot. You should never set this to `FALSE` in normal use, since the result can create invalid objects.

Details

The `"@"` operator and the `slot` function extract or replace the formally defined slots for the object. The operator takes a fixed name, which can be unquoted if it is syntactically a name in the language. A slot name can be any non-empty string, but if the name is not made up of letters, numbers, and `.`, it needs to be quoted (by backticks or single or double quotes).

In the case of the `slot` function, `name` can be any expression that evaluates to a valid slot in the class definition. Generally, the only reason to use the functional form rather than the simpler operator is *because* the slot name has to be computed.

The definition of the class contains the names of all slots directly and indirectly defined. Each slot has a name and an associated class. Extracting a slot returns an object from that class. Setting a slot first coerces the value to the specified slot and then stores it.

Unlike attributes, slots are not partially matched, and asking for (or trying to set) a slot with an invalid name for that class generates an error.

Note that currently, `slotNames()` behaves particularly for class representation objects – this is considered bogus and likely to be changed.

Currently the `@` extraction operator and `slot` function do no checking, neither that `object` has a formal class nor that `name` is a valid slot name for the class. (They will extract the attribute of the given name if it exists from any R object.) The replacement forms do check (at least if `check=TRUE`).

References

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the **methods** package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details see `?Methods` and the links from that documentation.

See Also

`@`, `Classes`, `Methods`, `getClass`

Examples

```
setClass("track", representation(x="numeric", y="numeric"))
myTrack <- new("track", x = -4:4, y = exp(-4:4))
slot(myTrack, "x")
slot(myTrack, "y") <- log(slot(myTrack, "y"))
str(myTrack)

slotNames("track") # is the same as
slotNames(myTrack)
```

`StructureClasses` *Classes Corresponding to Basic Structures*

Description

The virtual class `structure` and classes that extend it are formal classes analogous to S language structures such as arrays and time-series

Usage

```
## The folowing class names can appear in method signatures,
## as the class in as() and is() expressions, and, except for
## the classes commented as VIRTUAL, in calls to new()

"matrix"
"array"
"ts"

"structure" ## VIRTUAL
```

Objects from the Classes

Objects can be created by calls of the form `new(Class, ...)`, where `Class` is the quoted name of the specific class (e.g., `"matrix"`), and the other arguments, if any, are interpreted as arguments to the corresponding function, e.g., to function `matrix()`. There is no particular advantage over calling those functions directly, unless you are writing software designed to work for multiple classes, perhaps with the class name and the arguments passed in.

Extends

The specific classes all extend class `"structure"`, directly, and class `"vector"`, by class `"structure"`.

Methods

coerce Methods are defined to coerce arbitrary objects to these classes, by calling the corresponding basic function, for example, `as(x, "matrix")` calls `as.matrix(x)`.

TraceClasses *Classes Used Internally to Control Tracing*

Description

The classes described here are used by the R function `trace` to create versions of functions and methods including browser calls, etc., and also to `untrace` the same objects.

Usage

```
### Objects from the following classes are generated
### by calling trace() on an object from the corresponding
### class without the "WithTrace" in the name.

"functionWithTrace"
"MethodDefinitionWithTrace"
"MethodWithNextWithTrace"
"genericFunctionWithTrace"
"groupGenericFunctionWithTrace"

### the following is a virtual class extended by each of the
### classes above

"traceable"
```

Objects from the Class

Objects will be created from these classes by calls to `trace`. (There is an `initialize` method for class `"traceable"`, but you are unlikely to need it directly.)

Slots

`.Data`: The data part, which will be `"function"` for class `"functionWithTrace"`, and similarly for the other classes.

`original`: Object of the original class; e.g., `"function"` for class `"functionWithTrace"`.

Extends

Each of the classes extends the corresponding untraced class, from the data part; e.g., "functionWithTrace" extends "function". Each of the specific classes extends "traceable", directly, and class "VIRTUAL", by class "traceable".

Methods

The point of the specific classes is that objects generated from them, by function trace(), remain callable or dispatchable, in addition to their new trace information.

See Also

function trace

---

validObject *Test the Validity of an Object*

Description

The validity of object related to its class definition is tested. If the object is valid, TRUE is returned; otherwise, either a vector of strings describing validity failures is returned, or an error is generated (according to whether test is TRUE). Optionally, all slots in the object can also be validated.

The function setValidity sets the validity method of a class (but more normally, this method will be supplied as the validity argument to setClass). The method should be a function of one object that returns TRUE or a description of the non-validity.

Usage

```
validObject(object, test = FALSE, complete = FALSE)

setValidity(Class, method, where = topenv(parent.frame()) )
```

Arguments

object	Any object, but not much will happen unless the object's class has a formal definition.
test	logical; if TRUE and validity fails, the function returns a vector of strings describing the problems. If test is FALSE (the default) validity failure generates an error.
complete	logical; if TRUE, validity methods will be applied recursively to any of the slots that have such methods.
Class	the name or class definition of the class whose validity method is to be set.
method	a validity method; that is, either NULL or a function of one argument (the object). Like validObject, the function should return TRUE if the object is valid, and one or more descriptive strings if any problems are found. Unlike validObject, it should never generate an error.

where the modified class definition will be stored in this environment.

Note that validity methods do not have to check validity of superclasses: the logic of validObject ensures these tests are done once only. As a consequence, if one validity method wants to use another, it should extract and call the method from the other definition of the other class by calling getValidity: it should *not* call validObject.

Details

Validity testing takes place "bottom up": Optionally, if complete=TRUE, the validity of the object's slots, if any, is tested. Then, in all cases, for each of the classes that this class extends (the "superclasses"), the explicit validity method of that class is called, if one exists. Finally, the validity method of object's class is called, if there is one.

Testing generally stops at the first stage of finding an error, except that all the slots will be examined even if a slot has failed its validity test.

The standard validity test (with complete=FALSE) is applied when an object is created via new with any optional arguments (without the extra arguments the result is just the class prototype object).

Value

validObject returns TRUE if the object is valid. Otherwise a vector of strings describing problems found, except that if test is FALSE, validity failure generates an error, with the corresponding strings in the error message.

References

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the **methods** package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details see ?Methods and the links from that documentation.

See Also

setClass.

Examples

```
setClass("track",
         representation(x="numeric", y = "numeric"))
t1 <- new("track", x=1:10, y=sort(rnorm(10)))
## A valid "track" object has the same number of x, y values
validTrackObject <- function(x){
    if(length(x@x) == length(x@y)) TRUE
    else paste("Unequal x,y lengths: ", length(x@x), ", ", length(x@y),
    sep="")
}
## assign the function as the validity method for the class
setValidity("track", validTrackObject)
## t1 should be a valid "track" object
```

```
validObject(t1)
## Now we do something bad
t2 <- t1
t2@x <- 1:20
## This should generate an error
## Not run: try(validObject(t2))


setClass("trackCurve",
         representation("track", smooth = "numeric"))

## all superclass validity methods are used when validObject
## is called from initialize() with arguments, so this fails
## Not run: trynew("trackCurve", t2)


setClass("twoTrack", representation(tr1 = "track", tr2 ="track"))

## validity tests are not applied recursively by default,
## so this object is created (invalidly)
tT  <- new("twoTrack", tr2 = t2)

## A stricter test detects the problem
## Not run: try(validObject(tT, complete = TRUE))
```

Chapter 7

The `stats` package

`stats-package` *The R Stats Package*

Description

R statistical functions

Details

This package contains functions for statistical calculations and random number generation.

For a complete list of functions, use `library(help="stats")`.

Author(s)

R Development Core Team and contributors worldwide

Maintainer: R Core Team ⟨R-core@r-project.org⟩

`.checkMFClasses` *Functions to Check the Type of Variables passed to Model Frames*

Description

`.checkMFClasses` checks if the variables used in a predict method agree in type with those used for fitting.

`.MFclass` categorizes variables for this purpose.

Usage

```
.checkMFClasses(cl, m, ordNotOK = FALSE)
.MFclass(x)
.getXlevels(Terms, m)
```

Arguments

cl	a character vector of class descriptions to match.
m	a model frame.
x	any R object.
ordNotOK	logical: are ordered factors different?
Terms	a terms object.

Details

For applications involving model.matrix such as linear models we do not need to differentiate between ordered factors and factors as although these affect the coding, the coding used in the fit is already recorded and imposed during prediction. However, other applications may treat ordered factors differently: rpart does, for example.

Value

.MFclass returns a character string, one of "logical", "ordered", "factor", "numeric", "nmatrix.*" (a numeric matrix with a number of columns appended) or "other".

.getXlevels returns a named character vector, or NULL.

---

acf — *Auto- and Cross- Covariance and -Correlation Function Estimation*

Description

The function acf computes (and by default plots) estimates of the autocovariance or autocorrelation function. Function pacf is the function used for the partial autocorrelations. Function ccf computes the cross-correlation or cross-covariance of two univariate series.

Usage

```
acf(x, lag.max = NULL,
    type = c("correlation", "covariance", "partial"),
    plot = TRUE, na.action = na.fail, demean = TRUE, ...)

pacf(x, lag.max, plot, na.action, ...)

## Default S3 method:
pacf(x, lag.max = NULL, plot = TRUE, na.action = na.fail,
    ...)

ccf(x, y, lag.max = NULL, type = c("correlation", "covariance"),
    plot = TRUE, na.action = na.fail, ...)

## S3 method for class 'acf':
x[i, j]
```

Arguments

`x, y`	a univariate or multivariate (not `ccf`) numeric time series object or a numeric vector or matrix, or an `"acf"` object.
`lag.max`	maximum lag at which to calculate the acf. Default is $10\log_{10}(N/m)$ where N is the number of observations and m the number of series. Will be automatically limited to one less than the number of observations in the series.
`type`	character string giving the type of acf to be computed. Allowed values are `"correlation"` (the default), `"covariance"` or `"partial"`.
`plot`	logical. If `TRUE` (the default) the acf is plotted.
`na.action`	function to be called to handle missing values. `na.pass` can be used.
`demean`	logical. Should the covariances be about the sample means?
`...`	further arguments to be passed to `plot.acf`.
`i`	a set of lags (time differences) to retain.
`j`	a set of series (names or numbers) to retain.

Details

For `type = "correlation"` and `"covariance"`, the estimates are based on the sample covariance. (The lag 0 autocorrelation is fixed at 1 by convention.)

By default, no missing values are allowed. If the `na.action` function passes through missing values (as `na.pass` does), the covariances are computed from the complete cases. This means that the estimate computed may well not be a valid autocorrelation sequence, and may contain missing values. Missing values are not allowed when computing the PACF of a multivariate time series.

The partial correlation coefficient is estimated by fitting autoregressive models of successively higher orders up to `lag.max`.

The generic function `plot` has a method for objects of class `"acf"`.

The lag is returned and plotted in units of time, and not numbers of observations.

There are `print` and subsetting methods for objects of class `"acf"`.

Value

An object of class `"acf"`, which is a list with the following elements:

`lag`	A three dimensional array containing the lags at which the acf is estimated.
`acf`	An array with the same dimensions as `lag` containing the estimated acf.
`type`	The type of correlation (same as the `type` argument).
`n.used`	The number of observations in the time series.
`series`	The name of the series `x`.
`snames`	The series names for a multivariate time series.

The lag `k` value returned by `ccf(x,y)` estimates the correlation between `x[t+k]` and `y[t]`.

The result is returned invisibly if `plot` is `TRUE`.

Author(s)

Original: Paul Gilbert, Martyn Plummer. Extensive modifications and univariate case of `pacf` by B.D. Ripley.

See Also

plot.acf, ARMAacf for the exact autocorrelations of a given ARMA process.

Examples

```
## Examples from Venables & Ripley
acf(lh)
acf(lh, type = "covariance")
pacf(lh)

acf(ldeaths)
acf(ldeaths, ci.type = "ma")
acf(ts.union(mdeaths, fdeaths))
ccf(mdeaths, fdeaths, ylab="cross-correlation") # just the cross-correlations.

presidents # contains missing values
acf(presidents, na.action = na.pass)
pacf(presidents, na.action = na.pass)
```

---

acf2AR *Compute an AR Process Exactly Fitting an ACF*

Description

Compute an AR process exactly fitting an autocorrelation function.

Usage

```
acf2AR(acf)
```

Arguments

acf	An autocorrelation or autocovariance sequence.

Value

A matrix, with one row for the computed AR(p) coefficients for `1 <= p <= length(acf)`.

See Also

ARMAacf, ar.yw which does this from an empirical ACF.

Examples

```
(Acf <- ARMAacf(c(0.6, 0.3, -0.2)))
acf2AR(Acf)
```

`add1` *Add or Drop All Possible Single Terms to a Model*

Description

Compute all the single terms in the `scope` argument that can be added to or dropped from the model, fit those models and compute a table of the changes in fit.

Usage

```
add1(object, scope, ...)

## Default S3 method:
add1(object, scope, scale = 0, test = c("none", "Chisq"),
     k = 2, trace = FALSE, ...)

## S3 method for class 'lm':
add1(object, scope, scale = 0, test = c("none", "Chisq", "F"),
     x = NULL, k = 2, ...)

## S3 method for class 'glm':
add1(object, scope, scale = 0, test = c("none", "Chisq", "F"),
     x = NULL, k = 2, ...)

drop1(object, scope, ...)

## Default S3 method:
drop1(object, scope, scale = 0, test = c("none", "Chisq"),
      k = 2, trace = FALSE, ...)

## S3 method for class 'lm':
drop1(object, scope, scale = 0, all.cols = TRUE,
      test = c("none", "Chisq", "F"), k = 2, ...)

## S3 method for class 'glm':
drop1(object, scope, scale = 0, test = c("none", "Chisq", "F"),
      k = 2, ...)
```

Arguments

`object`	a fitted model object.
`scope`	a formula giving the terms to be considered for adding or dropping.
`scale`	an estimate of the residual mean square to be used in computing C_p. Ignored if `0` or `NULL`.
`test`	should the results include a test statistic relative to the original model? The F test is only appropriate for `lm` and `aov` models or perhaps for `glm` fits with estimated dispersion. The χ^2 test can be an exact test (`lm` models with known scale) or a likelihood-ratio test or a test of the reduction in scaled deviance depending on the method.
`k`	the penalty constant in AIC / C_p.

trace	if TRUE, print out progress reports.
x	a model matrix containing columns for the fitted model and all terms in the upper scope. Useful if add1 is to be called repeatedly. **Warning:** no checks are done on its validity.
all.cols	(Provided for compatibility with S.) Logical to specify whether all columns of the design matrix should be used. If FALSE then non-estimable columns are dropped, but the result is not usually statistically meaningful.
...	further arguments passed to or from other methods.

Details

For drop1 methods, a missing scope is taken to be all terms in the model. The hierarchy is respected when considering terms to be added or dropped: all main effects contained in a second-order interaction must remain, and so on.

In a scope formula . means 'what is already there'.

The methods for lm and glm are more efficient in that they do not recompute the model matrix and call the fit methods directly.

The default output table gives AIC, defined as minus twice log likelihood plus $2p$ where p is the rank of the model (the number of effective parameters). This is only defined up to an additive constant (like log-likelihoods). For linear Gaussian models with fixed scale, the constant is chosen to give Mallows' C_p, $RSS/scale + 2p - n$. Where C_p is used, the column is labelled as Cp rather than AIC.

The F tests for the "glm" methods are based on analysis of deviance tests, so if the dispersion is estimated it is based on the residual deviance, unlike the F tests of anova.glm.

Value

An object of class "anova" summarizing the differences in fit between the models.

Warning

The model fitting must apply the models to the same dataset. Most methods will attempt to use a subset of the data with no missing values for any of the variables if na.action=na.omit, but this may give biased results. Only use these functions with data containing missing values with great care.

Note

These are not fully equivalent to the functions in S. There is no keep argument, and the methods used are not quite so computationally efficient.

Their authors' definitions of Mallows' C_p and Akaike's AIC are used, not those of the authors of the models chapter of S.

Author(s)

The design was inspired by the S functions of the same names described in Chambers (1992).

References

Chambers, J. M. (1992) *Linear models.* Chapter 4 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

step, aov, lm, extractAIC, anova

Examples

```
example(step)#-> swiss
add1(lm1, ~ I(Education^2) + .^2)
drop1(lm1, test="F")  # So called 'type II' anova

example(glm)
drop1(glm.D93, test="Chisq")
drop1(glm.D93, test="F")
```

---

addmargins *Puts Arbitrary Margins on Multidimensional Tables or Arrays*

Description

For a given table one can specify which of the classifying factors to expand by one or more levels to hold margins to be calculated. One may for example form sums and means over the first dimension and medians over the second. The resulting table will then have two extra levels for the first dimension and one extra level for the second. The default is to sum over all margins in the table. Other possibilities may give results that depend on the order in which the margins are computed. This is flagged in the printed output from the function.

Usage

```
addmargins(A, margin = 1:length(dim(A)), FUN = sum, quiet = FALSE)
```

Arguments

A	table or array. The function uses the presence of the "dim" and "dimnames" attributes of A.
margin	vector of dimensions over which to form margins. Margins are formed in the order in which dimensions are specified in margin.
FUN	list of the same length as margin, each element of the list being either a function or a list of functions. Names of the list elements will appear as levels in dimnames of the result. Unnamed list elements will have names constructed: the name of a function or a constructed name based on the position in the table.
quiet	logical which suppresses the message telling the order in which the margins were computed.

Details

If the functions used to form margins are not commutative the result depends on the order in which margins are computed. Annotation of margins is done via naming the FUN list.

Value

A table or array with the same number of dimensions as A, but with extra levels of the dimensions mentioned in margin. The number of levels added to each dimension is the length of the entries in FUN. A message with the order of computation of margins is printed.

Author(s)

Bendix Carstensen, Steno Diabetes Center & Department of Biostatistics, University of Copenhagen, http://www.biostat.ku.dk/~bxc, autumn 2003. Margin naming enhanced by Duncan Murdoch.

See Also

table, ftable, margin.table.

Examples

```
Aye <- sample(c("Yes", "Si", "Oui"), 177, replace = TRUE)
Bee <- sample(c("Hum", "Buzz"), 177, replace = TRUE)
Sea <- sample(c("White", "Black", "Red", "Dead"), 177, replace = TRUE)
(A <- table(Aye, Bee, Sea))
addmargins(A)

ftable(A)
ftable(addmargins(A))

# Non-commutative functions - note differences between resulting tables:
ftable(addmargins(A, c(1,3),
       FUN = list(Sum = sum, list(Min = min, Max = max))))
ftable(addmargins(A, c(3,1),
       FUN = list(list(Min = min, Max = max), Sum = sum)))

# Weird function needed to return the N when computing percentages
sqsm <- function(x) sum(x)^2/100
B <- table(Sea, Bee)
round(sweep(addmargins(B, 1, list(list(All = sum, N = sqsm))), 2,
            apply(B, 2, sum)/100, "/"), 1)
round(sweep(addmargins(B, 2, list(list(All = sum, N = sqsm))), 1,
            apply(B, 1, sum)/100, "/"), 1)

# A total over Bee requires formation of the Bee-margin first:
mB <-  addmargins(B, 2, FUN = list(list(Total = sum)))
round(ftable(sweep(addmargins(mB, 1, list(list(All = sum, N = sqsm))), 2,
                   apply(mB,2,sum)/100, "/")), 1)

## Zero.Printing table+margins:
set.seed(1)
x <- sample( 1:7, 20, replace=TRUE)
y <- sample( 1:7, 20, replace=TRUE)
tx <- addmargins( table(x, y) )
print(tx, zero.print = ".")
```

---

aggregate *Compute Summary Statistics of Data Subsets*

Description

Splits the data into subsets, computes summary statistics for each, and returns the result in a convenient form.

Usage

```
aggregate(x, ...)

## Default S3 method:
aggregate(x, ...)

## S3 method for class 'data.frame':
aggregate(x, by, FUN, ...)

## S3 method for class 'ts':
aggregate(x, nfrequency = 1, FUN = sum, ndeltat = 1,
          ts.eps = getOption("ts.eps"), ...)
```

Arguments

x	an R object.
by	a list of grouping elements, each as long as the variables in x. Names for the grouping variables are provided if they are not given. The elements of the list will be coerced to factors (if they are not already factors).
FUN	a scalar function to compute the summary statistics which can be applied to all data subsets.
nfrequency	new number of observations per unit of time; must be a divisor of the frequency of x.
ndeltat	new fraction of the sampling period between successive observations; must be a divisor of the sampling interval of x.
ts.eps	tolerance used to decide if nfrequency is a sub-multiple of the original frequency.
...	further arguments passed to or used by methods.

Details

aggregate is a generic function with methods for data frames and time series.

The default method aggregate.default uses the time series method if x is a time series, and otherwise coerces x to a data frame and calls the data frame method.

aggregate.data.frame is the data frame method. If x is not a data frame, it is coerced to one. Then, each of the variables (columns) in x is split into subsets of cases (rows) of identical combinations of the components of by, and FUN is applied to each such subset with further arguments in ... passed to it. (I.e., tapply(VAR, by, FUN, ..., simplify = FALSE) is done for each variable VAR in x, conveniently wrapped into one call to lapply().) Empty subsets are removed, and the result is reformatted into a data frame containing the variables in by and x. The ones arising from by contain the unique combinations of grouping values used for determining the subsets, and the ones arising from x the corresponding summary statistics for the subset of the respective variables in x.

aggregate.ts is the time series method. If x is not a time series, it is coerced to one. Then, the variables in x are split into appropriate blocks of length frequency(x) / nfrequency, and FUN is applied to each such block, with further (named) arguments in ... passed to it. The result returned is a time series with frequency nfrequency holding the aggregated values.

Author(s)

Kurt Hornik

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

apply, lapply, tapply.

Examples

```
## Compute the averages for the variables in 'state.x77', grouped
## according to the region (Northeast, South, North Central, West) that
## each state belongs to.
aggregate(state.x77, list(Region = state.region), mean)

## Compute the averages according to region and the occurrence of more
## than 130 days of frost.
aggregate(state.x77,
          list(Region = state.region,
               Cold = state.x77[,"Frost"] > 130),
          mean)
## (Note that no state in 'South' is THAT cold.)

## Compute the average annual approval ratings for American presidents.
aggregate(presidents, nf = 1, FUN = mean)
## Give the summer less weight.
aggregate(presidents, nf = 1, FUN = weighted.mean, w = c(1, 1, 0.5, 1))
```

AIC *Akaike's An Information Criterion*

Description

Generic function calculating the Akaike information criterion for one or several fitted model objects for which a log-likelihood value can be obtained, according to the formula -2log-likelihood $+$ kn_{par}, where n_{par} represents the number of parameters in the fitted model, and $k = 2$ for the usual AIC, or $k = \log(n)$ (n the number of observations) for the so-called BIC or SBC (Schwarz's Bayesian criterion).

Usage

```
AIC(object, ..., k = 2)
```

Arguments

object	a fitted model object, for which there exists a logLik method to extract the corresponding log-likelihood, or an object inheriting from class logLik.
...	optionally more fitted model objects.
k	numeric, the "penalty" per parameter to be used; the default k = 2 is the classical AIC.

Details

The default method for AIC, AIC.default() entirely relies on the existence of a logLik method computing the log-likelihood for the given class.

When comparing fitted objects, the smaller the AIC, the better the fit.

The log-likelihood and hence the AIC is only defined up to an additive constant. Different constants have conventionally be used for different purposes and so extractAIC and AIC may give different values (and do for models of class "lm": see the help for extractAIC).

Value

If just one object is provided, returns a numeric value with the corresponding AIC (or BIC, or ..., depending on k); if more than one object are provided, returns a data.frame with rows corresponding to the objects and columns representing the number of parameters in the model (df) and the AIC.

Author(s)

Jose Pinheiro and Douglas Bates

References

Sakamoto, Y., Ishiguro, M., and Kitagawa G. (1986). *Akaike Information Criterion Statistics*. D. Reidel Publishing Company.

See Also

extractAIC, logLik.

Examples

```
lm1 <- lm(Fertility ~ . , data = swiss)
AIC(lm1)
stopifnot(all.equal(AIC(lm1),
                    AIC(logLik(lm1))))
## a version of BIC or Schwarz' BC :
AIC(lm1, k = log(nrow(swiss)))
```

---

alias *Find Aliases (Dependencies) in a Model*

Description

Find aliases (linearly dependent terms) in a linear model specified by a formula.

Usage

```
alias(object, ...)

## S3 method for class 'formula':
alias(object, data, ...)

## S3 method for class 'lm':
alias(object, complete = TRUE, partial = FALSE,
      partial.pattern = FALSE, ...)
```

Arguments

`object`	A fitted model object, for example from `lm` or `aov`, or a formula for `alias.formula`.
`data`	Optionally, a data frame to search for the objects in the formula.
`complete`	Should information on complete aliasing be included?
`partial`	Should information on partial aliasing be included?
`partial.pattern`	Should partial aliasing be presented in a schematic way? If this is done, the results are presented in a more compact way, usually giving the deciles of the coefficients.
`...`	further arguments passed to or from other methods.

Details

Although the main method is for class `"lm"`, `alias` is most useful for experimental designs and so is used with fits from `aov`. Complete aliasing refers to effects in linear models that cannot be estimated independently of the terms which occur earlier in the model and so have their coefficients omitted from the fit. Partial aliasing refers to effects that can be estimated less precisely because of correlations induced by the design.

Value

A list (of `class "listof"`) containing components

`Model`	Description of the model; usually the formula.
`Complete`	A matrix with columns corresponding to effects that are linearly dependent on the rows.
`Partial`	The correlations of the estimable effects, with a zero diagonal. An object of class `"mtable"` which has its own `print` method.

Note

The aliasing pattern may depend on the contrasts in use: Helmert contrasts are probably most useful.

The defaults are different from those in S.

Author(s)

The design was inspired by the S function of the same name described in Chambers *et al.* (1992).

References

Chambers, J. M., Freeny, A and Heiberger, R. M. (1992) *Analysis of variance; designed experiments.* Chapter 5 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

Examples

```
## From Venables and Ripley (2002) p.165.
data(npk, package="MASS")

op <- options(contrasts=c("contr.helmert", "contr.poly"))
npk.aov <- aov(yield ~ block + N*P*K, npk)
alias(npk.aov)
options(op)# reset
```

---

anova *Anova Tables*

Description

Compute analysis of variance (or deviance) tables for one or more fitted model objects.

Usage

```
anova(object, ...)
```

Arguments

`object`	an object containing the results returned by a model fitting function (e.g., `lm` or `glm`).
`...`	additional objects of the same type.

Value

This (generic) function returns an object of class `anova`. These objects represent analysis-of-variance and analysis-of-deviance tables. When given a single argument it produces a table which tests whether the model terms are significant.

When given a sequence of objects, `anova` tests the models against one another in the order specified.

The print method for `anova` objects prints tables in a "pretty" form.

Warning

The comparison between two or more models will only be valid if they are fitted to the same dataset. This may be a problem if there are missing values and R's default of `na.action = na.omit` is used.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*, Wadsworth & Brooks/Cole.

See Also

coefficients, effects, fitted.values, residuals, summary, drop1, add1.

---

anova.glm *Analysis of Deviance for Generalized Linear Model Fits*

Description

Compute an analysis of deviance table for one or more generalized linear model fits.

Usage

```
## S3 method for class 'glm':
anova(object, ..., dispersion = NULL, test = NULL)
```

Arguments

object, ...	objects of class glm, typically the result of a call to glm, or a list of objects for the "glmlist" method.
dispersion	the dispersion parameter for the fitting family. By default it is obtained from the object(s).
test	a character string, (partially) matching one of "Chisq", "F" or "Cp". See stat.anova.

Details

Specifying a single object gives a sequential analysis of deviance table for that fit. That is, the reductions in the residual deviance as each term of the formula is added in turn are given in as the rows of a table, plus the residual deviances themselves.

If more than one object is specified, the table has a row for the residual degrees of freedom and deviance for each model. For all but the first model, the change in degrees of freedom and deviance is also given. (This only makes statistical sense if the models are nested.) It is conventional to list the models from smallest to largest, but this is up to the user.

The table will optionally contain test statistics (and P values) comparing the reduction in deviance for the row to the residuals. For models with known dispersion (e.g., binomial and Poisson fits) the chi-squared test is most appropriate, and for those with dispersion estimated by moments (e.g., gaussian, quasibinomial and quasipoisson fits) the F test is most appropriate. Mallows' C_p statistic is the residual deviance plus twice the estimate of σ^2 times the residual degrees of freedom, which is closely related to AIC (and a multiple of it if the dispersion is known).

The dispersion estimate will be taken from the largest model, using the value returned by summary.glm. As this will in most cases use a Chisquared-based estimate, the F tests are not based on the residual deviance in the analysis of deviance table shown.

Value

An object of class "anova" inheriting from class "data.frame".

Warning

The comparison between two or more models by `anova` or `anova.glmlist` will only be valid if they are fitted to the same dataset. This may be a problem if there are missing values and R's default of `na.action = na.omit` is used, and `anova.glmlist` will detect this with an error.

References

Hastie, T. J. and Pregibon, D. (1992) *Generalized linear models.* Chapter 6 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`glm`, `anova`.

`drop1` for so-called 'type II' anova where each term is dropped one at a time respecting their hierarchy.

Examples

```
## --- Continuing the Example from  '?glm':

anova(glm.D93)
anova(glm.D93, test = "Cp")
anova(glm.D93, test = "Chisq")
```

`anova.lm` *ANOVA for Linear Model Fits*

Description

Compute an analysis of variance table for one or more linear model fits.

Usage

```
## S3 method for class 'lm':
anova(object, ...)

anova.lmlist(object, ..., scale = 0, test = "F")
```

Arguments

`object, ...`	objects of class `lm`, usually, a result of a call to `lm`.
`test`	a character string specifying the test statistic to be used. Can be one of `"F"`, `"Chisq"` or `"Cp"`, with partial matching allowed, or `NULL` for no test.
`scale`	numeric. An estimate of the noise variance σ^2. If zero this will be estimated from the largest model considered.

Details

Specifying a single object gives a sequential analysis of variance table for that fit. That is, the reductions in the residual sum of squares as each term of the formula is added in turn are given in as the rows of a table, plus the residual sum of squares.

The table will contain F statistics (and P values) comparing the mean square for the row to the residual mean square.

If more than one object is specified, the table has a row for the residual degrees of freedom and sum of squares for each model. For all but the first model, the change in degrees of freedom and sum of squares is also given. (This only make statistical sense if the models are nested.) It is conventional to list the models from smallest to largest, but this is up to the user.

Optionally the table can include test statistics. Normally the F statistic is most appropriate, which compares the mean square for a row to the residual sum of squares for the largest model considered. If `scale` is specified chi-squared tests can be used. Mallows' C_p statistic is the residual sum of squares plus twice the estimate of σ^2 times the residual degrees of freedom.

Value

An object of class `"anova"` inheriting from class `"data.frame"`.

Warning

The comparison between two or more models will only be valid if they are fitted to the same dataset. This may be a problem if there are missing values and R's default of `na.action = na.omit` is used, and `anova.lmlist` will detect this with an error.

Note

Versions of R prior to 1.2.0 based F tests on pairwise comparisons, and this behaviour can still be obtained by a direct call to `anovalist.lm`.

References

Chambers, J. M. (1992) *Linear models.* Chapter 4 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

The model fitting function `lm`, `anova`.

`drop1` for so-called 'type II' anova where each term is dropped one at a time respecting their hierarchy.

Examples

```
## sequential table
fit <- lm(sr ~ ., data = LifeCycleSavings)
anova(fit)

## same effect via separate models
fit0 <- lm(sr ~ 1, data = LifeCycleSavings)
fit1 <- update(fit0, . ~ . + pop15)
fit2 <- update(fit1, . ~ . + pop75)
fit3 <- update(fit2, . ~ . + dpi)
fit4 <- update(fit3, . ~ . + ddpi)
```

```
anova(fit0, fit1, fit2, fit3, fit4, test="F")

anova(fit4, fit2, fit0, test="F") # unconventional order
```

`anova.mlm` *Comparisons between Multivariate Linear Models*

Description

Compute a (generalized) analysis of variance table for one or more multivariate linear models.

Usage

```
## S3 method for class 'mlm':
anova(object, ...,
      test = c("Pillai", "Wilks", "Hotelling-Lawley", "Roy", "Spherical"),
      Sigma = diag(nrow = p),
      T = Thin.row(proj(M) - proj(X)), M = diag(nrow = p), X = ~0,
      idata = data.frame(index = seq_len(p)))
```

Arguments

`object`	an object of class `"mlm"`.
`...`	further objects of class `"mlm"`.
`test`	choice of test statistic (see below).
`Sigma`	(only relevant if `test == "Spherical"`). Covariance matrix assumed proportional to `Sigma`.
`T`	transformation matrix. By default computed from `M` and `X`.
`M`	formula or matrix describing the outer projection (see below).
`X`	formula or matrix describing the inner projection (see below).
`idata`	data frame describing intra-block design.

Details

The `anova.mlm` method uses either a multivariate test statistic for the summary table, or a test based on sphericity assumptions (i.e. that the covariance is proportional to a given matrix).

For the multivariate test, Wilks' statistic is most popular in the literature, but the default Pillai–Bartlett statistic is recommended by Hand and Taylor (1987). See `summary.manova` for futher details.

For the `"Spherical"` test, proportionality is usually with the identity matrix but a different matrix can be specified using `Sigma`). Corrections for asphericity known as the Greenhouse–Geisser, respectively Huynh–Feldt, epsilons are given and adjusted F tests are performed.

It is common to transform the observations prior to testing. This typically involves transformation to intra-block differences, but more complicated within-block designs can be encountered, making more elaborate transformations necessary. A transformation matrix `T` can be given directly or specified as the difference between two projections onto the spaces spanned by `M` and `X`, which in turn can be given as matrices or as model formulas with respect to `idata` (the tests will be invariant to parametrization of the quotient space `M/X`).

As with `anova.lm`, all test statistics use the SSD matrix from the largest model considered as the (generalized) denominator.

Contrary to other `anova` methods, the intercept is not excluded from the display in the single-model case. When contrast transformations are involved, it often makes good sense to test for a zero intercept.

Value

An object of class `"anova"` inheriting from class `"data.frame"`

Note

The Huynh–Feldt epsilon differs from that calculated by SAS (as of v. 8.2) except when the DF is equal to the number of observations minus one. This is believed to be a bug in SAS, not in R.

References

Hand, D. J. and Taylor, C. C. (1987) *Multivariate Analysis of Variance and Repeated Measures.* Chapman and Hall.

See Also

`summary.manova`

Examples

```
example(SSD) # Brings in the mlmfit and reacttime objects

mlmfit0 <- update(mlmfit, ~0)

### Traditional tests of intrasubj. contrasts
## Using MANOVA techniques on contrasts:
anova(mlmfit, mlmfit0, X=~1)

## Assuming sphericity
anova(mlmfit, mlmfit0, X=~1, test="Spherical")

### tests using intra-subject 3x2 design
idata <- data.frame(deg=gl(3,1,6,labels=c(0,4,8)),
                    noise=gl(2,3,6,labels=c("A","P")))

anova(mlmfit, mlmfit0, X = ~ deg + noise, idata = idata, test = "Spherical")
anova(mlmfit, mlmfit0, M = ~ deg + noise, X = ~ noise, idata = idata,
          test="Spherical" )
anova(mlmfit, mlmfit0, M = ~ deg + noise, X = ~ deg, idata = idata,
          test="Spherical" )

f <- factor(rep(1:2,5)) # bogus, just for illustration
mlmfit2 <- update(mlmfit, ~f)
anova(mlmfit2, mlmfit, mlmfit0, X=~1, test="Spherical")
anova(mlmfit2, X=~1, test="Spherical") # one-model form, eqiv. to previous

### There seems to be a strong interaction in these data
plot(colMeans(reacttime))
```

`ansari.test` *Ansari-Bradley Test*

Description

Performs the Ansari-Bradley two-sample test for a difference in scale parameters.

Usage

```
ansari.test(x, ...)

## Default S3 method:
ansari.test(x, y, alternative = c("two.sided", "less", "greater"),
            exact = NULL, conf.int = FALSE, conf.level = 0.95, ...)

## S3 method for class 'formula':
ansari.test(formula, data, subset, na.action, ...)
```

Arguments

`x`	numeric vector of data values.
`y`	numeric vector of data values.
`alternative`	indicates the alternative hypothesis and must be one of `"two.sided"`, `"greater"` or `"less"`. You can specify just the initial letter.
`exact`	a logical indicating whether an exact p-value should be computed.
`conf.int`	a logical,indicating whether a confidence interval should be computed.
`conf.level`	confidence level of the interval.
`formula`	a formula of the form `lhs ~ rhs` where `lhs` is a numeric variable giving the data values and `rhs` a factor with two levels giving the corresponding groups.
`data`	an optional matrix or data frame (or similar: see `model.frame`) containing the variables in the formula `formula`. By default the variables are taken from `environment(formula)`.
`subset`	an optional vector specifying a subset of observations to be used.
`na.action`	a function which indicates what should happen when the data contain `NA`s. Defaults to `getOption("na.action")`.
`...`	further arguments to be passed to or from methods.

Details

Suppose that `x` and `y` are independent samples from distributions with densities $f((t-m)/s)/s$ and $f(t-m)$, respectively, where m is an unknown nuisance parameter and s, the ratio of scales, is the parameter of interest. The Ansari-Bradley test is used for testing the null that s equals 1, the two-sided alternative being that $s \neq 1$ (the distributions differ only in variance), and the one-sided alternatives being $s > 1$ (the distribution underlying `x` has a larger variance, `"greater"`) or $s < 1$ (`"less"`).

By default (if `exact` is not specified), an exact p-value is computed if both samples contain less than 50 finite values and there are no ties. Otherwise, a normal approximation is used.

Optionally, a nonparametric confidence interval and an estimator for s are computed. If exact p-values are available, an exact confidence interval is obtained by the algorithm described in Bauer (1972), and the Hodges-Lehmann estimator is employed. Otherwise, the returned confidence interval and point estimate are based on normal approximations.

Note that mid-ranks are used in the case of ties rather than average scores as employed in Hollander & Wolfe (1973). See, e.g., Hajek, Sidak and Sen (1999), pages 131ff, for more information.

Value

A list with class `"htest"` containing the following components:

`statistic`	the value of the Ansari-Bradley test statistic.
`p.value`	the p-value of the test.
`null.value`	the ratio of scales s under the null, 1.
`alternative`	a character string describing the alternative hypothesis.
`method`	the string `"Ansari-Bradley test"`.
`data.name`	a character string giving the names of the data.
`conf.int`	a confidence interval for the scale parameter. (Only present if argument `conf.int = TRUE`.)
`estimate`	an estimate of the ratio of scales. (Only present if argument `conf.int = TRUE`.)

Note

To compare results of the Ansari-Bradley test to those of the F test to compare two variances (under the assumption of normality), observe that s is the ratio of scales and hence s^2 is the ratio of variances (provided they exist), whereas for the F test the ratio of variances itself is the parameter of interest. In particular, confidence intervals are for s in the Ansari-Bradley test but for s^2 in the F test.

References

David F. Bauer (1972), Constructing confidence sets using rank statistics. *Journal of the American Statistical Association* **67**, 687–690.

Jaroslav Hajek, Zbynek Sidak & Pranab K. Sen (1999), *Theory of Rank Tests*. San Diego, London: Academic Press.

Myles Hollander & Douglas A. Wolfe (1973), *Nonparametric statistical inference*. New York: John Wiley & Sons. Pages 83–92.

See Also

`fligner.test` for a rank-based (nonparametric) k-sample test for homogeneity of variances; `mood.test` for another rank-based two-sample test for a difference in scale parameters; `var.test` and `bartlett.test` for parametric tests for the homogeneity in variance.

`ansari_test` in package **coin** for exact and approximate *conditional* p-values for the Ansari-Bradley test, as well as different methods for handling ties.

Examples

```
## Hollander & Wolfe (1973, p. 86f):
## Serum iron determination using Hyland control sera
ramsay <- c(111, 107, 100, 99, 102, 106, 109, 108, 104, 99,
            101, 96, 97, 102, 107, 113, 116, 113, 110, 98)
jung.parekh <- c(107, 108, 106, 98, 105, 103, 110, 105, 104,
            100, 96, 108, 103, 104, 114, 114, 113, 108, 106, 99)
ansari.test(ramsay, jung.parekh)

ansari.test(rnorm(10), rnorm(10, 0, 2), conf.int = TRUE)
```

---

aov *Fit an Analysis of Variance Model*

Description

Fit an analysis of variance model by a call to `lm` for each stratum.

Usage

```
aov(formula, data = NULL, projections = FALSE, qr = TRUE,
    contrasts = NULL, ...)
```

Arguments

`formula`	A formula specifying the model.
`data`	A data frame in which the variables specified in the formula will be found. If missing, the variables are searched for in the standard way.
`projections`	Logical flag: should the projections be returned?
`qr`	Logical flag: should the QR decomposition be returned?
`contrasts`	A list of contrasts to be used for some of the factors in the formula. These are not used for any `Error` term, and supplying contrasts for factors only in the `Error` term will give a warning.
`...`	Arguments to be passed to `lm`, such as `subset` or `na.action`.

Details

This provides a wrapper to `lm` for fitting linear models to balanced or unbalanced experimental designs.

The main difference from `lm` is in the way `print`, `summary` and so on handle the fit: this is expressed in the traditional language of the analysis of variance rather than that of linear models.

If the formula contains a single `Error` term, this is used to specify error strata, and appropriate models are fitted within each error stratum.

The formula can specify multiple responses.

Weights can be specified by a `weights` argument, but should not be used with an `Error` term, and are incompletely supported (e.g., not by `model.tables`).

Value

An object of class `c("aov", "lm")` or for multiple responses of class `c("maov", "aov", "mlm", "lm")` or for multiple error strata of class `"aovlist"`. There are `print` and `summary` methods available for these.

Note

`aov` is designed for balanced designs, and the results can be hard to interpret without balance: beware that missing values in the response(s) will likely lose the balance. If there are two or more error strata, the methods used are statistically inefficient without balance, and it may be better to use `lme`.

Balance can be checked with the `replications` function.

The default 'contrasts' in R are not orthogonal contrasts, and `aov` and its helper functions will work better with such contrasts: see the examples for how to select these.

Author(s)

The design was inspired by the S function of the same name described in Chambers *et al.* (1992).

References

Chambers, J. M., Freeny, A and Heiberger, R. M. (1992) *Analysis of variance; designed experiments.* Chapter 5 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`lm`, `summary.aov`, `replications`, `alias`, `proj`, `model.tables`, `TukeyHSD`

Examples

```
## From Venables and Ripley (2002) p.165.
data(npk, package="MASS")

## Set orthogonal contrasts.
op <- options(contrasts=c("contr.helmert", "contr.poly"))
( npk.aov <- aov(yield ~ block + N*P*K, npk) )
summary(npk.aov)
coefficients(npk.aov)

## to show the effects of re-ordering terms contrast the two fits
aov(yield ~ block + N * P + K, npk)
aov(terms(yield ~ block + N * P + K, keep.order=TRUE), npk)

## as a test, not particularly sensible statistically
npk.aovE <- aov(yield ~  N*P*K + Error(block), npk)
npk.aovE
summary(npk.aovE)
options(op)# reset to previous
```

`approxfun` *Interpolation Functions*

Description

Return a list of points which linearly interpolate given data points, or a function performing the linear (or constant) interpolation.

Usage

```
approx   (x, y = NULL, xout, method="linear", n=50,
          yleft, yright, rule = 1, f = 0, ties = mean)

approxfun(x, y = NULL,       method="linear",
          yleft, yright, rule = 1, f = 0, ties = mean)
```

Arguments

`x, y`	vectors giving the coordinates of the points to be interpolated. Alternatively a single plotting structure can be specified: see `xy.coords`.
`xout`	an optional set of values specifying where interpolation is to take place.
`method`	specifies the interpolation method to be used. Choices are `"linear"` or `"constant"`.
`n`	If `xout` is not specified, interpolation takes place at `n` equally spaced points spanning the interval [`min(x)`, `max(x)`].
`yleft`	the value to be returned when input `x` values are less than `min(x)`. The default is defined by the value of `rule` given below.
`yright`	the value to be returned when input `x` values are greater than `max(x)`. The default is defined by the value of `rule` given below.
`rule`	an integer describing how interpolation is to take place outside the interval [`min(x)`, `max(x)`]. If `rule` is `1` then `NA`s are returned for such points and if it is `2`, the value at the closest data extreme is used.
`f`	For `method="constant"` a number between 0 and 1 inclusive, indicating a compromise between left- and right-continuous step functions. If `y0` and `y1` are the values to the left and right of the point then the value is `y0*(1-f)+y1*f` so that `f=0` is right-continuous and `f=1` is left-continuous.
`ties`	Handling of tied `x` values. Either a function with a single vector argument returning a single number result or the string `"ordered"`.

Details

The inputs can contain missing values which are deleted, so at least two complete `(x, y)` pairs are required (for `method = "linear"`, one otherwise). If there are duplicated (tied) `x` values and `ties` is a function it is applied to the `y` values for each distinct `x` value. Useful functions in this context include `mean`, `min`, and `max`. If `ties="ordered"` the `x` values are assumed to be already ordered. The first `y` value will be used for interpolation to the left and the last one for interpolation to the right.

Value

`approx` returns a list with components `x` and `y`, containing `n` coordinates which interpolate the given data points according to the `method` (and `rule`) desired.

The function `approxfun` returns a function performing (linear or constant) interpolation of the given data points. For a given set of `x` values, this function will return the corresponding interpolated values. This is often more useful than `approx`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`spline` and `splinefun` for spline interpolation.

Examples

```
x <- 1:10
y <- rnorm(10)
par(mfrow = c(2,1))
plot(x, y, main = "approx(.) and approxfun(.)")
points(approx(x, y), col = 2, pch = "*")
points(approx(x, y, method = "constant"), col = 4, pch = "*")

f <- approxfun(x, y)
curve(f(x), 0, 10, col = "green")
points(x, y)
is.function(fc <- approxfun(x, y, method = "const")) # TRUE
curve(fc(x), 0, 10, col = "darkblue", add = TRUE)

## Show treatment of 'ties' :

x <- c(2,2:4,4,4,5,5,7,7,7)
y <- c(1:6, 5:4, 3:1)
approx(x,y, xout=x)$y # warning
(ay <- approx(x,y, xout=x, ties = "ordered")$y)
stopifnot(ay == c(2,2,3,6,6,6,4,4,1,1,1))
approx(x,y, xout=x, ties = min)$y
approx(x,y, xout=x, ties = max)$y
```

---

ar — *Fit Autoregressive Models to Time Series*

Description

Fit an autoregressive time series model to the data, by default selecting the complexity by AIC.

Usage

```
ar(x, aic = TRUE, order.max = NULL,
   method=c("yule-walker", "burg", "ols", "mle", "yw"),
   na.action, series, ...)

ar.burg(x, ...)
## Default S3 method:
ar.burg(x, aic = TRUE, order.max = NULL,
        na.action = na.fail, demean = TRUE, series,
        var.method = 1, ...)
## S3 method for class 'mts':
ar.burg(x, aic = TRUE, order.max = NULL,
        na.action = na.fail, demean = TRUE, series,
        var.method = 1, ...)

ar.yw(x, ...)
## Default S3 method:
ar.yw(x, aic = TRUE, order.max = NULL,
      na.action = na.fail, demean = TRUE, series, ...)
## S3 method for class 'mts':
ar.yw(x, aic = TRUE, order.max = NULL,
      na.action = na.fail, demean = TRUE, series,
      var.method = 1, ...)

ar.mle(x, aic = TRUE, order.max = NULL, na.action = na.fail,
       demean = TRUE, series, ...)

## S3 method for class 'ar':
predict(object, newdata, n.ahead = 1, se.fit = TRUE, ...)
```

Arguments

`x`	A univariate or multivariate time series.
`aic`	Logical flag. If `TRUE` then the Akaike Information Criterion is used to choose the order of the autoregressive model. If `FALSE`, the model of order `order.max` is fitted.
`order.max`	Maximum order (or order) of model to fit. Defaults to $10\log_{10}(N)$ where N is the number of observations except for `method="mle"` where it is the minimum of this quantity and 12.
`method`	Character string giving the method used to fit the model. Must be one of the strings in the default argument (the first few characters are sufficient). Defaults to `"yule-walker"`.
`na.action`	function to be called to handle missing values.
`demean`	should a mean be estimated during fitting?
`series`	names for the series. Defaults to `deparse(substitute(x))`.
`var.method`	the method to estimate the innovations variance (see Details).
`...`	additional arguments for specific methods.
`object`	a fit from `ar`.
`newdata`	data to which to apply the prediction.

n.ahead	number of steps ahead at which to predict.
se.fit	logical: return estimated standard errors of the prediction error?

Details

For definiteness, note that the AR coefficients have the sign in

$$x_t - \mu = a_1(x_{t-1} - \mu) + \cdots + a_p(x_{t-p} - \mu) + e_t$$

ar is just a wrapper for the functions ar.yw, ar.burg, ar.ols and ar.mle.

Order selection is done by AIC if aic is true. This is problematic, as of the methods here only ar.mle performs true maximum likelihood estimation. The AIC is computed as if the variance estimate were the MLE, omitting the determinant term from the likelihood. Note that this is not the same as the Gaussian likelihood evaluated at the estimated parameter values. In ar.yw the variance matrix of the innovations is computed from the fitted coefficients and the autocovariance of x.

ar.burg allows two methods to estimate the innovations variance and hence AIC. Method 1 is to use the update given by the Levinson-Durbin recursion (Brockwell and Davis, 1991, (8.2.6) on page 242), and follows S-PLUS. Method 2 is the mean of the sum of squares of the forward and backward prediction errors (as in Brockwell and Davis, 1996, page 145). Percival and Walden (1998) discuss both. In the multivariate case the estimated coefficients will depend (slightly) on the variance estimation method.

Remember that ar includes by default a constant in the model, by removing the overall mean of x before fitting the AR model, or (ar.mle) estimating a constant to subtract.

Value

For ar and its methods a list of class "ar" with the following elements:

order	The order of the fitted model. This is chosen by minimizing the AIC if aic=TRUE, otherwise it is order.max.
ar	Estimated autoregression coefficients for the fitted model.
var.pred	The prediction variance: an estimate of the portion of the variance of the time series that is not explained by the autoregressive model.
x.mean	The estimated mean of the series used in fitting and for use in prediction.
x.intercept	(ar.ols only.) The intercept in the model for x - x.mean.
aic	The value of the aic argument.
n.used	The number of observations in the time series.
order.max	The value of the order.max argument.
partialacf	The estimate of the partial autocorrelation function up to lag order.max.
resid	residuals from the fitted model, conditioning on the first order observations. The first order residuals are set to NA. If x is a time series, so is resid.
method	The value of the method argument.
series	The name(s) of the time series.
frequency	The frequency of the time series.
call	The matched call.
asy.var.coef	(univariate case, order > 0.) The asymptotic-theory variance matrix of the coefficient estimates.

For predict.ar, a time series of predictions, or if se.fit = TRUE, a list with components pred, the predictions, and se, the estimated standard errors. Both components are time series.

Note

Only the univariate case of `ar.mle` is implemented.

Fitting by `method="mle"` to long series can be very slow.

Author(s)

Martyn Plummer. Univariate case of `ar.yw`, `ar.mle` and C code for univariate case of `ar.burg` by B. D. Ripley.

References

Brockwell, P. J. and Davis, R. A. (1991) *Time Series and Forecasting Methods.* Second edition. Springer, New York. Section 11.4.

Brockwell, P. J. and Davis, R. A. (1996) *Introduction to Time Series and Forecasting.* Springer, New York. Sections 5.1 and 7.6.

Percival, D. P. and Walden, A. T. (1998) *Spectral Analysis for Physical Applications.* Cambridge University Press.

Whittle, P. (1963) On the fitting of multivariate autoregressions and the approximate canonical factorization of a spectral density matrix. *Biometrika* **40**, 129–134.

See Also

`ar.ols`, `arima0` for ARMA models; `acf2AR`, for AR construction from the ACF.

Examples

```
ar(lh)
ar(lh, method="burg")
ar(lh, method="ols")
ar(lh, FALSE, 4) # fit ar(4)

(sunspot.ar <- ar(sunspot.year))
predict(sunspot.ar, n.ahead=25)
## try the other methods too

ar(ts.union(BJsales, BJsales.lead))
## Burg is quite different here, as is OLS (see ar.ols)
ar(ts.union(BJsales, BJsales.lead), method="burg")
```

`ar.ols` *Fit Autoregressive Models to Time Series by OLS*

Description

Fit an autoregressive time series model to the data by ordinary least squares, by default selecting the complexity by AIC.

Usage

```
ar.ols(x, aic = TRUE, order.max = NULL, na.action = na.fail,
       demean = TRUE, intercept = demean, series, ...)
```

Arguments

x	A univariate or multivariate time series.
aic	Logical flag. If TRUE then the Akaike Information Criterion is used to choose the order of the autoregressive model. If FALSE, the model of order order.max is fitted.
order.max	Maximum order (or order) of model to fit. Defaults to $10\log_{10}(N)$ where N is the number of observations.
na.action	function to be called to handle missing values.
demean	should the AR model be for x minus its mean?
intercept	should a separate intercept term be fitted?
series	names for the series. Defaults to deparse(substitute(x)).
...	further arguments to be passed to or from methods.

Details

ar.ols fits the general AR model to a possibly non-stationary and/or multivariate system of series x. The resulting unconstrained least squares estimates are consistent, even if some of the series are non-stationary and/or co-integrated. For definiteness, note that the AR coefficients have the sign in

$$x_t - \mu = a_0 + a_1(x_{t-1} - \mu) + \cdots + a_p(x_{t-p} - \mu) + e_t$$

where a_0 is zero unless intercept is true, and μ is the sample mean if demean is true, zero otherwise.

Order selection is done by AIC if aic is true. This is problematic, as ar.ols does not perform true maximum likelihood estimation. The AIC is computed as if the variance estimate (computed from the variance matrix of the residuals) were the MLE, omitting the determinant term from the likelihood. Note that this is not the same as the Gaussian likelihood evaluated at the estimated parameter values.

Some care is needed if intercept is true and demean is false. Only use this is the series are roughly centred on zero. Otherwise the computations may be inaccurate or fail entirely.

Value

A list of class "ar" with the following elements:

order	The order of the fitted model. This is chosen by minimizing the AIC if aic=TRUE, otherwise it is order.max.
ar	Estimated autoregression coefficients for the fitted model.
var.pred	The prediction variance: an estimate of the portion of the variance of the time series that is not explained by the autoregressive model.
x.mean	The estimated mean (or zero if demean is false) of the series used in fitting and for use in prediction.
x.intercept	The intercept in the model for x - x.mean, or zero if intercept is false.
aic	The value of the aic argument.
n.used	The number of observations in the time series.
order.max	The value of the order.max argument.
partialacf	NULL. For compatibility with ar.

`resid`	residuals from the fitted model, conditioning on the first `order` observations. The first `order` residuals are set to `NA`. If `x` is a time series, so is `resid`.
`method`	The character string `"Unconstrained LS"`.
`series`	The name(s) of the time series.
`frequency`	The frequency of the time series.
`call`	The matched call.
`asy.se.coef`	The asymptotic-theory standard errors of the coefficient estimates.

Author(s)

Adrian Trapletti, Brian Ripley.

References

Luetkepohl, H. (1991): *Introduction to Multiple Time Series Analysis.* Springer Verlag, NY, pp. 368–370.

See Also

`ar`

Examples

```
ar(lh, method="burg")
ar.ols(lh)
ar.ols(lh, FALSE, 4) # fit ar(4)

ar.ols(ts.union(BJsales, BJsales.lead))

x <- diff(log(EuStockMarkets))
ar.ols(x, order.max=6, demean=FALSE, intercept=TRUE)
```

`arima` *ARIMA Modelling of Time Series*

Description

Fit an ARIMA model to a univariate time series.

Usage

```
arima(x, order = c(0, 0, 0),
      seasonal = list(order = c(0, 0, 0), period = NA),
      xreg = NULL, include.mean = TRUE, transform.pars = TRUE,
      fixed = NULL, init = NULL, method = c("CSS-ML", "ML", "CSS"),
      n.cond, optim.control = list(), kappa = 1e6)
```

Arguments

x	a univariate time series
order	A specification of the non-seasonal part of the ARIMA model: the three components (p, d, q) are the AR order, the degree of differencing, and the MA order.
seasonal	A specification of the seasonal part of the ARIMA model, plus the period (which defaults to frequency(x)). This should be a list with components order and period, but a specification of just a numeric vector of length 3 will be turned into a suitable list with the specification as the order.
xreg	Optionally, a vector or matrix of external regressors, which must have the same number of rows as x.
include.mean	Should the ARIMA model include a mean term? The default is TRUE for undifferenced series, FALSE for differenced ones (where a mean would not affect the fit nor predictions).
transform.pars	Logical. If true, the AR parameters are transformed to ensure that they remain in the region of stationarity. Not used for method = "CSS".
fixed	optional numeric vector of the same length as the total number of parameters. If supplied, only NA entries in fixed will be varied. transform.pars = TRUE will be overridden (with a warning) if any AR parameters are fixed. It may be wise to set transform.pars = FALSE when fixing MA parameters, especially near non-invertibility.
init	optional numeric vector of initial parameter values. Missing values will be filled in, by zeroes except for regression coefficients. Values already specified in fixed will be ignored.
method	Fitting method: maximum likelihood or minimize conditional sum-of-squares. The default (unless there are missing values) is to use conditional-sum-of-squares to find starting values, then maximum likelihood.
n.cond	Only used if fitting by conditional-sum-of-squares: the number of initial observations to ignore. It will be ignored if less than the maximum lag of an AR term.
optim.control	List of control parameters for optim.
kappa	the prior variance (as a multiple of the innovations variance) for the past observations in a differenced model. Do not reduce this.

Details

Different definitions of ARMA models have different signs for the AR and/or MA coefficients. The definition here has

$$X_t = a_1 X_{t-1} + \cdots + a_p X_{t-p} + e_t + b_1 e_{t-1} + \ldots + b_q e_{t-q}$$

and so the MA coefficients differ in sign from those of S-PLUS. Further, if include.mean is true, this formula applies to $X - m$ rather than X. For ARIMA models with differencing, the differenced series follows a zero-mean ARMA model. If a xreg term is included, a linear regression (with a constant term if include.mean is true) is fitted with an ARMA model for the error term.

The variance matrix of the estimates is found from the Hessian of the log-likelihood, and so may only be a rough guide.

Optimization is done by optim. It will work best if the columns in xreg are roughly scaled to zero mean and unit variance, but does attempt to estimate suitable scalings.

Value

A list of class "Arima" with components:

coef	a vector of AR, MA and regression coefficients, which can be extracted by the coef method.
sigma2	the MLE of the innovations variance.
var.coef	the estimated variance matrix of the coefficients coef, which can be extracted by the vcov method.
loglik	the maximized log-likelihood (of the differenced data), or the approximation to it used.
arma	A compact form of the specification, as a vector giving the number of AR, MA, seasonal AR and seasonal MA coefficients, plus the period and the number of non-seasonal and seasonal differences.
aic	the AIC value corresponding to the log-likelihood. Only valid for method = "ML" fits.
residuals	the fitted innovations.
call	the matched call.
series	the name of the series x.
code	the convergence value returned by optim.
n.cond	the number of initial observations not used in the fitting.
model	A list representing the Kalman Filter used in the fitting. See KalmanLike.

Fitting methods

The exact likelihood is computed via a state-space representation of the ARIMA process, and the innovations and their variance found by a Kalman filter. The initialization of the differenced ARMA process uses stationarity and is based on Gardner *et al.* (1980). For a differenced process the non-stationary components are given a diffuse prior (controlled by kappa). Observations which are still controlled by the diffuse prior (determined by having a Kalman gain of at least 1e4) are excluded from the likelihood calculations. (This gives comparable results to arima0 in the absence of missing values, when the observations excluded are precisely those dropped by the differencing.)

Missing values are allowed, and are handled exactly in method "ML".

If transform.pars is true, the optimization is done using an alternative parametrization which is a variation on that suggested by Jones (1980) and ensures that the model is stationary. For an AR(p) model the parametrization is via the inverse tanh of the partial autocorrelations: the same procedure is applied (separately) to the AR and seasonal AR terms. The MA terms are not constrained to be invertible during optimization, but they will be converted to invertible form after optimization if transform.pars is true.

Conditional sum-of-squares is provided mainly for expositional purposes. This computes the sum of squares of the fitted innovations from observation n.cond on, (where n.cond is at least the maximum lag of an AR term), treating all earlier innovations to be zero. Argument n.cond can be used to allow comparability between different fits. The "part log-likelihood" is the first term, half the log of the estimated mean square. Missing values are allowed, but will cause many of the innovations to be missing.

When regressors are specified, they are orthogonalized prior to fitting unless any of the coefficients is fixed. It can be helpful to roughly scale the regressors to zero mean and unit variance.

Note

The results are likely to be different from S-PLUS's `arima.mle`, which computes a conditional likelihood and does not include a mean in the model. Further, the convention used by `arima.mle` reverses the signs of the MA coefficients.

`arima` is very similar to `arima0` for ARMA models or for differenced models without missing values, but handles differenced models with missing values exactly. It is somewhat slower than `arima0`, particularly for seasonally differenced models.

References

Brockwell, P. J. and Davis, R. A. (1996) *Introduction to Time Series and Forecasting.* Springer, New York. Sections 3.3 and 8.3.

Durbin, J. and Koopman, S. J. (2001) *Time Series Analysis by State Space Methods.* Oxford University Press.

Gardner, G, Harvey, A. C. and Phillips, G. D. A. (1980) Algorithm AS154. An algorithm for exact maximum likelihood estimation of autoregressive-moving average models by means of Kalman filtering. *Applied Statistics* **29**, 311–322.

Harvey, A. C. (1993) *Time Series Models*, 2nd Edition, Harvester Wheatsheaf, sections 3.3 and 4.4.

Jones, R. H. (1980) Maximum likelihood fitting of ARMA models to time series with missing observations. *Technometrics* **20** 389–395.

See Also

`predict.Arima`, `arima.sim` for simulating from an ARIMA model, `tsdiag`, `arima0`, `ar`

Examples

```
arima(lh, order = c(1,0,0))
arima(lh, order = c(3,0,0))
arima(lh, order = c(1,0,1))

arima(lh, order = c(3,0,0), method = "CSS")

arima(USAccDeaths, order = c(0,1,1), seasonal = list(order=c(0,1,1)))
arima(USAccDeaths, order = c(0,1,1), seasonal = list(order=c(0,1,1)),
      method = "CSS") # drops first 13 observations.
# for a model with as few years as this, we want full ML

arima(LakeHuron, order = c(2,0,0), xreg = time(LakeHuron)-1920)

## presidents contains NAs
## graphs in example(acf) suggest order 1 or 3
(fit1 <- arima(presidents, c(1, 0, 0)))
tsdiag(fit1)
(fit3 <- arima(presidents, c(3, 0, 0)))  # smaller AIC
tsdiag(fit3)
```

`arima.sim` *Simulate from an ARIMA Model*

Description

Simulate from an ARIMA model.

Usage

```
arima.sim(model, n, rand.gen = rnorm, innov = rand.gen(n, ...),
          n.start = NA, start.innov = rand.gen(n.start, ...), ...)
```

Arguments

`model`	A list with component `ar` and/or `ma` giving the AR and MA coefficients respectively. Optionally a component `order` can be used. An empty list gives an ARIMA(0, 0, 0) model, that is white noise.
`n`	length of output series, before un-differencing.
`rand.gen`	optional: a function to generate the innovations.
`innov`	an optional times series of innovations. If not provided, `rand.gen` is used.
`n.start`	length of "burn-in" period. If `NA`, the default, a reasonable value is computed.
`start.innov`	an optional times series of innovations to be used for the burn-in period. If supplied there must be at least `n.start` values (and `n.start` is by default computed inside the function).
`...`	additional arguments for `rand.gen`. Most usefully, the standard deviation of the innovations generated by `rnorm` can be specified by `sd`.

Details

See `arima` for the precise definition of an ARIMA model.

The ARMA model is checked for stationarity.

ARIMA models are specified via the `order` component of `model`, in the same way as for `arima`. Other aspects of the `order` component are ignored, but inconsistent specifications of the MA and AR orders are detected. The un-differencing assumes previous values of zero, and to remind the user of this, those values are returned.

Random inputs for the "burn-in" period are generated by calling `rand.gen`.

Value

A time-series object of class `"ts"`.

See Also

`arima`

Examples

```
arima.sim(n = 63, list(ar = c(0.8897, -0.4858), ma = c(-0.2279, 0.2488)),
          sd = sqrt(0.1796))
# mildly long-tailed
arima.sim(n = 63, list(ar=c(0.8897, -0.4858), ma=c(-0.2279, 0.2488)),
          rand.gen = function(n, ...) sqrt(0.1796) * rt(n, df = 5))

# An ARIMA simulation
ts.sim <- arima.sim(list(order = c(1,1,0), ar = 0.7), n = 200)
ts.plot(ts.sim)
```

---

arima0 *ARIMA Modelling of Time Series – Preliminary Version*

Description

Fit an ARIMA model to a univariate time series, and forecast from the fitted model.

Usage

```
arima0(x, order = c(0, 0, 0),
       seasonal = list(order = c(0, 0, 0), period = NA),
       xreg = NULL, include.mean = TRUE, delta = 0.01,
       transform.pars = TRUE, fixed = NULL, init = NULL,
       method = c("ML", "CSS"), n.cond, optim.control = list())

## S3 method for class 'arima0':
predict(object, n.ahead = 1, newxreg, se.fit = TRUE, ...)
```

Arguments

`x`	a univariate time series
`order`	A specification of the non-seasonal part of the ARIMA model: the three components (p, d, q) are the AR order, the degree of differencing, and the MA order.
`seasonal`	A specification of the seasonal part of the ARIMA model, plus the period (which defaults to `frequency(x)`). This should be a list with components `order` and `period`, but a specification of just a numeric vector of length 3 will be turned into a suitable list with the specification as the `order`.
`xreg`	Optionally, a vector or matrix of external regressors, which must have the same number of rows as `x`.
`include.mean`	Should the ARIMA model include a mean term? The default is `TRUE` for undifferenced series, `FALSE` for differenced ones (where a mean would not affect the fit nor predictions).
`delta`	A value to indicate at which point 'fast recursions' should be used. See the Details section.
`transform.pars`	Logical. If true, the AR parameters are transformed to ensure that they remain in the region of stationarity. Not used for `method = "CSS"`.

fixed	optional numeric vector of the same length as the total number of parameters. If supplied, only NA entries in fixed will be varied. transform.pars = TRUE will be overridden (with a warning) if any ARMA parameters are fixed.
init	optional numeric vector of initial parameter values. Missing values will be filled in, by zeroes except for regression coefficients. Values already specified in fixed will be ignored.
method	Fitting method: maximum likelihood or minimize conditional sum-of-squares.
n.cond	Only used if fitting by conditional-sum-of-squares: the number of initial observations to ignore. It will be ignored if less than the maximum lag of an AR term.
optim.control	List of control parameters for optim.
object	The result of an arima0 fit.
newxreg	New values of xreg to be used for prediction. Must have at least n.ahead rows.
n.ahead	The number of steps ahead for which prediction is required.
se.fit	Logical: should standard errors of prediction be returned?
...	arguments passed to or from other methods.

Details

Different definitions of ARMA models have different signs for the AR and/or MA coefficients. The definition here has

$$X_t = a_1 X_{t-1} + \cdots + a_p X_{t-p} + e_t + b_1 e_{t-1} + \ldots + b_q e_{t-q}$$

and so the MA coefficients differ in sign from those of S-PLUS. Further, if include.mean is true, this formula applies to $X - m$ rather than X. For ARIMA models with differencing, the differenced series follows a zero-mean ARMA model.

The variance matrix of the estimates is found from the Hessian of the log-likelihood, and so may only be a rough guide, especially for fits close to the boundary of invertibility.

Optimization is done by optim. It will work best if the columns in xreg are roughly scaled to zero mean and unit variance, but does attempt to estimate suitable scalings.

Finite-history prediction is used. This is only statistically efficient if the MA part of the fit is invertible, so predict.arima0 will give a warning for non-invertible MA models.

Value

For arima0, a list of class "arima0" with components:

coef	a vector of AR, MA and regression coefficients,
sigma2	the MLE of the innovations variance.
var.coef	the estimated variance matrix of the coefficients coef.
loglik	the maximized log-likelihood (of the differenced data), or the approximation to it used.
arma	A compact form of the specification, as a vector giving the number of AR, MA, seasonal AR and seasonal MA coefficients, plus the period and the number of non-seasonal and seasonal differences.

aic	the AIC value corresponding to the log-likelihood. Only valid for method = "ML" fits.
residuals	the fitted innovations.
call	the matched call.
series	the name of the series x.
convergence	the value returned by optim.
n.cond	the number of initial observations not used in the fitting.

For predict.arima0, a time series of predictions, or if se.fit = TRUE, a list with components pred, the predictions, and se, the estimated standard errors. Both components are time series.

Fitting methods

The exact likelihood is computed via a state-space representation of the ARMA process, and the innovations and their variance found by a Kalman filter based on Gardner *et al.* (1980). This has the option to switch to 'fast recursions' (assume an effectively infinite past) if the innovations variance is close enough to its asymptotic bound. The argument delta sets the tolerance: at its default value the approximation is normally negligible and the speed-up considerable. Exact computations can be ensured by setting delta to a negative value.

If transform.pars is true, the optimization is done using an alternative parametrization which is a variation on that suggested by Jones (1980) and ensures that the model is stationary. For an AR(p) model the parametrization is via the inverse tanh of the partial autocorrelations: the same procedure is applied (separately) to the AR and seasonal AR terms. The MA terms are also constrained to be invertible during optimization by the same transformation if transform.pars is true. Note that the MLE for MA terms does sometimes occur for MA polynomials with unit roots: such models can be fitted by using transform.pars = FALSE and specifying a good set of initial values (often obtainable from a fit with transform.pars = TRUE).

As from R 1.5.0 missing values are allowed, but any missing values will force delta to be ignored and full recursions used. Note that missing values will be propogated by differencing, so the procedure used in this function is not fully efficient in that case.

Conditional sum-of-squares is provided mainly for expositional purposes. This computes the sum of squares of the fitted innovations from observation n.cond on, (where n.cond is at least the maximum lag of an AR term), treating all earlier innovations to be zero. Argument n.cond can be used to allow comparability between different fits. The "part log-likelihood" is the first term, half the log of the estimated mean square. Missing values are allowed, but will cause many of the innovations to be missing.

When regressors are specified, they are orthogonalized prior to fitting unless any of the coefficients is fixed. It can be helpful to roughly scale the regressors to zero mean and unit variance.

Note

This is a preliminary version, and will be replaced by arima.

The standard errors of prediction exclude the uncertainty in the estimation of the ARMA model and the regression coefficients.

The results are likely to be different from S-PLUS's arima.mle, which computes a conditional likelihood and does not include a mean in the model. Further, the convention used by arima.mle reverses the signs of the MA coefficients.

References

Brockwell, P. J. and Davis, R. A. (1996) *Introduction to Time Series and Forecasting.* Springer, New York. Sections 3.3 and 8.3.

Gardner, G, Harvey, A. C. and Phillips, G. D. A. (1980) Algorithm AS154. An algorithm for exact maximum likelihood estimation of autoregressive-moving average models by means of Kalman filtering. *Applied Statistics* **29**, 311–322.

Harvey, A. C. (1993) *Time Series Models*, 2nd Edition, Harvester Wheatsheaf, sections 3.3 and 4.4.

Harvey, A. C. and McKenzie, C. R. (1982) Algorithm AS182. An algorithm for finite sample prediction from ARIMA processes. *Applied Statistics* **31**, 180–187.

Jones, R. H. (1980) Maximum likelihood fitting of ARMA models to time series with missing observations. *Technometrics* **20** 389–395.

See Also

`arima`, `ar`, `tsdiag`

Examples

```
## Not run: arima0(lh, order = c(1,0,0))
arima0(lh, order = c(3,0,0))
arima0(lh, order = c(1,0,1))
predict(arima0(lh, order = c(3,0,0)), n.ahead = 12)

arima0(lh, order = c(3,0,0), method = "CSS")

# for a model with as few years as this, we want full ML
(fit <- arima0(USAccDeaths, order = c(0,1,1),
               seasonal = list(order=c(0,1,1)), delta = -1))
predict(fit, n.ahead = 6)

arima0(LakeHuron, order = c(2,0,0), xreg = time(LakeHuron)-1920)
## Not run:
## presidents contains NAs
## graphs in example(acf) suggest order 1 or 3
(fit1 <- arima0(presidents, c(1, 0, 0), delta = -1))  # avoid warning
tsdiag(fit1)
(fit3 <- arima0(presidents, c(3, 0, 0), delta = -1))  # smaller AIC
tsdiag(fit3)
## End(Not run)
```

ARMAacf *Compute Theoretical ACF for an ARMA Process*

Description

Compute the theoretical autocorrelation function or partial autocorrelation function for an ARMA process.

Usage

```
ARMAacf(ar = numeric(0), ma = numeric(0), lag.max = r, pacf = FALSE)
```

Arguments

ar	numeric vector of AR coefficients
ma	numeric vector of MA coefficients
lag.max	integer. Maximum lag required. Defaults to max(p, q+1), where p, q are the numbers of AR and MA terms respectively.
pacf	logical. Should the partial autocorrelations be returned?

Details

The methods used follow Brockwell & Davis (1991, section 3.3). Their equations (3.3.8) are solved for the autocovariances at lags $0, \ldots, \max(p, q+1)$, and the remaining autocorrelations are given by a recursive filter.

Value

A vector of (partial) autocorrelations, named by the lags.

References

Brockwell, P. J. and Davis, R. A. (1991) *Time Series: Theory and Methods*, Second Edition. Springer.

See Also

arima, ARMAtoMA, acf2AR for inverting part of ARMAacf; further filter.

Examples

```
ARMAacf(c(1.0, -0.25), 1.0, lag.max = 10)

## Example from Brockwell & Davis (1991, pp.92-4)
## answer 2^(-n) * (32/3 + 8 * n) /(32/3)
n <- 1:10; 2^(-n) * (32/3 + 8 * n) /(32/3)
ARMAacf(c(1.0, -0.25), 1.0, lag.max = 10, pacf = TRUE)
ARMAacf(c(1.0, -0.25), lag.max = 10, pacf = TRUE)

## Cov-Matrix of length-7 sub-sample of AR(1) example:
toeplitz(ARMAacf(0.8, lag.max = 7))
```

ARMAtoMA *Convert ARMA Process to Infinite MA Process*

Description

Convert ARMA process to infinite MA process.

Usage

```
ARMAtoMA(ar = numeric(0), ma = numeric(0), lag.max)
```

Arguments

ar	numeric vector of AR coefficients
ma	numeric vector of MA coefficients
lag.max	Largest MA(Inf) coefficient required.

Value

A vector of coefficients.

References

Brockwell, P. J. and Davis, R. A. (1991) *Time Series: Theory and Methods*, Second Edition. Springer.

See Also

arima, ARMAacf.

Examples

```
ARMAtoMA(c(1.0, -0.25), 1.0, 10)
## Example from Brockwell & Davis (1991, p.92)
## answer (1 + 3*n)*2^(-n)
n <- 1:10; (1 + 3*n)*2^(-n)
```

---

as.hclust *Convert Objects to Class hclust*

Description

Converts objects from other hierarchical clustering functions to class "hclust".

Usage

```
as.hclust(x, ...)
```

Arguments

x	Hierarchical clustering object
...	further arguments passed to or from other methods.

Details

Currently there is only support for converting objects of class "twins" as produced by the functions diana and agnes from the package **cluster**. The default method throws an error unless passed an "hclust" object.

Value

An object of class "hclust".

See Also

hclust, diana, agnes

Examples

```
x <- matrix(rnorm(30), ncol=3)
hc <- hclust(dist(x), method="complete")

if(require(cluster, quietly=TRUE)) {# is a recommended package
  ag <- agnes(x, method="complete")
  hcag <- as.hclust(ag)
  ## The dendrograms order slightly differently:
  op <- par(mfrow=c(1,2))
  plot(hc) ;  mtext("hclust", side=1)
  plot(hcag); mtext("agnes",  side=1)
}
```

---

asOneSidedFormula *Convert to One-Sided Formula*

Description

Names, expressions, numeric values, and character strings are converted to one-sided formulas. If `object` is a formula, it must be one-sided, in which case it is returned unaltered.

Usage

```
asOneSidedFormula(object)
```

Arguments

`object`	a one-sided formula, an expression, a numeric value, or a character string.

Value

a one-sided formula representing `object`

Author(s)

Jose Pinheiro and Douglas Bates

See Also

formula

Examples

```
asOneSidedFormula("age")
asOneSidedFormula(~ age)
```

ave *Group Averages Over Level Combinations of Factors*

Description

Subsets of `x[]` are averaged, where each subset consist of those observations with the same factor levels.

Usage

```
ave(x, ..., FUN = mean)
```

Arguments

`x`	A numeric.
`...`	Grouping variables, typically factors, all of the same `length` as `x`.
`FUN`	Function to apply for each factor level combination.

Value

A numeric vector, say `y` of length `length(x)`. If `...` is `g1,g2`, e.g., `y[i]` is equal to `FUN(x[j]`, for all `j` with `g1[j] == g1[i]` and `g2[j] == g2[i])`.

See Also

`mean`, `median`.

Examples

```
ave(1:3)# no grouping -> grand mean

attach(warpbreaks)
ave(breaks, wool)
ave(breaks, tension)
ave(breaks, tension, FUN = function(x)mean(x, trim=.1))
plot(breaks, main =
     "ave( Warpbreaks )  for   wool  x  tension  combinations")
lines(ave(breaks, wool, tension           ), type='s', col = "blue")
lines(ave(breaks, wool, tension, FUN=median), type='s', col = "green")
legend(40,70, c("mean","median"), lty=1,col=c("blue","green"), bg="gray90")
detach()
```

bandwidth *Bandwidth Selectors for Kernel Density Estimation*

Description

Bandwidth selectors for gaussian windows in `density`.

Usage

```
bw.nrd0(x)

bw.nrd(x)

bw.ucv(x, nb = 1000, lower = 0.1 * hmax, upper = hmax)

bw.bcv(x, nb = 1000, lower = 0.1 * hmax, upper = hmax)

bw.SJ(x, nb = 1000, lower = 0.1 * hmax, upper = hmax,
      method = c("ste", "dpi"))
```

Arguments

`x`	A data vector.
`nb`	number of bins to use.
`lower, upper`	Range over which to minimize. The default is almost always satisfactory. `hmax` is calculated internally from a normal reference bandwidth.
`method`	Either `"ste"` ("solve-the-equation") or `"dpi"` ("direct plug-in").

Details

`bw.nrd0` implements a rule-of-thumb for choosing the bandwidth of a Gaussian kernel density estimator. It defaults to 0.9 times the minimum of the standard deviation and the interquartile range divided by 1.34 times the sample size to the negative one-fifth power (= Silverman's "rule of thumb", Silverman (1986, page 48, eqn (3.31)) *unless* the quartiles coincide when a positive result will be guaranteed.

`bw.nrd` is the more common variation given by Scott (1992), using factor 1.06.

`bw.ucv` and `bw.bcv` implement unbiased and biased cross-validation respectively.

`bw.SJ` implements the methods of Sheather & Jones (1991) to select the bandwidth using pilot estimation of derivatives.

Value

A bandwidth on a scale suitable for the `bw` argument of `density`.

References

Scott, D. W. (1992) *Multivariate Density Estimation: Theory, Practice, and Visualization.* Wiley.

Sheather, S. J. and Jones, M. C. (1991) A reliable data-based bandwidth selection method for kernel density estimation. *Journal of the Royal Statistical Society series B*, **53**, 683–690.

Silverman, B. W. (1986) *Density Estimation.* London: Chapman and Hall.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*. Springer.

See Also

`density`.

`bandwidth.nrd`, `ucv`, `bcv` and `width.SJ` in package **MASS**, which are all scaled to the `width` argument of `density` and so give answers four times as large.

Examples

```
plot(density(precip, n = 1000))
rug(precip)
lines(density(precip, bw="nrd"), col = 2)
lines(density(precip, bw="ucv"), col = 3)
lines(density(precip, bw="bcv"), col = 4)
lines(density(precip, bw="SJ-ste"), col = 5)
lines(density(precip, bw="SJ-dpi"), col = 6)
legend(55, 0.035,
       legend = c("nrd0", "nrd", "ucv", "bcv", "SJ-ste", "SJ-dpi"),
       col = 1:6, lty = 1)
```

`bartlett.test` *Bartlett Test of Homogeneity of Variances*

Description

Performs Bartlett's test of the null that the variances in each of the groups (samples) are the same.

Usage

```
bartlett.test(x, ...)

## Default S3 method:
bartlett.test(x, g, ...)

## S3 method for class 'formula':
bartlett.test(formula, data, subset, na.action, ...)
```

Arguments

`x`	a numeric vector of data values, or a list of numeric data vectors representing the respective samples, or fitted linear model objects (inheriting from class `"lm"`).
`g`	a vector or factor object giving the group for the corresponding elements of `x`. Ignored if `x` is a list.

formula	a formula of the form lhs ~ rhs where lhs gives the data values and rhs the corresponding groups.
data	an optional matrix or data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).
subset	an optional vector specifying a subset of observations to be used.
na.action	a function which indicates what should happen when the data contain NAs. Defaults to getOption("na.action").
...	further arguments to be passed to or from methods.

Details

If x is a list, its elements are taken as the samples or fitted linear models to be compared for homogeneity of variances. In this case, the elements must either all be numeric data vectors or fitted linear model objects, g is ignored, and one can simply use bartlett.test(x) to perform the test. If the samples are not yet contained in a list, use bartlett.test(list(x, ...)).

Otherwise, x must be a numeric data vector, and g must be a vector or factor object of the same length as x giving the group for the corresponding elements of x.

Value

A list of class "htest" containing the following components:

statistic	Bartlett's K-squared test statistic.
parameter	the degrees of freedom of the approximate chi-squared distribution of the test statistic.
p.value	the p-value of the test.
method	the character string "Bartlett test of homogeneity of variances".
data.name	a character string giving the names of the data.

References

Bartlett, M. S. (1937). Properties of sufficiency and statistical tests. *Proceedings of the Royal Statistical Society Series A* **160**, 268–282.

See Also

var.test for the special case of comparing variances in two samples from normal distributions; fligner.test for a rank-based (nonparametric) k-sample test for homogeneity of variances; ansari.test and mood.test for two rank based two-sample tests for difference in scale.

Examples

```
plot(count ~ spray, data = InsectSprays)
bartlett.test(InsectSprays$count, InsectSprays$spray)
bartlett.test(count ~ spray, data = InsectSprays)
```

`Beta` *The Beta Distribution*

Description

Density, distribution function, quantile function and random generation for the Beta distribution with parameters `shape1` and `shape2` (and optional non-centrality parameter `ncp`).

Usage

```
dbeta(x, shape1, shape2, ncp = 0, log = FALSE)
pbeta(q, shape1, shape2, ncp = 0, lower.tail = TRUE, log.p = FALSE)
qbeta(p, shape1, shape2, ncp = 0, lower.tail = TRUE, log.p = FALSE)
rbeta(n, shape1, shape2, ncp = 0)
```

Arguments

`x, q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`shape1, shape2`	positive parameters of the Beta distribution.
`ncp`	non-centrality parameter.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \le x]$, otherwise, $P[X > x]$.

Details

The Beta distribution with parameters `shape1` $= a$ and `shape2` $= b$ has density

$$f(x) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} x^a (1-x)^b$$

for $a > 0$, $b > 0$ and $0 \le x \le 1$ where the boundary values at $x = 0$ or $x = 1$ are defined as by continuity (as limits).
The mean is $a/(a+b)$ and the variance is $ab/((a+b)^2(a+b+1))$.

`pbeta` is closely related to the incomplete beta function. As defined by Abramowitz and Stegun 6.6.1

$$B_x(a,b) = \int_0^x t^{a-1}(1-t)^{b-1} dt,$$

and 6.6.2 $I_x(a,b) = B_x(a,b)/B(a,b)$ where $B(a,b) = B_1(a,b)$ is the Beta function (`beta`).

$I_x(a,b)$ is `pbeta(x,a,b)`.

The non-central Beta distribution is defined (Johnson et al, 1995, pp. 502) as the distribution of $X/(X+Y)$ where $X \sim \chi^2_{2a}(\lambda)$ and $Y \sim \chi^2_{2b}$.

Value

`dbeta` gives the density, `pbeta` the distribution function, `qbeta` the quantile function, and `rbeta` generates random deviates.

Invalid arguments will result in return value `NaN`, with a warning.

Source

The central `dbeta` is based on a binomial probability, using code contributed by Catherine Loader (see `dbinom`) if either shape parameter is larger than one, otherwise directly from the definition. The non-central case is based on the derivation as a Poisson mixture of betas (Johnson *et al*, 1995, pp. 502–3).

The central `pbeta` uses a C translation of

Didonato, A. and Morris, A., Jr, (1992) Algorithm 708: Significant digit computation of the incomplete beta function ratios, *ACM Transactions on Mathematical Software*, **18**, 360–373. (See also Brown, B. and Lawrence Levy, L. (1994) Certification of algorithm 708: Significant digit computation of the incomplete beta, *ACM Transactions on Mathematical Software*, **20**, 393–397.)

The non-central `pbeta` uses a C translation of

Lenth, R. V. (1987) Algorithm AS226: Computing noncentral beta probabilities. *Appl. Statist*, **36**, 241–244,
incorporating AS R84 (1990), *Appl. Statist*, **39**, 311–2.

`qbeta` is based on a C translation of

Cran, G. W., K. J. Martin and G. E. Thomas (1977). Remark AS R19 and Algorithm AS 109, *Applied Statistics*, **26**, 111–114, and subsequent remarks (AS83 and correction).

`rbeta` is based on a C translation of

R. C. H. Cheng (1978). Generating beta variates with nonintegral shape parameters. *Communications of the ACM*, **21**, 317–322.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Abramowitz, M. and Stegun, I. A. (1972) *Handbook of Mathematical Functions.* New York: Dover. Chapter 6: Gamma and Related Functions.

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, volume 2, especially chapter 25. Wiley, New York.

See Also

`beta` for the Beta function, and `dgamma` for the Gamma distribution.

Examples

```
x <- seq(0, 1, length=21)
dbeta(x, 1, 1)
pbeta(x, 1, 1)
```

`binom.test` *Exact Binomial Test*

Description

Performs an exact test of a simple null hypothesis about the probability of success in a Bernoulli experiment.

Usage

```
binom.test(x, n, p = 0.5,
           alternative = c("two.sided", "less", "greater"),
           conf.level = 0.95)
```

Arguments

`x`	number of successes, or a vector of length 2 giving the numbers of successes and failures, respectively.
`n`	number of trials; ignored if `x` has length 2.
`p`	hypothesized probability of success.
`alternative`	indicates the alternative hypothesis and must be one of `"two.sided"`, `"greater"` or `"less"`. You can specify just the initial letter.
`conf.level`	confidence level for the returned confidence interval.

Details

Confidence intervals are obtained by a procedure first given in Clopper and Pearson (1934). This guarantees that the confidence level is at least `conf.level`, but in general does not give the shortest-length confidence intervals.

Value

A list with class `"htest"` containing the following components:

`statistic`	the number of successes.
`parameter`	the number of trials.
`p.value`	the p-value of the test.
`conf.int`	a confidence interval for the probability of success.
`estimate`	the estimated probability of success.
`null.value`	the probability of success under the null, `p`.
`alternative`	a character string describing the alternative hypothesis.
`method`	the character string `"Exact binomial test"`.
`data.name`	a character string giving the names of the data.

References

Clopper, C. J. & Pearson, E. S. (1934). The use of confidence or fiducial limits illustrated in the case of the binomial. *Biometrika*, **26**, 404–413.

William J. Conover (1971), *Practical nonparametric statistics*. New York: John Wiley & Sons. Pages 97–104.

Myles Hollander & Douglas A. Wolfe (1973), *Nonparametric statistical inference*. New York: John Wiley & Sons. Pages 15–22.

See Also

`prop.test` for a general (approximate) test for equal or given proportions.

Examples

```
## Conover (1971), p. 97f.
## Under (the assumption of) simple Mendelian inheritance, a cross
##  between plants of two particular genotypes produces progeny 1/4 of
##  which are "dwarf" and 3/4 of which are "giant", respectively.
##  In an experiment to determine if this assumption is reasonable, a
##  cross results in progeny having 243 dwarf and 682 giant plants.
##  If "giant" is taken as success, the null hypothesis is that p =
##  3/4 and the alternative that p != 3/4.
binom.test(c(682, 243), p = 3/4)
binom.test(682, 682 + 243, p = 3/4)   # The same.
## => Data are in agreement with the null hypothesis.
```

Binomial *The Binomial Distribution*

Description

Density, distribution function, quantile function and random generation for the binomial distribution with parameters `size` and `prob`.

Usage

```
dbinom(x, size, prob, log = FALSE)
pbinom(q, size, prob, lower.tail = TRUE, log.p = FALSE)
qbinom(p, size, prob, lower.tail = TRUE, log.p = FALSE)
rbinom(n, size, prob)
```

Arguments

`x, q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`size`	number of trials (zero or more).
`prob`	probability of success on each trial.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

The binomial distribution with `size` $= n$ and `prob` $= p$ has density

$$p(x) = \binom{n}{x} p^x (1-p)^{n-x}$$

for $x = 0, \ldots, n$.

If an element of `x` is not integer, the result of `dbinom` is zero, with a warning. $p(x)$ is computed using Loader's algorithm, see the reference below.

The quantile is defined as the smallest value x such that $F(x) \geq p$, where F is the distribution function.

Value

`dbinom` gives the density, `pbinom` gives the distribution function, `qbinom` gives the quantile function and `rbinom` generates random deviates.

If `size` is not an integer, `NaN` is returned.

Source

For `dbinom` a saddle-point expansion is used: see

Catherine Loader (2000). *Fast and Accurate Computation of Binomial Probabilities*; available from http://www.herine.net/stat/software/dbinom.html.

`pbinom` uses `pbeta`.

`qbinom` uses the Cornish–Fisher Expansion to include a skewness correction to a normal approximation, followed by a search.

`rbinom` is based on

Kachitvichyanukul, V. and Schmeiser, B. W. (1988) Binomial random variate generation. *Communications of the ACM*, **31**, 216–222.

See Also

`dnbinom` for the negative binomial, and `dpois` for the Poisson distribution.

Examples

```
# Compute P(45 < X < 55) for X Binomial(100,0.5)
sum(dbinom(46:54, 100, 0.5))

## Using "log = TRUE" for an extended range :
n <- 2000
k <- seq(0, n, by = 20)
plot (k, dbinom(k, n, pi/10, log=TRUE), type='l', ylab="log density",
      main = "dbinom(*, log=TRUE) is better than  log(dbinom(*))")
lines(k, log(dbinom(k, n, pi/10)), col='red', lwd=2)
## extreme points are omitted since dbinom gives 0.
mtext("dbinom(k, log=TRUE)", adj=0)
mtext("extended range", adj=0, line = -1, font=4)
mtext("log(dbinom(k))", col="red", adj=1)
```

`biplot` *Biplot of Multivariate Data*

Description

Plot a biplot on the current graphics device.

Usage

```
biplot(x, ...)

## Default S3 method:
biplot(x, y, var.axes = TRUE, col, cex = rep(par("cex"), 2),
       xlabs = NULL, ylabs = NULL, expand = 1,
       xlim  = NULL, ylim  = NULL, arrow.len = 0.1,
       main = NULL, sub = NULL, xlab = NULL, ylab = NULL, ...)
```

Arguments

`x`	The `biplot`, a fitted object. For `biplot.default`, the first set of points (a two-column matrix), usually associated with observations.
`y`	The second set of points (a two-column matrix), usually associated with variables.
`var.axes`	If `TRUE` the second set of points have arrows representing them as (unscaled) axes.
`col`	A vector of length 2 giving the colours for the first and second set of points respectively (and the corresponding axes). If a single colour is specified it will be used for both sets. If missing the default colour is looked for in the `palette`: if there it and the next colour as used, otherwise the first two colours of the paletter are used.
`cex`	The character expansion factor used for labelling the points. The labels can be of different sizes for the two sets by supplying a vector of length two.
`xlabs`	A vector of character strings to label the first set of points: the default is to use the row dimname of `x`, or `1:n` is the dimname is `NULL`.
`ylabs`	A vector of character strings to label the second set of points: the default is to use the row dimname of `y`, or `1:n` is the dimname is `NULL`.
`expand`	An expansion factor to apply when plotting the second set of points relative to the first. This can be used to tweak the scaling of the two sets to a physically comparable scale.
`arrow.len`	The length of the arrow heads on the axes plotted in `var.axes` is true. The arrow head can be suppressed by `arrow.len = 0`.
`xlim, ylim`	Limits for the x and y axes in the units of the first set of variables.
`main, sub, xlab, ylab, ...`	graphical parameters.

Details

A biplot is plot which aims to represent both the observations and variables of a matrix of multivariate data on the same plot. There are many variations on biplots (see the references) and perhaps the most widely used one is implemented by `biplot.princomp`. The function `biplot.default` merely provides the underlying code to plot two sets of variables on the same figure.

Graphical parameters can also be given to `biplot`: the size of `xlabs` and `ylabs` is controlled by `cex`.

Side Effects

a plot is produced on the current graphics device.

References

K. R. Gabriel (1971). The biplot graphical display of matrices with application to principal component analysis. *Biometrika* **58**, 453–467.

J.C. Gower and D. J. Hand (1996). *Biplots*. Chapman & Hall.

See Also

`biplot.princomp`, also for examples.

`biplot.princomp` *Biplot for Principal Components*

Description

Produces a biplot (in the strict sense) from the output of `princomp` or `prcomp`

Usage

```
## S3 method for class 'prcomp':
biplot(x, choices = 1:2, scale = 1, pc.biplot = FALSE, ...)

## S3 method for class 'princomp':
biplot(x, choices = 1:2, scale = 1, pc.biplot = FALSE, ...)
```

Arguments

`x`	an object of class `"princomp"`.
`choices`	length 2 vector specifying the components to plot. Only the default is a biplot in the strict sense.
`scale`	The variables are scaled by `lambda ^ scale` and the observations are scaled by `lambda ^ (1-scale)` where `lambda` are the singular values as computed by `princomp`. Normally `0 <= scale <= 1`, and a warning will be issued if the specified `scale` is outside this range.
`pc.biplot`	If true, use what Gabriel (1971) refers to as a "principal component biplot", with `lambda = 1` and observations scaled up by sqrt(n) and variables scaled down by sqrt(n). Then inner products between variables approximate covariances and distances between observations approximate Mahalanobis distance.
`...`	optional arguments to be passed to `biplot.default`.

Details

This is a method for the generic function `biplot`. There is considerable confusion over the precise definitions: those of the original paper, Gabriel (1971), are followed here. Gabriel and Odoroff (1990) use the same definitions, but their plots actually correspond to `pc.biplot = TRUE`.

Side Effects

a plot is produced on the current graphics device.

References

Gabriel, K. R. (1971). The biplot graphical display of matrices with applications to principal component analysis. *Biometrika*, **58**, 453–467.

Gabriel, K. R. and Odoroff, C. L. (1990). Biplots in biomedical research. *Statistics in Medicine*, **9**, 469–485.

See Also

`biplot`, `princomp`.

Examples

```
biplot(princomp(USArrests))
```

`birthday` *Probability of coincidences*

Description

Computes approximate answers to a generalised "birthday paradox" problem. `pbirthday` computes the probability of a coincidence and `qbirthday` computes the number of observations needed to have a specified probability of coincidence.

Usage

```
qbirthday(prob = 0.5, classes = 365, coincident = 2)
pbirthday(n, classes = 365, coincident = 2)
```

Arguments

`classes`	How many distinct categories the people could fall into
`prob`	The desired probability of coincidence
`n`	The number of people
`coincident`	The number of people to fall in the same category

Details

The birthday paradox is that a very small number of people, 23, suffices to have a 50-50 chance that two of them have the same birthday. This function generalises the calculation to probabilities other than 0.5, numbers of coincident events other than 2, and numbers of classes other than 365.

This formula is approximate, as the example below shows. For `coincident=2` the exact computation is straightforward and may be preferable.

Value

`qbirthday`	Number of people needed for a probability `prob` that `k` of them have the same one out of `classes` equiprobable labels.
`pbirthday`	Probability of the specified coincidence

References

Diaconis P, Mosteller F., "Methods for studying coincidences". JASA 84:853-861

Examples

```
 ## the standard version
qbirthday()
 ## same 4-digit PIN number
qbirthday(classes=10^4)
 ## 0.9 probability of three coincident birthdays
qbirthday(coincident=3, prob=0.9)
## Chance of 4 coincident birthdays in 150 people
pbirthday(150,coincident=4)
## 100 coincident birthdays in 1000 people: *very* rare:
pbirthday(1000, coincident=100)

## Accuracy compared to exact calculation
x1<-  sapply(10:100, pbirthday)
x2<- 1-sapply(10:100, function(n)prod((365:(365-n+1))/rep(365,n)))
par(mfrow=c(2,2))
plot(x1, x2, xlab="approximate", ylab="exact")
abline(0,1)
plot(x1, x1-x2, xlab="approximate", ylab="error")
abline(h=0)
plot(x1, x2, log="xy", xlab="approximate", ylab="exact")
abline(0,1)
plot(1-x1, 1-x2, log="xy", xlab="approximate", ylab="exact")
abline(0,1)
```

`Box.test` *Box-Pierce and Ljung-Box Tests*

Description

Compute the Box–Pierce or Ljung–Box test statistic for examining the null hypothesis of independence in a given time series.

Usage

```
Box.test(x, lag = 1, type = c("Box-Pierce", "Ljung-Box"))
```

Arguments

`x`	a numeric vector or univariate time series.
`lag`	the statistic will be based on `lag` autocorrelation coefficients.
`type`	test to be performed: partial matching is used.

Value

A list with class "htest" containing the following components:

statistic	the value of the test statistic.
parameter	the degrees of freedom of the approximate chi-squared distribution of the test statistic.
p.value	the p-value of the test.
method	a character string indicating which type of test was performed.
data.name	a character string giving the name of the data.

Note

Missing values are not handled.

Author(s)

A. Trapletti

References

Box, G. E. P. and Pierce, D. A. (1970), Distribution of residual correlations in autoregressive-integrated moving average time series models. *Journal of the American Statistical Association*, **65**, 1509–1526.

Ljung, G. M. and Box, G. E. P. (1978), On a measure of lack of fit in time series models. *Biometrika* **65**, 553–564.

Harvey, A. C. (1993) *Time Series Models*. 2nd Edition, Harvester Wheatsheaf, NY, pp. 44, 45.

Examples

```
x <- rnorm (100)
Box.test (x, lag = 1)
Box.test (x, lag = 1, type="Ljung")
```

C *Sets Contrasts for a Factor*

Description

Sets the "contrasts" attribute for the factor.

Usage

```
C(object, contr, how.many, ...)
```

Arguments

object	a factor or ordered factor
contr	which contrasts to use. Can be a matrix with one row for each level of the factor or a suitable function like contr.poly or a character string giving the name of the function
how.many	the number of contrasts to set, by default one less than nlevels(object).
...	additional arguments for the function contr.

Details

For compatibility with S, `contr` can be `treatment`, `helmert`, `sum` or `poly` (without quotes) as shorthand for `contr.treatment` and so on.

Value

The factor `object` with the `"contrasts"` attribute set.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical models.* Chapter 2 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`contrasts`, `contr.sum`, etc.

Examples

```
## reset contrasts to defaults
options(contrasts=c("contr.treatment", "contr.poly"))
tens <- with(warpbreaks, C(tension, poly, 1))
attributes(tens)
## tension SHOULD be an ordered factor, but as it is not we can use
aov(breaks ~ wool + tens + tension, data=warpbreaks)

## show the use of ...  The default contrast is contr.treatment here
summary(lm(breaks ~ wool + C(tension, base=2), data=warpbreaks))

# following on from help(esoph)
model3 <- glm(cbind(ncases, ncontrols) ~ agegp + C(tobgp, , 1) +
     C(alcgp, , 1), data = esoph, family = binomial())
summary(model3)
```

`cancor` *Canonical Correlations*

Description

Compute the canonical correlations between two data matrices.

Usage

```
cancor(x, y, xcenter = TRUE, ycenter = TRUE)
```

Arguments

`x`	numeric matrix ($n \times p_1$), containing the x coordinates.
`y`	numeric matrix ($n \times p_2$), containing the y coordinates.
`xcenter`	logical or numeric vector of length p_1, describing any centering to be done on the x values before the analysis. If `TRUE` (default), subtract the column means. If `FALSE`, do not adjust the columns. Otherwise, a vector of values to be subtracted from the columns.
`ycenter`	analogous to `xcenter`, but for the y values.

Details

The canonical correlation analysis seeks linear combinations of the `y` variables which are well explained by linear combinations of the `x` variables. The relationship is symmetric as 'well explained' is measured by correlations.

Value

A list containing the following components:

`cor`	correlations.
`xcoef`	estimated coefficients for the `x` variables.
`ycoef`	estimated coefficients for the `y` variables.
`xcenter`	the values used to adjust the `x` variables.
`ycenter`	the values used to adjust the `x` variables.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Hotelling H. (1936). Relations between two sets of variables. *Biometrika*, **28**, 321–327.

Seber, G. A. F. (1984). *Multivariate Observations*. New York: Wiley, p. 506f.

See Also

`qr`, `svd`.

Examples

```
pop <- LifeCycleSavings[, 2:3]
oec <- LifeCycleSavings[, -(2:3)]
cancor(pop, oec)

x <- matrix(rnorm(150), 50, 3)
y <- matrix(rnorm(250), 50, 5)
(cxy <- cancor(x, y))
all(abs(cor(x %*% cxy$xcoef,
            y %*% cxy$ycoef)[,1:3] - diag(cxy $ cor)) < 1e-15)
all(abs(cor(x %*% cxy$xcoef) - diag(3)) < 1e-15)
all(abs(cor(y %*% cxy$ycoef) - diag(5)) < 1e-15)
```

`case/variable.names`

Case and Variable Names of Fitted Models

Description

Simple utilities returning (non-missing) case names, and (non-eliminated) variable names.

Usage

```
case.names(object, ...)
## S3 method for class 'lm':
case.names(object, full = FALSE, ...)

variable.names(object, ...)
## S3 method for class 'lm':
variable.names(object, full = FALSE, ...)
```

Arguments

`object`	an R object, typically a fitted model.
`full`	logical; if `TRUE`, all names (including zero weights, ...) are returned.
`...`	further arguments passed to or from other methods.

Value

A character vector.

See Also

`lm`; further, `all.names`, `all.vars` for functions with a similar name but only slightly related purpose.

Examples

```
x <- 1:20
y <-  x + (x/4 - 2)^3 + rnorm(20, s=3)
names(y) <- paste("O",x,sep=".")
ww <- rep(1,20); ww[13] <- 0
summary(lmxy <- lm(y ~ x + I(x^2)+I(x^3) + I((x-10)^2),
                   weights = ww), cor = TRUE)
variable.names(lmxy)
variable.names(lmxy, full= TRUE)# includes the last
case.names(lmxy)
case.names(lmxy, full = TRUE)# includes the 0-weight case
```

`Cauchy` *The Cauchy Distribution*

Description

Density, distribution function, quantile function and random generation for the Cauchy distribution with location parameter `location` and scale parameter `scale`.

Usage

```
dcauchy(x, location = 0, scale = 1, log = FALSE)
pcauchy(q, location = 0, scale = 1, lower.tail = TRUE, log.p = FALSE)
qcauchy(p, location = 0, scale = 1, lower.tail = TRUE, log.p = FALSE)
rcauchy(n, location = 0, scale = 1)
```

Arguments

`x, q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`location, scale`	location and scale parameters.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

If `location` or `scale` are not specified, they assume the default values of `0` and `1` respectively.

The Cauchy distribution with location l and scale s has density

$$f(x) = \frac{1}{\pi s}\left(1 + \left(\frac{x-l}{s}\right)^2\right)^{-1}$$

for all x.

Value

`dcauchy`, `pcauchy`, and `qcauchy` are respectively the density, distribution function and quantile function of the Cauchy distribution. `rcauchy` generates random deviates from the Cauchy.

Source

`dcauchy`, `pcauchy` and `qcauchy` are all calculated from numerically stable versions of the definitions.

`rcauchy` uses inversion.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, volume 1, chapter 16. Wiley, New York.

See Also

`dt` for the t distribution which generalizes `dcauchy(*, l = 0, s = 1)`.

Examples

```
dcauchy(-1:4)
```

`chisq.test` *Pearson's Chi-squared Test for Count Data*

Description

`chisq.test` performs chi-squared contingency table tests and goodness-of-fit tests.

Usage

```
chisq.test(x, y = NULL, correct = TRUE,
           p = rep(1/length(x), length(x)), rescale.p = FALSE,
           simulate.p.value = FALSE, B = 2000)
```

Arguments

`x`	a vector or matrix.
`y`	a vector; ignored if `x` is a matrix.
`correct`	a logical indicating whether to apply continuity correction when computing the test statistic for 2x2 tables: one half is subtracted from all $\|O - E\|$ differences. No correction is done if `simulate.p.value = TRUE`.
`p`	a vector of probabilities of the same length of `x`. An error is given if any entry of `p` is negative.
`rescale.p`	a logical scalar; if TRUE then `p` is rescaled (if necessary) to sum to 1. If `rescale.p` is FALSE, and `p` does not sum to 1, an error is given.
`simulate.p.value`	a logical indicating whether to compute p-values by Monte Carlo simulation.
`B`	an integer specifying the number of replicates used in the Monte Carlo test.

Details

If `x` is a matrix with one row or column, or if `x` is a vector and `y` is not given, then a "goodness-of-fit test" is performed ("`x` is treated as a one-dimensional contingency table"). The entries of `x` must be non-negative integers. In this case, the hypothesis tested is whether the population probabilities equal those in `p`, or are all equal if `p` is not given.

If `x` is a matrix with at least two rows and columns, it is taken as a two-dimensional contingency table. Again, the entries of `x` must be non-negative integers. Otherwise, `x` and `y` must be vectors or factors of the same length; incomplete cases are removed, the objects are coerced into factor objects, and the contingency table is computed from these. Then, Pearson's chi-squared test of the null hypothesis that the joint distribution of the cell counts in a 2-dimensional contingency table is the product of the row and column marginals is performed.

If `simulate.p.value` is `FALSE`, the p-value is computed from the asymptotic chi-squared distribution of the test statistic; continuity correction is only used in the 2-by-2 case (if `correct` is `TRUE`, the default). Otherwise the p-value is computed for a Monte Carlo test (Hope, 1968) with `B` replicates.

In the contingency table case simulation is done by random sampling from the set of all contingency tables with given marginals, and works only if the marginals are strictly positive. (A C translation of the algorithm of Patefield (1981) is used.) Continuity correction is never used, and the statistic is quoted without it. Note that this is not the usual sampling situation for the chi-squared test but rather that for Fisher's exact test.

In the goodness-of-fit case simulation is done by random sampling from the discrete distribution specified by p, each sample being of size n = sum(x). This simulation is done in R and may be slow.

Value

A list with class "htest" containing the following components:

statistic	the value the chi-squared test statistic.
parameter	the degrees of freedom of the approximate chi-squared distribution of the test statistic, NA if the p-value is computed by Monte Carlo simulation.
p.value	the p-value for the test.
method	a character string indicating the type of test performed, and whether Monte Carlo simulation or continuity correction was used.
data.name	a character string giving the name(s) of the data.
observed	the observed counts.
expected	the expected counts under the null hypothesis.
residuals	the Pearson residuals, (observed - expected) / sqrt(expected).

References

Hope, A. C. A. (1968) A simplified Monte Carlo significance test procedure. *J. Roy, Statist. Soc. B* **30**, 582–598.

Patefield, W. M. (1981) Algorithm AS159. An efficient method of generating r x c tables with given row and column totals. *Applied Statistics* **30**, 91–97.

Examples

```
## Not really a good example
chisq.test(InsectSprays$count > 7, InsectSprays$spray)
                                # Prints test summary
chisq.test(InsectSprays$count > 7, InsectSprays$spray)$obs
                                # Counts observed
chisq.test(InsectSprays$count > 7, InsectSprays$spray)$exp
                                # Counts expected under the null

## Effect of simulating p-values
x <- matrix(c(12, 5, 7, 7), nc = 2)
chisq.test(x)$p.value           # 0.4233
chisq.test(x, simulate.p.value = TRUE, B = 10000)$p.value
                                # around 0.29!

## Testing for population probabilities
## Case A. Tabulated data
x <- c(A = 20, B = 15, C = 25)
chisq.test(x)
chisq.test(as.table(x))         # the same
x <- c(89,37,30,28,2)
p <- c(40,20,20,15,5)
try(
chisq.test(x, p = p)            # gives an error
)
```

```
chisq.test(x, p = p, rescale.p = TRUE)
                                # works
p <- c(0.40,0.20,0.20,0.19,0.01)
                                # Expected count in category 5
                                # is 1.86 < 5 ==> chi square approx.
chisq.test(x, p = p)            #               maybe doubtful, but is ok!
chisq.test(x, p = p,simulate.p.value = TRUE)

## Case B. Raw data
x <- trunc(5 * runif(100))
chisq.test(table(x))            # NOT 'chisq.test(x)'!
```

`Chisquare` *The (non-central) Chi-Squared Distribution*

Description

Density, distribution function, quantile function and random generation for the chi-squared (χ^2) distribution with `df` degrees of freedom and optional non-centrality parameter `ncp`.

Usage

```
dchisq(x, df, ncp=0, log = FALSE)
pchisq(q, df, ncp=0, lower.tail = TRUE, log.p = FALSE)
qchisq(p, df, ncp=0, lower.tail = TRUE, log.p = FALSE)
rchisq(n, df, ncp=0)
```

Arguments

`x, q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`df`	degrees of freedom (non-negative, but can be non-integer).
`ncp`	non-centrality parameter (non-negative).
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

The chi-squared distribution with `df`$= n > 0$ degrees of freedom has density

$$f_n(x) = \frac{1}{2^{n/2}\Gamma(n/2)} x^{n/2-1} e^{-x/2}$$

for $x > 0$. The mean and variance are n and $2n$.

The non-central chi-squared distribution with `df`$= n$ degrees of freedom and non-centrality parameter `ncp` $= \lambda$ has density

$$f(x) = e^{-\lambda/2} \sum_{r=0}^{\infty} \frac{(\lambda/2)^r}{r!} f_{n+2r}(x)$$

for $x \geq 0$. For integer n, this is the distribution of the sum of squares of n normals each with variance one, λ being the sum of squares of the normal means; further,
$E(X) = n + \lambda$, $Var(X) = 2(n + 2 * \lambda)$, and $E((X - E(X))^3) = 8(n + 3 * \lambda)$.

Note that the degrees of freedom `df`$= n$, can be non-integer, and for non-centrality $\lambda > 0$, even $n = 0$; see Johnson et al. (1995, chapter 29).

Note that `ncp` values larger than about 1e5 may give inaccurate results with many warnings for `pchisq` and `qchisq`.

Value

`dchisq` gives the density, `pchisq` gives the distribution function, `qchisq` gives the quantile function, and `rchisq` generates random deviates.

Invalid arguments will result in return value `NaN`, with a warning.

Source

The central cases are computed via the gamma distribution.

The non-central `dchisq` and `rchisq` are computed as a Poisson mixture central of chi-squares (Johnson et al, 1995, p.436).

The non-central `pchisq` is for `ncp < 80` computed from the Poisson mixture of central chi-squares and for larger `ncp` based on a C translation of

Ding, C. G. (1992) Algorithm AS275: Computing the non-central chi-squared distribution function. *Appl.Statist.*, **41** 478–482.

which computes the lower tail only (so the upper tail suffers from cancellation).

The non-central `qchisq` is based on inversion of `pchisq`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, chapters 18 (volume 1) and 29 (volume 2). Wiley, New York.

See Also

A central chi-squared distribution with n degrees of freedom is the same as a Gamma distribution with `shape` $\alpha = n/2$ and `scale` $\sigma = 2$. Hence, see `dgamma` for the Gamma distribution.

Examples

```
dchisq(1, df=1:3)
pchisq(1, df= 3)
pchisq(1, df= 3, ncp = 0:4)# includes the above

x <- 1:10
## Chi-squared(df = 2) is a special exponential distribution
all.equal(dchisq(x, df=2), dexp(x, 1/2))
all.equal(pchisq(x, df=2), pexp(x, 1/2))

## non-central RNG -- df=0 is ok for ncp > 0:  Z0 has point mass at 0!
Z0 <- rchisq(100, df = 0, ncp = 2.)
graphics::stem(Z0)
```

```
## Not run:
## visual testing
## do P-P plots for 1000 points at various degrees of freedom
L <- 1.2; n <- 1000; pp <- ppoints(n)
op <- par(mfrow = c(3,3), mar= c(3,3,1,1)+.1, mgp= c(1.5,.6,0),
          oma = c(0,0,3,0))
for(df in 2^(4*rnorm(9))) {
  plot(pp, sort(pchisq(rr <- rchisq(n,df=df, ncp=L), df=df, ncp=L)),
       ylab="pchisq(rchisq(.),.)", pch=".")
  mtext(paste("df = ",formatC(df, digits = 4)), line= -2, adj=0.05)
  abline(0,1,col=2)
}
mtext(expression("P-P plots : Noncentral  "*
                 chi^2 *"(n=1000, df=X, ncp= 1.2)"),
      cex = 1.5, font = 2, outer=TRUE)
par(op)
## End(Not run)
```

`clearNames` *Remove the Names from an Object*

Description

This function sets the `names` attribute of `object` to `NULL` and returns the object.

Usage

```
clearNames(object)
```

Arguments

`object`	an object that may have a `names` attribute

Value

An object similar to `object` but without names.

Author(s)

Douglas Bates and Saikat DebRoy

See Also

`setNames`

Examples

```
lapply( women, mean )                # has a names attribute
clearNames( lapply( women, mean ) ) # removes the names
```

`cmdscale` *Classical (Metric) Multidimensional Scaling*

Description

Classical multidimensional scaling of a data matrix. Also known as *principal coordinates analysis* (Gower, 1966).

Usage

```
cmdscale(d, k = 2, eig = FALSE, add = FALSE, x.ret = FALSE)
```

Arguments

`d`	a distance structure such as that returned by `dist` or a full symmetric matrix containing the dissimilarities.
`k`	the dimension of the space which the data are to be represented in; must be in $\{1, 2, \ldots, n-1\}$.
`eig`	indicates whether eigenvalues should be returned.
`add`	logical indicating if an additive constant $c*$ should be computed, and added to the non-diagonal dissimilarites such that all $n-1$ eigenvalues are non-negative.
`x.ret`	indicates whether the doubly centered symmetric distance matrix should be returned.

Details

Multidimensional scaling takes a set of dissimilarities and returns a set of points such that the distances between the points are approximately equal to the dissimilarities.

The functions `isoMDS` and `sammon` in package **MASS** provide alternative ordination techniques.

When `add = TRUE`, an additive constant $c*$ is computed, and the dissimilarities $d_{ij} + c*$ are used instead of the original d_{ij}'s.

Whereas S (Becker *et al.*, 1988) computes this constant using an approximation suggested by Torgerson, R uses the analytical solution of Cailliez (1983), see also Cox and Cox (1994).

Value

If `eig = FALSE` and `x.ret = FALSE` (default), a matrix with `k` columns whose rows give the coordinates of the points chosen to represent the dissimilarities.

Otherwise, a list containing the following components.

`points`	a matrix with `k` columns whose rows give the coordinates of the points chosen to represent the dissimilarities.
`eig`	the $n-1$ eigenvalues computed during the scaling process if `eig` is true.
`x`	the doubly centered distance matrix if `x.ret` is true.
`GOF`	a numeric vector of length 2, equal to say (g_1, g_2), where $g_i = (\sum_{j=1}^{k} \lambda_j)/(\sum_{j=1}^{n} T_i(\lambda_j))$, where λ_j are the eigenvalues (sorted decreasingly), $T_1(v) = \lvert v \rvert$, and $T_2(v) = max(v, 0)$.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Cailliez, F. (1983) The analytical solution of the additive constant problem. *Psychometrika* **48**, 343–349.

Cox, T. F. and Cox, M. A. A. (1994) *Multidimensional Scaling*. Chapman and Hall.

Gower, J. C. (1966) Some distance properties of latent root and vector methods used in multivariate analysis. *Biometrika* **53**, 325–328.

Mardia, K. V., Kent, J. T. and Bibby, J. M. (1979). Chapter 14 of *Multivariate Analysis*, London: Academic Press.

Seber, G. A. F. (1984). *Multivariate Observations*. New York: Wiley.

Torgerson, W. S. (1958). *Theory and Methods of Scaling*. New York: Wiley.

See Also

`dist`. Also `isoMDS` and `sammon` in package **MASS**.

Examples

```
loc <- cmdscale(eurodist)
x <- loc[,1]
y <- -loc[,2]
plot(x, y, type="n", xlab="", ylab="", main="cmdscale(eurodist)")
text(x, y, rownames(loc), cex=0.8)

cmdsE <- cmdscale(eurodist, k=20, add = TRUE, eig = TRUE, x.ret = TRUE)
str(cmdsE)
```

`coef` *Extract Model Coefficients*

Description

`coef` is a generic function which extracts model coefficients from objects returned by modeling functions. `coefficients` is an *alias* for it.

Usage

```
coef(object, ...)
coefficients(object, ...)
```

Arguments

`object`	an object for which the extraction of model coefficients is meaningful.
`...`	other arguments.

Details

All object classes which are returned by model fitting functions should provide a `coef` method or use the default one. (Note that the method is for `coef` and not `coefficients`.)

Class `"aov"` has a `coef` method that does not report aliased coefficients (see `alias`).

Value

Coefficients extracted from the model object `object`.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

See Also

`fitted.values` and `residuals` for related methods; `glm`, `lm` for model fitting.

Examples

```
x <- 1:5; coef(lm(c(1:3,7,6) ~ x))
```

`complete.cases` *Find Complete Cases*

Description

Return a logical vector indicating which cases are complete, i.e., have no missing values.

Usage

```
complete.cases(...)
```

Arguments

`...`	a sequence of vectors, matrices and data frames.

Value

A logical vector specifying which observations/rows have no missing values across the entire sequence.

See Also

`is.na`, `na.omit`, `na.fail`.

Examples

```
x <- airquality[, -1] # x is a regression design matrix
y <- airquality[,  1] # y is the corresponding response

stopifnot(complete.cases(y) != is.na(y))
ok <- complete.cases(x,y)
sum(!ok) # how many are not "ok" ?
x <- x[ok,]
y <- y[ok]
```

confint *Confidence Intervals for Model Parameters*

Description

Computes confidence intervals for one or more parameters in a fitted model. There is a default and a method for objects inheriting from class "lm".

Usage

```
confint(object, parm, level = 0.95, ...)
```

Arguments

object	a fitted model object.
parm	a specification of which parameters are to be given confidence intervals, either a vector of numbers or a vector of names. If missing, all parameters are considered.
level	the confidence level required.
...	additional argument(s) for methods.

Details

confint is a generic function. The default method assumes asymptotic normality, and needs suitable coef and vcov methods to be available. The default method can be called directly for comparison with other methods.

For objects of class "lm" the direct formulae based on t values are used.

There are stub methods for classes "glm" and "nls" which invoke those in package **MASS** which are based on profile likelihoods.

Value

A matrix (or vector) with columns giving lower and upper confidence limits for each parameter. These will be labelled as (1-level)/2 and 1 - (1-level)/2 in % (by default 2.5% and 97.5%).

See Also

confint.glm and confint.nls in package **MASS**.

Examples

```
fit <- lm(100/mpg ~ disp + hp + wt + am, data=mtcars)
confint(fit)
confint(fit, "wt")

## from example(glm) (needs MASS to be present on the system)
counts <- c(18,17,15,20,10,20,25,13,12)
outcome <- gl(3,1,9); treatment <- gl(3,3)
glm.D93 <- glm(counts ~ outcome + treatment, family=poisson())
confint(glm.D93)
confint.default(glm.D93)  # based on asympotic normality
```

`constrOptim` *Linearly constrained optimisation*

Description

Minimise a function subject to linear inequality constraints using an adaptive barrier algorithm.

Usage

```
constrOptim(theta, f, grad, ui, ci, mu = 1e-04, control = list(),
            method = if(is.null(grad)) "Nelder-Mead" else "BFGS",
            outer.iterations = 100, outer.eps = 1e-05, ...)
```

Arguments

`theta`	Starting value: must be in the feasible region.
`f`	Function to minimise.
`grad`	Gradient of `f`.
`ui`	Constraints (see below).
`ci`	Constraints (see below).
`mu`	(Small) tuning parameter.
`control`	Passed to `optim`.
`method`	Passed to `optim`.
`outer.iterations`	Iterations of the barrier algorithm.
`outer.eps`	Criterion for relative convergence of the barrier algorithm.
`...`	Other arguments passed to `optim`, which will pass them to `f` and `grad` if it does not used them.

Details

The feasible region is defined by `ui %*% theta - ci >= 0`. The starting value must be in the interior of the feasible region, but the minimum may be on the boundary.

A logarithmic barrier is added to enforce the constraints and then `optim` is called. The barrier function is chosen so that the objective function should decrease at each outer iteration. Minima in the interior of the feasible region are typically found quite quickly, but a substantial number of outer iterations may be needed for a minimum on the boundary.

The tuning parameter `mu` multiplies the barrier term. Its precise value is often relatively unimportant. As `mu` increases the augmented objective function becomes closer to the original objective function but also less smooth near the boundary of the feasible region.

Any `optim` method that permits infinite values for the objective function may be used (currently all but "L-BFGS-B"). The gradient function must be supplied except with `method="Nelder-Mead"`.

As with `optim`, the default is to minimise and maximisation can be performed by setting `control$fnscale` to a negative value.

Value

As for optim, but with two extra components: barrier.value giving the value of the barrier function at the optimum and outer.iterations gives the number of outer iterations (calls to optim)

References

K. Lange *Numerical Analysis for Statisticians.* Springer 2001, p185ff

See Also

optim, especially method="L-BGFS-B" which does box-constrained optimisation.

Examples

```
## from optim
fr <- function(x) {   ## Rosenbrock Banana function
    x1 <- x[1]
    x2 <- x[2]
    100 * (x2 - x1 * x1)^2 + (1 - x1)^2
}
grr <- function(x) { ## Gradient of 'fr'
    x1 <- x[1]
    x2 <- x[2]
    c(-400 * x1 * (x2 - x1 * x1) - 2 * (1 - x1),
       200 *      (x2 - x1 * x1))
}

optim(c(-1.2,1), fr, grr)
#Box-constraint, optimum on the boundary
constrOptim(c(-1.2,0.9), fr, grr, ui=rbind(c(-1,0),c(0,-1)), ci=c(-1,-1))
#  x<=0.9,  y-x>0.1
constrOptim(c(.5,0), fr, grr, ui=rbind(c(-1,0),c(1,-1)), ci=c(-0.9,0.1))

## Solves linear and quadratic programming problems
## but needs a feasible starting value
#
# from example(solve.QP) in 'quadprog'
# no derivative
fQP <- function(b) {-sum(c(0,5,0)*b)+0.5*sum(b*b)}
Amat       <- matrix(c(-4,-3,0,2,1,0,0,-2,1),3,3)
bvec       <- c(-8,2,0)
constrOptim(c(2,-1,-1), fQP, NULL, ui=t(Amat),ci=bvec)
# derivative
gQP <- function(b) {-c(0,5,0)+b}
constrOptim(c(2,-1,-1), fQP, gQP, ui=t(Amat), ci=bvec)

## Now with maximisation instead of minimisation
hQP <- function(b) {sum(c(0,5,0)*b)-0.5*sum(b*b)}
constrOptim(c(2,-1,-1), hQP, NULL, ui=t(Amat), ci=bvec,
            control=list(fnscale=-1))
```

`contrast` *Contrast Matrices*

Description

Return a matrix of contrasts.

Usage

```
contr.helmert(n, contrasts = TRUE)
contr.poly(n, scores = 1:n, contrasts = TRUE)
contr.sum(n, contrasts = TRUE)
contr.treatment(n, base = 1, contrasts = TRUE)
contr.SAS(n, contrasts = TRUE)
```

Arguments

`n`	a vector of levels for a factor, or the number of levels.
`contrasts`	a logical indicating whether contrasts should be computed.
`scores`	the set of values over which orthogonal polynomials are to be computed.
`base`	an integer specifying which group is considered the baseline group. Ignored if `contrasts` is `FALSE`.

Details

These functions are used for creating contrast matrices for use in fitting analysis of variance and regression models. The columns of the resulting matrices contain contrasts which can be used for coding a factor with `n` levels. The returned value contains the computed contrasts. If the argument `contrasts` is `FALSE` a square indicator matrix (the dummy coding) is returned **except** for `contr.poly` (which include the 0-degree, i.e. constant, polynomial when `contrasts = FALSE`).

`cont.helmert` returns Helmert contrasts, which contrast the second level with the first, the third with the average of the first two, and so on. `contr.poly` returns contrasts based on orthogonal polynomials. `contr.sum` uses 'sum to zero contrasts'.

`contr.treatment` contrasts each level with the baseline level (specified by `base`): the baseline level is omitted. Note that this does not produce 'contrasts' as defined in the standard theory for linear models as they are not orthogonal to the intercept.

`contr.SAS` is a wrapper for `contr.treatment` that sets the base level to be the last level of the factor. The coefficients produced when using these contrasts should be equivalent to those produced by many (but not all) SAS procedures.

Value

A matrix with `n` rows and `k` columns, with `k=n-1` if `contrasts` is `TRUE` and `k=n` if `contrasts` is `FALSE`.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical models.* Chapter 2 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

contrasts, C, and aov, glm, lm.

Examples

```
(cH <- contr.helmert(4))
apply(cH, 2,sum) # column sums are 0!
crossprod(cH) # diagonal -- columns are orthogonal
contr.helmert(4, contrasts = FALSE) # just the 4 x 4 identity matrix

(cT <- contr.treatment(5))
all(crossprod(cT) == diag(4)) # TRUE: even orthonormal

(cP <- contr.SAS(5))
all(crossprod(cP) == diag(4)) # TRUE: even orthonormal

(cP <- contr.poly(3)) # Linear and Quadratic
zapsmall(crossprod(cP), dig=15) # orthonormal up to fuzz
```

---

contrasts *Get and Set Contrast Matrices*

Description

Set and view the contrasts associated with a factor.

Usage

```
contrasts(x, contrasts = TRUE)
contrasts(x, how.many) <- value
```

Arguments

x	a factor or a logical variable.
contrasts	logical. See Details.
how.many	How many contrasts should be made. Defaults to one less than the number of levels of x. This need not be the same as the number of columns of ctr.
value	either a numeric matrix whose columns give coefficients for contrasts in the levels of x, or the (quoted) name of a function which computes such matrices.

Details

If contrasts are not set for a factor the default functions from options("contrasts") are used.

A logical vector x is converted into a two-level factor with levels c(FALSE, TRUE) (regardless of which levels occur in the variable).

The argument contrasts is ignored if x has a matrix contrasts attribute set. Otherwise if contrasts = TRUE it is passed to a contrasts function such as contr.treatment and if contrasts = FALSE an identity matrix is returned.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical models.* Chapter 2 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`C`, `contr.helmert`, `contr.poly`, `contr.sum`, `contr.treatment`; `glm`, `aov`, `lm`.

Examples

```
example(factor)
fff <- ff[, drop=TRUE]  # reduce to 5 levels.
contrasts(fff) # treatment contrasts by default
contrasts(C(fff, sum))
contrasts(fff, contrasts = FALSE) # the 5x5 identity matrix

contrasts(fff) <- contr.sum(5); contrasts(fff)  # set sum contrasts
contrasts(fff, 2) <- contr.sum(5); contrasts(fff)  # set 2 contrasts
# supply 2 contrasts, compute 2 more to make full set of 4.
contrasts(fff) <- contr.sum(5)[,1:2]; contrasts(fff)
```

---

convolve *Fast Convolution*

Description

Use the Fast Fourier Transform to compute the several kinds of convolutions of two sequences.

Usage

```
convolve(x, y, conj = TRUE, type = c("circular", "open", "filter"))
```

Arguments

`x,y`	numeric sequences *of the same length* to be convolved.
`conj`	logical; if `TRUE`, take the complex *conjugate* before back-transforming (default, and used for usual convolution).
`type`	character; one of `"circular"`, `"open"`, `"filter"` (beginning of word is ok). For `circular`, the two sequences are treated as *circular*, i.e., periodic. For `open` and `filter`, the sequences are padded with `0`s (from left and right) first; `"filter"` returns the middle sub-vector of `"open"`, namely, the result of running a weighted mean of `x` with weights `y`.

Details

The Fast Fourier Transform, `fft`, is used for efficiency.

The input sequences `x` and `y` must have the same length if `circular` is true.

Note that the usual definition of convolution of two sequences `x` and `y` is given by `convolve(x, rev(y), type = "o")`.

Value

If `r <- convolve(x,y, type = "open")` and `n <- length(x)`, `m <- length(y)`, then

$$r_k = \sum_i x_{k-m+i} y_i$$

where the sum is over all valid indices i, for $k = 1, \ldots, n + m - 1$

If `type == "circular"`, $n = m$ is required, and the above is true for $i, k = 1, \ldots, n$ when $x_j := x_{n+j}$ for $j < 1$.

References

Brillinger, D. R. (1981) *Time Series: Data Analysis and Theory*, Second Edition. San Francisco: Holden-Day.

See Also

`fft`, `nextn`, and particularly `filter` (from the **stats** package) which may be more appropriate.

Examples

```
x <- c(0,0,0,100,0,0,0)
y <- c(0,0,1, 2 ,1,0,0)/4
zapsmall(convolve(x,y))         #  *NOT* what you first thought.
zapsmall(convolve(x, y[3:5], type="f")) # rather
x <- rnorm(50)
y <- rnorm(50)
# Circular convolution *has* this symmetry:
all.equal(convolve(x,y, conj = FALSE), rev(convolve(rev(y),x)))

n <- length(x <- -20:24)
y <- (x-10)^2/1000 + rnorm(x)/8

Han <- function(y) # Hanning
       convolve(y, c(1,2,1)/4, type = "filter")

plot(x,y, main="Using  convolve(.) for Hanning filters")
lines(x[-c(1  , n)      ], Han(y), col="red")
lines(x[-c(1:2, (n-1):n)], Han(Han(y)), lwd=2, col="dark blue")
```

`cophenetic` *Cophenetic Distances for a Hierarchical Clustering*

Description

Computes the cophenetic distances for a hierarchical clustering.

Usage

```
cophenetic(x)
## Default S3 method:
cophenetic(x)
## S3 method for class 'dendrogram':
cophenetic(x)
```

Arguments

x	an R object representing a hierarchical clustering. For the default method, an object of class hclust or with a method for as.hclust() such as agnes.

Details

The cophenetic distance between two observations that have been clustered is defined to be the intergroup dissimilarity at which the two observations are first combined into a single cluster. Note that this distance has many ties and restrictions.

It can be argued that a dendrogram is an appropriate summary of some data if the correlation between the original distances and the cophenetic distances is high. Otherwise, it should simply be viewed as the description of the output of the clustering algorithm.

cophenetic is a generic function. Support for classes which represent hierarchical clusterings (total indexed hierarchies) can be added by providing an as.hclust() or, more directly, a cophenetic() method for such a class.

The method for objects of class "dendrogram" requires that all leaves of the dendrogram object have non-null labels.

Value

An object of class dist.

Author(s)

Robert Gentleman

References

Sneath, P.H.A. and Sokal, R.R. (1973) *Numerical Taxonomy: The Principles and Practice of Numerical Classification*, p. 278 ff; Freeman, San Francisco.

See Also

dist, hclust

Examples

```
d1 <- dist(USArrests)
hc <- hclust(d1, "ave")
d2 <- cophenetic(hc)
cor(d1,d2) # 0.7659

## Example from Sneath & Sokal, Fig. 5-29, p.279
d0 <- c(1,3.8,4.4,5.1, 4,4.2,5, 2.6,5.3, 5.4)
attributes(d0) <- list(Size = 5, diag=TRUE)
class(d0) <- "dist"
names(d0) <- letters[1:5]
d0
str(upgma <- hclust(d0, method = "average"))
plot(upgma, hang = -1)
#
(d.coph <- cophenetic(upgma))
cor(d0, d.coph) # 0.9911
```

`cor` *Correlation, Variance and Covariance (Matrices)*

Description

`var`, `cov` and `cor` compute the variance of `x` and the covariance or correlation of `x` and `y` if these are vectors. If `x` and `y` are matrices then the covariances (or correlations) between the columns of `x` and the columns of `y` are computed.

`cov2cor` scales a covariance matrix into the corresponding correlation matrix *efficiently*.

Usage

```
var(x, y = NULL, na.rm = FALSE, use)

cov(x, y = NULL, use = "all.obs",
    method = c("pearson", "kendall", "spearman"))

cor(x, y = NULL, use = "all.obs",
     method = c("pearson", "kendall", "spearman"))

cov2cor(V)
```

Arguments

`x`	a numeric vector, matrix or data frame.
`y`	`NULL` (default) or a vector, matrix or data frame with compatible dimensions to `x`. The default is equivalent to `y = x` (but more efficient).
`na.rm`	logical. Should missing values be removed?
`use`	an optional character string giving a method for computing covariances in the presence of missing values. This must be (an abbreviation of) one of the strings `"all.obs"`, `"complete.obs"` or `"pairwise.complete.obs"`.
`method`	a character string indicating which correlation coefficient (or covariance) is to be computed. One of `"pearson"` (default), `"kendall"`, or `"spearman"`, can be abbreviated.
`V`	symmetric numeric matrix, usually positive definite such as a covariance matrix.

Details

For `cov` and `cor` one must *either* give a matrix or data frame for `x` *or* give both `x` and `y`.

`var` is just another interface to `cov`, where `na.rm` is used to determine the default for `use` when that is unspecified. If `na.rm` is `TRUE` then the complete observations (rows) are used (`use = "complete"`) to compute the variance. Otherwise (`use = "all"`), `var` will give an error if there are missing values.

If `use` is `"all.obs"`, then the presence of missing observations will produce an error. If `use` is `"complete.obs"` then missing values are handled by casewise deletion. Finally, if `use` has the value `"pairwise.complete.obs"` then the correlation between each pair of variables is computed using all complete pairs of observations on those variables. This can result in covariance or correlation matrices which are not positive semidefinite. `"pairwise.complete.obs"` only works with the `"pearson"` method for `cov` and `var`.

The denominator $n - 1$ is used which gives an unbiased estimator of the (co)variance for i.i.d. observations. These functions return NA when there is only one observation (whereas S-PLUS has been returning NaN), and fail if x has length zero.

For cor(), if method is "kendall" or "spearman", Kendall's τ or Spearman's ρ statistic is used to estimate a rank-based measure of association. These are more robust and have been recommended if the data do not necessarily come from a bivariate normal distribution.
For cov(), a non-Pearson method is unusual but available for the sake of completeness. Note that "spearman" basically computes cor(R(x), R(y)) (or cov(.,.)) where R(u) := rank(u, na.last="keep"). In the case of missing values, the ranks are calculated depending on the value of use, either based on complete observations, or based on pairwise completeness with reranking for each pair.

Scaling a covariance matrix into a correlation one can be achieved in many ways, mathematically most appealing by multiplication with a diagonal matrix from left and right, or more efficiently by using sweep(.., FUN = "/") twice. The cov2cor function is even a bit more efficient, and provided mostly for didactical reasons.

Value

For r <- cor(*, use = "all.obs"), it is now guaranteed that all(r <= 1).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

cor.test for confidence intervals (and tests).

cov.wt for *weighted* covariance computation.

sd for standard deviation (vectors).

Examples

```
var(1:10)# 9.166667

var(1:5,1:5)# 2.5

## Two simple vectors
cor(1:10,2:11)# == 1

## Correlation Matrix of Multivariate sample:
(Cl <- cor(longley))
## Graphical Correlation Matrix:
symnum(Cl) # highly correlated

## Spearman's rho  and  Kendall's tau
symnum(clS <- cor(longley, method = "spearman"))
symnum(clK <- cor(longley, method = "kendall"))
## How much do they differ?
i <- lower.tri(Cl)
cor(cbind(P = Cl[i], S = clS[i], K = clK[i]))

## cov2cor() scales a covariance matrix by its diagonal
##           to become the correlation matrix.
```

```
cov2cor # see the function definition {and learn ..}
stopifnot(all.equal(C1, cov2cor(cov(longley))),
          all.equal(cor(longley, method="kendall"),
            cov2cor(cov(longley, method="kendall"))))

##--- Missing value treatment:
C1 <- cov(swiss)
range(eigen(C1, only=TRUE)$val) # 6.19  1921
swM <- swiss
swM[1,2] <- swM[7,3] <- swM[25,5] <- NA # create 3 "missing"
try(cov(swM)) # Error: missing obs...
C2 <- cov(swM, use = "complete")
range(eigen(C2, only=TRUE)$val) # 6.46  1930
C3 <- cov(swM, use = "pairwise")
range(eigen(C3, only=TRUE)$val) # 6.19  1938

symnum(cor(swM, method = "kendall", use = "complete"))
## Kendall's tau doesn't change much:
symnum(cor(swiss, method = "kendall"))
```

---

cor.test — *Test for Association/Correlation Between Paired Samples*

Description

Test for association between paired samples, using one of Pearson's product moment correlation coefficient, Kendall's τ or Spearman's ρ.

Usage

```
cor.test(x, ...)

## Default S3 method:
cor.test(x, y,
         alternative = c("two.sided", "less", "greater"),
         method = c("pearson", "kendall", "spearman"),
         exact = NULL, conf.level = 0.95, ...)

## S3 method for class 'formula':
cor.test(formula, data, subset, na.action, ...)
```

Arguments

`x, y`	numeric vectors of data values. `x` and `y` must have the same length.
`alternative`	indicates the alternative hypothesis and must be one of `"two.sided"`, `"greater"` or `"less"`. You can specify just the initial letter. `"greater"` corresponds to positive association, `"less"` to negative association.
`method`	a character string indicating which correlation coefficient is to be used for the test. One of `"pearson"`, `"kendall"`, or `"spearman"`, can be abbreviated.
`exact`	a logical indicating whether an exact p-value should be computed. Used for Kendall's τ and Spearman's ρ. See the Details for the meaning of `NULL` (the default).

conf.level	confidence level for the returned confidence interval. Currently only used for the Pearson product moment correlation coefficient if there are at least 4 complete pairs of observations.
formula	a formula of the form ~ u + v, where each of u and v are numeric variables giving the data values for one sample. The samples must be of the same length.
data	an optional matrix or data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).
subset	an optional vector specifying a subset of observations to be used.
na.action	a function which indicates what should happen when the data contain NAs. Defaults to getOption("na.action").
...	further arguments to be passed to or from methods.

Details

The three methods each estimate the association between paired samples and compute a test of the value being zero. They use different measures of association, all in the range $[-1, 1]$ with 0 indicating no association. These are sometimes referred to as tests of no *correlation*, but that term is often confined to the default method.

If method is "pearson", the test statistic is based on Pearson's product moment correlation coefficient cor(x, y) and follows a t distribution with length(x)-2 degrees of freedom if the samples follow independent normal distributions. If there are at least 4 complete pairs of observation, an asymptotic confidence interval is given based on Fisher's Z transform.

If method is "kendall" or "spearman", Kendall's τ or Spearman's ρ statistic is used to estimate a rank-based measure of association. These tests may be used if the data do not necessarily come from a bivariate normal distribution.

For Kendall's test, by default (if exact is NULL), an exact p-value is computed if there are less than 50 paired samples containing finite values and there are no ties. Otherwise, the test statistic is the estimate scaled to zero mean and unit variance, and is approximately normally distributed.

For Spearman's test, p-values are computed using algorithm AS 89.

Value

A list with class "htest" containing the following components:

statistic	the value of the test statistic.
parameter	the degrees of freedom of the test statistic in the case that it follows a t distribution.
p.value	the p-value of the test.
estimate	the estimated measure of association, with name "cor", "tau", or "rho" correspoding to the method employed.
null.value	the value of the association measure under the null hypothesis, always 0.
alternative	a character string describing the alternative hypothesis.
method	a character string indicating how the association was measured.
data.name	a character string giving the names of the data.
conf.int	a confidence interval for the measure of association. Currently only given for Pearson's product moment correlation coefficient in case of at least 4 complete pairs of observations.

References

D. J. Best & D. E. Roberts (1975), Algorithm AS 89: The Upper Tail Probabilities of Spearman's ρ. *Applied Statistics*, **24**, 377–379.

Myles Hollander & Douglas A. Wolfe (1973), *Nonparametric statistical inference*. New York: John Wiley & Sons. Pages 185–194 (Kendall and Spearman tests).

Examples

```
## Hollander & Wolfe (1973), p. 187f.
## Assessment of tuna quality.  We compare the Hunter L measure of
##  lightness to the averages of consumer panel scores (recoded as
##  integer values from 1 to 6 and averaged over 80 such values) in
##  9 lots of canned tuna.

x <- c(44.4, 45.9, 41.9, 53.3, 44.7, 44.1, 50.7, 45.2, 60.1)
y <- c( 2.6,  3.1,  2.5,  5.0,  3.6,  4.0,  5.2,  2.8,  3.8)

##  The alternative hypothesis of interest is that the
##  Hunter L value is positively associated with the panel score.

cor.test(x, y, method = "kendall", alternative = "greater")
## => p=0.05972

cor.test(x, y, method = "kendall", alternative = "greater",
         exact = FALSE) # using large sample approximation
## => p=0.04765

## Compare this to
cor.test(x, y, method = "spearm", alternative = "g")
cor.test(x, y,                    alternative = "g")

## Formula interface.
pairs(USJudgeRatings)
cor.test(~ CONT + INTG, data = USJudgeRatings)
```

---

cov.wt *Weighted Covariance Matrices*

Description

Returns a list containing estimates of the weighted covariance matrix and the mean of the data, and optionally of the (weighted) correlation matrix.

Usage

```
cov.wt(x, wt = rep(1/nrow(x), nrow(x)), cor = FALSE, center = TRUE,
       method = c("unbiased", "ML"))
```

Arguments

x	a matrix or data frame. As usual, rows are observations and columns are variables.

wt	a non-negative and non-zero vector of weights for each observation. Its length must equal the number of rows of x.
cor	a logical indicating whether the estimated correlation weighted matrix will be returned as well.
center	either a logical or a numeric vector specifying the centers to be used when computing covariances. If TRUE, the (weighted) mean of each variable is used, if FALSE, zero is used. If center is numeric, its length must equal the number of columns of x.
method	string specifying how the result is scaled, see *Details* below.

Details

By default, `method = "unbiased"`, The covariance matrix is divided by one minus the sum of squares of the weights, so if the weights are the default ($1/n$) the conventional unbiased estimate of the covariance matrix with divisor ($n - 1$) is obtained. This differs from the behaviour in S-PLUS which corresponds to `method = "ML"` and does not divide.

Value

A list containing the following named components:

cov	the estimated (weighted) covariance matrix
center	an estimate for the center (mean) of the data.
n.obs	the number of observations (rows) in x.
wt	the weights used in the estimation. Only returned if given as an argument.
cor	the estimated correlation matrix. Only returned if cor is TRUE.

See Also

cov and var.

Examples

```
 (xy <- cbind(x = 1:10, y = c(1:3, 8:5, 8:10)))
w1 <- c(0,0,0,1,1,1,1,1,0,0)
cov.wt(xy, wt = w1) # i.e. method = "unbiased"
cov.wt(xy, wt = w1, method = "ML", cor = TRUE)
```

---

cpgram *Plot Cumulative Periodogram*

Description

Plots a cumulative periodogram.

Usage

```
cpgram(ts, taper = 0.1,
       main = paste("Series: ", deparse(substitute(ts))),
       ci.col = "blue")
```

Arguments

ts	a univariate time series
taper	proportion tapered in forming the periodogram
main	main title
ci.col	colour for confidence band.

Value

None.

Side Effects

Plots the cumulative periodogram in a square plot.

Note

From package **MASS**.

Author(s)

B.D. Ripley

Examples

```
par(pty = "s", mfrow = c(1,2))
cpgram(lh)
lh.ar <- ar(lh, order.max = 9)
cpgram(lh.ar$resid, main = "AR(3) fit to lh")

cpgram(ldeaths)
```

---

cutree *Cut a tree into groups of data*

Description

Cuts a tree, e.g., as resulting from hclust, into several groups either by specifying the desired number(s) of groups or the cut height(s).

Usage

```
cutree(tree, k = NULL, h = NULL)
```

Arguments

tree	a tree as produced by hclust. cutree() only expects a list with components merge, height, and labels, of appropriate content each.
k	an integer scalar or vector with the desired number of groups
h	numeric scalar or vector with heights where the tree should be cut.

At least one of k or h must be specified, k overrides h if both are given.

Value

`cutree` returns a vector with group memberships if `k` or `h` are scalar, otherwise a matrix with group meberships is returned where each column corresponds to the elements of `k` or `h`, respectively (which are also used as column names).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`hclust`, `dendrogram` for cutting trees themselves.

Examples

```
hc <- hclust(dist(USArrests))

cutree(hc, k=1:5)#k = 1 is trivial
cutree(hc, h=250)

## Compare the 2 and 3 grouping:
g24 <- cutree(hc, k = c(2,4))
table(g24[,"2"], g24[,"4"])
```

`decompose` *Classical Seasonal Decomposition by Moving Averages*

Description

Decompose a time series into seasonal, trend and irregular components using moving averages. Deals with additive or multiplicative seasonal component.

Usage

```
decompose(x, type = c("additive", "multiplicative"), filter = NULL)
```

Arguments

`x`	A time series.
`type`	The type of seasonal component.
`filter`	A vector of filter coefficients in reverse time order (as for AR or MA coefficients), used for filtering out the seasonal component. If `NULL`, a moving average with symmetric window is performed.

Details

The additive model used is:

Y[t] = T[t] + S[t] + e[t]

The multiplicative model used is:

Y[t] = T[t] * S[t] + e[t]

Value

An object of class "decomposed.ts" with following components:

seasonal	The seasonal component (i.e., the repeated seasonal figure)
figure	The estimated seasonal figure only
trend	The trend component
random	The remainder part
type	The value of type

Note

The function stl provides a much more sophisticated decomposition.

Author(s)

David Meyer ⟨David.Meyer@wu-wien.ac.at⟩

See Also

stl

Examples

```
m <- decompose(co2)
m$figure
plot(m)
```

---

delete.response *Modify Terms Objects*

Description

delete.response returns a terms object for the same model but with no response variable.

drop.terms removes variables from the right-hand side of the model. There is also a "[.terms" method to perform the same function (with keep.response=TRUE).

reformulate creates a formula from a character vector.

Usage

```
delete.response(termobj)

reformulate(termlabels, response = NULL)

drop.terms(termobj, dropx = NULL, keep.response = FALSE)
```

Arguments

termobj	A terms object
termlabels	character vector giving the right-hand side of a model formula.
response	character string, symbol or call giving the left-hand side of a model formula.
dropx	vector of positions of variables to drop from the right-hand side of the model.
keep.response	Keep the response in the resulting object?

Value

delete.response and drop.terms return a terms object.

reformulate returns a formula.

See Also

terms

Examples

```
ff <- y ~ z + x + w
tt <- terms(ff)
tt
delete.response(tt)
drop.terms(tt, 2:3, keep.response = TRUE)
tt[-1]
tt[2:3]
reformulate(attr(tt, "term.labels"))

## keep LHS :
reformulate("x*w", ff[[2]])
fS <- surv(ft, case) ~ a + b
reformulate(c("a", "b*f"), fS[[2]])

stopifnot(identical(      ~ var, reformulate("var")),
          identical(~ a + b + c, reformulate(letters[1:3])),
          identical(  y ~ a + b, reformulate(letters[1:2], "y"))
         )
```

---

dendrapply *Apply a Function to All Nodes of a Dendrogram*

Description

Apply function FUN to each node of a dendrogram recursively. When y <- dendrapply(x, fn), then y is a dendrogram of the same graph structure as x and each for each node, y.node[j] <- FUN(x.node[j], ...) (where y.node[j] is an (invalid!) notation for the j-th node of y.

Usage

```
dendrapply(X, FUN, ...)
```

Arguments

X	an object of class "dendrogram".
FUN	an R function to be applied to each dendrogram node, typically working on its attributes alone, returning an altered version of the same node.
...	potential further arguments passed to FUN.

Value

Usually a dendrogram of the same (graph) structure as X. For that, the function must be conceptually of the form `FUN <- function(X) { attributes(X) <- .....; X }`, i.e. returning the node with some attributes added or changed.

Note

this is still somewhat experimental, and suggestions for enhancements (or nice examples of usage) are very welcome.

Author(s)

Martin Maechler

See Also

as.dendrogram, lapply for applying a function to each component of a list.

Examples

```
## a smallish simple dendrogram
dhc <- as.dendrogram(hc <- hclust(dist(USArrests), "ave"))
(dhc21 <- dhc[[2]][[1]])

## too simple:
dendrapply(dhc21, function(n) str(attributes(n)))

## toy example to set colored leaf labels :
local({
  colLab <<- function(n) {
      if(is.leaf(n)) {
        a <- attributes(n)
        i <<- i+1
        attr(n, "nodePar") <-
            c(a$nodePar, list(lab.col = mycols[i], lab.font= i%%3))
      }
      n
  }
  mycols <- grDevices::rainbow(attr(dhc21,"members"))
  i <- 0
 })
dL <- dendrapply(dhc21, colLab)
op <- par(mfrow=2:1)
 plot(dhc21)
 plot(dL) ## --> colored labels!
par(op)
```

dendrogram *General Tree Structures*

Description

Class "dendrogram" provides general functions for handling tree-like structures. It is intended as a replacement for similar functions in hierarchical clustering and classification/regression trees, such that all of these can use the same engine for plotting or cutting trees.

The code is still in testing stage and the API may change in the future.

Usage

```
as.dendrogram(object, ...)
## S3 method for class 'hclust':
as.dendrogram(object, hang = -1, ...)

## S3 method for class 'dendrogram':
plot(x, type = c("rectangle", "triangle"),
      center = FALSE,
      edge.root = is.leaf(x) || !is.null(attr(x,"edgetext")),
      nodePar = NULL, edgePar = list(),
      leaflab = c("perpendicular", "textlike", "none"),
      dLeaf = NULL, xlab = "", ylab = "", xaxt = "n", yaxt = "s",
      horiz = FALSE, frame.plot = FALSE, ...)

## S3 method for class 'dendrogram':
cut(x, h, ...)

## S3 method for class 'dendrogram':
print(x, digits, ...)

## S3 method for class 'dendrogram':
rev(x)

## S3 method for class 'dendrogram':
str(object, max.level = 0, digits.d = 3,
    give.attr = FALSE, wid = getOption("width"),
    nest.lev = 0, indent.str = "", stem = "--", ...)

is.leaf(object)
```

Arguments

object	any R object that can be made into one of class "dendrogram".
x	object of class "dendrogram".
hang	numeric scalar indicating how the *height* of leaves should be computed from the heights of their parents; see plot.hclust.
type	type of plot.
center	logical; if TRUE, nodes are plotted centered with respect to the leaves in the branch. Otherwise (default), plot them in the middle of all direct child nodes.

edge.root	logical; if true, draw an edge to the root node.
nodePar	a list of plotting parameters to use for the nodes (see points) or NULL by default which does not draw symbols at the nodes. The list may contain components named pch, cex, col, and/or bg each of which can have length two for specifying separate attributes for *inner* nodes and *leaves*.
edgePar	a list of plotting parameters to use for the edge segments and labels (if there's an edgetext). The list may contain components named col, lty and lwd (for the segments), p.col, p.lwd, and p.lty (for the polygon around the text) and t.col for the text color. As with nodePar, each can have length two for differentiating leaves and inner nodes.
leaflab	a string specifying how leaves are labeled. The default "perpendicular" write text vertically (by default). "textlike" writes text horizontally (in a rectangle), and "none" suppresses leaf labels.
dLeaf	a number specifying the **d**istance in user coordinates between the tip of a leaf and its label. If NULL as per default, 3/4 of a letter width or height is used.
horiz	logical indicating if the dendrogram should be drawn *horizontally* or not.
frame.plot	logical indicating if a box around the plot should be drawn, see plot.default.
h	height at which the tree is cut.
..., xlab, ylab, xaxt, yaxt	graphical parameters, or arguments for other methods.
digits	integer specifiying the precision for printing, see print.default.
max.level, digits.d, give.attr, wid, nest.lev, indent.str	arguments to str, see str.default(). Note that give.attr = FALSE still shows height and members attributes for each node.
stem	a string used for str() specifying the *stem* to use for each dendrogram branch.

Details

Warning: This documentation is preliminary.

The dendrogram is directly represented as a nested list where each component corresponds to a branch of the tree. Hence, the first branch of tree z is z[[1]], the second branch of the corresponding subtree is z[[1]][[2]] etc.. Each node of the tree carries some information needed for efficient plotting or cutting as attributes, of which only members, height and leaf for leaves are compulsory:

members total number of leaves in the branch

height numeric non-negative height at which the node is plotted.

midpoint numeric horizontal distance of the node from the left border (the leftmost leaf) of the branch (unit 1 between all leaves). This is used for plot(*, center=FALSE).

label character; the label of the node

x.member for cut()$upper, the number of *former* members; more generally a substitute for the members component used for "horizontal" (when horiz = FALSE, else "vertical") alignment.

edgetext character; the label for the edge leading to the node

nodePar a named list (of length-1 components) specifying node-specific attributes for points plotting, see the nodePar argument above.

edgePar a named list (of length-1 components) specifying attributes for segments plotting of the edge leading to the node, and drawing of the edgetext if available, see the edgePar argument above.

leaf logical, if TRUE, the node is a leaf of the tree.

cut.dendrogram() returns a list with components $upper and $lower, the first is a truncated version of the original tree, also of class dendrogram, the latter a list with the branches obtained from cutting the tree, each a dendrogram.

There are [[, print, and str methods for "dendrogram" objects where the first one (extraction) ensures that selecting sub-branches keeps the class.

Objects of class "hclust" can be converted to class "dendrogram" using method as.dendrogram.

rev.dendrogram simply returns the dendrogram x with reversed nodes, see also reorder.dendrogram.

is.leaf(object) is logical indicating if object is a leaf (the most simple dendrogram). plotNode() and plotNodeLimit() are helper functions.

Warning

Some operations on dendrograms (including plotting) make use of recursion. For very deep trees It may be necessary to increase options("expressions"): if you do you are likely to need to set the C stack size larger than the OS default if possible (which it is not on Windows).

Note

When using type = "triangle", center = TRUE often looks better.

See Also

order.dendrogram also on the labels method for dendrograms.

Examples

```
hc <- hclust(dist(USArrests), "ave")
(dend1 <- as.dendrogram(hc)) # "print()" method
str(dend1)          # "str()" method
str(dend1, max = 2) # only the first two sub-levels

op <- par(mfrow= c(2,2), mar = c(5,2,1,4))
plot(dend1)
## "triangle" type and show inner nodes:
plot(dend1, nodePar=list(pch = c(1,NA), cex=0.8, lab.cex = 0.8),
      type = "t", center=TRUE)
plot(dend1, edgePar=list(col = 1:2, lty = 2:3), dLeaf=1, edge.root = TRUE)
plot(dend1, nodePar=list(pch = 2:1,cex=.4*2:1, col = 2:3), horiz=TRUE)

dend2 <- cut(dend1, h=70)
plot(dend2$upper)
## leafs are wrong horizontally:
plot(dend2$upper, nodePar=list(pch = c(1,7), col = 2:1))
##  dend2$lower is *NOT* a dendrogram, but a list of .. :
plot(dend2$lower[[3]], nodePar=list(col=4), horiz = TRUE, type = "tr")
## "inner" and "leaf" edges in different type & color :
plot(dend2$lower[[2]], nodePar=list(col=1),# non empty list
```

```
     edgePar = list(lty=1:2, col=2:1), edge.root=TRUE)
par(op)
str(d3 <- dend2$lower[[2]][[2]][[1]])

nP <- list(col=3:2, cex=c(2.0, 0.75), pch= 21:22, bg= c("light blue", "pink"),
           lab.cex = 0.75, lab.col = "tomato")
plot(d3, nodePar= nP, edgePar = list(col="gray", lwd=2), horiz = TRUE)
addE <- function(n) {
      if(!is.leaf(n)) {
        attr(n, "edgePar") <- list(p.col="plum")
        attr(n, "edgetext") <- paste(attr(n,"members"),"members")
      }
      n
}
d3e <- dendrapply(d3, addE)
plot(d3e, nodePar= nP)
plot(d3e, nodePar= nP, leaflab = "textlike")
```

`density` *Kernel Density Estimation*

Description

The (S3) generic function `density` computes kernel density estimates. Its default method does so with the given kernel and bandwidth for univariate observations.

Usage

```
density(x, ...)
## Default S3 method:
density(x, bw = "nrd0", adjust = 1,
        kernel = c("gaussian", "epanechnikov", "rectangular",
                   "triangular", "biweight", "cosine", "optcosine"),
        weights = NULL, window = kernel, width,
        give.Rkern = FALSE,
        n = 512, from, to, cut = 3, na.rm = FALSE, ...)
```

Arguments

`x`	the data from which the estimate is to be computed.
`bw`	the smoothing bandwidth to be used. The kernels are scaled such that this is the standard deviation of the smoothing kernel. (Note this differs from the reference books cited below, and from S-PLUS.) `bw` can also be a character string giving a rule to choose the bandwidth. See `bw.nrd`. The specified (or computed) value of `bw` is multiplied by `adjust`.
`adjust`	the bandwidth used is actually `adjust*bw`. This makes it easy to specify values like "half the default" bandwidth.

`kernel, window`
: a character string giving the smoothing kernel to be used. This must be one of `"gaussian"`, `"rectangular"`, `"triangular"`, `"epanechnikov"`, `"biweight"`, `"cosine"` or `"optcosine"`, with default `"gaussian"`, and may be abbreviated to a unique prefix (single letter).

  `"cosine"` is smoother than `"optcosine"`, which is the usual "cosine" kernel in the literature and almost MSE-efficient. However, `"cosine"` is the version used by S.

`weights`
: numeric vector of non-negative observation weights, hence of same length as `x`. The default `NULL` is equivalent to `weights = rep(1/nx, nx)` where `nx` is the length of (the finite entries of) `x[]`.

`width`
: this exists for compatibility with S; if given, and `bw` is not, will set `bw` to `width` if this is a character string, or to a kernel-dependent multiple of `width` if this is numeric.

`give.Rkern`
: logical; if true, *no* density is estimated, and the "canonical bandwidth" of the chosen `kernel` is returned instead.

`n`
: the number of equally spaced points at which the density is to be estimated. When `n > 512`, it is rounded up to the next power of 2 for efficiency reasons (`fft`).

`from,to`
: the left and right-most points of the grid at which the density is to be estimated.

`cut`
: by default, the values of `left` and `right` are `cut` bandwidths beyond the extremes of the data. This allows the estimated density to drop to approximately zero at the extremes.

`na.rm`
: logical; if `TRUE`, missing values are removed from `x`. If `FALSE` any missing values cause an error.

`...`
: further arguments for (non-default) methods.

Details

The algorithm used in `density.default` disperses the mass of the empirical distribution function over a regular grid of at least 512 points and then uses the fast Fourier transform to convolve this approximation with a discretized version of the kernel and then uses linear approximation to evaluate the density at the specified points.

The statistical properties of a kernel are determined by $\sigma_K^2 = \int t^2 K(t)dt$ which is always $= 1$ for our kernels (and hence the bandwidth `bw` is the standard deviation of the kernel) and $R(K) = \int K^2(t)dt$.
MSE-equivalent bandwidths (for different kernels) are proportional to $\sigma_K R(K)$ which is scale invariant and for our kernels equal to $R(K)$. This value is returned when `give.Rkern = TRUE`. See the examples for using exact equivalent bandwidths.

Infinite values in `x` are assumed to correspond to a point mass at `+/-Inf` and the density estimate is of the sub-density on `(-Inf, +Inf)`.

Value

If `give.Rkern` is true, the number $R(K)$, otherwise an object with class `"density"` whose underlying structure is a list containing the following components.

`x`
: the `n` coordinates of the points where the density is estimated.

`y`
: the estimated density values.

`bw`
: the bandwidth used.

n	the sample size after elimination of missing values.
call	the call which produced the result.
data.name	the deparsed name of the x argument.
has.na	logical, for compatibility (always FALSE).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole (for S version).

Scott, D. W. (1992) *Multivariate Density Estimation. Theory, Practice and Visualization*. New York: Wiley.

Sheather, S. J. and Jones M. C. (1991) A reliable data-based bandwidth selection method for kernel density estimation. *J. Roy. Statist. Soc.* **B**, 683–690.

Silverman, B. W. (1986) *Density Estimation*. London: Chapman and Hall.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*. New York: Springer.

See Also

bw.nrd, plot.density, hist.

Examples

```
plot(density(c(-20,rep(0,98),20)), xlim = c(-4,4))# IQR = 0

# The Old Faithful geyser data
d <- density(faithful$eruptions, bw = "sj")
d
plot(d)

plot(d, type = "n")
polygon(d, col = "wheat")

## Missing values:
x <- xx <- faithful$eruptions
x[i.out <- sample(length(x), 10)] <- NA
doR <- density(x, bw = 0.15, na.rm = TRUE)
lines(doR, col = "blue")
points(xx[i.out], rep(0.01, 10))

## Weighted observations:
fe <- sort(faithful$eruptions) # has quite a few non-unique values
## use 'counts / n' as weights:
dw <- density(unique(fe), weights = table(fe)/length(fe), bw = d$bw)
str(dw) ## smaller n: only 126, but identical estimate:
stopifnot(all.equal(d[1:3], dw[1:3]))

(kernels <- eval(formals(density.default)$kernel))

## show the kernels in the R parametrization
plot (density(0, bw = 1), xlab = "",
      main="R's density() kernels with bw = 1")
for(i in 2:length(kernels))
   lines(density(0, bw = 1, kern =  kernels[i]), col = i)
legend(1.5,.4, legend = kernels, col = seq(kernels),
```

```
       lty = 1, cex = .8, y.int = 1)

## show the kernels in the S parametrization
plot(density(0, from=-1.2, to=1.2, width=2, kern="gaussian"), type="l",
     ylim = c(0, 1), xlab="", main="R's density() kernels with width = 1")
for(i in 2:length(kernels))
   lines(density(0, width=2, kern =  kernels[i]), col = i)
legend(0.6, 1.0, legend = kernels, col = seq(kernels), lty = 1)

##-------- Semi-advanced theoretic from here on --------------------------

(RKs <- cbind(sapply(kernels, function(k)density(kern = k, give.Rkern = TRUE))))
100*round(RKs["epanechnikov",]/RKs, 4) ## Efficiencies

bw <- bw.SJ(precip) ## sensible automatic choice
plot(density(precip, bw = bw),
     main = "same sd bandwidths, 7 different kernels")
for(i in 2:length(kernels))
   lines(density(precip, bw = bw, kern =  kernels[i]), col = i)

## Bandwidth Adjustment for "Exactly Equivalent Kernels"
h.f <- sapply(kernels, function(k)density(kern = k, give.Rkern = TRUE))
(h.f <- (h.f["gaussian"] / h.f)^ .2)
## -> 1, 1.01, .995, 1.007,... close to 1 => adjustment barely visible..

plot(density(precip, bw = bw),
     main = "equivalent bandwidths, 7 different kernels")
for(i in 2:length(kernels))
   lines(density(precip, bw = bw, adjust = h.f[i], kern = kernels[i]),
         col = i)
legend(55, 0.035, legend = kernels, col = seq(kernels), lty = 1)
```

`deriv` *Symbolic and Algorithmic Derivatives of Simple Expressions*

Description

Compute derivatives of simple expressions, symbolically.

Usage

```
    D (expr, name)
 deriv(expr, ...)
deriv3(expr, ...)

 ## Default S3 method:
 deriv(expr, namevec, function.arg = NULL, tag = ".expr",
       hessian = FALSE, ...)
 ## S3 method for class 'formula':
 deriv(expr, namevec, function.arg = NULL, tag = ".expr",
       hessian = FALSE, ...)

## Default S3 method:
```

```
deriv3(expr, namevec, function.arg = NULL, tag = ".expr",
       hessian = TRUE, ...)
## S3 method for class 'formula':
deriv3(expr, namevec, function.arg = NULL, tag = ".expr",
       hessian = TRUE, ...)
```

Arguments

`expr`	A `expression` or `call` or (except `D`) a formula with no lhs.
`name, namevec`	character vector, giving the variable names (only one for `D()`) with respect to which derivatives will be computed.
`function.arg`	If specified and non-`NULL`, a character vector of arguments for a function return, or a function (with empty body) or `TRUE`, the latter indicating that a function with argument names `namevec` should be used.
`tag`	character; the prefix to be used for the locally created variables in result.
`hessian`	a logical value indicating whether the second derivatives should be calculated and incorporated in the return value.
`...`	arguments to tbe passed to or from methods.

Details

`D` is modelled after its S namesake for taking simple symbolic derivatives.

`deriv` is a *generic* function with a default and a `formula` method. It returns a `call` for computing the `expr` and its (partial) derivatives, simultaneously. It uses so-called "*algorithmic derivatives*". If `function.arg` is a function, its arguments can have default values, see the `fx` example below.

Currently, `deriv.formula` just calls `deriv.default` after extracting the expression to the right of `~`.

`deriv3` and its methods are equivalent to `deriv` and its methods except that `hessian` defaults to `TRUE` for `deriv3`.

The internal code knows about the arithmetic operators `+`, `-`, `*`, `/` and `^`, and the single-variable functions `exp`, `log`, `sin`, `cos`, `tan`, `sinh`, `cosh`, `sqrt`, `pnorm`, `dnorm`, `asin`, `acos`, `atan`, `gamma` and `lgamma`. (Note that only the standard normal distribution is considered.)

Value

`D` returns a call and therefore can easily be iterated for higher derivatives.

`deriv` and `deriv3` normally return an `expression` object whose evaluation returns the function values with a `"gradient"` attribute containing the gradient matrix. If `hessian` is `TRUE` the evaluation also returns a `"hessian"` attribute containing the Hessian array.

If `function.arg` is not `NULL`, `deriv` and `deriv3` return a function with those arguments rather than an expression.

References

Griewank, A. and Corliss, G. F. (1991) *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. SIAM proceedings, Philadelphia.

Bates, D. M. and Chambers, J. M. (1992) *Nonlinear models.* Chapter 10 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

nlm and optim for numeric minimization which could make use of derivatives,

Examples

```
## formula argument :
dx2x <- deriv(~ x^2, "x") ; dx2x
## Not run:
expression({
         .value <- x^2
         .grad <- array(0, c(length(.value), 1), list(NULL, c("x")))
         .grad[, "x"] <- 2 * x
         attr(.value, "gradient") <- .grad
         .value
})
## End(Not run)
mode(dx2x)
x <- -1:2
eval(dx2x)

## Something 'tougher':
trig.exp <- expression(sin(cos(x + y^2)))
( D.sc <- D(trig.exp, "x") )
all.equal(D(trig.exp[[1]], "x"), D.sc)

( dxy <- deriv(trig.exp, c("x", "y")) )
y <- 1
eval(dxy)
eval(D.sc)

## function returned:
deriv((y ~ sin(cos(x) * y)), c("x","y"), func = TRUE)

## function with defaulted arguments:
(fx <- deriv(y ~ b0 + b1 * 2^(-x/th), c("b0", "b1", "th"),
             function(b0, b1, th, x = 1:7){} ) )
fx(2,3,4)

## Higher derivatives
deriv3(y ~ b0 + b1 * 2^(-x/th), c("b0", "b1", "th"),
     c("b0", "b1", "th", "x") )

## Higher derivatives:
DD <- function(expr,name, order = 1) {
   if(order < 1) stop("'order' must be >= 1")
   if(order == 1) D(expr,name)
   else DD(D(expr, name), name, order - 1)
}
DD(expression(sin(x^2)), "x", 3)
## showing the limits of the internal "simplify()" :
## Not run:
-sin(x^2) * (2 * x) * 2 + ((cos(x^2) * (2 * x) * (2 * x) + sin(x^2) *
    2) * (2 * x) + sin(x^2) * (2 * x) * 2)
## End(Not run)
```

deviance *Model Deviance*

Description

Returns the deviance of a fitted model object.

Usage

```
deviance(object, ...)
```

Arguments

object	an object for which the deviance is desired.
...	additional optional argument.

Details

This is a generic function which can be used to extract deviances for fitted models. Consult the individual modeling functions for details on how to use this function.

Value

The value of the deviance extracted from the object object.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S.* Wadsworth & Brooks/Cole.

See Also

df.residual, extractAIC, glm, lm.

df.residual *Residual Degrees-of-Freedom*

Description

Returns the residual degrees-of-freedom extracted from a fitted model object.

Usage

```
df.residual(object, ...)
```

Arguments

object	an object for which the degrees-of-freedom are desired.
...	additional optional arguments.

Details

This is a generic function which can be used to extract residual degrees-of-freedom for fitted models. Consult the individual modeling functions for details on how to use this function.

The default method just extracts the `df.residual` component.

Value

The value of the residual degrees-of-freedom extracted from the object `x`.

See Also

`deviance`, `glm`, `lm`.

`diffinv` *Discrete Integration: Inverse of Differencing*

Description

Computes the inverse function of the lagged differences function `diff`.

Usage

```
diffinv(x, ...)

## Default S3 method:
diffinv(x, lag = 1, differences = 1, xi, ...)
## S3 method for class 'ts':
diffinv(x, lag = 1, differences = 1, xi, ...)
```

Arguments

`x`	a numeric vector, matrix, or time series.
`lag`	a scalar lag parameter.
`differences`	an integer representing the order of the difference.
`xi`	a numeric vector, matrix, or time series containing the initial values for the integrals. If missing, zeros are used.
`...`	arguments passed to or from other methods.

Details

`diffinv` is a generic function with methods for class `"ts"` and `default` for vectors and matrices.

Missing values are not handled.

Value

A numeric vector, matrix, or time series (the latter for the `"ts"` method) representing the discrete integral of `x`.

Author(s)

A. Trapletti

See Also

`diff`

Examples

```
s <- 1:10
d <- diff(s)
diffinv(d, xi = 1)
```

`dist` *Distance Matrix Computation*

Description

This function computes and returns the distance matrix computed by using the specified distance measure to compute the distances between the rows of a data matrix.

Usage

```
dist(x, method = "euclidean", diag = FALSE, upper = FALSE, p = 2)

as.dist(m, diag = FALSE, upper = FALSE)
## Default S3 method:
as.dist(m, diag = FALSE, upper = FALSE)

## S3 method for class 'dist':
print(x, diag = NULL, upper = NULL,
      digits = getOption("digits"), justify = "none", right = TRUE, ...)

## S3 method for class 'dist':
as.matrix(x)
```

Arguments

`x`	a numeric matrix, data frame or `"dist"` object.
`method`	the distance measure to be used. This must be one of `"euclidean"`, `"maximum"`, `"manhattan"`, `"canberra"`, `"binary"` or `"minkowski"`. Any unambiguous substring can be given.
`diag`	logical value indicating whether the diagonal of the distance matrix should be printed by `print.dist`.
`upper`	logical value indicating whether the upper triangle of the distance matrix should be printed by `print.dist`.
`p`	The power of the Minkowski distance.
`m`	An object with distance information to be converted to a `"dist"` object. For the default method, a `"dist"` object, or a matrix (of distances) or an object which can be coerced to such a matrix using `as.matrix()`. (Only the lower triangle of the matrix is used, the rest is ignored).

`digits, justify`
: passed to `format` inside of `print()`.

`right, ...`
: further arguments, passed to the (next) `print` method.

Details

Available distance measures are (written for two vectors x and y):

`euclidean`: Usual square distance between the two vectors (2 norm).

`maximum`: Maximum distance between two components of x and y (supremum norm)

`manhattan`: Absolute distance between the two vectors (1 norm).

`canberra`: $\sum_i |x_i - y_i| / |x_i + y_i|$. Terms with zero numerator and denominator are omitted from the sum and treated as if the values were missing.

`binary`: (aka *asymmetric binary*): The vectors are regarded as binary bits, so non-zero elements are 'on' and zero elements are 'off'. The distance is the *proportion* of bits in which only one is on amongst those in which at least one is on.

`minkowski`: The p norm, the pth root of the sum of the pth powers of the differences of the components.

Missing values are allowed, and are excluded from all computations involving the rows within which they occur. Further, when `Inf` values are involved, all pairs of values are excluded when their contribution to the distance gave `NaN` or `NA`.
If some columns are excluded in calculating a Euclidean, Manhattan, Canberra or Minkowski distance, the sum is scaled up proportionally to the number of columns used. If all pairs are excluded when calculating a particular distance, the value is `NA`.

The `"dist"` method of `as.matrix()` and `as.dist()` can be used for conversion between objects of class `"dist"` and conventional distance matrices.

`as.dist()` is a generic function. Its default method handles objects inheriting from class `"dist"`, or coercible to matrices using `as.matrix()`. Support for classes representing distances (also known as dissimilarities) can be added by providing an `as.matrix()` or, more directly, an `as.dist` method for such a class.

Value

`dist` returns an object of class `"dist"`.

The lower triangle of the distance matrix stored by columns in a vector, say `do`. If `n` is the number of observations, i.e., `n <- attr(do, "Size")`, then for $i < j <= n$, the dissimilarity between (row) i and j is `do[n*(i-1) - i*(i-1)/2 + j-i]`. The length of the vector is $n*(n-1)/2$, i.e., of order n^2.

The object has the following attributes (besides `"class"` equal to `"dist"`):

`Size`
: integer, the number of observations in the dataset.

`Labels`
: optionally, contains the labels, if any, of the observations of the dataset.

`Diag, Upper`
: logicals corresponding to the arguments `diag` and `upper` above, specifying how the object should be printed.

`call`
: optionally, the `call` used to create the object.

`method`
: optionally, the distance method used; resulting from `dist()`, the (`match.arg()`ed) `method` argument.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Mardia, K. V., Kent, J. T. and Bibby, J. M. (1979) *Multivariate Analysis.* Academic Press.

Borg, I. and Groenen, P. (1997) *Modern Multidimensional Scaling. Theory and Applications.* Springer.

See Also

`daisy` in the **cluster** package with more possibilities in the case of *mixed* (continuous / categorical) variables. `hclust`.

Examples

```
x <- matrix(rnorm(100), nrow=5)
dist(x)
dist(x, diag = TRUE)
dist(x, upper = TRUE)
m <- as.matrix(dist(x))
d <- as.dist(m)
stopifnot(d == dist(x))

## Use correlations between variables ``as distance''
dd <- as.dist((1 - cor(USJudgeRatings))/2)
round(1000 * dd) # (prints more nicely)
plot(hclust(dd)) # to see a dendrogram of clustered variables

## example of binary and canberra distances.
x <- c(0, 0, 1, 1, 1, 1)
y <- c(1, 0, 1, 1, 0, 1)
dist(rbind(x,y), method= "binary")
## answer 0.4 = 2/5
dist(rbind(x,y), method= "canberra")
## answer 2 * (6/5)

## To find the names
labels(eurodist)

## Examples involving "Inf" :
## 1)
x[6] <- Inf
(m2 <- rbind(x,y))
dist(m2, method="binary")# warning, answer 0.5 = 2/4
## These all give "Inf":
stopifnot(Inf == dist(m2, method= "euclidean"),
          Inf == dist(m2, method= "maximum"),
          Inf == dist(m2, method= "manhattan"))
##  "Inf" is same as very large number:
x1 <- x; x1[6] <- 1e100
stopifnot(dist(cbind(x ,y), method="canberra") ==
    print(dist(cbind(x1,y), method="canberra")))

## 2)
y[6] <- Inf #-> 6-th pair is excluded
dist(rbind(x,y), method="binary")   # warning; 0.5
```

```
dist(rbind(x,y), method="canberra") # 3
dist(rbind(x,y), method="maximum")  # 1
dist(rbind(x,y), method="manhattan")# 2.4
```

`dummy.coef` *Extract Coefficients in Original Coding*

Description

This extracts coefficients in terms of the original levels of the coefficients rather than the coded variables.

Usage

```
dummy.coef(object, ...)

## S3 method for class 'lm':
dummy.coef(object, use.na = FALSE, ...)

## S3 method for class 'aovlist':
dummy.coef(object, use.na = FALSE, ...)
```

Arguments

`object`	a linear model fit.
`use.na`	logical flag for coefficients in a singular model. If `use.na` is true, undetermined coefficients will be missing; if false they will get one possible value.
`...`	arguments passed to or from other methods.

Details

A fitted linear model has coefficients for the contrasts of the factor terms, usually one less in number than the number of levels. This function re-expresses the coefficients in the original coding; as the coefficients will have been fitted in the reduced basis, any implied constraints (e.g., zero sum for `contr.helmert` or `contr.sum` will be respected. There will be little point in using `dummy.coef` for `contr.treatment` contrasts, as the missing coefficients are by definition zero.

The method used has some limitations, and will give incomplete results for terms such as `poly(x, 2))`. However, it is adequate for its main purpose, `aov` models.

Value

A list giving for each term the values of the coefficients. For a multistratum `aov` model, such a list for each stratum.

Warning

This function is intended for human inspection of the output: it should not be used for calculations. Use coded variables for all calculations.

The results differ from S for singular values, where S can be incorrect.

See Also

aov, model.tables

Examples

```
options(contrasts=c("contr.helmert", "contr.poly"))
## From Venables and Ripley (2002) p.165.
data(npk, package="MASS")
npk.aov <- aov(yield ~ block + N*P*K, npk)
dummy.coef(npk.aov)

npk.aovE <- aov(yield ~  N*P*K + Error(block), npk)
dummy.coef(npk.aovE)
```

---

ecdf *Empirical Cumulative Distribution Function*

Description

Compute or plot an empirical cumulative distribution function.

Usage

```
ecdf(x)

## S3 method for class 'ecdf':
plot(x, ..., ylab="Fn(x)", verticals = FALSE,
     col.01line = "gray70")

## S3 method for class 'ecdf':
print(x, digits= getOption("digits") - 2, ...)
```

Arguments

x	numeric vector of "observations" in ecdf; for the methods, an object of class "ecdf", typically.
...	arguments to be passed to subsequent methods, i.e., plot.stepfun for the plot method.
ylab	label for the y-axis.
verticals	see plot.stepfun.
col.01line	numeric or character specifying the color of the horizontal lines at y = 0 and 1, see colors.
digits	number of significant digits to use, see print.

Details

The e.c.d.f. (empirical cumulative distribution function) F_n is a step function with jumps i/n at observation values, where i is the number of tied observations at that value. Missing values are ignored.

For observations `x`$= (x_1, x_2, \ldots x_n)$, F_n is the fraction of observations less or equal to t, i.e.,

$$F_n(t) = \#\{x_i \le t\}\ /n = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}_{[x_i \le t]}.$$

The function `plot.ecdf` which implements the `plot` method for `ecdf` objects, is implemented via a call to `plot.stepfun`; see its documentation.

Value

For `ecdf`, a function of class `"ecdf"`, inheriting from the `"stepfun"` class.

Author(s)

Martin Maechler, ⟨maechler@stat.math.ethz.ch⟩.
Corrections by R-core.

See Also

`stepfun`, the more general class of step functions, `approxfun` and `splinefun`.

Examples

```
##-- Simple didactical  ecdf  example:
Fn <- ecdf(rnorm(12))
Fn
tt <- seq(-2,2, by = 0.1)
12* Fn(tt) # Fn is a `simple' function {with values k/12}
summary(Fn)
knots(Fn)# the unique data values {12 of them if there were no ties}

y <- round(rnorm(12),1); y[3] <- y[1]
Fn12 <- ecdf(y)
Fn12
knots(Fn12)# unique values (always less than 12!)
summary(Fn12)
summary.stepfun(Fn12)

## Advanced: What's inside the function closure?
print(ls.Fn12 <- ls(env= environment(Fn12)))
##[1] "f"  "method"  "n"  "x"  "y"  "yleft"  "yright"
utils::ls.str(environment(Fn12))

###----------------- Plotting --------------------------

op <- par(mfrow=c(3,1), mgp=c(1.5, 0.8,0), mar= .1+c(3,3,2,1))

F10 <- ecdf(rnorm(10))
summary(F10)

plot(F10)
```

```
plot(F10, verticals= TRUE, do.p = FALSE)

plot(Fn12 , lwd = 2) ; mtext("lwd = 2", adj=1)
xx <- unique(sort(c(seq(-3,2, length=201), knots(Fn12))))
lines(xx, Fn12(xx), col='blue')
abline(v=knots(Fn12),lty=2,col='gray70')

plot(xx, Fn12(xx), type='o', cex=.1)#- plot.default {ugly}
plot(Fn12, col.h='red', add= TRUE)  #- plot method
abline(v=knots(Fn12),lty=2,col='gray70')
## luxury plot
plot(Fn12, verticals=TRUE, col.p='blue', col.h='red',col.v='bisque')

##-- this works too (automatic call to  ecdf(.)):
plot.ecdf(rnorm(24))
title("via  simple  plot.ecdf(x)", adj=1)

par(op)
```

`eff.aovlist` *Compute Efficiencies of Multistratum Analysis of Variance*

Description

Computes the efficiencies of fixed-effect terms in an analysis of variance model with multiple strata.

Usage

```
eff.aovlist(aovlist)
```

Arguments

`aovlist`	The result of a call to `aov` with an `Error` term.

Details

Fixed-effect terms in an analysis of variance model with multiple strata may be estimable in more than one stratum, in which case there is less than complete information in each. The efficiency for a term is the fraction of the maximum possible precision (inverse variance) obtainable by estimating in just that stratum. Under the assumption of balance, this is the same for all contrasts involving that term.

This function is used to pick strata in which to estimate terms in `model.tables.aovlist` and `se.contrast.aovlist`.

In many cases terms will only occur in one stratum, when all the efficiencies will be one: this is detected and no further calculations are done.

The calculation used requires orthogonal contrasts for each term, and will throw an error if non-orthogonal contrasts (e.g. treatment contrasts or an unbalanced design) are detected.

Value

A matrix giving for each non-pure-error stratum (row) the efficiencies for each fixed-effect term in the model.

References

Heiberger, R. M. (1989) *Computation for the Analysis of Designed Experiments*. Wiley.

See Also

aov, model.tables.aovlist, se.contrast.aovlist

Examples

```
## An example of Yates (1932), a 2^3 design in 2 blocks replicated 4 times

Block <- gl(8, 4)
A<-factor(c(0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1))
B<-factor(c(0,0,1,1,0,0,1,1,0,1,0,1,1,0,1,0,0,0,1,1,0,0,1,1,0,0,1,1))
C<-factor(c(0,1,1,0,1,0,0,1,0,0,1,1,0,0,1,1,0,1,0,1,1,0,1,0,0,0,1,1,1,1,0,0))
Yield <- c(101, 373, 398, 291, 312, 106, 265, 450, 106, 306, 324, 449,
           272, 89, 407, 338, 87, 324, 279, 471, 323, 128, 423, 334,
           131, 103, 445, 437, 324, 361, 302, 272)
aovdat <- data.frame(Block, A, B, C, Yield)

old <- getOption("contrasts")
options(contrasts=c("contr.helmert", "contr.poly"))
(fit <- aov(Yield ~ A*B*C + Error(Block), data = aovdat))
eff.aovlist(fit)
options(contrasts = old)
```

---

effects *Effects from Fitted Model*

Description

Returns (orthogonal) effects from a fitted model, usually a linear model. This is a generic function, but currently only has a methods for objects inheriting from classes "lm" and "glm".

Usage

```
effects(object, ...)

## S3 method for class 'lm':
effects(object, set.sign = FALSE, ...)
```

Arguments

object	an R object; typically, the result of a model fitting function such as lm.
set.sign	logical. If TRUE, the sign of the effects corresponding to coefficients in the model will be set to agree with the signs of the corresponding coefficients, otherwise the sign is arbitrary.
...	arguments passed to or from other methods.

Details

For a linear model fitted by `lm` or `aov`, the effects are the uncorrelated single-degree-of-freedom values obtained by projecting the data onto the successive orthogonal subspaces generated by the QR decomposition during the fitting process. The first r (the rank of the model) are associated with coefficients and the remainder span the space of residuals (but are not associated with particular residuals).

Empty models do not have effects.

Value

A (named) numeric vector of the same length as `residuals`, or a matrix if there were multiple responses in the fitted model, in either case of class `"coef"`.

The first r rows are labelled by the corresponding coefficients, and the remaining rows are unlabelled. Note that in rank-deficient models the "corresponding" coefficients will be in a different order if pivoting occurred.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S.* Wadsworth & Brooks/Cole.

See Also

`coef`

Examples

```
y <- c(1:3,7,5)
x <- c(1:3,6:7)
( ee <- effects(lm(y ~ x)) )
c(round(ee - effects(lm(y+10 ~ I(x-3.8))),3))# just the first is different
```

`embed` *Embedding a Time Series*

Description

Embeds the time series `x` into a low-dimensional Euclidean space.

Usage

```
embed (x, dimension = 1)
```

Arguments

`x`	a numeric vector, matrix, or time series.
`dimension`	a scalar representing the embedding dimension.

Details

Each row of the resulting matrix consists of sequences `x[t]`, `x[t-1]`, ..., `x[t-dimension+1]`, where `t` is the original index of `x`. If `x` is a matrix, i.e., `x` contains more than one variable, then `x[t]` consists of the `t`th observation on each variable.

Value

A matrix containing the embedded time series `x`.

Author(s)

A. Trapletti, B.D. Ripley

Examples

```
x <- 1:10
embed (x, 3)
```

`expand.model.frame` *Add new variables to a model frame*

Description

Evaluates new variables as if they had been part of the formula of the specified model. This ensures that the same `na.action` and `subset` arguments are applied and allows, for example, `x` to be recovered for a model using `sin(x)` as a predictor.

Usage

```
expand.model.frame(model, extras,
                   envir = environment(formula(model)),
                   na.expand = FALSE)
```

Arguments

`model`	a fitted model
`extras`	one-sided formula or vector of character strings describing new variables to be added
`envir`	an environment to evaluate things in
`na.expand`	logical; see below

Details

If `na.expand=FALSE` then `NA` values in the extra variables will be passed to the `na.action` function used in `model`. This may result in a shorter data frame (with `na.omit`) or an error (with `na.fail`). If `na.expand=TRUE` the returned data frame will have precisely the same rows as `model.frame(model)`, but the columns corresponding to the extra variables may contain `NA`.

Value

A data frame.

See Also

`model.frame`,`predict`

Examples

```
model <- lm(log(Volume) ~ log(Girth) + log(Height), data=trees)
expand.model.frame(model, ~ Girth) # prints data.frame like

dd <- data.frame(x=1:5, y=rnorm(5), z=c(1,2,NA,4,5))
model <- glm(y ~ x, data=dd, subset=1:4, na.action=na.omit)
expand.model.frame(model, "z", na.expand=FALSE) # = default
expand.model.frame(model, "z", na.expand=TRUE)
```

Exponential *The Exponential Distribution*

Description

Density, distribution function, quantile function and random generation for the exponential distribution with rate `rate` (i.e., mean `1/rate`).

Usage

```
dexp(x, rate = 1, log = FALSE)
pexp(q, rate = 1, lower.tail = TRUE, log.p = FALSE)
qexp(p, rate = 1, lower.tail = TRUE, log.p = FALSE)
rexp(n, rate = 1)
```

Arguments

`x, q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`rate`	vector of rates.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

If `rate` is not specified, it assumes the default value of `1`.

The exponential distribution with rate λ has density

$$f(x) = \lambda e^{-\lambda x}$$

for $x \geq 0$.

Value

`dexp` gives the density, `pexp` gives the distribution function, `qexp` gives the quantile function, and `rexp` generates random deviates.

Note

The cumulative hazard $H(t) = -\log(1 - F(t))$ is `-pexp(t, r, lower = FALSE, log = TRUE)`.

Source

`dexp`, `pexp` and `qexp` are all calculated from numerically stable versions of the definitions.

`rexp` uses

Ahrens, J. H. and Dieter, U. (1972). Computer methods for sampling from the exponential and normal distributions. *Communications of the ACM*, **15**, 873–882.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, volume 1, chapter 19. Wiley, New York.

See Also

`exp` for the exponential function, `dgamma` for the gamma distribution and `dweibull` for the Weibull distribution, both of which generalize the exponential.

Examples

```
dexp(1) - exp(-1) #-> 0
```

`extractAIC` *Extract AIC from a Fitted Model*

Description

Computes the (generalized) Akaike **A**n **I**nformation **C**riterion for a fitted parametric model.

Usage

```
extractAIC(fit, scale, k = 2, ...)
```

Arguments

`fit`	fitted model, usually the result of a fitter like `lm`.
`scale`	optional numeric specifying the scale parameter of the model, see `scale` in `step`. Currently only used in the `"lm"` method, where `scale` specifies the estimate of the error variance, and `scale = 0` indicates that it is to be estimated by maximum likelihood.
`k`	numeric specifying the "weight" of the *equivalent degrees of freedom* ($\equiv$ `edf`) part in the AIC formula.
`...`	further arguments (currently unused in base R).

Details

This is a generic function, with methods in base R for `"aov"`, `"coxph"`, `"glm"`, `"lm"`, `"negbin"` and `"survreg"` classes.

The criterion used is

$$AIC = -2\log L + k \times \text{edf},$$

where L is the likelihood and `edf` the equivalent degrees of freedom (i.e., the number of free parameters for usual parametric models) of `fit`.

For linear models with unknown scale (i.e., for `lm` and `aov`), $-2\log L$ is computed from the *deviance* and uses a different additive constant to `logLik` and hence `AIC`. If RSS denotes the (weighted) residual sum of squares then `extractAIC` uses for $-2\log L$ the formulae $RSS/s - n$ (corresponding to Mallows' C_p) in the case of known scale s and $n\log(RSS/n)$ for unknown scale. `AIC` only handles unknown scale and uses the formula $n\log(RSS/n) - n + n\log 2\pi - \sum \log w$ where w are the weights.

For `glm` fits the family's `aic()` function to compute the AIC: see the note under `logLik` about the assumptions this makes.

`k = 2` corresponds to the traditional AIC, using `k = log(n)` provides the BIC (Bayesian IC) instead.

Value

A numeric vector of length 2, giving

`edf`	the "**e**quivalent **d**egrees of **f**reedom" for the fitted model `fit`.
`AIC`	the (generalized) Akaike Information Criterion for `fit`.

Note

This function is used in `add1`, `drop1` and `step` and similar functions in package **MASS** from which it was adopted.

Author(s)

B. D. Ripley

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* New York: Springer (4th ed).

See Also

`AIC`, `deviance`, `add1`, `step`

Examples

```
example(glm)
extractAIC(glm.D93)#>>  5  15.129
```

`factanal` *Factor Analysis*

Description

Perform maximum-likelihood factor analysis on a covariance matrix or data matrix.

Usage

```
factanal(x, factors, data = NULL, covmat = NULL, n.obs = NA,
         subset, na.action,
         start = NULL, scores = c("none", "regression", "Bartlett"),
         rotation = "varimax", control = NULL, ...)
```

Arguments

`x`	A formula or a numeric matrix or an object that can be coerced to a numeric matrix.
`factors`	The number of factors to be fitted.
`data`	An optional data frame (or similar: see `model.frame`), used only if `x` is a formula. By default the variables are taken from `environment(formula)`.
`covmat`	A covariance matrix, or a covariance list as returned by `cov.wt`. Of course, correlation matrices are covariance matrices.
`n.obs`	The number of observations, used if `covmat` is a covariance matrix.
`subset`	A specification of the cases to be used, if `x` is used as a matrix or formula.
`na.action`	The `na.action` to be used if `x` is used as a formula.
`start`	`NULL` or a matrix of starting values, each column giving an initial set of uniquenesses.
`scores`	Type of scores to produce, if any. The default is none, `"regression"` gives Thompson's scores, `"Bartlett"` given Bartlett's weighted least-squares scores. Partial matching allows these names to be abbreviated.
`rotation`	character. `"none"` or the name of a function to be used to rotate the factors: it will be called with first argument the loadings matrix, and should return a list with component `loadings` giving the rotated loadings, or just the rotated loadings.
`control`	A list of control values, **nstart** The number of starting values to be tried if `start = NULL`. Default 1. **trace** logical. Output tracing information? Default `FALSE`. **lower** The lower bound for uniquenesses during optimization. Should be > 0. Default 0.005. **opt** A list of control values to be passed to `optim`'s `control` argument. **rotate** a list of additional arguments for the rotation function.
`...`	Components of `control` can also be supplied as named arguments to `factanal`.

Details

The factor analysis model is

$$x = \Lambda f + e$$

for a p–element row-vector x, a $p \times k$ matrix of *loadings*, a k–element vector of *scores* and a p–element vector of errors. None of the components other than x is observed, but the major restriction is that the scores be uncorrelated and of unit variance, and that the errors be independent with variances Φ, the *uniquenesses*. Thus factor analysis is in essence a model for the covariance matrix of x,

$$\Sigma = \Lambda'\Lambda + \Psi$$

There is still some indeterminacy in the model for it is unchanged if Λ is replaced by $G\Lambda$ for any orthogonal matrix G. Such matrices G are known as *rotations* (although the term is applied also to non-orthogonal invertible matrices).

If `covmat` is supplied it is used. Otherwise `x` is used if it is a matrix, or a formula `x` is used with `data` to construct a model matrix, and that is used to construct a covariance matrix. (It makes no sense for the formula to have a response, and all the variables must be numeric.) Once a covariance matrix is found or calculated from `x`, it is converted to a correlation matrix for analysis. The correlation matrix is returned as component `correlation` of the result.

The fit is done by optimizing the log likelihood assuming multivariate normality over the uniquenesses. (The maximizing loadings for given uniquenesses can be found analytically: Lawley & Maxwell (1971, p. 27).) All the starting values supplied in `start` are tried in turn and the best fit obtained is used. If `start = NULL` then the first fit is started at the value suggested by Jöreskog (1963) and given by Lawley & Maxwell (1971, p. 31), and then `control$nstart - 1` other values are tried, randomly selected as equal values of the uniquenesses.

The uniquenesses are technically constrained to lie in $[0, 1]$, but near-zero values are problematical, and the optimization is done with a lower bound of `control$lower`, default 0.005 (Lawley & Maxwell, 1971, p. 32).

Scores can only be produced if a data matrix is supplied and used. The first method is the regression method of Thomson (1951), the second the weighted least squares method of Bartlett (1937, 8). Both are estimates of the unobserved scores f. Thomson's method regresses (in the population) the unknown f on x to yield

$$\hat{f} = \Lambda'\Sigma^{-1}x$$

and then substitutes the sample estimates of the quantities on the right-hand side. Bartlett's method minimizes the sum of squares of standardized errors over the choice of f, given (the fitted) Λ.

If `x` is a formula then the standard NA-handling is applied to the scores (if requested): see `napredict`.

Value

An object of class `"factanal"` with components

`loadings`	A matrix of loadings, one column for each factor. The factors are ordered in decreasing order of sums of squares of loadings, and given the sign that will make the sum of the loadings positive.
`uniquenesses`	The uniquenesses computed.
`correlation`	The correlation matrix used.
`criteria`	The results of the optimization: the value of the negative log-likelihood and information on the iterations used.
`factors`	The argument `factors`.

dof	The number of degrees of freedom of the factor analysis model.
method	The method: always "mle".
scores	If requested, a matrix of scores. napredict is applied to handle the treatment of values omitted by the na.action.
n.obs	The number of observations if available, or NA.
call	The matched call.
na.action	If relevant.
STATISTIC, PVAL	The significance-test statistic and P value, if if can be computed.

Note

There are so many variations on factor analysis that it is hard to compare output from different programs. Further, the optimization in maximum likelihood factor analysis is hard, and many other examples we compared had less good fits than produced by this function. In particular, solutions which are Heywood cases (with one or more uniquenesses essentially zero) are much often common than most texts and some other programs would lead one to believe.

References

Bartlett, M. S. (1937) The statistical conception of mental factors. *British Journal of Psychology*, **28**, 97–104.

Bartlett, M. S. (1938) Methods of estimating mental factors. *Nature*, **141**, 609–610.

Jöreskog, K. G. (1963) *Statistical Estimation in Factor Analysis.* Almqvist and Wicksell.

Lawley, D. N. and Maxwell, A. E. (1971) *Factor Analysis as a Statistical Method.* Second edition. Butterworths.

Thomson, G. H. (1951) *The Factorial Analysis of Human Ability.* London University Press.

See Also

print.loadings, varimax, princomp, ability.cov, Harman23.cor, Harman74.cor

Examples

```
# A little demonstration, v2 is just v1 with noise,
# and same for v4 vs. v3 and v6 vs. v5
# Last four cases are there to add noise
# and introduce a positive manifold (g factor)
v1 <- c(1,1,1,1,1,1,1,1,1,1,3,3,3,3,3,4,5,6)
v2 <- c(1,2,1,1,1,1,2,1,2,1,3,4,3,3,3,4,6,5)
v3 <- c(3,3,3,3,3,1,1,1,1,1,1,1,1,1,1,5,4,6)
v4 <- c(3,3,4,3,3,1,1,2,1,1,1,1,2,1,1,5,6,4)
v5 <- c(1,1,1,1,1,3,3,3,3,3,1,1,1,1,1,6,4,5)
v6 <- c(1,1,1,2,1,3,3,3,4,3,1,1,1,2,1,6,5,4)
m1 <- cbind(v1,v2,v3,v4,v5,v6)
cor(m1)
factanal(m1, factors=3) # varimax is the default
factanal(m1, factors=3, rotation="promax")
# The following shows the g factor as PC1
prcomp(m1)
```

```
## formula interface
factanal(~v1+v2+v3+v4+v5+v6, factors = 3,
         scores = "Bartlett")$scores

## a realistic example from Barthlomew (1987, pp. 61-65)
example(ability.cov)
```

`factor.scope` *Compute Allowed Changes in Adding to or Dropping from a Formula*

Description

`add.scope` and `drop.scope` compute those terms that can be individually added to or dropped from a model while respecting the hierarchy of terms.

Usage

```
add.scope(terms1, terms2)

drop.scope(terms1, terms2)

factor.scope(factor, scope)
```

Arguments

`terms1`	the terms or formula for the base model.
`terms2`	the terms or formula for the upper (`add.scope`) or lower (`drop.scope`) scope. If missing for `drop.scope` it is taken to be the null formula, so all terms (except any intercept) are candidates to be dropped.
`factor`	the `"factor"` attribute of the terms of the base object.
`scope`	a list with one or both components `drop` and `add` giving the `"factor"` attribute of the lower and upper scopes respectively.

Details

`factor.scope` is not intended to be called directly by users.

Value

For `add.scope` and `drop.scope` a character vector of terms labels. For `factor.scope`, a list with components `drop` and `add`, character vectors of terms labels.

See Also

`add1`, `drop1`, `aov`, `lm`

Examples

```
add.scope( ~ a + b + c + a:b,  ~ (a + b + c)^3)
# [1] "a:c" "b:c"
drop.scope( ~ a + b + c + a:b)
# [1] "c"   "a:b"
```

family *Family Objects for Models*

Description

Family objects provide a convenient way to specify the details of the models used by functions such as glm. See the documentation for glm for the details on how such model fitting takes place.

Usage

```
family(object, ...)

binomial(link = "logit")
gaussian(link = "identity")
Gamma(link = "inverse")
inverse.gaussian(link = "1/mu^2")
poisson(link = "log")
quasi(link = "identity", variance = "constant")
quasibinomial(link = "logit")
quasipoisson(link = "log")
```

Arguments

link	a specification for the model link function. This can be a name/expression, a literal character string, a length-one character vector or an object of class "link-glm" (provided it is not specified via one of the standard names given next). The gaussian family accepts the links "identity", "log" and "inverse"; the binomial family the links "logit", "probit", "cauchit", (corresponding to logistic, normal and Cauchy CDFs respectively) "log" and "cloglog" (complementary log-log); the Gamma family the links "inverse", "identity" and "log"; the poisson family the links "log", "identity", and "sqrt" and the inverse.gaussian family the links "1/mu^2", "inverse", "identity" and "log". The quasi family accepts the links "logit", "probit", "cloglog", "identity", "inverse", "log", "1/mu^2" and "sqrt", and the function power can be used to create a power link function.
variance	for all families other than quasi, the variance function is determined by the family. The quasi family will accept the literal character string (or unquoted as a name/expression) specifications "constant", "mu(1-mu)", "mu", "mu^2" and "mu^3", a length-one character vector taking one of those values, or a list containing components varfun, validmu, dev.resids, initialize and name.
object	the function family accesses the family objects which are stored within objects created by modelling functions (e.g., glm).
...	further arguments passed to methods.

Details

family is a generic function with methods for classes "glm" and "lm" (the latter returning gaussian()).

The quasibinomial and quasipoisson families differ from the binomial and poisson families only in that the dispersion parameter is not fixed at one, so they can "model" over-dispersion. For the binomial case see McCullagh and Nelder (1989, pp. 124–8). Although they show that there is (under some restrictions) a model with variance proportional to mean as in the quasi-binomial model, note that glm does not compute maximum-likelihood estimates in that model. The behaviour of S is closer to the quasi- variants.

Value

An object of class "family" (which has a concise print method). This is a list with elements

family	character: the family name.
link	character: the link name.
linkfun	function: the link.
linkinv	function: the inverse of the link function.
variance	function: the variance as a function of the mean.
dev.resids	function giving the deviance residuals as a function of (y, mu, wt).
aic	function giving the AIC value if appropriate (but NA for the quasi- families). See logLik for the assumptions made about the dispersion parameter.
mu.eta	function: derivative function(eta) $d\mu/d\eta$.
initialize	expression. This needs to set up whatever data objects are needed for the family as well as n (needed for AIC in the binomial family) and mustart (see glm.
valid.mu	logical function. Returns TRUE if a mean vector mu is within the domain of variance.
valid.eta	logical function. Returns TRUE if a linear predictor eta is within the domain of linkinv.

Note

The link and variance arguments have rather awkward semantics for back-compatibility. The recommended way is to supply them is as quoted character strings, but they can also be supplied unquoted (as names or expressions). In addition, they can also be supplied as a length-one character vector giving the name of one of the options, or as a list (for link, of class "link-glm").

This is potentially ambiguous: supplying link=logit could mean the unquoted name of a link or the value of object logit. It is interpreted if possible as the name of an allowed link, then as an object. (You can force the interpretation to always be the value of an object via logit[1].)

Author(s)

The design was inspired by S functions of the same names described in Hastie & Pregibon (1992) (except quasibinomial and quasipoisson).

References

McCullagh P. and Nelder, J. A. (1989) *Generalized Linear Models.* London: Chapman and Hall.

Dobson, A. J. (1983) *An Introduction to Statistical Modelling.* London: Chapman and Hall.

Cox, D. R. and Snell, E. J. (1981). *Applied Statistics; Principles and Examples.* London: Chapman and Hall.

Hastie, T. J. and Pregibon, D. (1992) *Generalized linear models.* Chapter 6 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

glm, power, make.link.

Examples

```
nf <- gaussian()# Normal family
nf
str(nf)# internal STRucture

gf <- Gamma()
gf
str(gf)
gf$linkinv
gf$variance(-3:4) #- == (.)^2

## quasipoisson. compare with example(glm)
counts <- c(18,17,15,20,10,20,25,13,12)
outcome <- gl(3,1,9)
treatment <- gl(3,3)
d.AD <- data.frame(treatment, outcome, counts)
glm.qD93 <- glm(counts ~ outcome + treatment, family=quasipoisson())
glm.qD93
anova(glm.qD93, test="F")
summary(glm.qD93)
## for Poisson results use
anova(glm.qD93, dispersion = 1, test="Chisq")
summary(glm.qD93, dispersion = 1)

## Example of user-specified link, a logit model for p^days
## See Shaffer, T.  2004. Auk 121(2): 526-540.
logexp <- function(days = 1)
{
    linkfun <- function(mu) qlogis(mu^(1/days))
    linkinv <- function(eta) plogis(eta)^days
    mu.eta <- function(eta) days * plogis(eta)^(days-1) *
      .Call("logit_mu_eta", eta, PACKAGE = "stats")
    valideta <- function(eta) TRUE
    link <- paste("logexp(", days, ")", sep="")
    structure(list(linkfun = linkfun, linkinv = linkinv,
                   mu.eta = mu.eta, valideta = valideta, name = link),
              class = "link-glm")
}
binomial(logexp(3))
## in practice this would be used with a vector of 'days', in
## which case use an offset of 0 in the corresponding formula
## to get the null deviance right.

## tests of quasi
x <- rnorm(100)
y <- rpois(100, exp(1+x))
glm(y ~x, family=quasi(var="mu", link="log"))
# which is the same as
glm(y ~x, family=poisson)
glm(y ~x, family=quasi(var="mu^2", link="log"))
## Not run: glm(y ~x, family=quasi(var="mu^3", link="log")) # should fail
y <- rbinom(100, 1, plogis(x))
```

```
# needs to set a starting value for the next fit
glm(y ~x, family=quasi(var="mu(1-mu)", link="logit"), start=c(0,1))
```

`FDist` *The F Distribution*

Description

Density, distribution function, quantile function and random generation for the F distribution with `df1` and `df2` degrees of freedom (and optional non-centrality parameter `ncp`).

Usage

```
df(x, df1, df2, ncp = 0, log = FALSE)
pf(q, df1, df2, ncp = 0, lower.tail = TRUE, log.p = FALSE)
qf(p, df1, df2, ncp = 0, lower.tail = TRUE, log.p = FALSE)
rf(n, df1, df2, ncp = 0)
```

Arguments

`x, q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`df1, df2`	degrees of freedom. `Inf` is allowed.
`ncp`	non-centrality parameter.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \le x]$, otherwise, $P[X > x]$.

Details

The F distribution with `df1` $= n_1$ and `df2` $= n_2$ degrees of freedom has density

$$f(x) = \frac{\Gamma(n_1/2 + n_2/2)}{\Gamma(n_1/2)\Gamma(n_2/2)} \left(\frac{n_1}{n_2}\right)^{n_1/2} x^{n_1/2-1} \left(1 + \frac{n_1 x}{n_2}\right)^{-(n_1+n_2)/2}$$

for $x > 0$.

It is the distribution of the ratio of the mean squares of n_1 and n_2 independent standard normals, and hence of the ratio of two independent chi-squared variates each divided by its degrees of freedom. Since the ratio of a normal and the root mean-square of m independent normals has a Student's t_m distribution, the square of a t_m variate has a F distribution on 1 and m degrees of freedom.

The non-central F distribution is again the ratio of mean squares of independent normals of unit variance, but those in the numerator are allowed to have non-zero means and `ncp` is the sum of squares of the means. See Chisquare for further details on non-central distributions.

Value

`df` gives the density, `pf` gives the distribution function `qf` gives the quantile function, and `rf` generates random deviates.

Invalid arguments will result in return value `NaN`, with a warning.

Source

For `df`, and `cdf == 0`, computed via a binomial probability, code contributed by Catherine Loader (see `dbinom`); for `cdf != 0`, computed via a `dbeta`, code contributed by Peter Ruckdeschel.

For `pf`, via `pbeta` (or for large `df2`, via `pchisq`).

For `qf`, via `qchisq` for large `df2`, else via `qbeta`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, volume 2, chapters 27 and 30. Wiley, New York.

See Also

`dchisq` for chi-squared and `dt` for Student's t distributions.

Examples

```
## the density of the square of a t_m is 2*dt(x, m)/(2*x)
# check this is the same as the density of F_{1,m}
x <- seq(0.001, 5, len=100)
all.equal(df(x^2, 1, 5), dt(x, 5)/x)

## Identity:  qf(2*p - 1, 1, df)) == qt(p, df)^2)  for  p >= 1/2
p <- seq(1/2, .99, length=50); df <- 10
rel.err <- function(x,y) ifelse(x==y,0, abs(x-y)/mean(abs(c(x,y))))
quantile(rel.err(qf(2*p - 1, df1=1, df2=df), qt(p, df)^2), .90)# ~= 7e-9
```

`fft` — *Fast Discrete Fourier Transform*

Description

Performs the Fast Fourier Transform of an array.

Usage

```
fft(z, inverse = FALSE)
mvfft(z, inverse = FALSE)
```

Arguments

`z`	a real or complex array containing the values to be transformed.
`inverse`	if `TRUE`, the unnormalized inverse transform is computed (the inverse has a + in the exponent of e, but here, we do *not* divide by `1/length(x)`).

Value

When `z` is a vector, the value computed and returned by `fft` is the unnormalized univariate Fourier transform of the sequence of values in `z`. When `z` contains an array, `fft` computes and returns the multivariate (spatial) transform. If `inverse` is `TRUE`, the (unnormalized) inverse Fourier transform is returned, i.e., if `y <- fft(z)`, then `z` is `fft(y, inverse = TRUE) / length(y)`.

By contrast, `mvfft` takes a real or complex matrix as argument, and returns a similar shaped matrix, but with each column replaced by its discrete Fourier transform. This is useful for analyzing vector-valued series.

The FFT is fastest when the length of the series being transformed is highly composite (i.e., has many factors). If this is not the case, the transform may take a long time to compute and will use a large amount of memory.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Singleton, R. C. (1979) Mixed Radix Fast Fourier Transforms, in *Programs for Digital Signal Processing*, IEEE Digital Signal Processing Committee eds. IEEE Press.

See Also

`convolve`, `nextn`.

Examples

```
x <- 1:4
fft(x)
fft(fft(x), inverse = TRUE)/length(x)
```

`filter` *Linear Filtering on a Time Series*

Description

Applies linear filtering to a univariate time series or to each series separately of a multivariate time series.

Usage

```
filter(x, filter, method = c("convolution", "recursive"),
       sides = 2, circular = FALSE, init)
```

Arguments

`x`	a univariate or multivariate time series.
`filter`	a vector of filter coefficients in reverse time order (as for AR or MA coefficients).
`method`	Either `"convolution"` or `"recursive"` (and can be abbreviated). If `"convolution"` a moving average is used: if `"recursive"` an autoregression is used.

`sides`	for convolution filters only. If `sides=1` the filter coefficients are for past values only; if `sides=2` they are centred around lag 0. In this case the length of the filter should be odd, but if it is even, more of the filter is forward in time than backward.
`circular`	for convolution filters only. If `TRUE`, wrap the filter around the ends of the series, otherwise assume external values are missing (`NA`).
`init`	for recursive filters only. Specifies the initial values of the time series just prior to the start value, in reverse time order. The default is a set of zeros.

Details

Missing values are allowed in `x` but not in `filter` (where they would lead to missing values everywhere in the output).

Note that there is an implied coefficient 1 at lag 0 in the recursive filter, which gives

$$y_i = x_i + f_1 y_{i-1} + \cdots + f_p y_{i-p}$$

No check is made to see if recursive filter is invertible: the output may diverge if it is not.

The convolution filter is

$$y_i = f_1 x_{i+o} + \cdots + f_p x_{i+o-(p-1)}$$

where `o` is the offset: see `sides` for how it is determined.

Value

A time series object.

Note

`convolve(, type="filter")` uses the FFT for computations and so *may* be faster for long filters on univariate series, but it does not return a time series (and so the time alignment is unclear), nor does it handle missing values. `filter` is faster for a filter of length 100 on a series of length 1000, for example.

See Also

`convolve`, `arima.sim`

Examples

```
x <- 1:100
filter(x, rep(1, 3))
filter(x, rep(1, 3), sides = 1)
filter(x, rep(1, 3), sides = 1, circular = TRUE)

filter(presidents, rep(1,3))
```

`fisher.test` *Fisher's Exact Test for Count Data*

Description

Performs Fisher's exact test for testing the null of independence of rows and columns in a contingency table with fixed marginals.

Usage

```
fisher.test(x, y = NULL, workspace = 200000, hybrid = FALSE,
            control = list(), or = 1, alternative = "two.sided",
            conf.int = TRUE, conf.level = 0.95,
            simulate.p.value = FALSE, B = 2000)
```

Arguments

`x`	either a two-dimensional contingency table in matrix form, or a factor object.
`y`	a factor object; ignored if `x` is a matrix.
`workspace`	an integer specifying the size of the workspace used in the network algorithm. In units of 4 bytes. Only used for non-simulated p-values larger than 2×2 tables.
`hybrid`	a logical. Only used for larger than 2×2 tables, in which cases it indicated whether the exact probabilities (default) or a hybrid approximation thereof should be computed. See Details.
`control`	a list with named components for low level algorithm control. At present the only one used is `"mult"`, a positive integer $>= 2$ with default 30 used only for larger than 2×2 tables. This says how many times as much space should be allocated to paths as to keys: see file 'fexact.c' in the sources of this package.
`or`	the hypothesized odds ratio. Only used in the 2×2 case.
`alternative`	indicates the alternative hypothesis and must be one of `"two.sided"`, `"greater"` or `"less"`. You can specify just the initial letter. Only used in the 2×2 case.
`conf.int`	logical indicating if a confidence interval should be computed (and returned).
`conf.level`	confidence level for the returned confidence interval. Only used in the 2×2 case if `conf.int = TRUE`.
`simulate.p.value`	a logical indicating whether to compute p-values by Monte Carlo simulation, in larger than 2×2 tables.
`B`	an integer specifying the number of replicates used in the Monte Carlo test.

Details

If `x` is a matrix, it is taken as a two-dimensional contingency table, and hence its entries should be nonnegative integers. Otherwise, both `x` and `y` must be vectors of the same length. Incomplete cases are removed, the vectors are coerced into factor objects, and the contingency table is computed from these.

For 2×2 cases, p-values are obtained directly using the (central or non-central) hypergeometric distribution. Otherwise, computations are based on a C version of the FORTRAN subroutine FEXACT which implements the network developed by Mehta and Patel (1986) and improved by Clarkson, Fan and Joe (1993). The FORTRAN code can be obtained from http://www.netlib.org/toms/643. Note this fails (with an error message) when the entries of the table are too large. (It transposes the table if necessary so it has no more rows than columns. One constraint is that the product of the row marginals be less than $2^{31} - 1$.)

For 2×2 tables, the null of conditional independence is equivalent to the hypothesis that the odds ratio equals one. 'Exact' inference can be based on observing that in general, given all marginal totals fixed, the first element of the contingency table has a non-central hypergeometric distribution with non-centrality parameter given by the odds ratio (Fisher, 1935). The alternative for a one-sided test is based on the odds ratio, so `alternative = "greater"` is a test of the odds ratio being bigger than `or`.

Two-sided tests are based on the probabilities of the tables, and take as 'more extreme' all tables with probabilities less than or equal to that of the observed table, the p-value being the sum of such probabilities.

For larger than 2×2 tables and `hybrid = TRUE`, asymptotic chi-squared probabilities are only used if the "Cochran conditions" are satisfied, that is if no cell has count zero, and more than 80% of the cells have counts at least 5.

Simulation is done conditional on the row and column marginals, and works only if the marginals are strictly positive. (A C translation of the algorithm of Patefield (1981) is used.)

Value

A list with class `"htest"` containing the following components:

`p.value`	the p-value of the test.
`conf.int`	a confidence interval for the odds ratio. Only present in the 2×2 case if argument `conf.int = TRUE`.
`estimate`	an estimate of the odds ratio. Note that the *conditional* Maximum Likelihood Estimate (MLE) rather than the unconditional MLE (the sample odds ratio) is used. Only present in the 2×2 case.
`null.value`	the odds ratio under the null, `or`. Only present in the 2×2 case.
`alternative`	a character string describing the alternative hypothesis.
`method`	the character string `"Fisher's Exact Test for Count Data"`.
`data.name`	a character string giving the names of the data.

References

Agresti, A. (1990) *Categorical data analysis*. New York: Wiley. Pages 59–66.

Fisher, R. A. (1935) The logic of inductive inference. *Journal of the Royal Statistical Society Series A* **98**, 39–54.

Fisher, R. A. (1962) Confidence limits for a cross-product ratio. *Australian Journal of Statistics* **4**, 41.

Fisher, R. A. (1970) *Statistical Methods for Research Workers.* Oliver & Boyd.

Mehta, C. R. and Patel, N. R. (1986) Algorithm 643. FEXACT: A Fortran subroutine for Fisher's exact test on unordered $r * c$ contingency tables. *ACM Transactions on Mathematical Software*, **12**, 154–161.

Clarkson, D. B., Fan, Y. and Joe, H. (1993) A Remark on Algorithm 643: FEXACT: An Algorithm for Performing Fisher's Exact Test in $r \times c$ Contingency Tables. *ACM Transactions on Mathematical Software*, **19**, 484–488.

Patefield, W. M. (1981) Algorithm AS159. An efficient method of generating r x c tables with given row and column totals. *Applied Statistics* **30**, 91–97.

See Also

chisq.test

Examples

```
## Agresti (1990), p. 61f, Fisher's Tea Drinker
## A British woman claimed to be able to distinguish whether milk or
##  tea was added to the cup first.  To test, she was given 8 cups of
##  tea, in four of which milk was added first.  The null hypothesis
##  is that there is no association between the true order of pouring
##  and the woman's guess, the alternative that there is a positive
##  association (that the odds ratio is greater than 1).
TeaTasting <-
matrix(c(3, 1, 1, 3),
       nr = 2,
       dimnames = list(Guess = c("Milk", "Tea"),
                       Truth = c("Milk", "Tea")))
fisher.test(TeaTasting, alternative = "greater")
## => p=0.2429, association could not be established

## Fisher (1962, 1970), Criminal convictions of like-sex twins
Convictions <-
matrix(c(2, 10, 15, 3),
       nr = 2,
       dimnames =
       list(c("Dizygotic", "Monozygotic"),
            c("Convicted", "Not convicted")))
Convictions
fisher.test(Convictions, alternative = "less")
fisher.test(Convictions, conf.int = FALSE)
fisher.test(Convictions, conf.level = 0.95)$conf.int
fisher.test(Convictions, conf.level = 0.99)$conf.int

## A r x c table  Agresti (2002, p. 57) Job Satisfaction
Job <- matrix(c(1,2,1,0, 3,3,6,1, 10,10,14,9, 6,7,12,11), 4, 4,
dimnames = list(income=c("< 15k", "15-25k", "25-40k", "> 40k"),
                satisfaction=c("VeryD", "LittleD", "ModerateS", "VeryS")))
fisher.test(Job)
fisher.test(Job, simulate=TRUE, B=1e5)
```

`fitted` *Extract Model Fitted Values*

Description

`fitted` is a generic function which extracts fitted values from objects returned by modeling functions. `fitted.values` is an alias for it.

All object classes which are returned by model fitting functions should provide a `fitted` method. (Note that the generic is `fitted` and not `fitted.values`.)

Methods can make use of `napredict` methods to compensate for the omission of missing values. The default and `nls` methods do.

Usage

```
fitted(object, ...)
fitted.values(object, ...)
```

Arguments

`object`	an object for which the extraction of model fitted values is meaningful.
`...`	other arguments.

Value

Fitted values extracted from the object `x`.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

See Also

`coefficients`, `glm`, `lm`, `residuals`.

`fivenum` *Tukey Five-Number Summaries*

Description

Returns Tukey's five number summary (minimum, lower-hinge, median, upper-hinge, maximum) for the input data.

Usage

```
fivenum(x, na.rm = TRUE)
```

Arguments

`x`	numeric, maybe including `NA`s and $\pm$`Inf`s.
`na.rm`	logical; if `TRUE`, all `NA` and `NaN`s are dropped, before the statistics are computed.

Value

A numeric vector of length 5 containing the summary information. See `boxplot.stats` for more details.

See Also

`IQR`, `boxplot.stats`, `median`, `quantile`, `range`.

Examples

```
fivenum(c(rnorm(100),-1:1/0))
```

`fligner.test` *Fligner-Killeen Test of Homogeneity of Variances*

Description

Performs a Fligner-Killeen (median) test of the null that the variances in each of the groups (samples) are the same.

Usage

```
fligner.test(x, ...)

## Default S3 method:
fligner.test(x, g, ...)

## S3 method for class 'formula':
fligner.test(formula, data, subset, na.action, ...)
```

Arguments

`x`	a numeric vector of data values, or a list of numeric data vectors.
`g`	a vector or factor object giving the group for the corresponding elements of `x`. Ignored if `x` is a list.
`formula`	a formula of the form `lhs ~ rhs` where `lhs` gives the data values and `rhs` the corresponding groups.
`data`	an optional matrix or data frame (or similar: see `model.frame`) containing the variables in the formula `formula`. By default the variables are taken from `environment(formula)`.
`subset`	an optional vector specifying a subset of observations to be used.
`na.action`	a function which indicates what should happen when the data contain `NA`s. Defaults to `getOption("na.action")`.
`...`	further arguments to be passed to or from methods.

Details

If x is a list, its elements are taken as the samples to be compared for homogeneity of variances, and hence have to be numeric data vectors. In this case, g is ignored, and one can simply use fligner.test(x) to perform the test. If the samples are not yet contained in a list, use fligner.test(list(x, ...)).

Otherwise, x must be a numeric data vector, and g must be a vector or factor object of the same length as x giving the group for the corresponding elements of x.

The Fligner-Killeen (median) test has been determined in a simulation study as one of the many tests for homogeneity of variances which is most robust against departures from normality, see Conover, Johnson & Johnson (1981). It is a k-sample simple linear rank which uses the ranks of the absolute values of the centered samples and weights $a(i) = \mathrm{qnorm}((1+i/(n+1))/2)$. The version implemented here uses median centering in each of the samples (F-K:med X^2 in the reference).

Value

A list of class "htest" containing the following components:

statistic	the Fligner-Killeen:med X^2 test statistic.
parameter	the degrees of freedom of the approximate chi-squared distribution of the test statistic.
p.value	the p-value of the test.
method	the character string "Fligner-Killeen test of homogeneity of variances".
data.name	a character string giving the names of the data.

References

William J. Conover & Mark E. Johnson & Myrle M. Johnson (1981). A comparative study of tests for homogeneity of variances, with applications to the outer continental shelf bidding data. *Technometrics* **23**, 351–361.

See Also

ansari.test and mood.test for rank-based two-sample test for a difference in scale parameters; var.test and bartlett.test for parametric tests for the homogeneity of variances.

Examples

```
plot(count ~ spray, data = InsectSprays)
fligner.test(InsectSprays$count, InsectSprays$spray)
fligner.test(count ~ spray, data = InsectSprays)
## Compare this to bartlett.test()
```

`formula` *Model Formulae*

Description

The generic function `formula` and its specific methods provide a way of extracting formulae which have been included in other objects.

`as.formula` is almost identical, additionally preserving attributes when `object` already inherits from `"formula"`. The default value of the `env` argument is used only when the formula would otherwise lack an environment.

Usage

```
formula(x, ...)
as.formula(object, env = parent.frame())
```

Arguments

`x, object`	R object.
`...`	further arguments passed to or from other methods.
`env`	the environment to associate with the result.

Details

The models fit by, e.g., the `lm` and `glm` functions are specified in a compact symbolic form. The `~` operator is basic in the formation of such models. An expression of the form `y ~ model` is interpreted as a specification that the response `y` is modelled by a linear predictor specified symbolically by `model`. Such a model consists of a series of terms separated by `+` operators. The terms themselves consist of variable and factor names separated by `:` operators. Such a term is interpreted as the interaction of all the variables and factors appearing in the term.

In addition to `+` and `:`, a number of other operators are useful in model formulae. The `*` operator denotes factor crossing: `a*b` interpreted as `a+b+a:b`. The `^` operator indicates crossing to the specified degree. For example `(a+b+c)^2` is identical to `(a+b+c)*(a+b+c)` which in turn expands to a formula containing the main effects for `a`, `b` and `c` together with their second-order interactions. The `%in%` operator indicates that the terms on its left are nested within those on the right. For example `a + b %in% a` expands to the formula `a + a:b`. The `-` operator removes the specified terms, so that `(a+b+c)^2 - a:b` is identical to `a + b + c + b:c + a:c`. It can also used to remove the intercept term: `y ~ x - 1` is a line through the origin. A model with no intercept can be also specified as `y ~ x + 0` or `y ~ 0 + x`.

While formulae usually involve just variable and factor names, they can also involve arithmetic expressions. The formula `log(y) ~ a + log(x)` is quite legal. When such arithmetic expressions involve operators which are also used symbolically in model formulae, there can be confusion between arithmetic and symbolic operator use.

To avoid this confusion, the function `I()` can be used to bracket those portions of a model formula where the operators are used in their arithmetic sense. For example, in the formula `y ~ a + I(b+c)`, the term `b+c` is to be interpreted as the sum of `b` and `c`.

Variable names can be quoted by backticks `` `like this` `` in formulae, although there is no guarantee that all code using formulae will accept such non-syntactic names.

When `formula` is called on a fitted model object, either a specific method is used (such as that for class `"nls"`) of the default method. The default first looks for a `"formula"` component of the object (and evaluates it), then a `"terms"` component, then a `formula` parameter of the call (and evaluates its value) and finally a `"formula"` attribute.

Value

All the functions above produce an object of class `"formula"` which contains a symbolic model formula.

Environments

A formula object has an associated environment, and this environment (rather than the parent environment) is used by `model.frame` to evaluate variables that are not found in the supplied `data` argument.

Formulas created with the ~ operator use the environment in which they were created. Formulas created with `as.formula` will use the `env` argument for their environment. Pre-existing formulas extracted with `as.formula` will only have their environment changed if `env` is given explicitly.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical models.* Chapter 2 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`I`.

For formula manipulation: `terms`, and `all.vars`; for typical use: `lm`, `glm`, and `coplot`.

Examples

```
class(fo <- y ~ x1*x2) # "formula"
fo
typeof(fo)# R internal : "language"
terms(fo)

environment(fo)
environment(as.formula("y ~ x"))
environment(as.formula("y ~ x",env=new.env()))

## Create a formula for a model with a large number of variables:
xnam <- paste("x", 1:25, sep="")
(fmla <- as.formula(paste("y ~ ", paste(xnam, collapse= "+"))))
```

---

formula.nls *Extract Model Formula from nls Object*

Description

Returns the model used to fit `object`.

Usage

```
## S3 method for class 'nls':
formula(x, ...)
```

Arguments

x	an object inheriting from class "nls", representing a nonlinear least squares fit.
...	further arguments passed to or from other methods.

Value

a formula representing the model used to obtain object.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

nls, formula

Examples

```
fm1 <- nls(circumference ~ A/(1+exp((B-age)/C)), Orange,
           start = list(A=160, B=700, C = 350))
formula(fm1)
```

---

friedman.test *Friedman Rank Sum Test*

Description

Performs a Friedman rank sum test with unreplicated blocked data.

Usage

```
friedman.test(y, ...)

## Default S3 method:
friedman.test(y, groups, blocks, ...)

## S3 method for class 'formula':
friedman.test(formula, data, subset, na.action, ...)
```

Arguments

y	either a numeric vector of data values, or a data matrix.
groups	a vector giving the group for the corresponding elements of y if this is a vector; ignored if y is a matrix. If not a factor object, it is coerced to one.
blocks	a vector giving the block for the corresponding elements of y if this is a vector; ignored if y is a matrix. If not a factor object, it is coerced to one.
formula	a formula of the form a ~ b | c, where a, b and c give the data values and corresponding groups and blocks, respectively.
data	an optional matrix or data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).
subset	an optional vector specifying a subset of observations to be used.
na.action	a function which indicates what should happen when the data contain NAs. Defaults to getOption("na.action").
...	further arguments to be passed to or from methods.

Details

friedman.test can be used for analyzing unreplicated complete block designs (i.e., there is exactly one observation in y for each combination of levels of groups and blocks) where the normality assumption may be violated.

The null hypothesis is that apart from an effect of blocks, the location parameter of y is the same in each of the groups.

If y is a matrix, groups and blocks are obtained from the column and row indices, respectively. NA's are not allowed in groups or blocks; if y contains NA's, corresponding blocks are removed.

Value

A list with class "htest" containing the following components:

statistic	the value of Friedman's chi-squared statistic.
parameter	the degrees of freedom of the approximate chi-squared distribution of the test statistic.
p.value	the p-value of the test.
method	the character string "Friedman rank sum test".
data.name	a character string giving the names of the data.

References

Myles Hollander & Douglas A. Wolfe (1973), *Nonparametric statistical inference*. New York: John Wiley & Sons. Pages 139–146.

See Also

quade.test.

Examples

```
## Hollander & Wolfe (1973), p. 140ff.
## Comparison of three methods ("round out", "narrow angle", and
##  "wide angle") for rounding first base.  For each of 18 players
##  and the three method, the average time of two runs from a point on
##  the first base line 35ft from home plate to a point 15ft short of
##  second base is recorded.
RoundingTimes <-
matrix(c(5.40, 5.50, 5.55,
         5.85, 5.70, 5.75,
         5.20, 5.60, 5.50,
         5.55, 5.50, 5.40,
         5.90, 5.85, 5.70,
         5.45, 5.55, 5.60,
         5.40, 5.40, 5.35,
         5.45, 5.50, 5.35,
         5.25, 5.15, 5.00,
         5.85, 5.80, 5.70,
         5.25, 5.20, 5.10,
         5.65, 5.55, 5.45,
         5.60, 5.35, 5.45,
         5.05, 5.00, 4.95,
         5.50, 5.50, 5.40,
         5.45, 5.55, 5.50,
         5.55, 5.55, 5.35,
         5.45, 5.50, 5.55,
         5.50, 5.45, 5.25,
         5.65, 5.60, 5.40,
         5.70, 5.65, 5.55,
         6.30, 6.30, 6.25),
       nr = 22,
       byrow = TRUE,
       dimnames = list(1 : 22,
                       c("Round Out", "Narrow Angle", "Wide Angle")))
friedman.test(RoundingTimes)
## => strong evidence against the null that the methods are equivalent
##    with respect to speed

wb <- aggregate(warpbreaks$breaks,
                by = list(w = warpbreaks$wool,
                          t = warpbreaks$tension),
                FUN = mean)
wb
friedman.test(wb$x, wb$w, wb$t)
friedman.test(x ~ w | t, data = wb)
```

---

ftable *Flat Contingency Tables*

Description

Create “flat” contingency tables.

Usage

```
ftable(x, ...)

## Default S3 method:
ftable(..., exclude = c(NA, NaN), row.vars = NULL, col.vars = NULL)
```

Arguments

x, ...	R objects which can be interpreted as factors (including character strings), or a list (or data frame) whose components can be so interpreted, or a contingency table object of class "table" or "ftable".
exclude	values to use in the exclude argument of factor when interpreting non-factor objects.
row.vars	a vector of integers giving the numbers of the variables, or a character vector giving the names of the variables to be used for the rows of the flat contingency table.
col.vars	a vector of integers giving the numbers of the variables, or a character vector giving the names of the variables to be used for the columns of the flat contingency table.

Details

ftable creates "flat" contingency tables. Similar to the usual contingency tables, these contain the counts of each combination of the levels of the variables (factors) involved. This information is then re-arranged as a matrix whose rows and columns correspond to unique combinations of the levels of the row and column variables (as specified by row.vars and col.vars, respectively). The combinations are created by looping over the variables in reverse order (so that the levels of the "left-most" variable vary the slowest). Displaying a contingency table in this flat matrix form (via print.ftable, the print method for objects of class "ftable") is often preferable to showing it as a higher-dimensional array.

ftable is a generic function. Its default method, ftable.default, first creates a contingency table in array form from all arguments except row.vars and col.vars. If the first argument is of class "table", it represents a contingency table and is used as is; if it is a flat table of class "ftable", the information it contains is converted to the usual array representation using as.ftable. Otherwise, the arguments should be R objects which can be interpreted as factors (including character strings), or a list (or data frame) whose components can be so interpreted, which are cross-tabulated using table. Then, the arguments row.vars and col.vars are used to collapse the contingency table into flat form. If neither of these two is given, the last variable is used for the columns. If both are given and their union is a proper subset of all variables involved, the other variables are summed out.

When the arguments are R expressions interpreted as factors, additional arguments will be passed to table to control how the variable names are displayed; see the last example below.

Function ftable.formula provides a formula method for creating flat contingency tables.

There are methods for as.table and as.data.frame.

Value

ftable returns an object of class "ftable", which is a matrix with counts of each combination of the levels of variables with information on the names and levels of the (row and columns) variables stored as attributes "row.vars" and "col.vars".

See Also

ftable.formula for the formula interface (which allows a data = . argument); read.ftable for information on reading, writing and coercing flat contingency tables; table for "ordinary" cross-tabulation; xtabs for formula-based cross-tabulation.

Examples

```
## Start with a contingency table.
ftable(Titanic, row.vars = 1:3)
ftable(Titanic, row.vars = 1:2, col.vars = "Survived")
ftable(Titanic, row.vars = 2:1, col.vars = "Survived")

## Start with a data frame.
x <- ftable(mtcars[c("cyl", "vs", "am", "gear")])
x
ftable(x, row.vars = c(2, 4))

## Start with expressions, use table()'s "dnn" to change labels
ftable(mtcars$cyl, mtcars$vs, mtcars$am, mtcars$gear, row.vars = c(2, 4),
       dnn = c("Cylinders", "V/S", "Transmission", "Gears"))
```

---

ftable.formula *Formula Notation for Flat Contingency Tables*

Description

Produce or manipulate a flat contingency table using formula notation.

Usage

```
## S3 method for class 'formula':
ftable(formula, data = NULL, subset, na.action, ...)
```

Arguments

formula	a formula object with both left and right hand sides specifying the column and row variables of the flat table.
data	a data frame, list or environment (or similar: see model.frame) containing the variables to be cross-tabulated, or a contingency table (see below).
subset	an optional vector specifying a subset of observations to be used. Ignored if data is a contingency table.
na.action	a function which indicates what should happen when the data contain NAs. Ignored if data is a contingency table.
...	further arguments to the default ftable method may also be passed as arguments, see ftable.default.

Details

This is a method of the generic function `ftable`.

The left and right hand side of `formula` specify the column and row variables, respectively, of the flat contingency table to be created. Only the + operator is allowed for combining the variables. A `.` may be used once in the formula to indicate inclusion of all the "remaining" variables.

If `data` is an object of class `"table"` or an array with more than 2 dimensions, it is taken as a contingency table, and hence all entries should be nonnegative. Otherwise, if it is not a flat contingency table (i.e., an object of class `"ftable"`), it should be a data frame or matrix, list or environment containing the variables to be cross-tabulated. In this case, `na.action` is applied to the data to handle missing values, and, after possibly selecting a subset of the data as specified by the `subset` argument, a contingency table is computed from the variables.

The contingency table is then collapsed to a flat table, according to the row and column variables specified by `formula`.

Value

A flat contingency table which contains the counts of each combination of the levels of the variables, collapsed into a matrix for suitably displaying the counts.

See Also

`ftable`, `ftable.default`; `table`.

Examples

```
Titanic
x <- ftable(Survived ~ ., data = Titanic)
x
ftable(Sex ~ Class + Age, data = x)
```

GammaDist *The Gamma Distribution*

Description

Density, distribution function, quantile function and random generation for the Gamma distribution with parameters `shape` and `scale`.

Usage

```
dgamma(x, shape, rate = 1, scale = 1/rate, log = FALSE)
pgamma(q, shape, rate = 1, scale = 1/rate, lower.tail = TRUE,
       log.p = FALSE)
qgamma(p, shape, rate = 1, scale = 1/rate, lower.tail = TRUE,
       log.p = FALSE)
rgamma(n, shape, rate = 1, scale = 1/rate)
```

Arguments

x, q	vector of quantiles.
p	vector of probabilities.
n	number of observations. If length(n) > 1, the length is taken to be the number required.
rate	an alternative way to specify the scale.
shape, scale	shape and scale parameters. Must be strictly positive.
log, log.p	logical; if TRUE, probabilities/densities p are returned as $log(p)$.
lower.tail	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

If scale is omitted, it assumes the default value of 1.

The Gamma distribution with parameters shape $= \alpha$ and scale $= \sigma$ has density

$$f(x) = \frac{1}{\sigma^\alpha \Gamma(\alpha)} x^{\alpha-1} e^{-x/\sigma}$$

for $x \geq 0$, $\alpha > 0$ and $\sigma > 0$. (Here $\Gamma(\alpha)$ is the function implemented by R's gamma() and defined in its help.)

The mean and variance are $E(X) = \alpha\sigma$ and $Var(X) = \alpha\sigma^2$.

The cumulative hazard $H(t) = -\log(1 - F(t))$ is -pgamma(t, ..., lower = FALSE, log = TRUE).

Note that for smallish values of shape (and moderate scale) a large parts of the mass of the Gamma distribution is on values of x so near zero that they will be represented as zero in computer arithmetic. So rgamma can well return values which will be represented as zero. (This will also happen for very large values of scale since the actual generation is done for scale=1.)

Value

dgamma gives the density, pgamma gives the distribution function, qgamma gives the quantile function, and rgamma generates random deviates.

Invalid arguments will result in return value NaN, with a warning.

Note

The S parametrization is via shape and rate: S has no scale parameter.

pgamma is closely related to the incomplete gamma function. As defined by Abramowitz and Stegun 6.5.1

$$P(a, x) = \frac{1}{\Gamma(a)} \int_0^x t^{a-1} e^{-t} dt$$

$P(a, x)$ is pgamma(x, a). Other authors (for example Karl Pearson in his 1922 tables) omit the normalizing factor, defining the incomplete gamma function as pgamma(x, a) * gamma(a).

Source

`dgamma` is computed via the Poisson density, using code contributed by Catherine Loader (see `dbinom`).

Prior to R 2.1.0 `pgamma` used AS239 (Shea, 1988). It currently uses an unpublished (and not otherwise documented) algorithm 'mainly by Morten Welinder'.

`qgamma` is based on a C translation of

Best, D. J. and D. E. Roberts (1975). Algorithm AS91. Percentage points of the chi-squared distribution. *Applied Statistics*, **24**, 385–388.

plus a final Newton step to improve the approximation.

`rgamma` for `shape >= 1` uses

Ahrens, J. H. and Dieter, U. (1982). Generating gamma variates by a modified rejection technique. *Communications of the ACM*, **25**, 47–54,

and for `0 < shape < 1` uses

Ahrens, J. H. and Dieter, U. (1974). Computer methods for sampling from gamma, beta, Poisson and binomial distributions. *Computing*, **12**, 223–246.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Shea, B. L. (1988) Algorithm AS 239, Chi-squared and incomplete Gamma integral, *Applied Statistics (JRSS C)* **37**, 466–473.

Abramowitz, M. and Stegun, I. A. (1972) *Handbook of Mathematical Functions.* New York: Dover. Chapter 6: Gamma and Related Functions.

See Also

`gamma` for the gamma function, `dbeta` for the Beta distribution and `dchisq` for the chi-squared distribution which is a special case of the Gamma distribution.

Examples

```
-log(dgamma(1:4, shape=1))
p <- (1:9)/10
pgamma(qgamma(p,shape=2), shape=2)
1 - 1/exp(qgamma(p, shape=1))

# even for shape = 0.001 about half the mass is on numbers
# that cannot be represented accurately (and most of those as zero)
pgamma(.Machine$double.xmin, 0.001)
pgamma(5e-324, 0.001)  # on most machines this is the smallest
                       # representable non-zero number
table(rgamma(1e4, 0.001) == 0)/1e4
```

`Geometric` *The Geometric Distribution*

Description

Density, distribution function, quantile function and random generation for the geometric distribution with parameter `prob`.

Usage

```
dgeom(x, prob, log = FALSE)
pgeom(q, prob, lower.tail = TRUE, log.p = FALSE)
qgeom(p, prob, lower.tail = TRUE, log.p = FALSE)
rgeom(n, prob)
```

Arguments

`x, q`	vector of quantiles representing the number of failures in a sequence of Bernoulli trials before success occurs.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`prob`	probability of success in each trial. `0 < prob <= 1`.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

The geometric distribution with `prob` $= p$ has density

$$p(x) = p(1-p)^x$$

for $x = 0, 1, 2, \ldots$, $0 < p \leq 1$.

If an element of `x` is not integer, the result of `pgeom` is zero, with a warning.

The quantile is defined as the smallest value x such that $F(x) \geq p$, where F is the distribution function.

Value

`dgeom` gives the density, `pgeom` gives the distribution function, `qgeom` gives the quantile function, and `rgeom` generates random deviates.

Invalid `prob` will result in return value `NaN`, with a warning.

Source

`dgeom` computes via `dbinom`, using code contributed by Catherine Loader (see `dbinom`).

`pgeom` and `qgeom` are based on the closed-form formulae.

`rgeom` uses the derivation as an exponential mixture of Poissons, see

Devroye, L. (1986) *Non-Uniform Random Variate Generation.* Springer-Verlag, New York. Page 480.

See Also

dnbinom for the negative binomial which generalizes the geometric distribution.

Examples

```
qgeom((1:9)/10, prob = .2)
Ni <- rgeom(20, prob = 1/4); table(factor(Ni, 0:max(Ni)))
```

---

getInitial *Get Initial Parameter Estimates*

Description

This function evaluates initial parameter estimates for a nonlinear regression model. If data is a parameterized data frame or pframe object, its parameters attribute is returned. Otherwise the object is examined to see if it contains a call to a selfStart object whose initial attribute can be evaluated.

Usage

```
getInitial(object, data, ...)
```

Arguments

object	a formula or a selfStart model that defines a nonlinear regression model
data	a data frame in which the expressions in the formula or arguments to the selfStart model can be evaluated
...	optional additional arguments

Value

A named numeric vector or list of starting estimates for the parameters. The construction of many selfStart models is such that these "starting" estimates are, in fact, the converged parameter estimates.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

nls, selfStart, selfStart.default, selfStart.formula

Examples

```
PurTrt <- Puromycin[ Puromycin$state == "treated", ]
getInitial( rate ~ SSmicmen( conc, Vm, K ), PurTrt )
```

`glm` *Fitting Generalized Linear Models*

Description

`glm` is used to fit generalized linear models, specified by giving a symbolic description of the linear predictor and a description of the error distribution.

Usage

```
glm(formula, family = gaussian, data, weights, subset,
    na.action, start = NULL, etastart, mustart,
    offset, control = glm.control(...), model = TRUE,
    method = "glm.fit", x = FALSE, y = TRUE, contrasts = NULL, ...)

glm.fit(x, y, weights = rep(1, nobs),
        start = NULL, etastart = NULL, mustart = NULL,
        offset = rep(0, nobs), family = gaussian(),
        control = glm.control(), intercept = TRUE)

## S3 method for class 'glm':
weights(object, type = c("prior", "working"), ...)
```

Arguments

`formula`	a symbolic description of the model to be fit. The details of model specification are given below.
`family`	a description of the error distribution and link function to be used in the model. This can be a character string naming a family function, a family function or the result of a call to a family function. (See `family` for details of family functions.)
`data`	an optional data frame, list or environment (or object coercible by `as.data.frame` to a data frame) containing the variables in the model. If not found in `data`, the variables are taken from `environment(formula)`, typically the environment from which `glm` is called.
`weights`	an optional vector of weights to be used in the fitting process. Should be `NULL` or a numeric vector.
`subset`	an optional vector specifying a subset of observations to be used in the fitting process.
`na.action`	a function which indicates what should happen when the data contain `NA`s. The default is set by the `na.action` setting of `options`, and is `na.fail` if that is unset. The "factory-fresh" default is `na.omit`. Another possible value is `NULL`, no action. Value `na.exclude` can be useful.
`start`	starting values for the parameters in the linear predictor.
`etastart`	starting values for the linear predictor.
`mustart`	starting values for the vector of means.

offset	this can be used to specify an *a priori* known component to be included in the linear predictor during fitting. This should be NULL or a numeric vector of length either one or equal to the number of cases. One or more offset terms can be included in the formula instead or as well, and if both are specified their sum is used. See model.offset.
control	a list of parameters for controlling the fitting process. See the documentation for glm.control for details.
model	a logical value indicating whether *model frame* should be included as a component of the returned value.
method	the method to be used in fitting the model. The default method "glm.fit" uses iteratively reweighted least squares (IWLS). The only current alternative is "model.frame" which returns the model frame and does no fitting.
x, y	For glm: logical values indicating whether the response vector and model matrix used in the fitting process should be returned as components of the returned value. For glm.fit: x is a design matrix of dimension n * p, and y is a vector of observations of length n.
contrasts	an optional list. See the contrasts.arg of model.matrix.default.
object	an object inheriting from class "glm".
type	character, partial matching allowed. Type of weights to extract from the fitted model object.
intercept	logical. Should an intercept be included in the *null* model?
...	further arguments passed to or from other methods.

Details

A typical predictor has the form response ~ terms where response is the (numeric) response vector and terms is a series of terms which specifies a linear predictor for response. For binomial and quasibinomial families the response can also be specified as a factor (when the first level denotes failure and all others success) or as a two-column matrix with the columns giving the numbers of successes and failures. A terms specification of the form first + second indicates all the terms in first together with all the terms in second with duplicates removed. The terms in the formula will be re-ordered so that main effects come first, followed by the interactions, all second-order, all third-order and so on: to avoid this pass a terms object as the formula.

A specification of the form first:second indicates the the set of terms obtained by taking the interactions of all terms in first with all terms in second. The specification first*second indicates the *cross* of first and second. This is the same as first + second + first:second.

glm.fit is the workhorse function.

If more than one of etastart, start and mustart is specified, the first in the list will be used. It is often advisable to supply starting values for a quasi family, and also for families with unusual links such as gaussian("log").

All of weights, subset, offset, etastart and mustart are evaluated in the same way as variables in formula, that is first in data and then in the environment of formula.

Value

`glm` returns an object of class inheriting from `"glm"` which inherits from the class `"lm"`. See later in this section.

The function `summary` (i.e., `summary.glm`) can be used to obtain or print a summary of the results and the function `anova` (i.e., `anova.glm`) to produce an analysis of variance table.

The generic accessor functions `coefficients`, `effects`, `fitted.values` and `residuals` can be used to extract various useful features of the value returned by `glm`.

`weights` extracts a vector of weights, one for each case in the fit (after subsetting and `na.action`).

An object of class `"glm"` is a list containing at least the following components:

`coefficients` a named vector of coefficients

`residuals` the *working* residuals, that is the residuals in the final iteration of the IWLS fit. Since cases with zero weights are omitted, their working residuals are `NA`.

`fitted.values` the fitted mean values, obtained by transforming the linear predictors by the inverse of the link function.

`rank` the numeric rank of the fitted linear model.

`family` the `family` object used.

`linear.predictors` the linear fit on link scale.

`deviance` up to a constant, minus twice the maximized log-likelihood. Where sensible, the constant is chosen so that a saturated model has deviance zero.

`aic` Akaike's *An Information Criterion*, minus twice the maximized log-likelihood plus twice the number of coefficients (so assuming that the dispersion is known).

`null.deviance` The deviance for the null model, comparable with `deviance`. The null model will include the offset, and an intercept if there is one in the model. Note that this will be incorrect if the link function depends on the data other than through the fitted mean: specify a zero offset to force a correct calculation.

`iter` the number of iterations of IWLS used.

`weights` the *working* weights, that is the weights in the final iteration of the IWLS fit.

`prior.weights` the case weights initially supplied.

`df.residual` the residual degrees of freedom.

`df.null` the residual degrees of freedom for the null model.

`y` the `y` vector used. (It is a vector even for a binomial model.)

`converged` logical. Was the IWLS algorithm judged to have converged?

`boundary` logical. Is the fitted value on the boundary of the attainable values?

`call` the matched call.

`formula` the formula supplied.

`terms` the `terms` object used.

`data` the `data argument`.

`offset` the offset vector used.

control	the value of the control argument used.
method	the name of the fitter function used, currently always "glm.fit".
contrasts	(where relevant) the contrasts used.
xlevels	(where relevant) a record of the levels of the factors used in fitting.

In addition, non-empty fits will have components qr, R and effects relating to the final weighted linear fit.

Objects of class "glm" are normally of class c("glm", "lm"), that is inherit from class "lm", and well-designed methods for class "lm" will be applied to the weighted linear model at the final iteration of IWLS. However, care is needed, as extractor functions for class "glm" such as residuals and weights do **not** just pick out the component of the fit with the same name.

If a binomial glm model is specified by giving a two-column response, the weights returned by prior.weights are the total numbers of cases (factored by the supplied case weights) and the component y of the result is the proportion of successes.

Author(s)

The original R implementation of glm was written by Simon Davies working for Ross Ihaka at the University of Auckland, but has since been extensively re-written by members of the R Core team.

The design was inspired by the S function of the same name described in Hastie & Pregibon (1992).

References

Dobson, A. J. (1990) *An Introduction to Generalized Linear Models.* London: Chapman and Hall.

Hastie, T. J. and Pregibon, D. (1992) *Generalized linear models.* Chapter 6 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

McCullagh P. and Nelder, J. A. (1989) *Generalized Linear Models.* London: Chapman and Hall.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* New York: Springer.

See Also

anova.glm, summary.glm, etc. for glm methods, and the generic functions anova, summary, effects, fitted.values, and residuals. Further, lm for non-generalized *linear* models.

esoph, infert and predict.glm have examples of fitting binomial glms.

Examples

```
## Dobson (1990) Page 93: Randomized Controlled Trial :
counts <- c(18,17,15,20,10,20,25,13,12)
outcome <- gl(3,1,9)
treatment <- gl(3,3)
print(d.AD <- data.frame(treatment, outcome, counts))
glm.D93 <- glm(counts ~ outcome + treatment, family=poisson())
anova(glm.D93)
summary(glm.D93)

## an example with offsets from Venables & Ripley (2002, p.189)
data(anorexia, package="MASS")

anorex.1 <- glm(Postwt ~ Prewt + Treat + offset(Prewt),
                family = gaussian, data = anorexia)
```

```
summary(anorex.1)

# A Gamma example, from McCullagh & Nelder (1989, pp. 300-2)
clotting <- data.frame(
    u = c(5,10,15,20,30,40,60,80,100),
    lot1 = c(118,58,42,35,27,25,21,19,18),
    lot2 = c(69,35,26,21,18,16,13,12,12))
summary(glm(lot1 ~ log(u), data=clotting, family=Gamma))
summary(glm(lot2 ~ log(u), data=clotting, family=Gamma))

## Not run:
## for an example of the use of a terms object as a formula
demo(glm.vr)
## End(Not run)
```

`glm.control` *Auxiliary for Controlling GLM Fitting*

Description

Auxiliary function as user interface for `glm` fitting. Typically only used when calling `glm` or `glm.fit`.

Usage

```
glm.control(epsilon = 1e-8, maxit = 25, trace = FALSE)
```

Arguments

`epsilon`	positive convergence tolerance ϵ; the iterations converge when $\|dev - dev_{old}\|/(\|dev\| + 0.1) < \epsilon$.
`maxit`	integer giving the maximal number of IWLS iterations.
`trace`	logical indicating if output should be produced for each iteration.

Details

If `epsilon` is small, it is also used as the tolerance for the least squares solution.

When `trace` is true, calls to `cat` produce the output for each IWLS iteration. Hence, `options(digits = *)` can be used to increase the precision, see the example.

Value

A list with the arguments as components.

References

Hastie, T. J. and Pregibon, D. (1992) *Generalized linear models.* Chapter 6 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`glm.fit`, the fitting procedure used by `glm`.

Examples

```
### A variation on  example(glm) :

## Annette Dobson's example ...
counts <- c(18,17,15,20,10,20,25,13,12)
outcome <- gl(3,1,9)
treatment <- gl(3,3)
oo <- options(digits = 12) # to see more when tracing :
glm.D93X <- glm(counts ~ outcome + treatment, family=poisson(),
                trace = TRUE, epsilon = 1e-14)
options(oo)
coef(glm.D93X) # the last two are closer to 0 than in ?glm's  glm.D93
```

---

glm.summaries *Accessing Generalized Linear Model Fits*

Description

These functions are all methods for class glm or summary.glm objects.

Usage

```
## S3 method for class 'glm':
family(object, ...)

## S3 method for class 'glm':
residuals(object, type = c("deviance", "pearson", "working",
                           "response", "partial"), ...)
```

Arguments

object	an object of class glm, typically the result of a call to glm.
type	the type of residuals which should be returned. The alternatives are: "deviance" (default), "pearson", "working", "response", and "partial".
...	further arguments passed to or from other methods.

Details

The references define the types of residuals: Davison & Snell is a good reference for the usages of each.

The partial residuals are a matrix of working residuals, with each column formed by omitting a term from the model.

How residuals treats cases with missing values in the original fit is determined by the na.action argument of that fit. If na.action = na.omit omitted cases will not appear in the residuals, whereas if na.action = na.exclude they will appear, with residual value NA. See also naresid.

For fits done with y = FALSE the response values are computed from other components.

References

Davison, A. C. and Snell, E. J. (1991) *Residuals and diagnostics.* In: Statistical Theory and Modelling. In Honour of Sir David Cox, FRS, eds. Hinkley, D. V., Reid, N. and Snell, E. J., Chapman & Hall.

Hastie, T. J. and Pregibon, D. (1992) *Generalized linear models.* Chapter 6 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

McCullagh P. and Nelder, J. A. (1989) *Generalized Linear Models.* London: Chapman and Hall.

See Also

`glm` for computing `glm.obj`, `anova.glm`; the corresponding *generic* functions, `summary.glm`, `coef`, `deviance`, `df.residual`, `effects`, `fitted`, `residuals`.

`hclust` *Hierarchical Clustering*

Description

Hierarchical cluster analysis on a set of dissimilarities and methods for analyzing it.

Usage

```
hclust(d, method = "complete", members=NULL)

## S3 method for class 'hclust':
plot(x, labels = NULL, hang = 0.1,
     axes = TRUE, frame.plot = FALSE, ann = TRUE,
     main = "Cluster Dendrogram",
     sub = NULL, xlab = NULL, ylab = "Height", ...)

plclust(tree, hang = 0.1, unit = FALSE, level = FALSE, hmin = 0,
        square = TRUE, labels = NULL, plot. = TRUE,
        axes = TRUE, frame.plot = FALSE, ann = TRUE,
        main = "", sub = NULL, xlab = NULL, ylab = "Height")
```

Arguments

`d`	a dissimilarity structure as produced by `dist`.
`method`	the agglomeration method to be used. This should be (an unambiguous abbreviation of) one of `"ward"`, `"single"`, `"complete"`, `"average"`, `"mcquitty"`, `"median"` or `"centroid"`.
`members`	`NULL` or a vector with length size of `d`. See the Details section.
`x,tree`	an object of the type produced by `hclust`.
`hang`	The fraction of the plot height by which labels should hang below the rest of the plot. A negative value will cause the labels to hang down from 0.
`labels`	A character vector of labels for the leaves of the tree. By default the row names or row numbers of the original data are used. If `labels=FALSE` no labels at all are plotted.

`axes, frame.plot, ann`
: logical flags as in `plot.default`.

`main, sub, xlab, ylab`
: character strings for `title`. `sub` and `xlab` have a non-NULL default when there's a `tree$call`.

`...`
: Further graphical arguments.

`unit`
: logical. If true, the splits are plotted at equally-spaced heights rather than at the height in the object.

`hmin`
: numeric. All heights less than `hmin` are regarded as being `hmin`: this can be used to suppress detail at the bottom of the tree.

`level, square, plot.`
: as yet unimplemented arguments of `plclust` for S-PLUS compatibility.

Details

This function performs a hierarchical cluster analysis using a set of dissimilarities for the n objects being clustered. Initially, each object is assigned to its own cluster and then the algorithm proceeds iteratively, at each stage joining the two most similar clusters, continuing until there is just a single cluster. At each stage distances between clusters are recomputed by the Lance–Williams dissimilarity update formula according to the particular clustering method being used.

A number of different clustering methods are provided. *Ward's* minimum variance method aims at finding compact, spherical clusters. The *complete linkage* method finds similar clusters. The *single linkage* method (which is closely related to the minimal spanning tree) adopts a 'friends of friends' clustering strategy. The other methods can be regarded as aiming for clusters with characteristics somewhere between the single and complete link methods. Note however, that methods `"median"` and `"centroid"` are *not* leading to a *monotone distance* measure, or equivalently the resulting dendrograms can have so called *inversions* (which are hard to interpret).

If `members!=NULL`, then `d` is taken to be a dissimilarity matrix between clusters instead of dissimilarities between singletons and `members` gives the number of observations per cluster. This way the hierarchical cluster algorithm can be "started in the middle of the dendrogram", e.g., in order to reconstruct the part of the tree above a cut (see examples). Dissimilarities between clusters can be efficiently computed (i.e., without `hclust` itself) only for a limited number of distance/linkage combinations, the simplest one being squared Euclidean distance and centroid linkage. In this case the dissimilarities between the clusters are the squared Euclidean distances between cluster means.

In hierarchical cluster displays, a decision is needed at each merge to specify which subtree should go on the left and which on the right. Since, for n observations there are $n-1$ merges, there are $2^{(n-1)}$ possible orderings for the leaves in a cluster tree, or dendrogram. The algorithm used in `hclust` is to order the subtree so that the tighter cluster is on the left (the last, i.e., most recent, merge of the left subtree is at a lower value than the last merge of the right subtree). Single observations are the tightest clusters possible, and merges involving two observations place them in order by their observation sequence number.

Value

An object of class **hclust** which describes the tree produced by the clustering process. The object is a list with components:

`merge`
: an $n-1$ by 2 matrix. Row i of `merge` describes the merging of clusters at step i of the clustering. If an element j in the row is negative, then observation $-j$ was merged at this stage. If j is positive then the merge was with the cluster formed at the (earlier) stage j of the algorithm. Thus negative entries in `merge` indicate

	agglomerations of singletons, and positive entries indicate agglomerations of non-singletons.
`height`	a set of $n - 1$ non-decreasing real values. The clustering *height*: that is, the value of the criterion associated with the clustering `method` for the particular agglomeration.
`order`	a vector giving the permutation of the original observations suitable for plotting, in the sense that a cluster plot using this ordering and matrix `merge` will not have crossings of the branches.
`labels`	labels for each of the objects being clustered.
`call`	the call which produced the result.
`method`	the cluster method that has been used.
`dist.method`	the distance that has been used to create `d` (only returned if the distance object has a `"method"` attribute).

There are `print`, `plot` and `identify` (see `identify.hclust`) methods and the `rect.hclust()` function for `hclust` objects. The `plclust()` function is basically the same as the plot method, `plot.hclust`, primarily for back compatibility with S-plus. Its extra arguments are not yet implemented.

Author(s)

The `hclust` function is based on Fortran code contributed to STATLIB by F. Murtagh.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (S version.)

Everitt, B. (1974). *Cluster Analysis*. London: Heinemann Educ. Books.

Hartigan, J. A. (1975). *Clustering Algorithms*. New York: Wiley.

Sneath, P. H. A. and R. R. Sokal (1973). *Numerical Taxonomy*. San Francisco: Freeman.

Anderberg, M. R. (1973). *Cluster Analysis for Applications*. Academic Press: New York.

Gordon, A. D. (1999). *Classification*. Second Edition. London: Chapman and Hall / CRC

Murtagh, F. (1985). "Multidimensional Clustering Algorithms", in *COMPSTAT Lectures 4*. Wuerzburg: Physica-Verlag (for algorithmic details of algorithms used).

McQuitty, L.L. (1966). Similarity Analysis by Reciprocal Pairs for Discrete and Continuous Data. *Educational and Psychological Measurement*, **26**, 825–831.

See Also

`identify.hclust`, `rect.hclust`, `cutree`, `dendrogram`, `kmeans`.

For the Lance–Williams formula and methods that apply it generally, see `agnes` from package **cluster**.

Examples

```
hc <- hclust(dist(USArrests), "ave")
plot(hc)
plot(hc, hang = -1)

## Do the same with centroid clustering and squared Euclidean distance,
```

```
## cut the tree into ten clusters and reconstruct the upper part of the
## tree from the cluster centers.
hc <- hclust(dist(USArrests)^2, "cen")
memb <- cutree(hc, k = 10)
cent <- NULL
for(k in 1:10){
  cent <- rbind(cent, colMeans(USArrests[memb == k, , drop = FALSE]))
}
hc1 <- hclust(dist(cent)^2, method = "cen", members = table(memb))
opar <- par(mfrow = c(1, 2))
plot(hc,  labels = FALSE, hang = -1, main = "Original Tree")
plot(hc1, labels = FALSE, hang = -1, main = "Re-start from 10 clusters")
par(opar)
```

`heatmap` *Draw a Heat Map*

Description

A heat map is a false color image (basically `image(t(x))`) with a dendrogram added to the left side and to the top. Typically, reordering of the rows and columns according to some set of values (row or column means) within the restrictions imposed by the dendrogram is carried out.

Usage

```
heatmap(x, Rowv=NULL, Colv=if(symm)"Rowv" else NULL,
        distfun = dist, hclustfun = hclust,
        reorderfun = function(d,w) reorder(d,w),
        add.expr, symm = FALSE, revC = identical(Colv, "Rowv"),
        scale=c("row", "column", "none"), na.rm = TRUE,
        margins = c(5, 5), ColSideColors, RowSideColors,
        cexRow = 0.2 + 1/log10(nr), cexCol = 0.2 + 1/log10(nc),
        labRow = NULL, labCol = NULL, main = NULL,
        xlab = NULL, ylab = NULL,
        keep.dendro = FALSE, verbose = getOption("verbose"), ...)
```

Arguments

`x`	numeric matrix of the values to be plotted.
`Rowv`	determines if and how the *row* dendrogram should be computed and reordered. Either a `dendrogram` or a vector of values used to reorder the row dendrogram or `NA` to suppress any row dendrogram (and reordering) or by default, `NULL`, see *Details* below.
`Colv`	determines if and how the *column* dendrogram should be reordered. Has the same options as the `Rowv` argument above and *additionally* when `x` is a square matrix, `Colv = "Rowv"` means that columns should be treated identically to the rows.
`distfun`	function used to compute the distance (dissimilarity) between both rows and columns. Defaults to `dist`.
`hclustfun`	function used to compute the hierarchical clustering when `Rowv` or `Colv` are not dendrograms. Defaults to `hclust`.

reorderfun	function(d,w) of dendrogram and weights for reordering the row and column dendrograms. The default uses reorder.dendrogram.
add.expr	expression that will be evaluated after the call to image. Can be used to add components to the plot.
symm	logical indicating if x should be treated **symm**etrically; can only be true when x is a square matrix.
revC	logical indicating if the column order should be reversed for plotting, such that e.g., for the symmetric case, the symmetry axis is as usual.
scale	character indicating if the values should be centered and scaled in either the row direction or the column direction, or none. The default is "row" if symm false, and "none" otherwise.
na.rm	logical indicating whether NA's should be removed.
margins	numeric vector of length 2 containing the margins (see par(mar= *)) for column and row names, respectively.
ColSideColors	(optional) character vector of length ncol(x) containing the color names for a horizontal side bar that may be used to annotate the columns of x.
RowSideColors	(optional) character vector of length nrow(x) containing the color names for a vertical side bar that may be used to annotate the rows of x.
cexRow, cexCol	positive numbers, used as cex.axis in for the row or column axis labeling. The defaults currently only use number of rows or columns, respectively.
labRow, labCol	character vectors with row and column labels to use; these default to rownames(x) or colnames(x), respectively.
main, xlab, ylab	main, x- and y-axis titles; defaults to none.
keep.dendro	logical indicating if the dendrogram(s) should be kept as part of the result (when Rowv and/or Colv are not NA).
verbose	logical indicating if information should be printed.
...	additional arguments passed on to image, e.g., col specifying the colors.

Details

If either Rowv or Colv are dendrograms they are honored (and not reordered). Otherwise, dendrograms are computed as dd <- as.dendrogram(hclustfun(distfun(X))) where X is either x or t(x).

If either is a vector (of "weights") then the appropriate dendrogram is reordered according to the supplied values subject to the constraints imposed by the dendrogram, by reorder(dd, Rowv), in the row case. If either is missing, as by default, then the ordering of the corresponding dendrogram is by the mean value of the rows/columns, i.e., in the case of rows, Rowv <- rowMeans(x, na.rm=na.rm). If either is NULL, *no reordering* will be done for the corresponding side.

By default (scale = "row") the rows are scaled to have mean zero and standard deviation one. There is some empirical evidence from genomic plotting that this is useful.

The default colors are not pretty. Consider using enhancements such as the **RColorBrewer** package, http://cran.r-project.org/src/contrib/PACKAGES.html#RColorBrewer.

Value

Invisibly, a list with components

`rowInd`	**r**ow index permutation vector as returned by `order.dendrogram`.
`colInd`	**c**olumn index permutation vector.
`Rowv`	the row dendrogram; only if input `Rowv` was not NA and `keep.dendro` is true.
`Colv`	the column dendrogram; only if input `Colv` was not NA and `keep.dendro` is true.

Note

Unless `Rowv = NA` (or `Colw = NA`), the original rows and columns are reordered *in any case* to match the dendrogram, e.g., the rows by `order.dendrogram(Rowv)` where `Rowv` is the (possibly `reorder()`ed) row dendrogram.

`heatmap()` uses `layout` and draws the `image` in the lower right corner of a 2x2 layout. Consequentially, it can **not** be used in a multi column/row layout, i.e., when `par(mfrow= *)` or `(mfcol= *)` has been called.

Author(s)

Andy Liaw, original; R. Gentleman, M. Maechler, W. Huber, revisions.

See Also

`image`, `hclust`

Examples

```
require(graphics)
x  <- as.matrix(mtcars)
rc <- rainbow(nrow(x), start=0, end=.3)
cc <- rainbow(ncol(x), start=0, end=.3)
hv <- heatmap(x, col = cm.colors(256), scale="column",
              RowSideColors = rc, ColSideColors = cc, margin=c(5,10),
              xlab = "specification variables", ylab= "Car Models",
              main = "heatmap(<Mtcars data>, ..., scale = \"column\")")
str(hv) # the two re-ordering index vectors

## no column dendrogram (nor reordering) at all:
heatmap(x, Colv = NA, col = cm.colors(256), scale="column",
        RowSideColors = rc, margin=c(5,10),
        xlab = "specification variables", ylab= "Car Models",
        main = "heatmap(<Mtcars data>, ..., scale = \"column\")")

## "no nothing"
heatmap(x, Rowv = NA, Colv = NA, scale="column",
        main = "heatmap(*, NA, NA) ~= image(t(x))")

round(Ca <- cor(attitude), 2)
symnum(Ca) # simple graphic
heatmap(Ca,             symm = TRUE, margin=c(6,6))# with reorder()
heatmap(Ca, Rowv=FALSE, symm = TRUE, margin=c(6,6))# _NO_ reorder()
```

```
## For variable clustering, rather use distance based on cor():
symnum( cU <- cor(USJudgeRatings) )

hU <- heatmap(cU, Rowv = FALSE, symm = TRUE, col = topo.colors(16),
              distfun = function(c) as.dist(1 - c), keep.dendro = TRUE)
## The Correlation matrix with same reordering:
round(100 * cU[hU[[1]], hU[[2]]])
## The column dendrogram:
str(hU$Colv)
```

`HoltWinters` *Holt-Winters Filtering*

Description

Computes Holt-Winters Filtering of a given time series. Unknown parameters are determined by minimizing the squared prediction error.

Usage

```
HoltWinters(x, alpha = NULL, beta = NULL, gamma = NULL,
            seasonal = c("additive", "multiplicative"),
            start.periods = 3, l.start = NULL, b.start = NULL,
            s.start = NULL,
            optim.start = c(alpha = 0.3, beta = 0.1, gamma = 0.1),
            optim.control = list())
```

Arguments

`x`	An object of class `ts`
`alpha`	$alpha$ parameter of Holt-Winters Filter
`beta`	$beta$ parameter of Holt-Winters Filter. If set to 0, the function will do exponential smoothing.
`gamma`	$gamma$ parameter used for the seasonal component. If set to 0, an non-seasonal model is fitted.
`seasonal`	Character string to select an `"additive"` (the default) or `"multiplicative"` seasonal model. The first few characters are sufficient. (Only takes effect if `gamma` is non-zero).
`start.periods`	Start periods used in the autodetection of start values. Must be at least 3.
`l.start`	Start value for level (a[0]).
`b.start`	Start value for trend (b[0]).
`s.start`	Vector of start values for the seasonal component ($s_1[0]...s_p[0]$)
`optim.start`	Vector with named components `alpha`, `beta`, and `gamma` containing the starting values for the optimizer. Only the values needed must be specified.
`optim.control`	Optional list with additional control parameters passed to `optim`.

Details

The additive Holt-Winters prediction function (for time series with period length p) is

$$\hat{Y}[t+h] = a[t] + hb[t] + s[t+1+(h-1) \bmod p],$$

where $a[t]$, $b[t]$ and $s[t]$ are given by

$$a[t] = \alpha(Y[t] - s[t-p]) + (1-\alpha)(a[t-1] + b[t-1])$$

$$b[t] = \beta(a[t] - a[t-1]) + (1-\beta)b[t-1]$$

$$s[t] = \gamma(Y[t] - a[t]) + (1-\gamma)s[t-p]$$

The multiplicative Holt-Winters prediction function (for time series with period length p) is

$$\hat{Y}[t+h] = (a[t] + hb[t]) \times s[t+1+(h-1) \bmod p].$$

where $a[t]$, $b[t]$ and $s[t]$ are given by

$$a[t] = \alpha(Y[t]/s[t-p]) + (1-\alpha)(a[t-1] + b[t-1])$$

$$b[t] = \beta(a[t] - a[t-1]) + (1-\beta)b[t-1]$$

$$s[t] = \gamma(Y[t]/a[t]) + (1-\gamma)s[t-p]$$

The function tries to find the optimal values of α and/or β and/or γ by minimizing the squared one-step prediction error if they are omitted.

For seasonal models, start values for `a`, `b` and `s` are detected by performing a simple decomposition in trend and seasonal component using moving averages (see function `decompose`) on the `start.periods` first periods (a simple linear regression on the trend component is used for starting level and trend.). For level/trend-models (no seasonal component), start values for a and b are x[2] and x[2] - x[1], respectively. For level-only models (ordinary exponential smoothing), the start value for a is x[1].

Value

An object of class `"HoltWinters"`, a list with components:

`fitted`	A multiple time series with one column for the filtered series as well as for the level, trend and seasonal components, estimated contemporaneously (that is at time t and not at the end of the series).
`x`	The original series
`alpha`	alpha used for filtering
`beta`	beta used for filtering
`coefficients`	A vector with named components `a, b, s1, ..., sp` containing the estimated values for the level, trend and seasonal components
`seasonal`	The specified `seasonal`-parameter
`SSE`	The final sum of squared errors achieved in optimizing
`call`	The call used

Author(s)

David Meyer ⟨David.Meyer@wu-wien.ac.at⟩

References

C. C. Holt (1957) Forecasting seasonals and trends by exponentially weighted moving averages, ONR Research Memorandum, Carnigie Institute 52.

P. R. Winters (1960) Forecasting sales by exponentially weighted moving averages, *Management Science* **6**, 324–342.

See Also

`predict.HoltWinters`,`optim`

Examples

```
## Seasonal Holt-Winters
(m <- HoltWinters(co2))
plot(m)
plot(fitted(m))

(m <- HoltWinters(AirPassengers, seasonal = "mult"))
plot(m)

## Non-Seasonal Holt-Winters
x <- uspop + rnorm(uspop, sd = 5)
m <- HoltWinters(x, gamma = 0)
plot(m)

## Exponential Smoothing
m2 <- HoltWinters(x, gamma = 0, beta = 0)
lines(fitted(m2)[,1], col = 3)
```

`Hypergeometric` *The Hypergeometric Distribution*

Description

Density, distribution function, quantile function and random generation for the hypergeometric distribution.

Usage

```
dhyper(x, m, n, k, log = FALSE)
phyper(q, m, n, k, lower.tail = TRUE, log.p = FALSE)
qhyper(p, m, n, k, lower.tail = TRUE, log.p = FALSE)
rhyper(nn, m, n, k)
```

Arguments

`x, q`	vector of quantiles representing the number of white balls drawn without replacement from an urn which contains both black and white balls.
`m`	the number of white balls in the urn.
`n`	the number of black balls in the urn.
`k`	the number of balls drawn from the urn.

`p`	probability, it must be between 0 and 1.
`nn`	number of observations. If `length(nn) > 1`, the length is taken to be the number required.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

The hypergeometric distribution is used for sampling *without* replacement. The density of this distribution with parameters `m`, `n` and `k` (named Np, $N - Np$, and n, respectively in the reference below) is given by

$$p(x) = \binom{m}{x}\binom{n}{k-x} \Big/ \binom{m+n}{k}$$

for $x = 0, \ldots, k$.

Value

`dhyper` gives the density, `phyper` gives the distribution function, `qhyper` gives the quantile function, and `rhyper` generates random deviates.

Invalid arguments will result in return value `NaN`, with a warning.

Source

`dhyper` computes via binomial probabilities, using code contributed by Catherine Loader (see `dbinom`).

`phyper` is based on calculating `dhyper` and `phyper(...)/dhyper(...)` (as a summation), based on ideas of Ian Smith and Morten Welinder.

`qhyper` is based on inversion.

`rhyper` is based on a corrected version of

Kachitvichyanukul, V. and Schmeiser, B. (1985). Computer generation of hypergeometric random variates. *Journal of Statistical Computation and Simulation*, **22**, 127–145.

References

Johnson, N. L., Kotz, S., and Kemp, A. W. (1992) *Univariate Discrete Distributions*, Second Edition. New York: Wiley.

Examples

```
m <- 10; n <- 7; k <- 8
x <- 0:(k+1)
rbind(phyper(x, m, n, k), dhyper(x, m, n, k))
all(phyper(x, m, n, k) == cumsum(dhyper(x, m, n, k)))# FALSE
## but error is very small:
signif(phyper(x, m, n, k) - cumsum(dhyper(x, m, n, k)), dig=3)
```

`identify.hclust` *Identify Clusters in a Dendrogram*

Description

`identify.hclust` reads the position of the graphics pointer when the (first) mouse button is pressed. It then cuts the tree at the vertical position of the pointer and highlights the cluster containing the horizontal position of the pointer. Optionally a function is applied to the index of data points contained in the cluster.

Usage

```
## S3 method for class 'hclust':
identify(x, FUN = NULL, N = 20, MAXCLUSTER = 20, DEV.FUN = NULL,
         ...)
```

Arguments

`x`	an object of the type produced by `hclust`.
`FUN`	(optional) function to be applied to the index numbers of the data points in a cluster (see Details below).
`N`	the maximum number of clusters to be identified.
`MAXCLUSTER`	the maximum number of clusters that can be produced by a cut (limits the effective vertical range of the pointer).
`DEV.FUN`	(optional) integer scalar. If specified, the corresponding graphics device is made active before `FUN` is applied.
`...`	further arguments to `FUN`.

Details

By default clusters can be identified using the mouse and an `invisible` list of indices of the respective data points is returned.

If `FUN` is not `NULL`, then the index vector of data points is passed to this function as first argument, see the examples below. The active graphics device for `FUN` can be specified using `DEV.FUN`.

The identification process is terminated by pressing any mouse button other than the first, see also `identify`.

Value

Either a list of data point index vectors or a list of return values of `FUN`.

See Also

`hclust`, `rect.hclust`

Examples

```
## Not run:
hca <- hclust(dist(USArrests))
plot(hca)
(x <- identify(hca)) ##  Terminate with 2nd mouse button !!

hci <- hclust(dist(iris[,1:4]))
plot(hci)
identify(hci, function(k) print(table(iris[k,5])))

# open a new device (one for dendrogram, one for bars):
get(getOption("device"))() # << make that narrow (& small)
                           # and *beside* 1st one
nD <- dev.cur()            # to be for the barplot
dev.set(dev.prev())# old one for dendrogram
plot(hci)
## select subtrees in dendrogram and "see" the species distribution:
identify(hci, function(k) barplot(table(iris[k,5]),col=2:4), DEV.FUN = nD)
## End(Not run)
```

---

influence.measures *Regression Deletion Diagnostics*

Description

This suite of functions can be used to compute some of the regression (leave-one-out deletion) diagnostics for linear and generalized linear models discussed in Belsley, Kuh and Welsch (1980), Cook and Weisberg (1982), etc.

Usage

```
influence.measures(model)

rstandard(model, ...)
## S3 method for class 'lm':
rstandard(model, infl = lm.influence(model, do.coef = FALSE),
          sd = sqrt(deviance(model)/df.residual(model)), ...)
## S3 method for class 'glm':
rstandard(model, infl = lm.influence(model, do.coef = FALSE),
          ...)

rstudent(model, ...)
## S3 method for class 'lm':
rstudent(model, infl = lm.influence(model, do.coef = FALSE),
         res = infl$wt.res, ...)
## S3 method for class 'glm':
rstudent(model, infl = influence(model, do.coef = FALSE), ...)

dffits(model, infl = , res = )

dfbeta(model, ...)
## S3 method for class 'lm':
```

```
dfbeta(model, infl = lm.influence(model, do.coef = TRUE), ...)

dfbetas(model, ...)
## S3 method for class 'lm':
dfbetas(model, infl = lm.influence(model, do.coef = TRUE), ...)

covratio(model, infl = lm.influence(model, do.coef = FALSE),
         res = weighted.residuals(model))

cooks.distance(model, ...)
## S3 method for class 'lm':
cooks.distance(model, infl = lm.influence(model, do.coef = FALSE),
               res = weighted.residuals(model),
               sd = sqrt(deviance(model)/df.residual(model)),
               hat = infl$hat, ...)
## S3 method for class 'glm':
cooks.distance(model, infl = influence(model, do.coef = FALSE),
               res = infl$pear.res,
               dispersion = summary(model)$dispersion,
               hat = infl$hat, ...)

hatvalues(model, ...)
## S3 method for class 'lm':
hatvalues(model, infl = lm.influence(model, do.coef = FALSE), ...)

hat(x, intercept = TRUE)
```

Arguments

`model`	an R object, typically returned by `lm` or `glm`.
`infl`	influence structure as returned by `lm.influence` or `influence` (the latter only for the `glm` method of `rstudent` and `cooks.distance`).
`res`	(possibly weighted) residuals, with proper default.
`sd`	standard deviation to use, see default.
`dispersion`	dispersion (for `glm` objects) to use, see default.
`hat`	hat values H_{ii}, see default.
`x`	the X or design matrix.
`intercept`	should an intercept column be pre-prended to `x`?
`...`	further arguments passed to or from other methods.

Details

The primary high-level function is `influence.measures` which produces a class `"infl"` object tabular display showing the DFBETAS for each model variable, DFFITS, covariance ratios, Cook's distances and the diagonal elements of the hat matrix. Cases which are influential with respect to any of these measures are marked with an asterisk.

The functions `dfbetas`, `dffits`, `covratio` and `cooks.distance` provide direct access to the corresponding diagnostic quantities. Functions `rstandard` and `rstudent` give the standardized and Studentized residuals respectively. (These re-normalize the residuals to have unit variance, using an overall and leave-one-out measure of the error variance respectively.)

Values for generalized linear models are approximations, as described in Williams (1987) (except that Cook's distances are scaled as F rather than as chi-square values). The approximations can be poor when some cases have large influence.

The optional `infl`, `res` and `sd` arguments are there to encourage the use of these direct access functions, in situations where, e.g., the underlying basic influence measures (from `lm.influence` or the generic `influence`) are already available.

Note that cases with `weights == 0` are *dropped* from all these functions, but that if a linear model has been fitted with `na.action = na.exclude`, suitable values are filled in for the cases excluded during fitting.

The function `hat()` exists mainly for S (version 2) compatibility; we recommend using `hatvalues()` instead.

Note

For `hatvalues`, `dfbeta`, and `dfbetas`, the method for linear models also works for generalized linear models.

Author(s)

Several R core team members and John Fox, originally in his 'car' package.

References

Belsley, D. A., Kuh, E. and Welsch, R. E. (1980) *Regression Diagnostics*. New York: Wiley.

Cook, R. D. and Weisberg, S. (1982) *Residuals and Influence in Regression*. London: Chapman and Hall.

Williams, D. A. (1987) Generalized linear model diagnostics using the deviance and single case deletions. *Applied Statistics* **36**, 181–191.

Fox, J. (1997) *Applied Regression, Linear Models, and Related Methods*. Sage.

Fox, J. (2002) *An R and S-Plus Companion to Applied Regression*. Sage Publ.; http://www.socsci.mcmaster.ca/jfox/Books/Companion/.

See Also

`influence` (containing `lm.influence`).

Examples

```
## Analysis of the life-cycle savings data
## given in Belsley, Kuh and Welsch.
lm.SR <- lm(sr ~ pop15 + pop75 + dpi + ddpi, data = LifeCycleSavings)

inflm.SR <- influence.measures(lm.SR)
which(apply(inflm.SR$is.inf, 1, any)) # which observations 'are' influential
summary(inflm.SR) # only these
inflm.SR          # all
plot(rstudent(lm.SR) ~ hatvalues(lm.SR)) # recommended by some

## The 'infl' argument is not needed, but avoids recomputation:
rs <- rstandard(lm.SR)
iflSR <- influence(lm.SR)
identical(rs, rstandard(lm.SR, infl = iflSR))
## to "see" the larger values:
```

```
1000 * round(dfbetas(lm.SR, infl = iflSR), 3)

## Huber's data [Atkinson 1985]
xh <- c(-4:0, 10)
yh <- c(2.48, .73, -.04, -1.44, -1.32, 0)
summary(lmH <- lm(yh ~ xh))
(im <- influence.measures(lmH))
plot(xh,yh, main = "Huber's data: L.S. line and influential obs.")
abline(lmH); points(xh[im$is.inf], yh[im$is.inf], pch=20, col=2)
```

`integrate` *Integration of One-Dimensional Functions*

Description

Adaptive quadrature of functions of one variable over a finite or infinite interval.

Usage

```
integrate(f, lower, upper, subdivisions=100,
          rel.tol = .Machine$double.eps^0.25, abs.tol = rel.tol,
          stop.on.error = TRUE, keep.xy = FALSE, aux = NULL, ...)
```

Arguments

`f`	an R function taking a numeric first argument and returning a numeric vector of the same length. Returning a non-finite element will generate an error.
`lower, upper`	the limits of integration. Can be infinite.
`subdivisions`	the maximum number of subintervals.
`rel.tol`	relative accuracy requested.
`abs.tol`	absolute accuracy requested.
`stop.on.error`	logical. If true (the default) an error stops the function. If false some errors will give a result with a warning in the `message` component.
`keep.xy`	unused. For compatibility with S.
`aux`	unused. For compatibility with S.
`...`	additional arguments to be passed to `f`. Remember to use argument names *not* matching those of `integrate(.)`!

Details

If one or both limits are infinite, the infinite range is mapped onto a finite interval.

For a finite interval, globally adaptive interval subdivision is used in connection with extrapolation by the Epsilon algorithm.

`rel.tol` cannot be less than `max(50*.Machine$double.eps, 0.5e-28)` if `abs.tol <= 0`.

Value

A list of class "integrate" with components

value	the final estimate of the integral.
abs.error	estimate of the modulus of the absolute error.
subdivisions	the number of subintervals produced in the subdivision process.
message	"OK" or a character string giving the error message.
call	the matched call.

Note

Like all numerical integration routines, these evaluate the function on a finite set of points. If the function is approximately constant (in particular, zero) over nearly all its range it is possible that the result and error estimate may be seriously wrong.

When integrating over infinite intervals do so explicitly, rather than just using a large number as the endpoint. This increases the chance of a correct answer – any function whose integral over an infinite interval is finite must be near zero for most of that interval.

f must accept a vector of inputs and produce a vector of function evaluations at those points. The Vectorize function may be helpful to convert f to this form.

References

Based on QUADPACK routines dqags and dqagi by R. Piessens and E. deDoncker-Kapenga, available from Netlib.

See
R. Piessens, E. deDoncker-Kapenga, C. Uberhuber, D. Kahaner (1983) *Quadpack: a Subroutine Package for Automatic Integration*; Springer Verlag.

See Also

The function adapt in the **adapt** package on CRAN, for multivariate integration.

Examples

```
integrate(dnorm, -1.96, 1.96)
integrate(dnorm, -Inf, Inf)

## a slowly-convergent integral
integrand <- function(x) {1/((x+1)*sqrt(x))}
integrate(integrand, lower = 0, upper = Inf)

## don't do this if you really want the integral from 0 to Inf
integrate(integrand, lower = 0, upper = 10)
integrate(integrand, lower = 0, upper = 100000)
integrate(integrand, lower = 0, upper = 1000000, stop.on.error = FALSE)

## some functions do not handle vector input properly
f <- function(x) 2
try(integrate(f, 0, 1))
integrate(Vectorize(f), 0, 1)  ## correct
integrate(function(x) rep(2, length(x)), 0, 1)  ## correct
```

```
## integrate can fail if misused
integrate(dnorm,0,2)
integrate(dnorm,0,20)
integrate(dnorm,0,200)
integrate(dnorm,0,2000)
integrate(dnorm,0,20000) ## fails on many systems
integrate(dnorm,0,Inf)   ## works
```

`interaction.plot` *Two-way Interaction Plot*

Description

Plots the mean (or other summary) of the response for two-way combinations of factors, thereby illustrating possible interactions.

Usage

```
interaction.plot(x.factor, trace.factor, response, fun = mean,
                 type = c("l", "p", "b"), legend = TRUE,
                 trace.label = deparse(substitute(trace.factor)),
                 fixed = FALSE,
                 xlab = deparse(substitute(x.factor)),
                 ylab = ylabel,
                 ylim = range(cells, na.rm=TRUE),
                 lty = nc:1, col = 1, pch = c(1:9, 0, letters),
                 xpd = NULL, leg.bg = par("bg"), leg.bty = "n",
                 xtick = FALSE, xaxt = par("xaxt"), axes = TRUE,
                 ...)
```

Arguments

`x.factor`	a factor whose levels will form the x axis.
`trace.factor`	another factor whose levels will form the traces.
`response`	a numeric variable giving the response
`fun`	the function to compute the summary. Should return a single real value.
`type`	the type of plot: lines or points.
`legend`	logical. Should a legend be included?
`trace.label`	overall label for the legend.
`fixed`	logical. Should the legend be in the order of the levels of `trace.factor` or in the order of the traces at their right-hand ends?
`xlab,ylab`	the x and y label of the plot each with a sensible default.
`ylim`	numeric of length 2 giving the y limits for the plot.
`lty`	line type for the lines drawn, with sensible default.
`col`	the color to be used for plotting.
`pch`	a vector of plotting symbols or characters, with sensible default.
`xpd`	determines clipping behaviour for the `legend` used, see `par(xpd)`. Per default, the legend is *not* clipped at the figure border.

`leg.bg, leg.bty`
: arguments passed to `legend()`.

`xtick`
: logical. Should tick marks be used on the x axis?

`xaxt, axes, ...`
: graphics parameters to be passed to the plotting routines.

Details

By default the levels of `x.factor` are plotted on the x axis in their given order, with extra space left at the right for the legend (if specified). If `x.factor` is an ordered factor and the levels are numeric, these numeric values are used for the x axis.

The response and hence its summary can contain missing values. If so, the missing values and the line segments joining them are omitted from the plot (and this can be somewhat disconcerting).

The graphics parameters `xlab`, `ylab`, `ylim`, `lty`, `col` and `pch` are given suitable defaults (and `xlim` and `xaxs` are set and cannot be overriden). The defaults are to cycle through the line types, use the foreground colour, and to use the symbols 1:9, 0, and the capital letters to plot the traces.

Note

Some of the argument names and the precise behaviour are chosen for S-compatibility.

References

Chambers, J. M., Freeny, A and Heiberger, R. M. (1992) *Analysis of variance; designed experiments.* Chapter 5 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

Examples

```
with(ToothGrowth, {
interaction.plot(dose, supp, len, fixed=TRUE)
dose <- ordered(dose)
interaction.plot(dose, supp, len, fixed=TRUE, col = 2:3, leg.bty = "o")
interaction.plot(dose, supp, len, fixed=TRUE, col = 2:3, type = "p")
})

with(OrchardSprays, {
  interaction.plot(treatment, rowpos, decrease)
  interaction.plot(rowpos, treatment, decrease, cex.axis=0.8)
  ## order the rows by their mean effect
  rowpos <- factor(rowpos, levels=sort.list(tapply(decrease, rowpos, mean)))
  interaction.plot(rowpos, treatment, decrease, col = 2:9, lty = 1)
})

with(esoph, {
  interaction.plot(agegp, alcgp, ncases/ncontrols, main = "'esoph' Data")
  interaction.plot(agegp, tobgp, ncases/ncontrols, trace.label="tobacco",
                   fixed=TRUE, xaxt = "n")
})
## deal with NAs:
esoph[66,] # second to last age group: 65-74
esophNA <- esoph; esophNA$ncases[66] <- NA
with(esophNA, {
  interaction.plot(agegp, alcgp, ncases/ncontrols, col= 2:5)
                                # doesn't show *last* group either
```

```
  interaction.plot(agegp, alcgp, ncases/ncontrols, col= 2:5, type = "b")
  ## alternative take non-NA's  {"cheating"}
  interaction.plot(agegp, alcgp, ncases/ncontrols, col= 2:5,
                   fun = function(x) mean(x, na.rm=TRUE),
                   sub = "function(x) mean(x, na.rm=TRUE)")
})
rm(esophNA) # to clear up
```

`IQR` *The Interquartile Range*

Description

computes interquartile range of the `x` values.

Usage

```
IQR(x, na.rm = FALSE)
```

Arguments

`x`	a numeric vector.
`na.rm`	logical. Should missing values be removed?

Details

Note that this function computes the quartiles using the `quantile` function rather than following Tukey's recommendations, i.e., `IQR(x) = quantile(x,3/4) - quantile(x,1/4)`.

For normally $N(m, 1)$ distributed X, the expected value of `IQR(X)` is `2*qnorm(3/4) = 1.3490`, i.e., for a normal-consistent estimate of the standard deviation, use `IQR(x) / 1.349`.

References

Tukey, J. W. (1977). *Exploratory Data Analysis.* Reading: Addison-Wesley.

See Also

`fivenum`, `mad` which is more robust, `range`, `quantile`.

Examples

```
IQR(rivers)
```

is.empty.model *Check if a Model is Empty*

Description

R model notation allows models with no intercept and no predictors. These require special handling internally. `is.empty.model()` checks whether an object describes an empty model.

Usage

```
is.empty.model(x)
```

Arguments

`x`	A `terms` object or an object with a `terms` method.

Value

`TRUE` if the model is empty

See Also

`lm`,`glm`

Examples

```
y <- rnorm(20)
is.empty.model(y ~ 0)
is.empty.model(y ~ -1)
is.empty.model(lm(y ~ 0))
```

isoreg *Isotonic / Monotone Regression*

Description

Compute the isotonic (monotonely increasing nonparametric) least squares regression which is piecewise constant.

Usage

```
isoreg(x, y = NULL)
```

Arguments

`x, y`	in `isoreg`, coordinate vectors of the regression points. Alternatively a single "plotting" structure can be specified: see `xy.coords`.

Details

The algorithm determines the convex minorant $m(x)$ of the *cumulative* data (i.e., `cumsum(y)`) which is piecewise linear and the result is $m'(x)$, a step function with level changes at locations where the convex $m(x)$ touches the cumulative data polygon and changes slope.
`as.stepfun()` returns a `stepfun` object which can be more parsimonious.

Value

`isoreg()` returns an object of class `isoreg` which is basically a list with components

`x`	original (constructed) abscissa values `x`.
`y`	corresponding y values.
`yf`	fitted values corresponding to *ordered* x values.
`yc`	cumulative y values corresponding to *ordered* x values.
`iKnots`	integer vector giving indices where the fitted curve jumps, i.e., where the convex minorant has kinks.
`isOrd`	logical indicating if original x values were ordered increasingly already.
`ord`	`if(!isOrd)`: integer permutation `order(x)` of *original* `x`.
`call`	the `call` to `isoreg()` used.

Note

The code should be improved to accept *weights* additionally and solve the corresponding weighted least squares problem.
"Patches are welcome!"

References

Barlow, R. E., Bartholomew, D. J., Bremner, J. M., and Brunk, H. D. (1972) *Statistical inference under order restrictions*; Wiley, London.

Robertson, T., Wright,F. T. and Dykstra, R. L. (1988) *Order Restricted Statistical Inference*; Wiley, New York.

See Also

the plotting method `plot.isoreg` with more examples; `isoMDS()` from the **MASS** package internally uses isotonic regression.

Examples

```
(ir <- isoreg(c(1,0,4,3,3,5,4,2,0)))
plot(ir, plot.type = "row")

(ir3 <- isoreg(y3 <- c(1,0,4,3,3,5,4,2, 3)))# last "3", not "0"
(fi3 <- as.stepfun(ir3))
(ir4 <- isoreg(1:10, y4 <- c(5, 9, 1:2, 5:8, 3, 8)))
cat("R^2 =", formatC(sum(residuals(ir4)^2) / (9*var(y4)), dig=2),"\n")
```

KalmanLike *Kalman Filtering*

Description

Use Kalman Filtering to find the (Gaussian) log-likelihood, or for forecasting or smoothing.

Usage

```
KalmanLike(y, mod, nit = 0, fast=TRUE)
KalmanRun(y, mod, nit = 0, fast=TRUE)
KalmanSmooth(y, mod, nit = 0)
KalmanForecast(n.ahead = 10, mod, fast=TRUE)
makeARIMA(phi, theta, Delta, kappa = 1e6)
```

Arguments

`y`	a univariate time series.
`mod`	A list describing the state-space model: see Details.
`nit`	The time at which the initialization is computed. `nit = 0` implies that the initialization is for a one-step prediction, so `Pn` should not be computed at the first step.
`n.ahead`	The number of steps ahead for which prediction is required.
`phi, theta`	numeric vectors of length ≥ 0 giving AR and MA parameters.
`Delta`	vector of differencing coefficients, so an ARMA model is fitted to `y[t] - Delta[1]*y[t-1] - ...`.
`kappa`	the prior variance (as a multiple of the innovations variance) for the past observations in a differenced model.
`fast`	If `TRUE` the `mod` object may be modified.

Details

These functions work with a general univariate state-space model with state vector `a`, transitions `a <- T a + R e`, $e \sim \mathcal{N}(0, \kappa Q)$ and observation equation `y = Z'a + eta`, $(eta \equiv \eta), \eta \sim \mathcal{N}(0, \kappa h)$. The likelihood is a profile likelihood after estimation of κ.

The model is specified as a list with at least components

`T` the transition matrix

`Z` the observation coeficients

`h` the observation variance

`V` `RQR'`

`a` the current state estimate

`P` the current estimate of the state uncertainty matrix

`Pn` the estimate at time $t-1$ of the state uncertainty matrix

`KalmanSmooth` is the workhorse function for `tsSmooth`.

`makeARIMA` constructs the state-space model for an ARIMA model.

Value

For `KalmanLike`, a list with components `Lik` (the log-likelihood less some constants) and `s2`, the estimate of of κ.

For `KalmanRun`, a list with components `values`, a vector of length 2 giving the output of `KalmanLike`, `resid` (the residuals) and `states`, the contemporaneous state estimates, a matrix with one row for each time.

For `KalmanSmooth`, a list with two components. Component `smooth` is a `n` by `p` matrix of state estimates based on all the observations, with one row for each time. Component `var` is a `n` by `p` by `p` array of variance matrices.

For `KalmanForecast`, a list with components `pred`, the predictions, and `var`, the unscaled variances of the prediction errors (to be muliplied by `s2`).

For `makeARIMA`, a model list including components for its arguments.

Warning

These functions are designed to be called from other functions which check the validity of the arguments passed, so very little checking is done.

In particular, `KalmanLike` alters the objects passed as the elements `a`, `P` and `Pn` of `mod`, so these should not be shared. Use `fast=FALSE` to prevent this.

References

Durbin, J. and Koopman, S. J. (2001) *Time Series Analysis by State Space Methods.* Oxford University Press.

See Also

`arima`, `StructTS`. `tsSmooth`.

`kernapply` *Apply Smoothing Kernel*

Description

`kernapply` computes the convolution between an input sequence and a specific kernel.

Usage

```
kernapply(x, ...)

## Default S3 method:
kernapply(x, k, circular = FALSE, ...)
## S3 method for class 'ts':
kernapply(x, k, circular = FALSE, ...)
## S3 method for class 'vector':
kernapply(x, k, circular = FALSE, ...)

## S3 method for class 'tskernel':
kernapply(x, k, ...)
```

Arguments

`x`	an input vector, matrix, time series or kernel to be smoothed.
`k`	smoothing `"tskernel"` object.
`circular`	a logical indicating whether the input sequence to be smoothed is treated as circular, i.e., periodic.
`...`	arguments passed to or from other methods.

Value

A smoothed version of the input sequence.

Author(s)

A. Trapletti

See Also

`kernel`, `convolve`, `filter`, `spectrum`

Examples

```
## see 'kernel' for examples
```

`kernel` *Smoothing Kernel Objects*

Description

The `"tskernel"` class is designed to represent discrete symmetric normalized smoothing kernels. These kernels can be used to smooth vectors, matrices, or time series objects.

There are `print`, `plot` and `[` methods for these kernel objects.

Usage

```
kernel(coef, m, r, name)

df.kernel(k)
bandwidth.kernel(k)
is.tskernel(k)
## S3 method for class 'tskernel':
plot(x, type = "h", xlab = "k", ylab = "W[k]",
     main = attr(x,"name"), ...)
```

Arguments

coef	the upper half of the smoothing kernel coefficients (including coefficient zero) *or* the name of a kernel (currently "daniell", "dirichlet", "fejer" or "modified.daniell".
m	the kernel dimension(s). When m has length larger than one, it means the convolution of kernels of dimension m[j], for j in 1:length(m). Currently this is supported only for the named "*daniell" kernels.
name	the name the kernel will be called.
r	the kernel order for a Fejer kernel.
k,x	a "tskernel" object.
type, xlab, ylab, main, ...	arguments passed to plot.default.

Details

kernel is used to construct a general kernel or named specific kernels. The modified Daniell kernel halves the end coefficients (as used by S-PLUS).

The [method allows natural indexing of kernel objects with indices in (-m) : m. The normalization is such that for k <- kernel(*), sum(k[-k$m : k$m]) is one.

df.kernel returns the "equivalent degrees of freedom" of a smoothing kernel as defined in Brockwell and Davies (1991), page 362, and bandwidth.kernel returns the equivalent bandwidth as defined in Bloomfield (1976), p. 201, with a continuity correction.

Value

kernel() returns an object of class "tskernel" which is basically a list with the two components coef and the kernel dimension m. An additional attribute is "name".

Author(s)

A. Trapletti; modifications by B.D. Ripley

References

Bloomfield, P. (1976) *Fourier Analysis of Time Series: An Introduction.* Wiley.

Brockwell, P.J. and Davis, R.A. (1991) *Time Series: Theory and Methods.* Second edition. Springer, pp. 350–365.

See Also

kernapply

Examples

```
## Demonstrate a simple trading strategy for the
## financial time series German stock index DAX.
x <- EuStockMarkets[,1]
k1 <- kernel("daniell", 50)  # a long moving average
k2 <- kernel("daniell", 10)  # and a short one
plot(k1)
plot(k2)
```

```
x1 <- kernapply(x, k1)
x2 <- kernapply(x, k2)
plot(x)
lines(x1, col = "red")    # go long if the short crosses the long upwards
lines(x2, col = "green")  # and go short otherwise

## More interesting kernels
kd <- kernel("daniell", c(3,3))
kd # note the unusual indexing
kd[-2:2]
plot(kernel("fejer", 100, r=6))
plot(kernel("modified.daniell", c(7,5,3)))

# Reproduce example 10.4.3 from Brockwell and Davies (1991)
spectrum(sunspot.year, kernel=kernel("daniell", c(11,7,3)), log="no")
```

`kmeans` *K-Means Clustering*

Description

Perform k-means clustering on a data matrix.

Usage

```
kmeans(x, centers, iter.max = 10, nstart = 1,
       algorithm = c("Hartigan-Wong", "Lloyd", "Forgy", "MacQueen"))
```

Arguments

`x`	A numeric matrix of data, or an object that can be coerced to such a matrix (such as a numeric vector or a data frame with all numeric columns).
`centers`	Either the number of clusters or a set of initial (distinct) cluster centres. If a number, a random set of (distinct) rows in `x` is chosen as the initial centres.
`iter.max`	The maximum number of iterations allowed.
`nstart`	If `centers` is a number, how many random sets should be chosen?
`algorithm`	character: may be abbreviated.

Details

The data given by `x` is clustered by the k-means method, which aims to partition the points into k groups such that the sum of squares from points to the assigned cluster centres is minimized. At the minimum, all cluster centres are at the mean of their Voronoi sets (the set of data points which are nearest to the cluster centre).

The algorithm of Hartigan and Wong (1979) is used by default. Note that some authors use k-means to refer to a specific algorithm rather than the general method: most commonly the algorithm given by MacQueen (1967) but sometimes that given by Lloyd (1957) and Forgy (1965). The Hartigan–Wong algorithm generally does a better job than either of those, but trying several random starts is often recommended.

Except for the Lloyd–Forgy method, k clusters will always be returned if a number is specified. If an initial matrix of centres is supplied, it is possible that no point will be closest to one or more centres, which is currently an error for the Hartigan–Wong method.

Value

An object of class `"kmeans"` which is a list with components:

`cluster`	A vector of integers indicating the cluster to which each point is allocated.
`centers`	A matrix of cluster centres.
`withinss`	The within-cluster sum of squares for each cluster.
`size`	The number of points in each cluster.

There is a `print` method for this class.

References

Forgy, E. W. (1965) Cluster analysis of multivariate data: efficiency vs interpretability of classifications. *Biometrics* **21**, 768–769.

Hartigan, J. A. and Wong, M. A. (1979). A K-means clustering algorithm. *Applied Statistics* **28**, 100–108.

Lloyd, S. P. (1957, 1982) Least squares quantization in PCM. Technical Note, Bell Laboratories. Published in 1982 in *IEEE Transactions on Information Theory* **28**, 128–137.

MacQueen, J. (1967) Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, eds L. M. Le Cam & J. Neyman, **1**, pp. 281–297. Berkeley, CA: University of California Press.

Examples

```
# a 2-dimensional example
x <- rbind(matrix(rnorm(100, sd = 0.3), ncol = 2),
           matrix(rnorm(100, mean = 1, sd = 0.3), ncol = 2))
colnames(x) <- c("x", "y")
(cl <- kmeans(x, 2))
plot(x, col = cl$cluster)
points(cl$centers, col = 1:2, pch = 8, cex=2)

## random starts do help here with too many clusters
(cl <- kmeans(x, 5, nstart = 25))
plot(x, col = cl$cluster)
points(cl$centers, col = 1:5, pch = 8)
```

`kruskal.test` *Kruskal-Wallis Rank Sum Test*

Description

Performs a Kruskal-Wallis rank sum test.

Usage

```
kruskal.test(x, ...)

## Default S3 method:
kruskal.test(x, g, ...)

## S3 method for class 'formula':
kruskal.test(formula, data, subset, na.action, ...)
```

Arguments

x	a numeric vector of data values, or a list of numeric data vectors.
g	a vector or factor object giving the group for the corresponding elements of x. Ignored if x is a list.
formula	a formula of the form lhs ~ rhs where lhs gives the data values and rhs the corresponding groups.
data	an optional matrix or data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).
subset	an optional vector specifying a subset of observations to be used.
na.action	a function which indicates what should happen when the data contain NAs. Defaults to getOption("na.action").
...	further arguments to be passed to or from methods.

Details

kruskal.test performs a Kruskal-Wallis rank sum test of the null that the location parameters of the distribution of x are the same in each group (sample). The alternative is that they differ in at least one.

If x is a list, its elements are taken as the samples to be compared, and hence have to be numeric data vectors. In this case, g is ignored, and one can simply use kruskal.test(x) to perform the test. If the samples are not yet contained in a list, use kruskal.test(list(x, ...)).

Otherwise, x must be a numeric data vector, and g must be a vector or factor object of the same length as x giving the group for the corresponding elements of x.

Value

A list with class "htest" containing the following components:

statistic	the Kruskal-Wallis rank sum statistic.
parameter	the degrees of freedom of the approximate chi-squared distribution of the test statistic.
p.value	the p-value of the test.
method	the character string "Kruskal-Wallis rank sum test".
data.name	a character string giving the names of the data.

References

Myles Hollander & Douglas A. Wolfe (1973), *Nonparametric statistical inference*. New York: John Wiley & Sons. Pages 115–120.

See Also

The Wilcoxon rank sum test (wilcox.test) as the special case for two samples; lm together with anova for performing one-way location analysis under normality assumptions; with Student's t test (t.test) as the special case for two samples.

Examples

```
## Hollander & Wolfe (1973), 116.
## Mucociliary efficiency from the rate of removal of dust in normal
##  subjects, subjects with obstructive airway disease, and subjects
##  with asbestosis.
x <- c(2.9, 3.0, 2.5, 2.6, 3.2) # normal subjects
y <- c(3.8, 2.7, 4.0, 2.4)      # with obstructive airway disease
z <- c(2.8, 3.4, 3.7, 2.2, 2.0) # with asbestosis
kruskal.test(list(x, y, z))
## Equivalently,
x <- c(x, y, z)
g <- factor(rep(1:3, c(5, 4, 5)),
            labels = c("Normal subjects",
                       "Subjects with obstructive airway disease",
                       "Subjects with asbestosis"))
kruskal.test(x, g)

## Formula interface.
boxplot(Ozone ~ Month, data = airquality)
kruskal.test(Ozone ~ Month, data = airquality)
```

---

ks.test *Kolmogorov-Smirnov Tests*

Description

Performs one or two sample Kolmogorov-Smirnov tests.

Usage

```
ks.test(x, y, ..., alternative = c("two.sided", "less", "greater"),
        exact = NULL)
```

Arguments

`x`	a numeric vector of data values.
`y`	either a numeric vector of data values, or a character string naming a distribution function.
`...`	parameters of the distribution specified (as a character string) by `y`.
`alternative`	indicates the alternative hypothesis and must be one of `"two.sided"` (default), `"less"`, or `"greater"`. You can specify just the initial letter of the value, but the argument name must be give in full. See Details for the meanings of the possible values.
`exact`	`NULL` or a logical indicating whether an exact p-value should be computed. See Details for the meaning of `NULL`. Not used for the one-sided two-sample case.

Details

If `y` is numeric, a two-sample test of the null hypothesis that `x` and `y` were drawn from the same *continuous* distribution is performed.

Alternatively, `y` can be a character string naming a continuous distribution function. In this case, a one-sample test is carried out of the null that the distribution function which generated `x` is distribution `y` with parameters specified by `...`.

The presence of ties generates a warning, since continuous distributions do not generate them.

The possible values `"two.sided"`, `"less"` and `"greater"` of `alternative` specify the null hypothesis that the true distribution function of `x` is equal to, not less than or not greater than the hypothesized distribution function (one-sample case) or the distribution function of `y` (two-sample case), respectively. This is a comparison of cumulative distribution functions, and the test statistic is the maximum difference in value, with the statistic in the `"greater"` alternative being $D^+ = max_u[F_x(u) - F_y(u)]$. Thus in the two-sample case `alternative="greater"` includes distributions for which `x` is stochastically *smaller* than `y` (the CDF of `x` lies above and hence to the left of that for `y`), in contrast to `t.test` or `wilcox.test`.

Exact p-values are not available for the one-sided two-sample case, or in the case of ties. If `exact = NULL` (the default), an exact p-value is computed if the sample size if less than 100 in the one-sample case, and if the product of the sample sizes is less than 10000 in the two-sample case. Otherwise, asymptotic distributions are used whose approximations may be inaccurate in small samples. In the one-sample two-sided case, exact p-values are obtained as described in Marsaglia, Tsang & Wang (2003). The formula of Birnbaum & Tingey (1951) is used for the one-sample one-sided case.

If a single-sample test is used, the parameters specified in `...` must be pre-specified and not estimated from the data. There is some more refined distribution theory for the KS test with estimated parameters (see Durbin, 1973), but that is not implemented in `ks.test`.

Value

A list with class `"htest"` containing the following components:

`statistic`	the value of the test statistic.
`p.value`	the p-value of the test.
`alternative`	a character string describing the alternative hypothesis.
`method`	a character string indicating what type of test was performed.
`data.name`	a character string giving the name(s) of the data.

References

Z. W. Birnbaum & Fred H. Tingey (1951), One-sided confidence contours for probability distribution functions. *The Annals of Mathematical Statistics*, **22**/4, 592–596.

William J. Conover (1971), *Practical Nonparametric Statistics*. New York: John Wiley & Sons. Pages 295–301 (one-sample "Kolmogorov" test), 309–314 (two-sample "Smirnov" test).

Durbin, J. (1973) *Distribution theory for tests based on the sample distribution function*. SIAM.

George Marsaglia, Wai Wan Tsang & Jingbo Wang (2003), Evaluating Kolmogorov's distribution. *Journal of Statistical Software*, **8**/18. http://www.jstatsoft.org/v08/i18/.

See Also

`shapiro.test` which performs the Shapiro-Wilk test for normality.

Examples

```
x <- rnorm(50)
y <- runif(30)
# Do x and y come from the same distribution?
ks.test(x, y)
# Does x come from a shifted gamma distribution with shape 3 and rate 2?
ks.test(x+2, "pgamma", 3, 2) # two-sided, exact
ks.test(x+2, "pgamma", 3, 2, exact = FALSE)
ks.test(x+2, "pgamma", 3, 2, alternative = "gr")

# test if x is stochastically larger than x2
x2 <- rnorm(50, -1)
plot(ecdf(x), xlim=range(c(x, x2)))
plot(ecdf(x2), add=TRUE, lty="dashed")
t.test(x, x2, alternative="g")
wilcox.test(x, x2, alternative="g")
ks.test(x, x2, alternative="l")
```

---

ksmooth *Kernel Regression Smoother*

Description

The Nadaraya-Watson kernel regression estimate.

Usage

```
ksmooth(x, y, kernel = c("box", "normal"), bandwidth = 0.5,
        range.x = range(x), n.points = max(100, length(x)), x.points)
```

Arguments

`x`	input x values
`y`	input y values
`kernel`	the kernel to be used.
`bandwidth`	the bandwidth. The kernels are scaled so that their quartiles (viewed as probability densities) are at $\pm$ `0.25*bandwidth`.
`range.x`	the range of points to be covered in the output.
`n.points`	the number of points at which to evaluate the fit.
`x.points`	points at which to evaluate the smoothed fit. If missing, `n.points` are chosen uniformly to cover `range.x`.

Value

A list with components

`x`	values at which the smoothed fit is evaluated. Guaranteed to be in increasing order.
`y`	fitted values corresponding to `x`.

Note

This function is implemented purely for compatibility with S, although it is nowhere near as slow as the S function. Better kernel smoothers are available in other packages.

Examples

```
with(cars, {
    plot(speed, dist)
    lines(ksmooth(speed, dist, "normal", bandwidth=2), col=2)
    lines(ksmooth(speed, dist, "normal", bandwidth=5), col=3)
})
```

`lag` *Lag a Time Series*

Description

Compute a lagged version of a time series, shifting the time base back by a given number of observations.

Usage

```
lag(x, ...)

## Default S3 method:
lag(x, k = 1, ...)
```

Arguments

`x`	A vector or matrix or univariate or multivariate time series
`k`	The number of lags (in units of observations).
`...`	further arguments to be passed to or from methods.

Details

Vector or matrix arguments `x` are coerced to time series.

`lag` is a generic function; this page documents its default method.

Value

A time series object.

Note

Note the sign of `k`: a series lagged by a positive `k` starts *earlier*.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`diff`, `deltat`

Examples

```
lag(ldeaths, 12) # starts one year earlier
```

`lag.plot` *Time Series Lag Plots*

Description

Plot time series against lagged versions of themselves. Helps visualizing "auto-dependence" even when auto-correlations vanish.

Usage

```
lag.plot(x, lags = 1, layout = NULL, set.lags = 1:lags,
         main = NULL, asp = 1,
         diag = TRUE, diag.col = "gray", type = "p", oma = NULL,
         ask = NULL, do.lines = (n <= 150), labels = do.lines, ...)
```

Arguments

`x`	time-series (univariate or multivariate)
`lags`	number of lag plots desired, see arg `set.lags`.
`layout`	the layout of multiple plots, basically the `mfrow` `par()` argument. The default uses about a square layout (see `n2mfrow` such that all plots are on one page.
`set.lags`	vector of positive integers allowing specification of the set of lags used; defaults to `1:lags`.
`main`	character with a main header title to be done on the top of each page.
`asp`	Aspect ratio to be fixed, see `plot.default`.
`diag`	logical indicating if the x=y diagonal should be drawn.
`diag.col`	color to be used for the diagonal `if(diag)`.
`type`	plot type to be used, but see `plot.ts` about its restricted meaning.
`oma`	outer margins, see `par`.
`ask`	logical or `NULL`; if true, the user is asked to confirm before a new page is started.
`do.lines`	logical indicating if lines should be drawn.
`labels`	logical indicating if labels should be used.
`...`	Further arguments to `plot.ts`. Several graphical parameters are set in this function and so cannot be changed: these include `xlab`, `ylab`, `mgp`, `col.lab` and `font.lab`: this also applies to the arguments `xy.labels` and `xy.lines`.

Details

If just one plot is produced, this is a conventional plot. If more than one plot is to be produced, `par(mfrow)` and several other graphics parameters will be set, so it is not (easily) possible to mix such lag plots with other plots on the same page.

If `ask = NULL`, `par(ask = TRUE)` will be called if more than one page of plots is to be produced and the device is interactive.

Note

It is more flexible and has different default behaviour than the S version. We use `main =` instead of `head =` for internal consistency.

Author(s)

Martin Maechler

See Also

`plot.ts` which is the basic work horse.

Examples

```
lag.plot(nhtemp, 8, diag.col = "forest green")
lag.plot(nhtemp, 5, main="Average Temperatures in New Haven")
## ask defaults to TRUE when we have more than one page:
lag.plot(nhtemp, 6, layout = c(2,1), asp = NA,
         main = "New Haven Temperatures", col.main = "blue")

## Multivariate (but non-stationary! ...)
lag.plot(freeny.x, lag = 3)
## Not run:
no lines for long series :
lag.plot(sqrt(sunspots), set = c(1:4, 9:12), pch = ".", col = "gold")
## End(Not run)
```

`line` *Robust Line Fitting*

Description

Fit a line robustly as recommended in *Exploratory Data Analysis*.

Usage

```
line(x, y)
```

Arguments

`x,y`	the arguments can be any way of specifying x-y pairs.

Value

An object of class "tukeyline".

Methods are available for the generic functions coef, residuals, fitted, and print.

References

Tukey, J. W. (1977). *Exploratory Data Analysis*, Reading Massachusetts: Addison-Wesley.

See Also

lm.

Examples

```
plot(cars)
(z <- line(cars))
abline(coef(z))
## Tukey-Anscombe Plot :
plot(residuals(z) ~ fitted(z), main = deparse(z$call))
```

---

lm *Fitting Linear Models*

Description

lm is used to fit linear models. It can be used to carry out regression, single stratum analysis of variance and analysis of covariance (although aov may provide a more convenient interface for these).

Usage

```
lm(formula, data, subset, weights, na.action,
   method = "qr", model = TRUE, x = FALSE, y = FALSE, qr = TRUE,
   singular.ok = TRUE, contrasts = NULL, offset, ...)
```

Arguments

formula	a symbolic description of the model to be fit. The details of model specification are given below.
data	an optional data frame, list or environment (or object coercible by as.data.frame to a data frame) containing the variables in the model. If not found in data, the variables are taken from environment(formula), typically the environment from which lm is called.
subset	an optional vector specifying a subset of observations to be used in the fitting process.
weights	an optional vector of weights to be used in the fitting process. Should be NULL or a numeric vector. If non-NULL, weighted least squares is used with weights weights (that is, minimizing sum(w*e^2)); otherwise ordinary least squares is used.

na.action	a function which indicates what should happen when the data contain NAs. The default is set by the na.action setting of options, and is na.fail if that is unset. The "factory-fresh" default is na.omit. Another possible value is NULL, no action. Value na.exclude can be useful.
method	the method to be used; for fitting, currently only method = "qr" is supported; method = "model.frame" returns the model frame (the same as with model = TRUE, see below).
model, x, y, qr	logicals. If TRUE the corresponding components of the fit (the model frame, the model matrix, the response, the QR decomposition) are returned.
singular.ok	logical. If FALSE (the default in S but not in R) a singular fit is an error.
contrasts	an optional list. See the contrasts.arg of model.matrix.default.
offset	this can be used to specify an *a priori* known component to be included in the linear predictor during fitting. This should be NULL or a numeric vector of length either one or equal to the number of cases. One or more offset terms can be included in the formula instead or as well, and if both are specified their sum is used. See model.offset.
...	additional arguments to be passed to the low level regression fitting functions (see below).

Details

Models for lm are specified symbolically. A typical model has the form response ~ terms where response is the (numeric) response vector and terms is a series of terms which specifies a linear predictor for response. A terms specification of the form first + second indicates all the terms in first together with all the terms in second with duplicates removed. A specification of the form first:second indicates the set of terms obtained by taking the interactions of all terms in first with all terms in second. The specification first*second indicates the *cross* of first and second. This is the same as first + second + first:second.

If the formula includes an offset, this is evaluated and subtracted from the response.

If response is a matrix a linear model is fitted separately by least-squares to each column of the matrix.

See model.matrix for some further details. The terms in the formula will be re-ordered so that main effects come first, followed by the interactions, all second-order, all third-order and so on: to avoid this pass a terms object as the formula (see aov and demo(glm.vr) for an example).

A formula has an implied intercept term. To remove this use either y ~ x - 1 or y ~ 0 + x. See formula for more details of allowed formulae.

lm calls the lower level functions lm.fit, etc, see below, for the actual numerical computations. For programming only, you may consider doing likewise.

All of weights, subset and offset are evaluated in the same way as variables in formula, that is first in data and then in the environment of formula.

Value

lm returns an object of class "lm" or for multiple responses of class c("mlm", "lm").

The functions summary and anova are used to obtain and print a summary and analysis of variance table of the results. The generic accessor functions coefficients, effects, fitted.values and residuals extract various useful features of the value returned by lm.

An object of class "lm" is a list containing at least the following components:

coefficients	a named vector of coefficients
residuals	the residuals, that is response minus fitted values.
fitted.values	the fitted mean values.
rank	the numeric rank of the fitted linear model.
weights	(only for weighted fits) the specified weights.
df.residual	the residual degrees of freedom.
call	the matched call.
terms	the terms object used.
contrasts	(only where relevant) the contrasts used.
xlevels	(only where relevant) a record of the levels of the factors used in fitting.
offset	the offset used (missing if none were used).
y	if requested, the response used.
x	if requested, the model matrix used.
model	if requested (the default), the model frame used.

In addition, non-null fits will have components assign, effects and (unless not requested) qr relating to the linear fit, for use by extractor functions such as summary and effects.

Using time series

Considerable care is needed when using lm with time series.

Unless na.action = NULL, the time series attributes are stripped from the variables before the regression is done. (This is necessary as omitting NAs would invalidate the time series attributes, and if NAs are omitted in the middle of the series the result would no longer be a regular time series.)

Even if the time series attributes are retained, they are not used to line up series, so that the time shift of a lagged or differenced regressor would be ignored. It is good practice to prepare a data argument by ts.intersect(..., dframe = TRUE), then apply a suitable na.action to that data frame and call lm with na.action = NULL so that residuals and fitted values are time series.

Note

Offsets specified by offset will not be included in predictions by predict.lm, whereas those specified by an offset term in the formula will be.

Author(s)

The design was inspired by the S function of the same name described in Chambers (1992). The implementation of model formula by Ross Ihaka was based on Wilkinson & Rogers (1973).

References

Chambers, J. M. (1992) *Linear models.* Chapter 4 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

Wilkinson, G. N. and Rogers, C. E. (1973) Symbolic descriptions of factorial models for analysis of variance. *Applied Statistics*, **22**, 392–9.

See Also

summary.lm for summaries and anova.lm for the ANOVA table; aov for a different interface.

The generic functions coef, effects, residuals, fitted, vcov.

predict.lm (via predict) for prediction, including confidence and prediction intervals; confint for confidence intervals of *parameters*.

lm.influence for regression diagnostics, and glm for **generalized** linear models.

The underlying low level functions, lm.fit for plain, and lm.wfit for weighted regression fitting.

More lm() examples are available e.g., in anscombe, attitude, freeny, LifeCycleSavings, longley, stackloss, swiss.

Examples

```
## Annette Dobson (1990) "An Introduction to Generalized Linear Models".
## Page 9: Plant Weight Data.
ctl <- c(4.17,5.58,5.18,6.11,4.50,4.61,5.17,4.53,5.33,5.14)
trt <- c(4.81,4.17,4.41,3.59,5.87,3.83,6.03,4.89,4.32,4.69)
group <- gl(2,10,20, labels=c("Ctl","Trt"))
weight <- c(ctl, trt)
anova(lm.D9 <- lm(weight ~ group))
summary(lm.D90 <- lm(weight ~ group - 1))# omitting intercept
summary(resid(lm.D9) - resid(lm.D90)) #- residuals almost identical

opar <- par(mfrow = c(2,2), oma = c(0, 0, 1.1, 0))
plot(lm.D9, las = 1)      # Residuals, Fitted, ...
par(opar)

## model frame :
stopifnot(identical(lm(weight ~ group, method = "model.frame"),
                    model.frame(lm.D9)))

### less simple examples in "See Also" above
```

---

lm.fit *Fitter Functions for Linear Models*

Description

These are the basic computing engines called by lm used to fit linear models. These should usually *not* be used directly unless by experienced users.

Usage

```
lm.fit (x, y,     offset = NULL, method = "qr", tol = 1e-7,
        singular.ok = TRUE, ...)

lm.wfit(x, y, w, offset = NULL, method = "qr", tol = 1e-7,
         singular.ok = TRUE, ...)
```

Arguments

`x`	design matrix of dimension `n * p`.
`y`	vector of observations of length `n`, or a matrix with `n` rows.
`w`	vector of weights (length `n`) to be used in the fitting process for the `wfit` functions. Weighted least squares is used with weights `w`, i.e., `sum(w * e^2)` is minimized.
`offset`	numeric of length `n`). This can be used to specify an *a priori* known component to be included in the linear predictor during fitting.
`method`	currently, only `method="qr"` is supported.
`tol`	tolerance for the `qr` decomposition. Default is 1e-7.
`singular.ok`	logical. If `FALSE`, a singular model is an error.
`...`	currently disregarded.

Value

a list with components

`coefficients`	`p` vector
`residuals`	`n` vector or matrix
`fitted.values`	`n` vector or matrix
`effects`	(not null fits)`n` vector of orthogonal single-df effects. The first `rank` of them correspond to non-aliased coeffcients, and are named accordingly.
`weights`	`n` vector — *only* for the `*wfit*` functions.
`rank`	integer, giving the rank
`df.residual`	degrees of freedom of residuals
`qr`	(not null fits) the QR decomposition, see `qr`.

See Also

`lm` which you should use for linear least squares regression, unless you know better.

Examples

```
set.seed(129)
n <- 7 ; p <- 2
X <- matrix(rnorm(n * p), n,p) # no intercept!
y <- rnorm(n)
w <- rnorm(n)^2

str(lmw <- lm.wfit(x=X, y=y, w=w))

str(lm. <- lm.fit (x=X, y=y))
```

`lm.influence` *Regression Diagnostics*

Description

This function provides the basic quantities which are used in forming a wide variety of diagnostics for checking the quality of regression fits.

Usage

```
influence(model, ...)
## S3 method for class 'lm':
influence(model, do.coef = TRUE, ...)
## S3 method for class 'glm':
influence(model, do.coef = TRUE, ...)

lm.influence(model, do.coef = TRUE)
```

Arguments

`model`	an object as returned by `lm` or `glm`.
`do.coef`	logical indicating if the changed `coefficients` (see below) are desired. These need $O(n^2p)$ computing time.
`...`	further arguments passed to or from other methods.

Details

The `influence.measures()` and other functions listed in **See Also** provide a more user oriented way of computing a variety of regression diagnostics. These all build on `lm.influence`. Note that for GLMs (other than the Gaussian family with identity link) these are based on one-step approximations which may be inadequate if a case has high influence.

An attempt is made to ensure that computed hat values that are probably one are treated as one, and the corresponding rows in `sigma` and `coefficients` are `NaN`. (Dropping such a case would normally result in a variable being dropped, so it is not possible to give simple drop-one diagnostics.)

`naresid` is applied to the results and so will fill in with `NA`s it the fit had `na.action = na.exclude`.

Value

A list containing the following components of the same length or number of rows n, which is the number of non-zero weights. Cases omitted in the fit are omitted unless a `na.action` method was used (such as `na.exclude`) which restores them.

`hat`	a vector containing the diagonal of the "hat" matrix.
`coefficients`	(unless `do.coef` is false) a matrix whose i-th row contains the change in the estimated coefficients which results when the i-th case is dropped from the regression. Note that aliased coefficients are not included in the matrix.

`sigma`	a vector whose i-th element contains the estimate of the residual standard deviation obtained when the i-th case is dropped from the regression. (The approximations needed for GLMs can result in this being `NaN`.)
`wt.res`	a vector of *weighted* (or for class `glm` rather *deviance*) residuals.

Note

The `coefficients` returned by the R version of `lm.influence` differ from those computed by S. Rather than returning the coefficients which result from dropping each case, we return the changes in the coefficients. This is more directly useful in many diagnostic measures.
Since these need $O(n^2p)$ computing time, they can be omitted by `do.coef = FALSE`.

Note that cases with `weights == 0` are *dropped* (contrary to the situation in S).

If a model has been fitted with `na.action=na.exclude` (see `na.exclude`), cases excluded in the fit *are* considered here.

References

See the list in the documentation for `influence.measures`.

Chambers, J. M. (1992) *Linear models.* Chapter 4 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`summary.lm` for `summary` and related methods;
`influence.measures`,
`hat` for the hat matrix diagonals,
`dfbetas`, `dffits`, `covratio`, `cooks.distance`, `lm`.

Examples

```
## Analysis of the life-cycle savings data
## given in Belsley, Kuh and Welsch.
summary(lm.SR <- lm(sr ~ pop15 + pop75 + dpi + ddpi,
                    data = LifeCycleSavings),
        corr = TRUE)
str(lmI <- lm.influence(lm.SR))

## For more "user level" examples, use example(influence.measures)
```

`lm.summaries` *Accessing Linear Model Fits*

Description

All these functions are `methods` for class `"lm"` objects.

Usage

```
## S3 method for class 'lm':
family(object, ...)

## S3 method for class 'lm':
formula(x, ...)

## S3 method for class 'lm':
residuals(object,
          type = c("working", "response", "deviance", "pearson",
                   "partial"),
          ...)

## S3 method for class 'lm':
labels(object, ...)

weights(object, ...)
```

Arguments

`object, x`	an object inheriting from class `lm`, usually the result of a call to `lm` or `aov`.
`...`	further arguments passed to or from other methods.
`type`	the type of residuals which should be returned.

Details

The generic accessor functions `coef`, `effects`, `fitted` and `residuals` can be used to extract various useful features of the value returned by `lm`.

The working and response residuals are "observed - fitted". The deviance and pearson residuals are weighted residuals, scaled by the square root of the weights used in fitting. The partial residuals are a matrix with each column formed by omitting a term from the model. In all these, zero weight cases are never omitted (as opposed to the standardized `rstudent` residuals, and the `weighted.residuals`).

How `residuals` treats cases with missing values in the original fit is determined by the `na.action` argument of that fit. If `na.action = na.omit` omitted cases will not appear in the residuals, whereas if `na.action = na.exclude` they will appear, with residual value `NA`. See also `naresid`.

The `"lm"` method for generic `labels` returns the term labels for estimable terms, that is the names of the terms with an least one estimable coefficient.

References

Chambers, J. M. (1992) *Linear models.* Chapter 4 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

The model fitting function `lm`, `anova.lm`.

`coef`, `deviance`, `df.residual`, `effects`, `fitted`, `glm` for **generalized** linear models, `influence` (etc on that page) for regression diagnostics, `weighted.residuals`, `residuals`, `residuals.glm`, `summary.lm`.

Examples

```
##-- Continuing the  lm(.) example:
coef(lm.D90)# the bare coefficients

## The 2 basic regression diagnostic plots [plot.lm(.) is preferred]
plot(resid(lm.D90), fitted(lm.D90))# Tukey-Anscombe's
abline(h=0, lty=2, col = 'gray')

qqnorm(residuals(lm.D90))
```

---

loadings *Print Loadings in Factor Analysis*

Description

Extract or print loadings in factor analysis (or principal components analysis).

Usage

```
loadings(x)

## S3 method for class 'loadings':
print(x, digits = 3, cutoff = 0.1, sort = FALSE, ...)

## S3 method for class 'factanal':
print(x, digits = 3, ...)
```

Arguments

`x`	an object of class `"factanal"` or `"princomp"` or the `loadings` component of such an object.
`digits`	number of decimal places to use in printing uniquenesses and loadings.
`cutoff`	loadings smaller than this (in absolute value) are suppressed.
`sort`	logical. If true, the variables are sorted by their importance on each factor. Each variable with any loading larger than 0.5 (in modulus) is assigned to the factor with the largest loading, and the variables are printed in the order of the factor they are assigned to, then those unassigned.
`...`	further arguments for other methods, such as `cutoff` and `sort` for `print.factanal`.

See Also

`factanal`, `princomp`

`loess` *Local Polynomial Regression Fitting*

Description

Fit a polynomial surface determined by one or more numerical predictors, using local fitting.

Usage

```
loess(formula, data, weights, subset, na.action, model = FALSE,
      span = 0.75, enp.target, degree = 2,
      parametric = FALSE, drop.square = FALSE, normalize = TRUE,
      family = c("gaussian", "symmetric"),
      method = c("loess", "model.frame"),
      control = loess.control(...), ...)
```

Arguments

`formula`	a formula specifying the numeric response and one to four numeric predictors (best specified via an interaction, but can also be specified additively).
`data`	an optional data frame, list or environment (or object coercible by `as.data.frame` to a data frame) containing the variables in the model. If not found in `data`, the variables are taken from `environment(formula)`, typically the environment from which `loess` is called.
`weights`	optional weights for each case.
`subset`	an optional specification of a subset of the data to be used.
`na.action`	the action to be taken with missing values in the response or predictors. The default is given by `getOption("na.action")`.
`model`	should the model frame be returned?
`span`	the parameter α which controls the degree of smoothing.
`enp.target`	an alternative way to specify `span`, as the approximate equivalent number of parameters to be used.
`degree`	the degree of the polynomials to be used, up to 2.
`parametric`	should any terms be fitted globally rather than locally? Terms can be specified by name, number or as a logical vector of the same length as the number of predictors.
`drop.square`	for fits with more than one predictor and `degree=2`, should the quadratic term (and cross-terms) be dropped for particular predictors? Terms are specified in the same way as for `parametric`.
`normalize`	should the predictors be normalized to a common scale if there is more than one? The normalization used is to set the 10% trimmed standard deviation to one. Set to false for spatial coordinate predictors and others know to be a common scale.
`family`	if `"gaussian"` fitting is by least-squares, and if `"symmetric"` a re-descending M estimator is used with Tukey's biweight function.
`method`	fit the model or just extract the model frame.
`control`	control parameters: see `loess.control`.
`...`	control parameters can also be supplied directly.

Details

Fitting is done locally. That is, for the fit at point x, the fit is made using points in a neighbourhood of x, weighted by their distance from x (with differences in 'parametric' variables being ignored when computing the distance). The size of the neighbourhood is controlled by α (set by `span` or `enp.target`). For $\alpha < 1$, the neighbourhood includes proportion α of the points, and these have tricubic weighting (proportional to $(1 - (\text{dist}/\text{maxdist})^3)^3$. For $\alpha > 1$, all points are used, with the 'maximum distance' assumed to be $\alpha^{1/p}$ times the actual maximum distance for p explanatory variables.

For the default family, fitting is by (weighted) least squares. For `family="symmetric"` a few iterations of an M-estimation procedure with Tukey's biweight are used. Be aware that as the initial value is the least-squares fit, this need not be a very resistant fit.

It can be important to tune the control list to achieve acceptable speed. See `loess.control` for details.

Value

An object of class `"loess"`.

Note

As this is based on the `cloess` package available at `netlib`, it is similar to but not identical to the `loess` function of S. In particular, conditioning is not implemented.

The memory usage of this implementation of `loess` is roughly quadratic in the number of points, with 1000 points taking about 10Mb.

Author(s)

B.D. Ripley, based on the `cloess` package of Cleveland, Grosse and Shyu avaliable at `http://www.netlib.org/a/`.

References

W.S. Cleveland, E. Grosse and W.M. Shyu (1992) Local regression models. Chapter 8 of *Statistical Models in S* eds J.M. Chambers and T.J. Hastie, Wadsworth & Brooks/Cole.

See Also

`loess.control`, `predict.loess`.

`lowess`, the ancestor of `loess` (with different defaults!).

Examples

```
cars.lo <- loess(dist ~ speed, cars)
predict(cars.lo, data.frame(speed = seq(5, 30, 1)), se = TRUE)
# to allow extrapolation
cars.lo2 <- loess(dist ~ speed, cars,
  control = loess.control(surface = "direct"))
predict(cars.lo2, data.frame(speed = seq(5, 30, 1)), se = TRUE)
```

`loess.control` *Set Parameters for Loess*

Description

Set control parameters for `loess` fits.

Usage

```
loess.control(surface = c("interpolate", "direct"),
              statistics = c("approximate", "exact"),
              trace.hat = c("exact", "approximate"),
              cell = 0.2, iterations = 4, ...)
```

Arguments

`surface`	should be fitted surface be computed exactly or via interpolation from a kd tree?
`statistics`	should the statistics be computed exactly or approximately? Exact computation can be very slow.
`trace.hat`	should the trace of the smoother matrix be computed exactly or approximately? It is recommended to use the approximation for more than about 1000 data points.
`cell`	if interpolation is used this controls the accuracy of the approximation via the maximum number of points in a cell in the kd tree. Cells with more than `floor(n*span*cell)` points are subdivided.
`iterations`	the number of iterations used in robust fitting.
`...`	further arguments which are ignored.

Value

A list with components

`surface`

`statistics`

`trace.hat`

`cell`

`iterations`

with meanings as explained under 'Arguments'.

See Also

`loess`

`Logistic` *The Logistic Distribution*

Description

Density, distribution function, quantile function and random generation for the logistic distribution with parameters `location` and `scale`.

Usage

```
dlogis(x, location = 0, scale = 1, log = FALSE)
plogis(q, location = 0, scale = 1, lower.tail = TRUE, log.p = FALSE)
qlogis(p, location = 0, scale = 1, lower.tail = TRUE, log.p = FALSE)
rlogis(n, location = 0, scale = 1)
```

Arguments

`x, q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`location, scale`	location and scale parameters.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

If `location` or `scale` are omitted, they assume the default values of `0` and `1` respectively.

The Logistic distribution with `location` $= \mu$ and `scale` $= \sigma$ has distribution function

$$F(x) = \frac{1}{1 + e^{-(x-\mu)/\sigma}}$$

and density

$$f(x) = \frac{1}{\sigma} \frac{e^{(x-\mu)/\sigma}}{(1 + e^{(x-\mu)/\sigma})^2}$$

It is a long-tailed distribution with mean μ and variance $\pi^2/3\sigma^2$.

Value

`dlogis` gives the density, `plogis` gives the distribution function, `qlogis` gives the quantile function, and `rlogis` generates random deviates.

Note

`qlogis(p)` is the same as the well known '*logit*' function, $logit(p) = \log(p/(1-p))$, and `plogis(x)` has consequently been called the "inverse logit".

The distribution function is a rescaled hyperbolic tangent, `plogis(x) == (1+tanh(x/2))/2`, and it is called *sigmoid function* in contexts such as neural networks.

Source

`[dpr]logis` are calculated directly from the definitions.

`rlogis` uses inversion.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, volume 2, chapter 23. Wiley, New York.

Examples

```
var(rlogis(4000, 0, s = 5))# approximately (+/- 3)
pi^2/3 * 5^2
```

`logLik` *Extract Log-Likelihood*

Description

This function is generic; method functions can be written to handle specific classes of objects. Classes which already have methods for this function include: `glm`, `lm`, `nls`, `Arima` and `gls`, `lme` and others in package **nlme**.

Usage

```
logLik(object, ...)

## S3 method for class 'lm':
logLik(object, REML = FALSE, ...)
```

Arguments

`object`	any object from which a log-likelihood value, or a contribution to a log-likelihood value, can be extracted.
`...`	some methods for this generic function require additional arguments.
`REML`	an optional logical value. If `TRUE` the restricted log-likelihood is returned, else, if `FALSE`, the log-likelihood is returned. Defaults to `FALSE`.

Details

For a `"glm"` fit the `family` does not have to specify how to calculate the log-likelihood, so this is based on the family's `aic()` function to compute the AIC. For the `gaussian`, `Gamma` and `inverse.gaussian` families it assumed that the dispersion of the GLM is estimated has been counted as a parameter in the AIC value, and for all other families it is assumed that the dispersion is known.

Note that this procedure is not completely accurate for the gamma and inverse gaussian families, as the estimate of dispersion used is not the MLE.

For `"lm"` fits it is assumed that the scale has been estimated (by maximum likelihood or REML), and all the constants in the log-likelihood are included.

Value

Returns an object, say `r`, of class `logLik` which is a number with attributes, `attr(r, "df")` (**d**egrees of **f**reedom) giving the number of (esimated) parameters in the model. There is a simple `print` method for `logLik` objects.

The details depend on the method function used; see the appropriate documentation.

Author(s)

Jose Pinheiro and Douglas Bates

References

For `logLik.lm`:

Harville, D.A. (1974). Bayesian inference for variance components using only error contrasts. *Biometrika*, **61**, 383–385.

See Also

`logLik.gls`, `logLik.lme`, in package **nlme**, etc.

Examples

```
x <- 1:5
lmx <- lm(x ~ 1)
logLik(lmx) # using print.logLik() method
str(logLik(lmx))

## lm method
(fm1 <- lm(rating ~ ., data = attitude))
logLik(fm1)
logLik(fm1, REML = TRUE)

res <- try(data(Orthodont, package="nlme"))
if(!inherits(res, "try-error")) {
  fm1 <- lm(distance ~ Sex * age, Orthodont)
  print(logLik(fm1))
  print(logLik(fm1, REML = TRUE))
}
```

`loglin` *Fitting Log-Linear Models*

Description

`loglin` is used to fit log-linear models to multidimensional contingency tables by Iterative Proportional Fitting.

Usage

```
loglin(table, margin, start = rep(1, length(table)), fit = FALSE,
       eps = 0.1, iter = 20, param = FALSE, print = TRUE)
```

Arguments

table	a contingency table to be fit, typically the output from table.
margin	a list of vectors with the marginal totals to be fit. (Hierarchical) log-linear models can be specified in terms of these marginal totals which give the "maximal" factor subsets contained in the model. For example, in a three-factor model, list(c(1, 2), c(1, 3)) specifies a model which contains parameters for the grand mean, each factor, and the 1-2 and 1-3 interactions, respectively (but no 2-3 or 1-2-3 interaction), i.e., a model where factors 2 and 3 are independent conditional on factor 1 (sometimes represented as '[12][13]'). The names of factors (i.e., names(dimnames(table))) may be used rather than numeric indices.
start	a starting estimate for the fitted table. This optional argument is important for incomplete tables with structural zeros in table which should be preserved in the fit. In this case, the corresponding entries in start should be zero and the others can be taken as one.
fit	a logical indicating whether the fitted values should be returned.
eps	maximum deviation allowed between observed and fitted margins.
iter	maximum number of iterations.
param	a logical indicating whether the parameter values should be returned.
print	a logical. If TRUE, the number of iterations and the final deviation are printed.

Details

The Iterative Proportional Fitting algorithm as presented in Haberman (1972) is used for fitting the model. At most iter iterations are performed, convergence is taken to occur when the maximum deviation between observed and fitted margins is less than eps. All internal computations are done in double precision; there is no limit on the number of factors (the dimension of the table) in the model.

Assuming that there are no structural zeros, both the Likelihood Ratio Test and Pearson test statistics have an asymptotic chi-squared distribution with df degrees of freedom.

Package **MASS** contains loglm, a front-end to loglin which allows the log-linear model to be specified and fitted in a formula-based manner similar to that of other fitting functions such as lm or glm.

Value

A list with the following components.

lrt	the Likelihood Ratio Test statistic.
pearson	the Pearson test statistic (X-squared).
df	the degrees of freedom for the fitted model. There is no adjustment for structural zeros.
margin	list of the margins that were fit. Basically the same as the input margin, but with numbers replaced by names where possible.
fit	An array like table containing the fitted values. Only returned if fit is TRUE.
param	A list containing the estimated parameters of the model. The "standard" constraints of zero marginal sums (e.g., zero row and column sums for a two factor parameter) are employed. Only returned if param is TRUE.

Author(s)

Kurt Hornik

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Haberman, S. J. (1972) Log-linear fit for contingency tables—Algorithm AS51. *Applied Statistics*, **21**, 218–225.

Agresti, A. (1990) *Categorical data analysis*. New York: Wiley.

See Also

`table`

Examples

```
## Model of joint independence of sex from hair and eye color.
fm <- loglin(HairEyeColor, list(c(1, 2), c(1, 3), c(2, 3)))
fm
1 - pchisq(fm$lrt, fm$df)
## Model with no three-factor interactions fits well.
```

`Lognormal` *The Log Normal Distribution*

Description

Density, distribution function, quantile function and random generation for the log normal distribution whose logarithm has mean equal to `meanlog` and standard deviation equal to `sdlog`.

Usage

```
dlnorm(x, meanlog = 0, sdlog = 1, log = FALSE)
plnorm(q, meanlog = 0, sdlog = 1, lower.tail = TRUE, log.p = FALSE)
qlnorm(p, meanlog = 0, sdlog = 1, lower.tail = TRUE, log.p = FALSE)
rlnorm(n, meanlog = 0, sdlog = 1)
```

Arguments

`x, q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`meanlog, sdlog`	mean and standard deviation of the distribution on the log scale with default values of `0` and `1` respectively.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

The log normal distribution has density

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma x} e^{-(\log(x)-\mu)^2/2\sigma^2}$$

where μ and σ are the mean and standard deviation of the logarithm. The mean is $E(X) = exp(\mu + 1/2\sigma^2)$, and the variance $Var(X) = exp(2\mu + \sigma^2)(exp(\sigma^2) - 1)$ and hence the coefficient of variation is $\sqrt{exp(\sigma^2) - 1}$ which is approximately σ when that is small (e.g., $\sigma < 1/2$).

Value

`dlnorm` gives the density, `plnorm` gives the distribution function, `qlnorm` gives the quantile function, and `rlnorm` generates random deviates.

Note

The cumulative hazard $H(t) = -\log(1 - F(t))$ is `-plnorm(t, r, lower = FALSE, log = TRUE)`.

Source

`dlnorm` is calculated from the definition (in Details). `[pqr]lnorm` are based on the relationship to the normal.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, volume 1, chapter 14. Wiley, New York.

See Also

`dnorm` for the normal distribution.

Examples

```
dlnorm(1) == dnorm(0)
```

`lowess` *Scatter Plot Smoothing*

Description

This function performs the computations for the *LOWESS* smoother which uses locally-weighted polynomial regression (see the references).

Usage

```
lowess(x, y = NULL, f = 2/3, iter = 3,
       delta = 0.01 * diff(range(xy$x[o])))
```

Arguments

`x, y`	vectors giving the coordinates of the points in the scatter plot. Alternatively a single plotting structure can be specified.
`f`	the smoother span. This gives the proportion of points in the plot which influence the smooth at each value. Larger values give more smoothness.
`iter`	the number of 'robustifying' iterations which should be performed. Using smaller values of `iter` will make `lowess` run faster.
`delta`	See Details. Defaults to 1/100th of the range of `x`.

Details

`lowess` is defined by a complex algorithm, the Ratfor original of which (by W. S. Cleveland) can be found in the R sources as file 'src/appl/lowess.doc'. Normally a local linear polynomial fit is used, but under some circumstances (see the file) a local constant fit can be used. 'Local' is defined by the distance to the `floor(f*n)` th nearest neighbour, and tricubic weighting is used for `x` which fall within the neighbourhood.

The initial fit is done using weighted least squares. If `iter > 0`, further weighted fits are done using the product of the weights from the proximity of the `x` values and case weights derived from the residuals at the previous iteration. Specifically, the case weight is Tukey's biweight, with cutoff 6 times the MAD of the residuals. (The current R implementation differs from the original in stopping iteration if the MAD is effectively zero since the algorithm is highly unstable in that case.)

`delta` is used to speed up computation: instead of computing the local polynomial fit at each data point it is not computed for points within `delta` of the last computed point, and linear interpolation is used to fill in the fitted values for the skipped points.

Value

`lowess` returns a list containing components `x` and `y` which give the coordinates of the smooth. The smooth can be added to a plot of the original points with the function `lines`: see the examples.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Cleveland, W. S. (1979) Robust locally weighted regression and smoothing scatterplots. *J. Amer. Statist. Assoc.* **74**, 829–836.

Cleveland, W. S. (1981) LOWESS: A program for smoothing scatterplots by robust locally weighted regression. *The American Statistician*, **35**, 54.

See Also

`loess`, a newer formula based version of `lowess` (with different defaults!).

Examples

```
plot(cars, main = "lowess(cars)")
lines(lowess(cars), col = 2)
lines(lowess(cars, f=.2), col = 3)
legend(5, 120, c(paste("f = ", c("2/3", ".2"))), lty = 1, col = 2:3)
```

`ls.diag` *Compute Diagnostics for 'lsfit' Regression Results*

Description

Computes basic statistics, including standard errors, t- and p-values for the regression coefficients.

Usage

```
ls.diag(ls.out)
```

Arguments

`ls.out`	Typically the result of `lsfit()`

Value

A `list` with the following numeric components.

`std.dev`	The standard deviation of the errors, an estimate of σ.
`hat`	diagonal entries h_{ii} of the hat matrix H
`std.res`	standardized residuals
`stud.res`	studentized residuals
`cooks`	Cook's distances
`dfits`	DFITS statistics
`correlation`	correlation matrix
`std.err`	standard errors of the regression coefficients
`cov.scaled`	Scaled covariance matrix of the coefficients
`cov.unscaled`	Unscaled covariance matrix of the coefficients

References

Belsley, D. A., Kuh, E. and Welsch, R. E. (1980) *Regression Diagnostics.* New York: Wiley.

See Also

`hat` for the hat matrix diagonals, `ls.print`, `lm.influence`, `summary.lm`, `anova`.

Examples

```
##-- Using the same data as the lm(.) example:
lsD9 <- lsfit(x = as.numeric(gl(2, 10, 20)), y = weight)
dlsD9 <- ls.diag(lsD9)
str(dlsD9, give.attr=FALSE)
abs(1 - sum(dlsD9$hat) / 2) < 10*.Machine$double.eps # sum(h.ii) = p
plot(dlsD9$hat, dlsD9$stud.res, xlim=c(0,0.11))
abline(h = 0, lty = 2, col = "lightgray")
```

ls.print *Print 'lsfit' Regression Results*

Description

Computes basic statistics, including standard errors, t- and p-values for the regression coefficients and prints them if `print.it` is `TRUE`.

Usage

```
ls.print(ls.out, digits = 4, print.it = TRUE)
```

Arguments

`ls.out`	Typically the result of `lsfit()`
`digits`	The number of significant digits used for printing
`print.it`	a logical indicating whether the result should also be printed

Value

A list with the components

`summary`	The ANOVA table of the regression
`coef.table`	matrix with regression coefficients, standard errors, t- and p-values

Note

Usually, you'd rather use `summary(lm(...))` and `anova(lm(...))` for obtaining similar output.

See Also

`ls.diag`, `lsfit`, also for examples; `lm`, `lm.influence` which usually are preferable.

lsfit *Find the Least Squares Fit*

Description

The least squares estimate of β in the model

$$\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$$

is found.

Usage

```
lsfit(x, y, wt = NULL, intercept = TRUE, tolerance = 1e-07,
      yname = NULL)
```

Arguments

`x`	a matrix whose rows correspond to cases and whose columns correspond to variables.
`y`	the responses, possibly a matrix if you want to fit multiple left hand sides.
`wt`	an optional vector of weights for performing weighted least squares.
`intercept`	whether or not an intercept term should be used.
`tolerance`	the tolerance to be used in the matrix decomposition.
`yname`	names to be used for the response variables.

Details

If weights are specified then a weighted least squares is performed with the weight given to the *j*th case specified by the *j*th entry in `wt`.

If any observation has a missing value in any field, that observation is removed before the analysis is carried out. This can be quite inefficient if there is a lot of missing data.

The implementation is via a modification of the LINPACK subroutines which allow for multiple left-hand sides.

Value

A list with the following named components:

`coef`	the least squares estimates of the coefficients in the model (β as stated above).
`residuals`	residuals from the fit.
`intercept`	indicates whether an intercept was fitted.
`qr`	the QR decomposition of the design matrix.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`lm` which usually is preferable; `ls.print`, `ls.diag`.

Examples

```
##-- Using the same data as the lm(.) example:
lsD9 <- lsfit(x = unclass(gl(2,10)), y = weight)
ls.print(lsD9)
```

`mad` *Median Absolute Deviation*

Description

Compute the median absolute deviation, i.e., the (lo-/hi-) median of the absolute deviations from the median, and (by default) adjust by a factor for asymptotically normal consistency.

Usage

```
mad(x, center = median(x), constant = 1.4826, na.rm = FALSE,
    low = FALSE, high = FALSE)
```

Arguments

`x`	a numeric vector.
`center`	Optionally, the centre: defaults to the median.
`constant`	scale factor.
`na.rm`	if `TRUE` then `NA` values are stripped from `x` before computation takes place.
`low`	if `TRUE`, compute the "lo-median", i.e., for even sample size, do not average the two middle values, but take the smaller one.
`high`	if `TRUE`, compute the "hi-median", i.e., take the larger of the two middle values for even sample size.

Details

The actual value calculated is `constant * cMedian(abs(x - center))` with the default value of `center` being `median(x)`, and `cMedian` being the usual, the "low" or "high" median, see the arguments description for `low` and `high` above.

The default `constant = 1.4826` (approximately $1/\Phi^{-1}(\frac{3}{4})$ = `1/qnorm(3/4)`) ensures consistency, i.e.,

$$E[mad(X_1, \ldots, X_n)] = \sigma$$

for X_i distributed as $N(\mu, \sigma^2)$ and large n.

If `na.rm` is `TRUE` then `NA` values are stripped from `x` before computation takes place. If this is not done then an `NA` value in `x` will cause `mad` to return `NA`.

See Also

`IQR` which is simpler but less robust, `median`, `var`.

Examples

```
mad(c(1:9))
print(mad(c(1:9),     constant=1)) ==
      mad(c(1:8,100), constant=1)       # = 2 ; TRUE
x <- c(1,2,3, 5,7,8)
sort(abs(x - median(x)))
c(mad(x, co=1), mad(x, co=1, lo = TRUE), mad(x, co=1, hi = TRUE))
```

`mahalanobis` *Mahalanobis Distance*

Description

Returns the squared Mahalanobis distance of all rows in `x` and the vector $\mu =$`center` with respect to $\Sigma =$`cov`. This is (for vector `x`) defined as

$$D^2 = (x - \mu)' \Sigma^{-1} (x - \mu)$$

Usage

```
mahalanobis(x, center, cov, inverted=FALSE, ...)
```

Arguments

`x`	vector or matrix of data with, say, p columns.
`center`	mean vector of the distribution or second data vector of length p.
`cov`	covariance matrix ($p \times p$) of the distribution.
`inverted`	logical. If `TRUE`, `cov` is supposed to contain the *inverse* of the covariance matrix.
`...`	passed to `solve` for computing the inverse of the covariance matrix (if `inverted` is false).

See Also

`cov`, `var`

Examples

```
ma <- cbind(1:6, 1:3)
(S <-  var(ma))
mahalanobis(c(0,0), 1:2, S)

x <- matrix(rnorm(100*3), ncol = 3)
stopifnot(mahalanobis(x, 0, diag(ncol(x))) == rowSums(x*x))
        ##- Here, D^2 = usual squared Euclidean distances

Sx <- cov(x)
D2 <- mahalanobis(x, colMeans(x), Sx)
plot(density(D2, bw=.5),
     main="Squared Mahalanobis distances, n=100, p=3") ; rug(D2)
qqplot(qchisq(ppoints(100), df=3), D2,
       main = expression("Q-Q plot of Mahalanobis" * ~D^2 *
                         " vs. quantiles of" * ~ chi[3]^2))
abline(0, 1, col = 'gray')
```

`make.link` *Create a Link for GLM families*

Description

This function is used with the `family` functions in `glm()`. Given a link, it returns a link function, an inverse link function, the derivative $d\mu/d\eta$ and a function for domain checking.

Usage

```
make.link(link)
```

Arguments

`link`	character or numeric; one of `"logit"`, `"probit"`, `"cloglog"`, `"identity"`, `"log"`, `"sqrt"`, `"1/mu^2"`, `"inverse"`, or (deprecated) a non-negative number, say λ resulting in `power` link $= \mu^\lambda$. Also (deprecated) a string like `"power(0.5)"` to indicate a call to `power`.

Value

A object of class `"link-glm"`, a list with components

`linkfun`	Link function `function(mu)`
`linkinv`	Inverse link function `function(eta)`
`mu.eta`	Derivative `function(eta)` $d\mu/d\eta$
`valideta`	`function(eta)` { `TRUE` if `eta` is in the domain of `linkinv` }.
`name`	a name to be used for the link

.

See Also

`power`, `glm`, `family`.

Examples

```
str(make.link("logit"))
```

`makepredictcall` *Utility Function for Safe Prediction*

Description

A utility to help `model.frame.default` create the right matrices when predicting from models with terms like `poly` or `ns`.

Usage

```
makepredictcall(var, call)
```

Arguments

`var`	A variable.
`call`	The term in the formula, as a call.

Details

This is a generic function with methods for `poly`, `bs` and `ns`: the default method handles `scale`. If `model.frame.default` encounters such a term when creating a model frame, it modifies the `predvars` attribute of the terms supplied to replace the term with one that will work for predicting new data. For example `makepredictcall.ns` adds arguments for the knots and intercept.

To make use of this, have your model-fitting function return the `terms` attribute of the model frame, or copy the `predvars` attribute of the `terms` attribute of the model frame to your `terms` object.

To extend this, make sure the term creates variables with a class, and write a suitable method for that class.

Value

A replacement for `call` for the `predvars` attribute of the terms.

See Also

`model.frame`, `poly`, `scale`; `bs` and `ns` in package **splines**, `cars`

Examples

```
## using poly: this did not work in R < 1.5.0
fm <- lm(weight ~ poly(height, 2), data = women)
plot(women, xlab = "Height (in)", ylab = "Weight (lb)")
ht <- seq(57, 73, len = 200)
lines(ht, predict(fm, data.frame(height=ht)))

## see also example(cars)

## see bs and ns for spline examples.
```

`manova` *Multivariate Analysis of Variance*

Description

A class for the multivariate analysis of variance.

Usage

```
manova(...)
```

Arguments

`...`	Arguments to be passed to `aov`.

Details

Class `"manova"` differs from class `"aov"` in selecting a different `summary` method. Function `manova` calls `aov` and then add class `"manova"` to the result object for each stratum.

Value

See `aov` and the comments in Details here.

Note

`manova` does not support multistratum analysis of variance, so the formula should not include an `Error` term.

References

Krzanowski, W. J. (1988) *Principles of Multivariate Analysis. A User's Perspective.* Oxford.

Hand, D. J. and Taylor, C. C. (1987) *Multivariate Analysis of Variance and Repeated Measures.* Chapman and Hall.

See Also

`aov`, `summary.manova`, the latter containing examples.

`mantelhaen.test` *Cochran-Mantel-Haenszel Chi-Squared Test for Count Data*

Description

Performs a Cochran-Mantel-Haenszel chi-squared test of the null that two nominal variables are conditionally independent in each stratum, assuming that there is no three-way interaction.

Usage

```
mantelhaen.test(x, y = NULL, z = NULL,
                alternative = c("two.sided", "less", "greater"),
                correct = TRUE, exact = FALSE, conf.level = 0.95)
```

Arguments

`x`	either a 3-dimensional contingency table in array form where each dimension is at least 2 and the last dimension corresponds to the strata, or a factor object with at least 2 levels.
`y`	a factor object with at least 2 levels; ignored if `x` is an array.
`z`	a factor object with at least 2 levels identifying to which stratum the corresponding elements in `x` and `y` belong; ignored if `x` is an array.
`alternative`	indicates the alternative hypothesis and must be one of `"two.sided"`, `"greater"` or `"less"`. You can specify just the initial letter. Only used in the 2 by 2 by K case.
`correct`	a logical indicating whether to apply continuity correction when computing the test statistic. Only used in the 2 by 2 by K case.
`exact`	a logical indicating whether the Mantel-Haenszel test or the exact conditional test (given the strata margins) should be computed. Only used in the 2 by 2 by K case.
`conf.level`	confidence level for the returned confidence interval. Only used in the 2 by 2 by K case.

Details

If `x` is an array, each dimension must be at least 2, and the entries should be nonnegative integers. `NA`'s are not allowed. Otherwise, `x`, `y` and `z` must have the same length. Triples containing `NA`'s are removed. All variables must take at least two different values.

Value

A list with class `"htest"` containing the following components:

`statistic`	Only present if no exact test is performed. In the classical case of a 2 by 2 by K table (i.e., of dichotomous underlying variables), the Mantel-Haenszel chi-squared statistic; otherwise, the generalized Cochran-Mantel-Haenszel statistic.
`parameter`	the degrees of freedom of the approximate chi-squared distribution of the test statistic (1 in the classical case). Only present if no exact test is performed.
`p.value`	the p-value of the test.

`conf.int`	a confidence interval for the common odds ratio. Only present in the 2 by 2 by K case.
`estimate`	an estimate of the common odds ratio. If an exact test is performed, the conditional Maximum Likelihood Estimate is given; otherwise, the Mantel-Haenszel estimate. Only present in the 2 by 2 by K case.
`null.value`	the common odds ratio under the null of independence, `1`. Only present in the 2 by 2 by K case.
`alternative`	a character string describing the alternative hypothesis. Only present in the 2 by 2 by K case.
`method`	a character string indicating the method employed, and whether or not continuity correction was used.
`data.name`	a character string giving the names of the data.

Note

The asymptotic distribution is only valid if there is no three-way interaction. In the classical 2 by 2 by K case, this is equivalent to the conditional odds ratios in each stratum being identical. Currently, no inference on homogeneity of the odds ratios is performed.

See also the example below.

References

Alan Agresti (1990). *Categorical data analysis*. New York: Wiley. Pages 230–235.

Alan Agresti (2002). *Categorical data analysis* (second edition). New York: Wiley.

Examples

```
## Agresti (1990), pages 231--237, Penicillin and Rabbits
## Investigation of the effectiveness of immediately injected or 1.5
##  hours delayed penicillin in protecting rabbits against a lethal
##  injection with beta-hemolytic streptococci.
Rabbits <-
array(c(0, 0, 6, 5,
        3, 0, 3, 6,
        6, 2, 0, 4,
        5, 1, 6, 0,
        2, 5, 0, 0),
      dim = c(2, 2, 5),
      dimnames = list(
          Delay = c("None", "1.5h"),
          Response = c("Cured", "Died"),
          Penicillin.Level = c("1/8", "1/4", "1/2", "1", "4")))
Rabbits
## Classical Mantel-Haenszel test
mantelhaen.test(Rabbits)
## => p = 0.047, some evidence for higher cure rate of immediate
##               injection
## Exact conditional test
mantelhaen.test(Rabbits, exact = TRUE)
## => p - 0.040
## Exact conditional test for one-sided alternative of a higher
## cure rate for immediate injection
mantelhaen.test(Rabbits, exact = TRUE, alternative = "greater")
```

```
## => p = 0.020

## UC Berkeley Student Admissions
mantelhaen.test(UCBAdmissions)
## No evidence for association between admission and gender
## when adjusted for department.  However,
apply(UCBAdmissions, 3, function(x) (x[1,1]*x[2,2])/(x[1,2]*x[2,1]))
## This suggests that the assumption of homogeneous (conditional)
## odds ratios may be violated.  The traditional approach would be
## using the Woolf test for interaction:
woolf <- function(x) {
  x <- x + 1 / 2
  k <- dim(x)[3]
  or <- apply(x, 3, function(x) (x[1,1]*x[2,2])/(x[1,2]*x[2,1]))
  w <-  apply(x, 3, function(x) 1 / sum(1 / x))
  1 - pchisq(sum(w * (log(or) - weighted.mean(log(or), w)) ^ 2), k - 1)
}
woolf(UCBAdmissions)
## => p = 0.003, indicating that there is significant heterogeneity.
## (And hence the Mantel-Haenszel test cannot be used.)

## Agresti (2002), p. 287f and p. 297.
## Job Satisfaction example.
Satisfaction <-
    as.table(array(c(1, 2, 0, 0, 3, 3, 1, 2,
                     11, 17, 8, 4, 2, 3, 5, 2,
                     1, 0, 0, 0, 1, 3, 0, 1,
                     2, 5, 7, 9, 1, 1, 3, 6),
                   dim = c(4, 4, 2),
                   dimnames =
                   list(Income =
                        c("<5000", "5000-15000",
                          "15000-25000", ">25000"),
                        "Job Satisfaction" =
                        c("V_D", "L_S", "M_S", "V_S"),
                        Gender = c("Female", "Male"))))
## (Satisfaction categories abbreviated for convenience.)
ftable(. ~ Gender + Income, Satisfaction)
## Table 7.8 in Agresti (2002), p. 288.
mantelhaen.test(Satisfaction)
## See Table 7.12 in Agresti (2002), p. 297.
```

`mauchly.test` *Mauchly's Test of Sphericity*

Description

Tests whether a Wishart-distributed covariance matrix (or transformation thereof) is proportional to a given matrix.

Usage

```
mauchly.test(object, Sigma = diag(nrow = p),
   T = Thin.row(proj(M) - proj(X)), M = diag(nrow = p), X = ~0,
   idata = data.frame(index = seq_len(p)), ...)
```

Arguments

object	object of class SSD or mlm.
Sigma	matrix to be proportional to.
T	transformation matrix. By default computed from M and X.
M	formula or matrix describing the outer projection (see below).
X	formula or matrix describing the inner projection (see below).
idata	data frame describing intra-block design.
...	arguments to be passed to or from other methods.

Details

Mauchly's test test for whether a covariance matrix can be assumed to be proportional to a given matrix.

It is common to transform the observations prior to testing. This typically involves transformation to intra-block differences, but more complicated within-block designs can be encountered, making more elaborate transformations necessary. A transformation matrix T can be given directly or specified as the difference between two projections onto the spaces spanned by M and X, which in turn can be given as matrices or as model formulas with respect to idata (the tests will be invariant to parametrization of the quotient space M/X).

The common use of this test is in repeated measurements designs, with X=~1. This is almost, but not quite the same as testing for compund symmetry in the untransformed covariance matrix.

This is a generic function with methods for classes "mlm" and "SSD".

Value

An object of class "htest"

Note

The p-value differs slightly from that of SAS because a second order term is included in the asymptotic approximation in R.

References

T. W. Anderson (1958). *An Introduction to Multivariate Statistical Analysis.* Wiley.

See Also

SSD, anova.mlm

Examples

```
example(SSD) # Brings in the mlmfit and reacttime objects

### traditional test of intrasubj. contrasts
mauchly.test(mlmfit, X=~1)

### tests using intra-subject 3x2 design
idata <- data.frame(deg=gl(3,1,6, labels=c(0,4,8)),
                    noise=gl(2,3,6, labels=c("A","P")))
mauchly.test(mlmfit, X = ~ deg + noise, idata = idata)
mauchly.test(mlmfit, M = ~ deg + noise, X = ~ noise, idata=idata)
```

`mcnemar.test` *McNemar's Chi-squared Test for Count Data*

Description

Performs McNemar's chi-squared test for symmetry of rows and columns in a two-dimensional contingency table.

Usage

```
mcnemar.test(x, y = NULL, correct = TRUE)
```

Arguments

`x`	either a two-dimensional contingency table in matrix form, or a factor object.
`y`	a factor object; ignored if `x` is a matrix.
`correct`	a logical indicating whether to apply continuity correction when computing the test statistic.

Details

The null is that the probabilities of being classified into cells `[i,j]` and `[j,i]` are the same.

If `x` is a matrix, it is taken as a two-dimensional contingency table, and hence its entries should be nonnegative integers. Otherwise, both `x` and `y` must be vectors of the same length. Incomplete cases are removed, the vectors are coerced into factor objects, and the contingency table is computed from these.

Continuity correction is only used in the 2-by-2 case if `correct` is `TRUE`.

Value

A list with class `"htest"` containing the following components:

`statistic`	the value of McNemar's statistic.
`parameter`	the degrees of freedom of the approximate chi-squared distribution of the test statistic.
`p.value`	the p-value of the test.
`method`	a character string indicating the type of test performed, and whether continuity correction was used.
`data.name`	a character string giving the name(s) of the data.

References

Alan Agresti (1990). *Categorical data analysis*. New York: Wiley. Pages 350–354.

Examples

```
## Agresti (1990), p. 350.
## Presidential Approval Ratings.
##  Approval of the President's performance in office in two surveys,
##  one month apart, for a random sample of 1600 voting-age Americans.
Performance <-
matrix(c(794, 86, 150, 570),
       nr = 2,
       dimnames = list("1st Survey" = c("Approve", "Disapprove"),
                       "2nd Survey" = c("Approve", "Disapprove")))
Performance
mcnemar.test(Performance)
## => significant change (in fact, drop) in approval ratings
```

`median` *Median Value*

Description

Compute the sample median.

Usage

```
median(x, na.rm = FALSE)
```

Arguments

`x`	an object for which a method has been defined, or a numeric vector containing the values whose median is to be computed.
`na.rm`	a logical value indicating whether `NA` values should be stripped before the computation proceeds.

Details

This is a generic function for which methods can be written. However, the default method makes use of `sort` and `mean`, both of which are generic, and so the default method will work for most classes (e.g. `"Date"`) for which a median is a reasonable concept.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`quantile` for general quantiles.

Examples

```
median(1:4)# = 2.5 [even number]
median(c(1:3,100,1000))# = 3 [odd, robust]
```

`medpolish` *Median Polish of a Matrix*

Description

Fits an additive model using Tukey's *median polish* procedure.

Usage

```
medpolish(x, eps = 0.01, maxiter = 10, trace.iter = TRUE,
          na.rm = FALSE)
```

Arguments

`x`	a numeric matrix.
`eps`	real number greater than 0. A tolerance for convergence: see **Details**.
`maxiter`	the maximum number of iterations
`trace.iter`	logical. Should progress in convergence be reported?
`na.rm`	logical. Should missing values be removed?

Details

The model fitted is additive (constant + rows + columns). The algorithm works by alternately removing the row and column medians, and continues until the proportional reduction in the sum of absolute residuals is less than `eps` or until there have been `maxiter` iterations. The sum of absolute residuals is printed at each iteration of the fitting process, if `trace.iter` is `TRUE`. If `na.rm` is `FALSE` the presence of any `NA` value in `x` will cause an error, otherwise `NA` values are ignored.

`medpolish` returns an object of class `medpolish` (see below). There are printing and plotting methods for this class, which are invoked via by the generics `print` and `plot`.

Value

An object of class `medpolish` with the following named components:

`overall`	the fitted constant term.
`row`	the fitted row effects.
`col`	the fitted column effects.
`residuals`	the residuals.
`name`	the name of the dataset.

References

Tukey, J. W. (1977). *Exploratory Data Analysis*, Reading Massachusetts: Addison-Wesley.

See Also

`median`; `aov` for a *mean* instead of *median* decomposition.

Examples

```
## Deaths from sport parachuting;  from ABC of EDA, p.224:
deaths <-
    rbind(c(14,15,14),
          c( 7, 4, 7),
          c( 8, 2,10),
          c(15, 9,10),
          c( 0, 2, 0))
dimnames(deaths) <- list(c("1-24", "25-74", "75-199", "200++", "NA"),
                         paste(1973:1975))
deaths
(med.d <- medpolish(deaths))
plot(med.d)
## Check decomposition:
all(deaths == med.d$overall + outer(med.d$row,med.d$col, "+") + med.d$resid)
```

---

model.extract *Extract Components from a Model Frame*

Description

Returns the response, offset, subset, weights or other special components of a model frame passed as optional arguments to `model.frame`.

Usage

```
model.extract(frame, component)
model.offset(x)
model.response(data, type = "any")
model.weights(x)
```

Arguments

`frame, x, data`	A model frame.
`component`	literal character string or name. The name of a component to extract, such as `"weights"`, `"subset"`.
`type`	One of `"any"`, `"numeric"`, `"double"`. Using either of latter two coerces the result to have storage mode `"double"`.

Details

`model.extract` is provided for compatibility with S, which does not have the more specific functions. It is also useful to extract e.g. the `etastart` and `mustart` components of a `glm` fit.

`model.offset` and `model.response` are equivalent to `model.extract(, "offset")` and `model.extract(, "response")` respectively. `model.offset` sums any terms specified by `offset` terms in the formula or by `offset` arguments in the call producing the model frame: it does check that the offset is numeric.

`model.weights` is slightly different from `model.frame(, "weights")` in not naming the vector it returns.

Value

The specified component of the model frame, usually a vector.

See Also

model.frame, offset

Examples

```
a <- model.frame(cbind(ncases,ncontrols) ~ agegp+tobgp+alcgp, data=esoph)
model.extract(a, "response")
stopifnot(model.extract(a, "response") == model.response(a))

a <- model.frame(ncases/(ncases+ncontrols) ~ agegp+tobgp+alcgp,
                 data = esoph, weights = ncases+ncontrols)
model.response(a)
model.extract(a, "weights")

a <- model.frame(cbind(ncases,ncontrols) ~ agegp,
                 something = tobgp, data = esoph)
names(a)
stopifnot(model.extract(a, "something") == esoph$tobgp)
```

---

model.frame *Extracting the "Environment" of a Model Formula*

Description

model.frame (a generic function) and its methods return a data.frame with the variables needed to use formula and any ... arguments.

Usage

```
model.frame(formula, ...)

## Default S3 method:
model.frame(formula, data = NULL,
            subset = NULL, na.action = na.fail,
            drop.unused.levels = FALSE, xlev = NULL, ...)

## S3 method for class 'aovlist':
model.frame(formula, data = NULL, ...)

## S3 method for class 'glm':
model.frame(formula, ...)

## S3 method for class 'lm':
model.frame(formula, ...)
```

Arguments

formula	a model formula or terms object or an R object.
data	a data.frame, list or environment (or object coercible by as.data.frame to a data.frame), containing the variables in formula. Neither a matrix nor an array will be accepted.
subset	a specification of the rows to be used: defaults to all rows. This can be any valid indexing vector (see [.data.frame) for the rows of data or if that is not supplied, a data frame made up of the variables used in formula.
na.action	how NAs are treated. The default is first, any na.action attribute of data, second a na.action setting of options, and third na.fail if that is unset. The "factory-fresh" default is na.omit. Another possible value is NULL.
drop.unused.levels	should factors have unused levels dropped? Defaults to FALSE.
xlev	a named list of character vectors giving the full set of levels to be assumed for each factor.
...	further arguments such as data, na.action, subset. Any additional arguments such as offset and weights which reach the default method are used to create further columns in the model frame, with parenthesised names such as "(offset)".

Details

Exactly what happens depends on the class and attributes of the object formula. If this is an object of fitted-model class such as "lm", the method will either returned the saved model frame used when fitting the model (if any, often selected by argument model = TRUE) or pass the call used when fitting on to the default method. The default method itself can cope with rather standard model objects such as those of classes "lqs" and "ppr" from package **MASS** if no other arguments are supplied.

The rest of this section applies only to the default method.

If either formula or data is already a model frame (a data frame with a "terms" attribute and the other is missing, the model frame is returned. Unless formula is a terms object, terms is called on it. (If you wish to use the keep.order argument of terms.formula, pass a terms object rather than a formula.)

Row names for the model frame are taken from the data argument if present, then from the names of the response in the formula (or rownames if it is a matrix), if there is one.

All the variables in formula, subset and in ... are looked for first in data and then in the environment of formula (see the help for formula() for further details) and collected into a data frame. Then the subset expression is evaluated, and it is is used as a row index to the data frame. Then the na.action function is applied to the data frame (and may well add attributes). The levels of any factors in the data frame are adjusted according to the drop.unused.levels and xlev arguments.

Unless na.action = NULL, time-series attributes will be removed from the variables found (since they will be wrong if NAs are removed).

Note that *all* the variables in the formula are included in the data frame, even those preceded by −.

Only variables whose type is raw, logical, integer, real, complex or character can be included in a model frame: this includes classed variables such as factors (whose underlying type is integer), but excludes lists.

Value

A data.frame containing the variables used in formula plus those specified ....

References

Chambers, J. M. (1992) *Data for models.* Chapter 3 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

model.matrix for the "design matrix", formula for formulas and expand.model.frame for model.frame manipulation.

Examples

```
data.class(model.frame(dist ~ speed, data = cars))
```

---

model.matrix *Construct Design Matrices*

Description

model.matrix creates a design matrix.

Usage

```
model.matrix(object, ...)

## Default S3 method:
model.matrix(object, data = environment(object),
             contrasts.arg = NULL, xlev = NULL, ...)
```

Arguments

object	an object of an appropriate class. For the default method, a model formula or terms object.
data	a data frame created with model.frame. If another sort of object, model.frame is called first.
contrasts.arg	A list, whose entries are contrasts suitable for input to the contrasts replacement function and whose names are the names of columns of data containing factors.
xlev	to be used as argument of model.frame if data has no "terms" attribute.
...	further arguments passed to or from other methods.

Details

`model.matrix` creates a design matrix from the description given in `terms(object)`, using the data in `data` which must contain variables with the same names as would be created by a call to `model.frame(object)` or, more precisely, by evaluating `attr(terms(object), "variables")`. If it is a data frame, there may be other columns and the order of columns is not important. Any character variables are coerced to factors, with a warning. After coercion, all the variables used in RHD of the formula must be logical, integer, numeric or factor.

If `contrasts.arg` is specified for a factor it overrides the default factor coding for that variable and any `"contrasts"` attribute set by `C` or `contrasts`.

In an interaction term, the variable whose levels vary fastest is the first one to appear in the formula (and not in the term), so in `~ a + b + b:a` the interaction will have `a` varying fastest.

By convention, if the response variable also appears on the right-hand side of the formula it is dropped (with a warning), although interactions involving the term are retained.

Value

The design matrix for a regression model with the specified formula and data.

There is an attribute `"assign"`, an integer vector with an entry for each column in the matrix giving the term in the formula which gave rise to the column. Value `0` corresponds to the intercept (if any), and positive values to terms in the order given by the `terms.labels` attribute of the `terms` structure corresponding to `object`.

If there are any factors in terms in the model, there is an attribute `"contrasts"`, a named list with an entry for each factor. This specifies the contrasts that would be used in terms in which the factor is coded by contrasts (in some terms dummy coding may be used), either as a character vector naming a function or as a numeric matrix.

References

Chambers, J. M. (1992) *Data for models.* Chapter 3 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`model.frame`, `model.extract`, `terms`

Examples

```
ff <- log(Volume) ~ log(Height) + log(Girth)
str(m <- model.frame(ff, trees))
mat <- model.matrix(ff, m)

dd <- data.frame(a = gl(3,4), b = gl(4,1,12))# balanced 2-way
options("contrasts")
model.matrix(~ a + b, dd)
model.matrix(~ a + b, dd, contrasts = list(a="contr.sum"))
model.matrix(~ a + b, dd, contrasts = list(a="contr.sum", b="contr.poly"))
m.orth <- model.matrix(~a+b, dd, contrasts = list(a="contr.helmert"))
crossprod(m.orth)# m.orth is  ALMOST  orthogonal
```

`model.tables` *Compute Tables of Results from an Aov Model Fit*

Description

Computes summary tables for model fits, especially complex `aov` fits.

Usage

```
model.tables(x, ...)

## S3 method for class 'aov':
model.tables(x, type = "effects", se = FALSE, cterms, ...)

## S3 method for class 'aovlist':
model.tables(x, type = "effects", se = FALSE, ...)
```

Arguments

`x`	a model object, usually produced by `aov`
`type`	type of table: currently only `"effects"` and `"means"` are implemented.
`se`	should standard errors be computed?
`cterms`	A character vector giving the names of the terms for which tables should be computed. The default is all tables.
`...`	further arguments passed to or from other methods.

Details

For `type = "effects"` give tables of the coefficients for each term, optionally with standard errors.

For `type = "means"` give tables of the mean response for each combinations of levels of the factors in a term.

The `"aov"` method cannot be applied to components of a `"aovlist"` fit.

Value

An object of class `"tables.aov"`, as list which may contain components

`tables`	A list of tables for each requested term.
`n`	The replication information for each term.
`se`	Standard error information.

Warning

The implementation is incomplete, and only the simpler cases have been tested thoroughly.

Weighted `aov` fits are not supported.

See Also

`aov`, `proj`, `replications`, `TukeyHSD`, `se.contrast`

Examples

```
## From Venables and Ripley (2002) p.165.
N <- c(0,1,0,1,1,1,0,0,0,1,1,0,1,1,0,0,1,0,1,0,1,1,0,0)
P <- c(1,1,0,0,0,1,0,1,1,1,0,0,0,1,0,1,1,0,0,1,0,1,1,0)
K <- c(1,0,0,1,0,1,1,0,0,0,1,0,1,0,1,1,0,0,0,1,1,1,0,1,0)
yield <- c(49.5,62.8,46.8,57.0,59.8,58.5,55.5,56.0,62.8,55.8,69.5,
55.0, 62.0,48.8,45.5,44.2,52.0,51.5,49.8,48.8,57.2,59.0,53.2,56.0)

npk <- data.frame(block=gl(6,4), N=factor(N), P=factor(P),
                  K=factor(K), yield=yield)
options(contrasts=c("contr.helmert", "contr.treatment"))
npk.aov <- aov(yield ~ block + N*P*K, npk)
model.tables(npk.aov, "means", se = TRUE)

## as a test, not particularly sensible statistically
npk.aovE <- aov(yield ~  N*P*K + Error(block), npk)
model.tables(npk.aovE, se=TRUE)
model.tables(npk.aovE, "means")
```

---

monthplot *Plot a Seasonal or other Subseries from a Time Series*

Description

These functions plot seasonal (or other) subseries of a time series. For each season (or other category), a time series is plotted.

Usage

```
monthplot(x, ...)

## S3 method for class 'stl':
monthplot(x, labels = NULL, ylab = choice, choice = "seasonal", ...)

## S3 method for class 'StructTS':
monthplot(x, labels = NULL, ylab = choice, choice = "sea", ...)

## S3 method for class 'ts':
monthplot(x, labels = NULL, times = time(x), phase = cycle(x),
             ylab = deparse(substitute(x)), ...)

## Default S3 method:
monthplot(x, labels = 1:12,
          ylab = deparse(substitute(x)),
          times = 1:length(x),
          phase = (times - 1)%%length(labels) + 1, base = mean,
          axes = TRUE, type = c("l", "h"), box = TRUE,
          add = FALSE, ...)
```

Arguments

x	Time series or related object.
labels	Labels to use for each "season".
ylab	y label.
times	Time of each observation.
phase	Indicator for each "season".
base	Function to use for reference line for subseries.
choice	Which series of an stl or StructTS object?
...	Arguments to be passed to the default method or graphical parameters.
axes	Should axes be drawn (ignored if add=TRUE)?
type	Type of plot. The default is to join the points with lines, and "h" is for histogram-like vertical lines.
box	Should a box be drawn (ignored if add=TRUE?
add	Should thus just add on an existing plot.

Details

These functions extract subseries from a time series and plot them all in one frame. The ts, stl, and StructTS methods use the internally recorded frequency and start and finish times to set the scale and the seasons. The default method assumes observations come in groups of 12 (though this can be changed).

If the labels are not given but the phase is given, then the labels default to the unique values of the phase. If both are given, then the phase values are assumed to be indices into the labels array, i.e., they should be in the range from 1 to length(labels).

Value

These functions are executed for their side effect of drawing a seasonal subseries plot on the current graphical window.

Author(s)

Duncan Murdoch

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

ts, stl, StructTS

Examples

```
## The CO2 data
fit <- stl(log(co2), s.window = 20, t.window = 20)
plot(fit)
op <- par(mfrow = c(2,2))
monthplot(co2, ylab = "data", cex.axis = 0.8)
monthplot(fit, choice = "seasonal", cex.axis = 0.8)
monthplot(fit, choice = "trend", cex.axis = 0.8)
monthplot(fit, choice = "remainder", type = "h", cex.axis = 0.8)
par(op)

## The CO2 data, grouped quarterly
quarter <- (cycle(co2) - 1) %/% 3
monthplot(co2, phase = quarter)

## see also JohnsonJohnson
```

---

mood.test *Mood Two-Sample Test of Scale*

Description

Performs Mood's two-sample test for a difference in scale parameters.

Usage

```
mood.test(x, ...)

## Default S3 method:
mood.test(x, y, alternative = c("two.sided", "less", "greater"), ...)

## S3 method for class 'formula':
mood.test(formula, data, subset, na.action, ...)
```

Arguments

`x, y`	numeric vectors of data values.
`alternative`	indicates the alternative hypothesis and must be one of `"two.sided"` (default), `"greater"` or `"less"` all of which can be abbreviated.
`formula`	a formula of the form `lhs ~ rhs` where `lhs` is a numeric variable giving the data values and `rhs` a factor with two levels giving the corresponding groups.
`data`	an optional matrix or data frame (or similar: see `model.frame`) containing the variables in the formula `formula`. By default the variables are taken from `environment(formula)`.
`subset`	an optional vector specifying a subset of observations to be used.
`na.action`	a function which indicates what should happen when the data contain `NA`s. Defaults to `getOption("na.action")`.
`...`	further arguments to be passed to or from methods.

Details

The underlying model is that the two samples are drawn from $f(x - l)$ and $f((x - l)/s)/s$, respectively, where l is a common location parameter and s is a scale parameter.

The null hypothesis is $s = 1$.

There are more useful tests for this problem.

Value

A list with class `"htest"` containing the following components:

`statistic`	the value of the test statistic.
`p.value`	the p-value of the test.
`alternative`	a character string describing the alternative hypothesis.
`method`	the character string `"Mood two-sample test of scale"`.
`data.name`	a character string giving the names of the data.

References

William J. Conover (1971), *Practical nonparametric statistics*. New York: John Wiley & Sons. Pages 234f.

See Also

`fligner.test` for a rank-based (nonparametric) k-sample test for homogeneity of variances; `ansari.test` for another rank-based two-sample test for a difference in scale parameters; `var.test` and `bartlett.test` for parametric tests for the homogeneity in variance.

Examples

```
## Same data as for the Ansari-Bradley test:
## Serum iron determination using Hyland control sera
ramsay <- c(111, 107, 100, 99, 102, 106, 109, 108, 104, 99,
            101, 96, 97, 102, 107, 113, 116, 113, 110, 98)
jung.parekh <- c(107, 108, 106, 98, 105, 103, 110, 105, 104,
            100, 96, 108, 103, 104, 114, 114, 113, 108, 106, 99)
mood.test(ramsay, jung.parekh)
## Compare this to ansari.test(ramsay, jung.parekh)
```

Multinomial *The Multinomial Distribution*

Description

Generate multinomially distributed random number vectors and compute multinomial "density" probabilities.

Usage

```
rmultinom(n, size, prob)
dmultinom(x, size = NULL, prob, log = FALSE)
```

Arguments

`x`	vector of length K of integers in `0:size`.
`n`	number of random vectors to draw.
`size`	integer, say N, specifying the total number of objects that are put into K boxes in the typical multinomial experiment. For `dmultinom`, it defaults to `sum(x)`.
`prob`	numeric non-negative vector of length K, specifying the probability for the K classes; is internally normalized to sum 1.
`log`	logical; if TRUE, log probabilities are computed.

Details

If `x` is a K-component vector, `dmultinom(x, prob)` is the probability

$$P(X_1 = x_1, \ldots, X_K = x_k) = C \times \prod_{j=1}^{K} \pi_j^{x_j}$$

where C is the "multinomial coefficient" $C = N!/(x_1! \cdots x_K!)$ and $N = \sum_{j=1}^{K} x_j$.
By definition, each component X_j is binomially distributed as `Bin(size, prob[j])` for $j = 1, \ldots, K$.

The `rmultinom()` algorithm draws binomials from $Bin(n_j, P_j)$ sequentially, where $n_1 = N$ (N := `size`), $P_1 = \pi_1$ (π is `prob` scaled to sum 1), and for $j \geq 2$, recursively $n_j = N - \sum_{k=1}^{j-1} n_k$ and $P_j = \pi_j/(1 - \sum_{k=1}^{j-1} \pi_k)$.

Value

For `rmultinom()`, an integer `K x n` matrix where each column is a random vector generated according to the desired multinomial law, and hence summing to `size`. Whereas the *transposed* result would seem more natural at first, the returned matrix is more efficient because of columnwise storage.

Note

`dmultinom` is currently *not vectorized* at all and has no C interface (API); this may be amended in the future.

See Also

`rbinom` which is a special case conceptually.

Examples

```
rmultinom(10, size = 12, prob=c(0.1,0.2,0.8))

pr <- c(1,3,6,10) # normalization not necessary for generation
rmultinom(10, 20, prob = pr)

## all possible outcomes of Multinom(N = 3, K = 3)
X <- t(as.matrix(expand.grid(0:3, 0:3))); X <- X[, colSums(X) <= 3]
X <- rbind(X, 3:3 - colSums(X)); dimnames(X) <- list(letters[1:3], NULL)
X
round(apply(X, 2, function(x) dmultinom(x, prob = c(1,2,5))), 3)
```

na.action *NA Action*

Description

na.action is a generic function, and na.action.default its default method.

Usage

```
na.action(object, ...)
```

Arguments

object	any object whose NA action is given.
...	further arguments special methods could require.

Value

The "NA action" which should be applied to object whenever NAs are not desired.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S.* Wadsworth & Brooks/Cole.

See Also

options("na.action"), na.omit, na.fail, also for na.exclude, na.pass.

Examples

```
na.action(c(1, NA))
```

na.contiguous *Find Longest Contiguous Stretch of non-NAs*

Description

Find the longest consecutive stretch of non-missing values in a time series object. (In the event of a tie, the first such stretch.)

Usage

```
na.contiguous(object, ...)
```

Arguments

object	a univariate or multivariate time series.
...	further arguments passed to or from other methods.

Value

A time series without missing values. The class of `object` will be preserved.

See Also

`na.omit` and `na.omit.ts`; `na.fail`

Examples

```
na.contiguous(presidents)
```

`na.fail` *Handle Missing Values in Objects*

Description

These generic functions are useful for dealing with `NA`s in e.g., data frames. `na.fail` returns the object if it does not contain any missing values, and signals an error otherwise. `na.omit` returns the object with incomplete cases removed. `na.pass` returns the object unchanged.

Usage

```
na.fail(object, ...)
na.omit(object, ...)
na.exclude(object, ...)
na.pass(object, ...)
```

Arguments

`object`	an R object, typically a data frame
`...`	further arguments special methods could require.

Details

At present these will handle vectors, matrices and data frames comprising vectors and matrices (only).

If `na.omit` removes cases, the row numbers of the cases form the `"na.action"` attribute of the result, of class `"omit"`.

`na.exclude` differs from `na.omit` only in the class of the `"na.action"` attribute of the result, which is `"exclude"`. This gives different behaviour in functions making use of `naresid` and `napredict`: when `na.exclude` is used the residuals and predictions are padded to the correct length by inserting `NA`s for cases omitted by `na.exclude`.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

See Also

`na.action`; `options` with argument `na.action` for setting "NA actions"; and `lm` and `glm` for functions using these. `na.contiguous` as alternative for time series.

Examples

```
DF <- data.frame(x = c(1, 2, 3), y = c(0, 10, NA))
na.omit(DF)
m <- as.matrix(DF)
na.omit(m)
stopifnot(all(na.omit(1:3) == 1:3))  # does not affect objects with no NA's
try(na.fail(DF))#> Error: missing values in ...

options("na.action")
```

---

naprint *Adjust for Missing Values*

Description

Use missing value information to report the effects of an `na.action`.

Usage

```
naprint(x, ...)
```

Arguments

`x`	An object produced by an `na.action` function.
`...`	further arguments passed to or from other methods.

Details

This is a generic function, and the exact information differs by method. `naprint.omit` reports the number of rows omitted: `naprint.default` reports an empty string.

Value

A character string providing information on missing values, for example the number.

---

naresid *Adjust for Missing Values*

Description

Use missing value information to adjust residuals and predictions.

Usage

```
naresid(omit, x, ...)
napredict(omit, x, ...)
```

Arguments

omit	an object produced by an na.action function, typically the "na.action" attribute of the result of na.omit or na.exclude.
x	a vector, data frame, or matrix to be adjusted based upon the missing value information.
...	further arguments passed to or from other methods.

Details

These are utility functions used to allow predict, fitted and residuals methods for modelling functions to compensate for the removal of NAs in the fitting process. They are used by the default, "lm", "glm" and "nls" methods, and by further methods in packages **MASS**, **rpart** and **survival**. Also used for the scores returned by factanal, prcomp and princomp.

The default methods do nothing. The default method for the na.exclude action is to pad the object with NAs in the correct positions to have the same number of rows as the original data frame.

Currently naresid and napredict are identical, but future methods need not be. naresid is used for residuals, and napredict for fitted values and predictions.

Value

These return a similar object to x.

Note

Packages rpart and survival5 used to contain versions of these functions that had an na.omit action equivalent to that now used for na.exclude.

NegBinomial *The Negative Binomial Distribution*

Description

Density, distribution function, quantile function and random generation for the negative binomial distribution with parameters size and prob.

Usage

```
dnbinom(x, size, prob, mu, log = FALSE)
pnbinom(q, size, prob, mu, lower.tail = TRUE, log.p = FALSE)
qnbinom(p, size, prob, mu, lower.tail = TRUE, log.p = FALSE)
rnbinom(n, size, prob, mu)
```

Arguments

x	vector of (non-negative integer) quantiles.
q	vector of quantiles.
p	vector of probabilities.
n	number of observations. If length(n) > 1, the length is taken to be the number required.

size	target for number of successful trials, or dispersion parameter (the shape parameter of the gamma mixing distribution). Must be strictly positive.
prob	probability of success in each trial. 0 < prob <= 1.
mu	alternative parametrization via mean: see Details
log, log.p	logical; if TRUE, probabilities p are given as log(p).
lower.tail	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

The negative binomial distribution with size $= n$ and prob $= p$ has density

$$p(x) = \frac{\Gamma(x+n)}{\Gamma(n)x!}p^n(1-p)^x$$

for $x = 0, 1, 2, \ldots$, $n > 0$ and $0 < p \leq 1$.

This represents the number of failures which occur in a sequence of Bernoulli trials before a target number of successes is reached.

A negative binomial distribution can arise as a mixture of Poisson distributions with mean distributed as a Γ (pgamma) distribution with scale parameter (1 - prob)/prob and shape parameter size. (This definition allows non-integer values of size.) In this model prob = scale/(1+scale), and the mean is size * (1 - prob)/prob.

The alternative parametrization (often used in ecology) is by the *mean* mu, and size, the *dispersion parameter*, where prob = size/(size+mu). The variance is mu + mu^2/size in this parametrization or $n(1-p)/p^2$ in the first one.

If an element of x is not integer, the result of dnbinom is zero, with a warning.

The quantile is defined as the smallest value x such that $F(x) \geq p$, where F is the distribution function.

Value

dnbinom gives the density, pnbinom gives the distribution function, qnbinom gives the quantile function, and rnbinom generates random deviates.

Invalid size or prob will result in return value NaN, with a warning.

Source

dnbinom computes via binomial probabilities, using code contributed by Catherine Loader (see dbinom).

pnbinom uses pbeta.

qnbinom uses the Cornish–Fisher Expansion to include a skewness correction to a normal approximation, followed by a search.

rnbinom uses the derivation as a gamma mixture of Poissons, see

Devroye, L. (1986) *Non-Uniform Random Variate Generation.* Springer-Verlag, New York. Page 480.

See Also

dbinom for the binomial, dpois for the Poisson and dgeom for the geometric distribution, which is a special case of the negative binomial.

Examples

```
x <- 0:11
dnbinom(x, size = 1, prob = 1/2) * 2^(1 + x) # == 1
126 /  dnbinom(0:8, size  = 2, prob  = 1/2) #- theoretically integer

## Cumulative ('p') = Sum of discrete prob.s ('d');  Relative error :
summary(1 - cumsum(dnbinom(x, size = 2, prob = 1/2)) /
                  pnbinom(x, size  = 2, prob = 1/2))

x <- 0:15
size <- (1:20)/4
persp(x,size, dnb <- outer(x,size,function(x,s)dnbinom(x,s, pr= 0.4)),
      xlab = "x", ylab = "s", zlab="density", theta = 150)
title(tit <- "negative binomial density(x,s, pr = 0.4)  vs.  x & s")

image  (x,size, log10(dnb), main= paste("log [",tit,"]"))
contour(x,size, log10(dnb),add=TRUE)

## Alternative parametrization
x1 <- rnbinom(500, mu = 4, size = 1)
x2 <- rnbinom(500, mu = 4, size = 10)
x3 <- rnbinom(500, mu = 4, size = 100)
h1 <- hist(x1, breaks = 20, plot = FALSE)
h2 <- hist(x2, breaks = h1$breaks, plot = FALSE)
h3 <- hist(x3, breaks = h1$breaks, plot = FALSE)
barplot(rbind(h1$counts, h2$counts, h3$counts),
        beside = TRUE, col = c("red","blue","cyan"),
        names.arg = round(h1$breaks[-length(h1$breaks)]))
```

---

nextn *Highly Composite Numbers*

Description

`nextn` returns the smallest integer, greater than or equal to `n`, which can be obtained as a product of powers of the values contained in `factors`. `nextn` is intended to be used to find a suitable length to zero-pad the argument of `fft` to so that the transform is computed quickly. The default value for `factors` ensures this.

Usage

```
nextn(n, factors = c(2,3,5))
```

Arguments

`n`	an integer.
`factors`	a vector of positive integer factors.

See Also

`convolve`, `fft`.

Examples

```
nextn(1001) # 1024
table(sapply(599:630, nextn))
```

`nlm` *Non-Linear Minimization*

Description

This function carries out a minimization of the function `f` using a Newton-type algorithm. See the references for details.

Usage

```
nlm(f, p, hessian = FALSE, typsize=rep(1, length(p)), fscale=1,
    print.level = 0, ndigit=12, gradtol = 1e-6,
    stepmax = max(1000 * sqrt(sum((p/typsize)^2)), 1000),
    steptol = 1e-6, iterlim = 100, check.analyticals = TRUE, ...)
```

Arguments

`f`	the function to be minimized. If the function value has an attribute called `gradient` or both `gradient` and `hessian` attributes, these will be used in the calculation of updated parameter values. Otherwise, numerical derivatives are used. `deriv` returns a function with suitable `gradient` attribute. This should be a function of a vector of the length of `p` followed by any other arguments specified by the `...` argument.
`p`	starting parameter values for the minimization.
`hessian`	if `TRUE`, the hessian of `f` at the minimum is returned.
`typsize`	an estimate of the size of each parameter at the minimum.
`fscale`	an estimate of the size of `f` at the minimum.
`print.level`	this argument determines the level of printing which is done during the minimization process. The default value of `0` means that no printing occurs, a value of `1` means that initial and final details are printed and a value of 2 means that full tracing information is printed.
`ndigit`	the number of significant digits in the function `f`.
`gradtol`	a positive scalar giving the tolerance at which the scaled gradient is considered close enough to zero to terminate the algorithm. The scaled gradient is a measure of the relative change in `f` in each direction `p[i]` divided by the relative change in `p[i]`.
`stepmax`	a positive scalar which gives the maximum allowable scaled step length. `stepmax` is used to prevent steps which would cause the optimization function to overflow, to prevent the algorithm from leaving the area of interest in parameter space, or to detect divergence in the algorithm. `stepmax` would be chosen small enough to prevent the first two of these occurrences, but should be larger than any anticipated reasonable step.
`steptol`	A positive scalar providing the minimum allowable relative step length.

`iterlim` a positive integer specifying the maximum number of iterations to be performed before the program is terminated.

`check.analyticals` a logical scalar specifying whether the analytic gradients and Hessians, if they are supplied, should be checked against numerical derivatives at the initial parameter values. This can help detect incorrectly formulated gradients or Hessians.

`...` additional arguments to `f`.

Details

If a gradient or hessian is supplied but evaluates to the wrong mode or length, it will be ignored if `check.analyticals = TRUE` (the default) with a warning. The hessian is not even checked unless the gradient is present and passes the sanity checks.

From the three methods available in the original source, we always use method "1" which is line search.

Value

A list containing the following components:

`minimum` the value of the estimated minimum of `f`.

`estimate` the point at which the minimum value of `f` is obtained.

`gradient` the gradient at the estimated minimum of `f`.

`hessian` the hessian at the estimated minimum of `f` (if requested).

`code` an integer indicating why the optimization process terminated.

- **1:** relative gradient is close to zero, current iterate is probably solution.
- **2:** successive iterates within tolerance, current iterate is probably solution.
- **3:** last global step failed to locate a point lower than `estimate`. Either `estimate` is an approximate local minimum of the function or `steptol` is too small.
- **4:** iteration limit exceeded.
- **5:** maximum step size `stepmax` exceeded five consecutive times. Either the function is unbounded below, becomes asymptotic to a finite value from above in some direction or `stepmax` is too small.

`iterations` the number of iterations performed.

References

Dennis, J. E. and Schnabel, R. B. (1983) *Numerical Methods for Unconstrained Optimization and Nonlinear Equations.* Prentice-Hall, Englewood Cliffs, NJ.

Schnabel, R. B., Koontz, J. E. and Weiss, B. E. (1985) A modular system of algorithms for unconstrained minimization. *ACM Trans. Math. Software*, **11**, 419–440.

See Also

`optim` and `nlminb`.

`constrOptim` for constrained optimization, `optimize` for one-dimensional minimization and `uniroot` for root finding. `deriv` to calculate analytical derivatives.

For nonlinear regression, `nls` may be better.

Examples

```
f <- function(x) sum((x-1:length(x))^2)
nlm(f, c(10,10))
nlm(f, c(10,10), print.level = 2)
str(nlm(f, c(5), hessian = TRUE))

f <- function(x, a) sum((x-a)^2)
nlm(f, c(10,10), a=c(3,5))
f <- function(x, a)
{
    res <- sum((x-a)^2)
    attr(res, "gradient") <- 2*(x-a)
    res
}
nlm(f, c(10,10), a=c(3,5))

## more examples, including the use of derivatives.
## Not run: demo(nlm)
```

---

nlminb *Optimization using PORT routines*

Description

Unconstrained and constrained optimization using PORT routines.

Usage

```
nlminb(start, objective, gradient = NULL, hessian = NULL,
       scale = 1, control = list(), lower = -Inf, upper = Inf, ...)
```

Arguments

`start`	numeric vector, initial values for the parameters to be optimized.
`objective`	Function to be minimized. Must return a scalar value (possibly NA/Inf). The first argument to `objective` is the vector of parameters to be optimized, whose initial values are supplied through `start`. Further arguments (fixed during the course of the optimization) to `objective` may be specified as well (see `...`).
`gradient`	Optional function that takes the same arguments as `objective` and evaluates the gradient of `objective` at its first argument. Must return a vector as long as `start`.
`hessian`	Optional function that takes the same arguments as `objective` and evaluates the hessian of `objective` at its first argument. Must return a square matrix of order `length(start)`. Only the lower triangle is used.
`scale`	See PORT documentation (or leave alone).
`control`	A list of control parameters. See below for details.
`lower, upper`	vectors of lower and upper bounds, replicated to be as long as `start`. If unspecified, all parameters are assumed to be unconstrained.
`...`	Ffurther arguments to be supplied to `objective`.

Details

The PORT documentation is at http://netlib.bell-labs.com/cm/cs/cstr/153.pdf.

Value

A list with components:

`par`	The best set of parameters found.
`objective`	The value of `objective` corresponding to `par`.
`convergence`	An integer code. `0` indicates successful convergence.
`message`	A character string giving any additional information returned by the optimizer, or `NULL`. For details, see PORT documentation.
`iterations`	Number of iterations performed.
`evaluations`	Number of objective function and gradient function evaluations

Control parameters

Possible names in the `control` list and their default values are:

`eval.max` Maximum number of evaluations of the objective function allowed. Defaults to 200.

`iter.max` Maximum number of iterations allowed. Defaults to 150.

`trace` The value of the objective function and the parameters is printed every trace'th iteration. Defaults to 0 which indicates no trace information is to be printed.

`abs.tol` Absolute tolerance. Defaults to `1e-20`.

`rel.tol` Relative tolerance. Defaults to `1e-10`.

`x.tol` X tolerance. Defaults to `1.5e-8`.

`step.min` Minimum step size. Defaults to `2.2e-14`.

Author(s)

(of R port) Douglas Bates and Deepayan Sarkar.

References

http://netlib.bell-labs.com/netlib/port/

See Also

`optim` and `nlm`.

`optimize` for one-dimensional minimization and `constrOptim` for constrained optimization.

Examples

```
x <- rnbinom(100, mu = 10, size = 10)
hdev <- function(par) {
    -sum(dnbinom(x, mu = par[1], size = par[2], log = TRUE))
}
nlminb(c(9, 12), hdev)
nlminb(c(20, 20), hdev, lower = 0, upper = Inf)
nlminb(c(20, 20), hdev, lower = 0.001, upper = Inf)
```

```
## slightly modified from the S-PLUS help page for nlminb
# this example minimizes a sum of squares with known solution y
sumsq <- function( x, y) {sum((x-y)^2)}
y <- rep(1,5)
x0 <- rnorm(length(y))
nlminb(start = x0, obj = sumsq, y = y)
# now use bounds with a y that has some components outside the bounds
y <- c( 0, 2, 0, -2, 0)
nlminb(start = x0, obj = sumsq, lower = -1, upper = 1, y = y)
# try using the gradient
sumsq.g <- function(x,y) 2*(x-y)
nlminb(start = x0, obj = sumsq, grad = sumsq.g,
       lower = -1, upper = 1, y = y)
# now use the hessian, too
sumsq.h <- function(x,y) diag(2, nrow = length(x))
nlminb(start = x0, obj = sumsq, grad = sumsq.g, hes = sumsq.h,
       lower = -1, upper = 1, y = y)

## Rest lifted from optim help page

fr <- function(x) {   ## Rosenbrock Banana function
    x1 <- x[1]
    x2 <- x[2]
    100 * (x2 - x1 * x1)^2 + (1 - x1)^2
}
grr <- function(x) { ## Gradient of 'fr'
    x1 <- x[1]
    x2 <- x[2]
    c(-400 * x1 * (x2 - x1 * x1) - 2 * (1 - x1),
       200 *      (x2 - x1 * x1))
}
nlminb(c(-1.2,1), fr)
nlminb(c(-1.2,1), fr, grr)

flb <- function(x)
    { p <- length(x); sum(c(1, rep(4, p-1)) * (x - c(1, x[-p])^2)^2) }
## 25-dimensional box constrained
## par[24] is *not* at boundary
nlminb(rep(3, 25), flb,
          lower=rep(2, 25),
          upper=rep(4, 25))
```

`nls` *Nonlinear Least Squares*

Description

Determine the nonlinear (weighted) least-squares estimates of the parameters of a nonlinear model.

Usage

```
nls(formula, data, start, control, algorithm,
    trace, subset, weights, na.action, model,
    lower, upper, ...)
```

Arguments

formula	a nonlinear model formula including variables and parameters.
data	an optional data frame in which to evaluate the variables in formula. Can also be a list or an environment, but not a matrix.
start	a named list or named numeric vector of starting estimates. Since R 2.4.0, when start is missing, a very cheap guess for start is tried (if algorithm != "plinear").
control	an optional list of control settings. See nls.control for the names of the settable control values and their effect.
algorithm	character string specifying the algorithm to use. The default algorithm is a Gauss-Newton algorithm. Other possible values are "plinear" for the Golub-Pereyra algorithm for partially linear least-squares models and "port" for the 'nl2sol' algorithm from the Port package.
trace	logical value indicating if a trace of the iteration progress should be printed. Default is FALSE. If TRUE the residual (weighted) sum-of-squares and the parameter values are printed at the conclusion of each iteration. When the "plinear" algorithm is used, the conditional estimates of the linear parameters are printed after the nonlinear parameters. When the "port" algorithm is used the objective function value printed is half the residual (weighted) sum-of-squares.
subset	an optional vector specifying a subset of observations to be used in the fitting process.
weights	an optional numeric vector of (fixed) weights. When present, the objective function is weighted least squares.
na.action	a function which indicates what should happen when the data contain NAs. The default is set by the na.action setting of options, and is na.fail if that is unset. The "factory-fresh" default is na.omit. Value na.exclude can be useful.
model	logical. If true, the model frame is returned as part of the object. Default is FALSE.
lower, upper	vectors of lower and upper bounds, replicated to be as long as start. If unspecified, all parameters are assumed to be unconstrained. Bounds can only be used with the "port" algorithm. They are ignored, with a warning, if given for other algorithms.
...	Additional optional arguments. None are used at present.

Details

An nls object is a type of fitted model object. It has methods for the generic functions anova, coef, confint, deviance, df.residual, fitted, formula, logLik, predict, print, profile, residuals, summary, vcov and weights.

Variables in formula are looked for first in data, then the environment of formula and finally along the search path. Functions in formula are searched for first in the environment of formula and then along the search path.

Value

A list of

m	an nlsModel object incorporating the model.

`data`	the expression that was passed to `nls` as the data argument. The actual data values are present in the environment of the `m` component.
`call`	the matched call.
`na.action`	the `"na.action"` attribute (if any) of the model frame.
`dataClasses`	the `"dataClasses"` attribute (if any) of the `"terms"` attribute of the model frame.
`model`	if `model = TRUE`, the model frame.
`weights`	if `weights` is supplied, the weights.
`convergence, message`	for an `algorithm = "port"` fit only, a convergence code (always 0) and message.

Warning

Do not use `nls` on artificial "zero-residual" data.

The `nls` function uses a relative-offset convergence criterion that compares the numerical imprecision at the current parameter estimates to the residual sum-of-squares. This performs well on data of the form

$$y = f(x, \theta) + \epsilon$$

(with `var(eps) > 0`). It fails to indicate convergence on data of the form

$$y = f(x, \theta)$$

because the criterion amounts to comparing two components of the round-off error. If you wish to test `nls` on artificial data please add a noise component, as shown in the example below.

The `algorithm = "port"` code appears unfinished, and does not even check that the starting value is within the bounds. Use with caution, especially where bounds are supplied.

Author(s)

Douglas M. Bates and Saikat DebRoy

References

Bates, D. M. and Watts, D. G. (1988) *Nonlinear Regression Analysis and Its Applications*, Wiley

Bates, D. M. and Chambers, J. M. (1992) *Nonlinear models.* Chapter 10 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

http://netlib.bell-labs.com/netlib/port/ for the Port library documentation.

See Also

`summary.nls`, `predict.nls`, `profile.nls`.

Examples

```
DNase1 <- subset(DNase, Run == 1)

## using a selfStart model
fm1DNase1 <- nls(density ~ SSlogis(log(conc), Asym, xmid, scal), DNase1)
summary(fm1DNase1)
## the coefficients only:
```

```
coef(fm1DNase1)
## including their SE, etc:
coef(summary(fm1DNase1))

## using conditional linearity
fm2DNase1 <- nls(density ~ 1/(1 + exp((xmid - log(conc))/scal)),
                 data = DNase1,
                 start = list(xmid = 0, scal = 1),
                 alg = "plinear", trace = TRUE)
summary(fm2DNase1)

## without conditional linearity
fm3DNase1 <- nls(density ~ Asym/(1 + exp((xmid - log(conc))/scal)),
                 data = DNase1,
                 start = list(Asym = 3, xmid = 0, scal = 1),
                 trace = TRUE)
summary(fm3DNase1)

## using Port's nl2sol algorithm
fm4DNase1 <- nls(density ~ Asym/(1 + exp((xmid - log(conc))/scal)),
                 data = DNase1,
                 start = list(Asym = 3, xmid = 0, scal = 1),
                 trace = TRUE, algorithm = "port")
summary(fm4DNase1)

## weighted nonlinear regression
Treated <- Puromycin[Puromycin$state == "treated", ]
weighted.MM <- function(resp, conc, Vm, K)
{
    ## Purpose: exactly as white book p. 451 -- RHS for nls()
    ##  Weighted version of Michaelis-Menten model
    ## ---------------------------------------------------------
    ## Arguments: 'y', 'x' and the two parameters (see book)
    ## ---------------------------------------------------------
    ## Author: Martin Maechler, Date: 23 Mar 2001

    pred <- (Vm * conc)/(K + conc)
    (resp - pred) / sqrt(pred)
}

Pur.wt <- nls( ~ weighted.MM(rate, conc, Vm, K), data = Treated,
              start = list(Vm = 200, K = 0.1),
              trace = TRUE)
summary(Pur.wt)

## The two examples below show that you can fit a model to
## artificial data with noise but not to artificial data
## without noise.
x <- 1:10
y <- 2*x + 3                               # perfect fit
yeps <- y + rnorm(length(y), sd = 0.01) # added noise
nls(yeps ~ a + b*x, start = list(a = 0.12345, b = 0.54321),
    trace = TRUE)
## Not run:
## terminates in an error, because convergence cannot be confirmed:
nls(y ~ a + b*x, start = list(a = 0.12345, b = 0.54321),
     trace = TRUE)
```

```
## End(Not run)

## nls() internal very cheap guess for starting values can be sufficient:

x <- -(1:100)/10
y <- 100 + 10 * exp(x / 2) + rnorm(x)/10
nlmod <- nls(y ~  Const + A * exp(B * x), trace=TRUE)

plot(x,y, main = "nls(*), data, true function and fit, n=100")
curve(100 + 10 * exp(x / 2), col=4, add = TRUE)
lines(x, predict(nlmod), col=2)
```

`nls.control` *Control the Iterations in nls*

Description

Allow the user to set some characteristics of the `nls` nonlinear least squares algorithm.

Usage

```
nls.control(maxiter = 50, tol = 1e-05, minFactor = 1/1024)
```

Arguments

`maxiter`	A positive integer specifying the maximum number of iterations allowed.
`tol`	A positive numeric value specifying the tolerance level for the relative offset convergence criterion.
`minFactor`	A positive numeric value specifying the minimum step-size factor allowed on any step in the iteration. The increment is calculated with a Gauss-Newton algorithm and successively halved until the residual sum of squares has been decreased or until the step-size factor has been reduced below this limit.

Value

A `list` with exactly three components:

`maxiter`

`tol`

`minFactor`

with meanings as explained under 'Arguments'.

Author(s)

Douglas Bates and Saikat DebRoy

References

Bates and Watts (1988), *Nonlinear Regression Analysis and Its Applications*, Wiley.

See Also

nls

Examples

```
nls.control(minFactor = 1/2048)
```

NLSstAsymptotic *Fit the Asymptotic Regression Model*

Description

Fits the asymptotic regression model, in the form `b0 + b1*(1-exp(-exp(lrc) * x)` to the `xy` data. This can be used as a building block in determining starting estimates for more complicated models.

Usage

```
NLSstAsymptotic(xy)
```

Arguments

`xy`	a `sortedXyData` object

Value

A numeric value of length 3 with components labelled `b0`, `b1`, and `lrc`. `b0` is the estimated intercept on the `y`-axis, `b1` is the estimated difference between the asymptote and the `y`-intercept, and `lrc` is the estimated logarithm of the rate constant.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

SSasymp

Examples

```
Lob.329 <- Loblolly[ Loblolly$Seed == "329", ]
NLSstAsymptotic(sortedXyData(expression(age), expression(height), Lob.329))
```

NLSstClosestX *Inverse Interpolation*

Description

Use inverse linear interpolation to approximate the `x` value at which the function represented by `xy` is equal to `yval`.

Usage

```
NLSstClosestX(xy, yval)
```

Arguments

`xy`	a `sortedXyData` object
`yval`	a numeric value on the `y` scale

Value

A single numeric value on the `x` scale.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

`sortedXyData`, `NLSstLfAsymptote`, `NLSstRtAsymptote`, `selfStart`

Examples

```
DNase.2 <- DNase[ DNase$Run == "2", ]
DN.srt <- sortedXyData(expression(log(conc)), expression(density), DNase.2)
NLSstClosestX(DN.srt, 1.0)
```

NLSstLfAsymptote *Horizontal Asymptote on the Left Side*

Description

Provide an initial guess at the horizontal asymptote on the left side (i.e., small values of `x`) of the graph of `y` versus `x` from the `xy` object. Primarily used within `initial` functions for self-starting nonlinear regression models.

Usage

```
NLSstLfAsymptote(xy)
```

Arguments

`xy`	a `sortedXyData` object

Value

A single numeric value estimating the horizontal asymptote for small x.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

sortedXyData, NLSstClosestX, NLSstRtAsymptote, selfStart

Examples

```
DNase.2 <- DNase[ DNase$Run == "2", ]
DN.srt <- sortedXyData( expression(log(conc)), expression(density), DNase.2 )
NLSstLfAsymptote( DN.srt )
```

NLSstRtAsymptote *Horizontal Asymptote on the Right Side*

Description

Provide an initial guess at the horizontal asymptote on the right side (i.e., large values of x) of the graph of y versus x from the xy object. Primarily used within initial functions for self-starting nonlinear regression models.

Usage

```
NLSstRtAsymptote(xy)
```

Arguments

xy	a sortedXyData object

Value

A single numeric value estimating the horizontal asymptote for large x.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

sortedXyData, NLSstClosestX, NLSstRtAsymptote, selfStart

Examples

```
DNase.2 <- DNase[ DNase$Run == "2", ]
DN.srt <- sortedXyData( expression(log(conc)), expression(density), DNase.2 )
NLSstRtAsymptote( DN.srt )
```

`Normal` *The Normal Distribution*

Description

Density, distribution function, quantile function and random generation for the normal distribution with mean equal to `mean` and standard deviation equal to `sd`.

Usage

```
dnorm(x, mean=0, sd=1, log = FALSE)
pnorm(q, mean=0, sd=1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean=0, sd=1, lower.tail = TRUE, log.p = FALSE)
rnorm(n, mean=0, sd=1)
```

Arguments

`x,q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`mean`	vector of means.
`sd`	vector of standard deviations.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

If `mean` or `sd` are not specified they assume the default values of `0` and `1`, respectively.

The normal distribution has density

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu)^2/2\sigma^2}$$

where μ is the mean of the distribution and σ the standard deviation.

`qnorm` is based on Wichura's algorithm AS 241 which provides precise results up to about 16 digits.

Value

`dnorm` gives the density, `pnorm` gives the distribution function, `qnorm` gives the quantile function, and `rnorm` generates random deviates.

Source

For `pnorm`, based on

Cody, W. D. (1993) Algorithm 715: SPECFUN – A portable FORTRAN package of special function routines and test drivers. *ACM Transactions on Mathematical Software* **19**, 22–32.

For `qnorm`, the code is a C translation of

Wichura, M. J. (1988) Algorithm AS 241: The Percentage Points of the Normal Distribution. *Applied Statistics*, **37**, 477–484.

For `rnorm`, see RNG for how to select the algorithm and for references to the supplied methods.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, volume 1, chapter 13. Wiley, New York.

See Also

`runif` and `.Random.seed` about random number generation, and `dlnorm` for the *Log*normal distribution.

Examples

```
dnorm(0) == 1/ sqrt(2*pi)
dnorm(1) == exp(-1/2)/ sqrt(2*pi)
dnorm(1) == 1/ sqrt(2*pi*exp(1))

## Using "log = TRUE" for an extended range :
par(mfrow=c(2,1))
plot(function(x) dnorm(x, log=TRUE), -60, 50,
     main = "log { Normal density }")
curve(log(dnorm(x)), add=TRUE, col="red",lwd=2)
mtext("dnorm(x, log=TRUE)", adj=0)
mtext("log(dnorm(x))", col="red", adj=1)

plot(function(x) pnorm(x, log=TRUE), -50, 10,
     main = "log { Normal Cumulative }")
curve(log(pnorm(x)), add=TRUE, col="red",lwd=2)
mtext("pnorm(x, log=TRUE)", adj=0)
mtext("log(pnorm(x))", col="red", adj=1)

## if you want the so-called 'error function'
erf <- function(x) 2 * pnorm(x * sqrt(2)) - 1
## (see Abrahamowitz and Stegun 29.2.29)
## and the so-called 'complementary error function'
erfc <- function(x) 2 * pnorm(x * sqrt(2), lower = FALSE)
```

numericDeriv *Evaluate derivatives numerically*

Description

numericDeriv numerically evaluates the gradient of an expression.

Usage

```
numericDeriv(expr, theta, rho = parent.frame(), dir = 1)
```

Arguments

expr	The expression to be differentiated. The value of this expression should be a numeric vector.
theta	A character vector of names of numeric variables used in expr.
rho	An environment containing all the variables needed to evaluate expr.
dir	A numeric vector of directions to use for the finite differences.

Details

This is a front end to the C function numeric_deriv, which is described in *Writing R Extensions*.

The numeric variables must be of type real and not integer.

Value

The value of eval(expr, envir = rho) plus a matrix attribute called gradient. The columns of this matrix are the derivatives of the value with respect to the variables listed in theta.

Author(s)

Saikat DebRoy ⟨saikat@stat.wisc.edu⟩

Examples

```
myenv <- new.env()
assign("mean", 0., env = myenv)
assign("sd", 1., env = myenv)
assign("x", seq(-3., 3., len = 31), env = myenv)
numericDeriv(quote(pnorm(x, mean, sd)), c("mean", "sd"), myenv)
```

offset *Include an Offset in a Model Formula*

Description

An offset is a term to be added to a linear predictor, such as in a generalised linear model, with known coefficient 1 rather than an estimated coefficient.

Usage

```
offset(object)
```

Arguments

object	An offset to be included in a model frame

Details

There can be more than one offset in a model formula, but - is not supported for `offset` terms (and is equivalent to +).

Value

The input value.

See Also

`model.offset`, `model.frame`.

For examples see `glm` and `Insurance` in package **MASS**.

oneway.test *Test for Equal Means in a One-Way Layout*

Description

Test whether two or more samples from normal distributions have the same means. The variances are not necessarily assumed to be equal.

Usage

```
oneway.test(formula, data, subset, na.action, var.equal = FALSE)
```

Arguments

formula	a formula of the form lhs ~ rhs where lhs gives the sample values and rhs the corresponding groups.
data	an optional matrix or data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).
subset	an optional vector specifying a subset of observations to be used.
na.action	a function which indicates what should happen when the data contain NAs. Defaults to getOption("na.action").
var.equal	a logical variable indicating whether to treat the variances in the samples as equal. If TRUE, then a simple F test for the equality of means in a one-way analysis of variance is performed. If FALSE, an approximate method of Welch (1951) is used, which generalizes the commonly known 2-sample Welch test to the case of arbitrarily many samples.

Value

A list with class "htest" containing the following components:

statistic	the value of the test statistic.
parameter	the degrees of freedom of the exact or approximate F distribution of the test statistic.
p.value	the p-value of the test.
method	a character string indicating the test performed.
data.name	a character string giving the names of the data.

References

B. L. Welch (1951), On the comparison of several mean values: an alternative approach. *Biometrika*, **38**, 330–336.

See Also

The standard t test (t.test) as the special case for two samples; the Kruskal-Wallis test kruskal.test for a nonparametric test for equal location parameters in a one-way layout.

Examples

```
## Not assuming equal variances
oneway.test(extra ~ group, data = sleep)
## Assuming equal variances
oneway.test(extra ~ group, data = sleep, var.equal = TRUE)
## which gives the same result as
anova(lm(extra ~ group, data = sleep))
```

`optim` *General-purpose Optimization*

Description

General-purpose optimization based on Nelder–Mead, quasi-Newton and conjugate-gradient algorithms. It includes an option for box-constrained optimization and simulated annealing.

Usage

```
optim(par, fn, gr = NULL,
      method = c("Nelder-Mead", "BFGS", "CG", "L-BFGS-B", "SANN"),
      lower = -Inf, upper = Inf,
      control = list(), hessian = FALSE, ...)
```

Arguments

`par`	Initial values for the parameters to be optimized over.
`fn`	A function to be minimized (or maximized), with first argument the vector of parameters over which minimization is to take place. It should return a scalar result.
`gr`	A function to return the gradient for the `"BFGS"`, `"CG"` and `"L-BFGS-B"` methods. If it is `NULL`, a finite-difference approximation will be used. For the `"SANN"` method it specifies a function to generate a new candidate point. If it is `NULL` a default Gaussian Markov kernel is used.
`method`	The method to be used. See **Details**.
`lower, upper`	Bounds on the variables for the `"L-BFGS-B"` method.
`control`	A list of control parameters. See **Details**.
`hessian`	Logical. Should a numerically differentiated Hessian matrix be returned?
`...`	Further arguments to be passed to `fn` and `gr`. Beware of partial matching to earlier arguments.

Details

By default this function performs minimization, but it will maximize if `control$fnscale` is negative.

The default method is an implementation of that of Nelder and Mead (1965), that uses only function values and is robust but relatively slow. It will work reasonably well for non-differentiable functions.

Method `"BFGS"` is a quasi-Newton method (also known as a variable metric algorithm), specifically that published simultaneously in 1970 by Broyden, Fletcher, Goldfarb and Shanno. This uses function values and gradients to build up a picture of the surface to be optimized.

Method `"CG"` is a conjugate gradients method based on that by Fletcher and Reeves (1964) (but with the option of Polak–Ribiere or Beale–Sorenson updates). Conjugate gradient methods will generally be more fragile than the BFGS method, but as they do not store a matrix they may be successful in much larger optimization problems.

Method `"L-BFGS-B"` is that of Byrd *et. al.* (1995) which allows *box constraints*, that is each variable can be given a lower and/or upper bound. The initial value must satisfy the constraints.

This uses a limited-memory modification of the BFGS quasi-Newton method. If non-trivial bounds are supplied, this method will be selected, with a warning.

Nocedal and Wright (1999) is a comprehensive reference for the previous three methods.

Method `"SANN"` is by default a variant of simulated annealing given in Belisle (1992). Simulated-annealing belongs to the class of stochastic global optimization methods. It uses only function values but is relatively slow. It will also work for non-differentiable functions. This implementation uses the Metropolis function for the acceptance probability. By default the next candidate point is generated from a Gaussian Markov kernel with scale proportional to the actual temperature. If a function to generate a new candidate point is given, method `"SANN"` can also be used to solve combinatorial optimization problems. Temperatures are decreased according to the logarithmic cooling schedule as given in Belisle (1992, p. 890); specifically, the temperature is set to `temp / log(((t-1) %/% tmax)*tmax + exp(1))`, where `t` is the current iteration step and `temp` and `tmax` are specifiable via `control`, see below. Note that the `"SANN"` method depends critically on the settings of the control parameters. It is not a general-purpose method but can be very useful in getting to a good value on a very rough surface.

Function `fn` can return `NA` or `Inf` if the function cannot be evaluated at the supplied value, but the initial value must have a computable finite value of `fn`. (Except for method `"L-BFGS-B"` where the values should always be finite.)

`optim` can be used recursively, and for a single parameter as well as many.

The `control` argument is a list that can supply any of the following components:

`trace` Non-negative integer. If positive, tracing information on the progress of the optimization is produced. Higher values may produce more tracing information: for method `"L-BFGS-B"` there are six levels of tracing. (To understand exactly what these do see the source code: higher levels give more detail.)

`fnscale` An overall scaling to be applied to the value of `fn` and `gr` during optimization. If negative, turns the problem into a maximization problem. Optimization is performed on `fn(par)/fnscale`.

`parscale` A vector of scaling values for the parameters. Optimization is performed on `par/parscale` and these should be comparable in the sense that a unit change in any element produces about a unit change in the scaled value.

`ndeps` A vector of step sizes for the finite-difference approximation to the gradient, on `par/parscale` scale. Defaults to `1e-3`.

`maxit` The maximum number of iterations. Defaults to `100` for the derivative-based methods, and `500` for `"Nelder-Mead"`. For `"SANN"` `maxit` gives the total number of function evaluations. There is no other stopping criterion. Defaults to `10000`.

`abstol` The absolute convergence tolerance. Only useful for non-negative functions, as a tolerance for reaching zero.

`reltol` Relative convergence tolerance. The algorithm stops if it is unable to reduce the value by a factor of `reltol * (abs(val) + reltol)` at a step. Defaults to `sqrt(.Machine$double.eps)`, typically about `1e-8`.

`alpha, beta, gamma` Scaling parameters for the `"Nelder-Mead"` method. `alpha` is the reflection factor (default 1.0), `beta` the contraction factor (0.5) and `gamma` the expansion factor (2.0).

`REPORT` The frequency of reports for the `"BFGS"` and `"L-BFGS-B"` methods if `control$trace` is positive. Defaults to every 10 iterations.

`type` for the conjugate-gradients method. Takes value `1` for the Fletcher–Reeves update, `2` for Polak–Ribiere and `3` for Beale–Sorenson.

lmm is an integer giving the number of BFGS updates retained in the "L-BFGS-B" method, It defaults to 5.

factr controls the convergence of the "L-BFGS-B" method. Convergence occurs when the reduction in the objective is within this factor of the machine tolerance. Default is 1e7, that is a tolerance of about 1e-8.

pgtol helps control the convergence of the "L-BFGS-B" method. It is a tolerance on the projected gradient in the current search direction. This defaults to zero, when the check is suppressed.

temp controls the "SANN" method. It is the starting temperature for the cooling schedule. Defaults to 10.

tmax is the number of function evaluations at each temperature for the "SANN" method. Defaults to 10.

Any names given to par will be copied to the vectors passed to fn and gr.

Value

A list with components:

par	The best set of parameters found.
value	The value of fn corresponding to par.
counts	A two-element integer vector giving the number of calls to fn and gr respectively. This excludes those calls needed to compute the Hessian, if requested, and any calls to fn to compute a finite-difference approximation to the gradient.
convergence	An integer code. 0 indicates successful convergence. Error codes are **1** indicates that the iteration limit maxit had been reached. **10** indicates degeneracy of the Nelder–Mead simplex. **51** indicates a warning from the "L-BFGS-B" method; see component message for further details. **52** indicates an error from the "L-BFGS-B" method; see component message for further details.
message	A character string giving any additional information returned by the optimizer, or NULL.
hessian	Only if argument hessian is true. A symmetric matrix giving an estimate of the Hessian at the solution found. Note that this is the Hessian of the unconstrained problem even if the box constraints are active.

Note

optim will work with one-dimensional pars, but the default method does not work well (and will warn). Use optimize instead.

Source

The code for methods "Nelder-Mead", "BFGS" and "CG" was based originally on Pascal code in Nash (1990) that was translated by p2c and then hand-optimized. Dr Nash has agreed that the code can be made freely available.

The code for method "L-BFGS-B" is based on Fortran code by Zhu, Byrd, Lu-Chen and Nocedal obtained from Netlib (file 'opt/lbfgs_bcm.shar': another version is in 'toms/778').

The code for method "SANN" was contributed by A. Trapletti.

References

Belisle, C. J. P. (1992) Convergence theorems for a class of simulated annealing algorithms on R^d. *J Applied Probability*, **29**, 885–895.

Byrd, R. H., Lu, P., Nocedal, J. and Zhu, C. (1995) A limited memory algorithm for bound constrained optimization. *SIAM J. Scientific Computing*, **16**, 1190–1208.

Fletcher, R. and Reeves, C. M. (1964) Function minimization by conjugate gradients. *Computer Journal* **7**, 148–154.

Nash, J. C. (1990) *Compact Numerical Methods for Computers. Linear Algebra and Function Minimisation.* Adam Hilger.

Nelder, J. A. and Mead, R. (1965) A simplex algorithm for function minimization. *Computer Journal* **7**, 308–313.

Nocedal, J. and Wright, S. J. (1999) *Numerical Optimization*. Springer.

See Also

`nlm`, `nlminb`.

`optimize` for one-dimensional minimization and `constrOptim` for constrained optimization.

Examples

```
fr <- function(x) {   ## Rosenbrock Banana function
    x1 <- x[1]
    x2 <- x[2]
    100 * (x2 - x1 * x1)^2 + (1 - x1)^2
}
grr <- function(x) { ## Gradient of 'fr'
    x1 <- x[1]
    x2 <- x[2]
    c(-400 * x1 * (x2 - x1 * x1) - 2 * (1 - x1),
       200 *      (x2 - x1 * x1))
}
optim(c(-1.2,1), fr)
optim(c(-1.2,1), fr, grr, method = "BFGS")
optim(c(-1.2,1), fr, NULL, method = "BFGS", hessian = TRUE)
optim(c(-1.2,1), fr, grr, method = "CG")
optim(c(-1.2,1), fr, grr, method = "CG", control=list(type=2))
optim(c(-1.2,1), fr, grr, method = "L-BFGS-B")

flb <- function(x)
    { p <- length(x); sum(c(1, rep(4, p-1)) * (x - c(1, x[-p])^2)^2) }
## 25-dimensional box constrained
optim(rep(3, 25), flb, NULL, "L-BFGS-B",
      lower=rep(2, 25), upper=rep(4, 25)) # par[24] is *not* at boundary

## "wild" function , global minimum at about -15.81515
fw <- function (x)
    10*sin(0.3*x)*sin(1.3*x^2) + 0.00001*x^4 + 0.2*x+80
plot(fw, -50, 50, n=1000, main = "optim() minimising 'wild function'")

res <- optim(50, fw, method="SANN",
             control=list(maxit=20000, temp=20, parscale=20))
res
## Now improve locally
```

```
(r2 <- optim(res$par, fw, method="BFGS"))
points(r2$par, r2$val, pch = 8, col = "red", cex = 2)

## Combinatorial optimization: Traveling salesman problem
library(stats) # normally loaded

eurodistmat <- as.matrix(eurodist)

distance <- function(sq) {  # Target function
    sq2 <- embed(sq, 2)
    return(sum(eurodistmat[cbind(sq2[,2],sq2[,1])]))
}

genseq <- function(sq) {  # Generate new candidate sequence
    idx <- seq(2, NROW(eurodistmat)-1, by=1)
    changepoints <- sample(idx, size=2, replace=FALSE)
    tmp <- sq[changepoints[1]]
    sq[changepoints[1]] <- sq[changepoints[2]]
    sq[changepoints[2]] <- tmp
    return(sq)
}

sq <- c(1,2:NROW(eurodistmat),1)  # Initial sequence
distance(sq)

set.seed(2222) # chosen to get a good soln quickly
res <- optim(sq, distance, genseq, method="SANN",
             control = list(maxit=6000, temp=2000, trace=TRUE))
res  # Near optimum distance around 12842

loc <- cmdscale(eurodist)
rx <- range(x <- loc[,1])
ry <- range(y <- -loc[,2])
tspinit <- loc[sq,]
tspres <- loc[res$par,]
s <- seq(NROW(tspres)-1)

plot(x, y, type="n", asp=1, xlab="", ylab="",
     main="initial solution of traveling salesman problem")
arrows(tspinit[s,1], -tspinit[s,2], tspinit[s+1,1], -tspinit[s+1,2],
       angle=10, col="green")
text(x, y, labels(eurodist), cex=0.8)

plot(x, y, type="n", asp=1, xlab="", ylab="",
     main="optim() 'solving' traveling salesman problem")
arrows(tspres[s,1], -tspres[s,2], tspres[s+1,1], -tspres[s+1,2],
       angle=10, col="red")
text(x, y, labels(eurodist), cex=0.8)
```

`optimize` *One Dimensional Optimization*

Description

The function `optimize` searches the interval from `lower` to `upper` for a minimum or maximum of the function `f` with respect to its first argument.

`optimise` is an alias for `optimize`.

Usage

```
optimize(f = , interval = , lower = min(interval),
         upper = max(interval), maximum = FALSE,
         tol = .Machine$double.eps^0.25, ...)
optimise(f = , interval = , lower = min(interval),
         upper = max(interval), maximum = FALSE,
         tol = .Machine$double.eps^0.25, ...)
```

Arguments

`f`	the function to be optimized. The function is either minimized or maximized over its first argument depending on the value of `maximum`.
`interval`	a vector containing the end-points of the interval to be searched for the minimum.
`lower`	the lower end point of the interval to be searched.
`upper`	the upper end point of the interval to be searched.
`maximum`	logical. Should we maximize or minimize (the default)?
`tol`	the desired accuracy.
`...`	additional named or unnamed arguments to be passed to `f` (but beware of partial matching to other arguments).

Details

The method used is a combination of golden section search and successive parabolic interpolation. Convergence is never much slower than that for a Fibonacci search. If `f` has a continuous second derivative which is positive at the minimum (which is not at `lower` or `upper`), then convergence is superlinear, and usually of the order of about 1.324.

The function `f` is never evaluated at two points closer together than $\epsilon|x_0| + (tol/3)$, where ϵ is approximately `sqrt(.Machine$double.eps)` and x_0 is the final abscissa `optimize()$minimum`.
If `f` is a unimodal function and the computed values of `f` are always unimodal when separated by at least $\epsilon\,|x| + (tol/3)$, then x_0 approximates the abscissa of the global minimum of `f` on the interval `lower,upper` with an error less than $\epsilon|x_0| + tol$.
If `f` is not unimodal, then `optimize()` may approximate a local, but perhaps non-global, minimum to the same accuracy.

The first evaluation of `f` is always at $x_1 = a+(1-\phi)(b-a)$ where `(a,b) = (lower, upper)` and $\phi = (\sqrt{5}-1)/2 = 0.61803..$ is the golden section ratio. Almost always, the second evaluation is at $x_2 = a + \phi(b-a)$. Note that a local minimum inside $[x_1, x_2]$ will be found as solution, even when `f` is constant in there, see the last example.

`f` will be called as `f(x, ...)` for a numeric value of x.

Value

A list with components `minimum` (or `maximum`) and `objective` which give the location of the minimum (or maximum) and the value of the function at that point.

Source

A C translation of Fortran code http://www.netlib.org/fmm/fmin.f based on the Algol 60 procedure localmin given in the reference.

References

Brent, R. (1973) *Algorithms for Minimization without Derivatives.* Englewood Cliffs N.J.: Prentice-Hall.

See Also

nlm, uniroot.

Examples

```
f <- function (x,a) (x-a)^2
xmin <- optimize(f, c(0, 1), tol = 0.0001, a = 1/3)
xmin

## See where the function is evaluated:
optimize(function(x) x^2*(print(x)-1), l=0, u=10)

## "wrong" solution with unlucky interval and piecewise constant f():
f  <- function(x) ifelse(x > -1, ifelse(x < 4, exp(-1/abs(x - 1)), 10), 10)
fp <- function(x) { print(x); f(x) }

plot(f, -2,5, ylim = 0:1, col = 2)
optimize(fp, c(-4, 20))# doesn't see the minimum
optimize(fp, c(-7, 20))# ok
```

---

order.dendrogram *Ordering or Labels of the Leaves in a Dendrogram*

Description

Theses functions return the order (index) or the "label" attribute for the leaves in a dendrogram. These indices can then be used to access the appropriate components of any additional data.

Usage

```
order.dendrogram(x)

## S3 method for class 'dendrogram':
labels(object, ...)
```

Arguments

x, object	a dendrogram (see as.dendrogram).
...	additional arguments

Details

The indices or labels for the leaves in left to right order are retrieved.

Value

A vector with length equal to the number of leaves in the dendrogram is returned. From `r <- order.dendrogram()`, each element is the index into the original data (from which the dendrogram was computed).

Author(s)

R. Gentleman (`order.dendrogam` and Martin Maechler (`labels.dendrogram`).

See Also

`reorder`, `dendrogram`.

Examples

```
set.seed(123)
x <- rnorm(10)
hc <- hclust(dist(x))
hc$order
dd <- as.dendrogram(hc)
order.dendrogram(dd) ## the same :
stopifnot(hc$order == order.dendrogram(dd))

d2 <- as.dendrogram(hclust(dist(USArrests)))
labels(d2) ## in this case the same as
stopifnot(labels(d2) == rownames(USArrests)[order.dendrogram(d2)])
```

---

p.adjust *Adjust P-values for Multiple Comparisons*

Description

Given a set of p-values, returns p-values adjusted using one of several methods.

Usage

```
p.adjust(p, method = p.adjust.methods, n = length(p))

p.adjust.methods
# c("holm", "hochberg", "hommel", "bonferroni", "BH", "BY",
#   "fdr", "none")
```

Arguments

`p`	vector of p-values (possibly with `NA`s).
`method`	correction method
`n`	number of comparisons, must be at least `length(p)`; only set this (to non-default) when you know what you are doing!

Details

The adjustment methods include the Bonferroni correction ("bonferroni") in which the p-values are multiplied by the number of comparisons. Less conservative corrections are also included by Holm (1979) ("holm"), Hochberg (1988) ("hochberg"), Hommel (1988) ("hommel"), Benjamini & Hochberg (1995) ("BH"), and Benjamini & Yekutieli (2001) ("BY"), respectively. A pass-through option ("none") is also included. The set of methods are contained in the p.adjust.methods vector for the benefit of methods that need to have the method as an option and pass it on to p.adjust.

The first four methods are designed to give strong control of the family wise error rate. There seems no reason to use the unmodified Bonferroni correction because it is dominated by Holm's method, which is also valid under arbitrary assumptions.

Hochberg's and Hommel's methods are valid when the hypothesis tests are independent or when they are non-negatively associated (Sarkar, 1998; Sarkar and Chang, 1997). Hommel's method is more powerful than Hochberg's, but the difference is usually small and the Hochberg p-values are faster to compute.

The "BH" and "BY" method of Benjamini, Hochberg, and Yekutieli control the false discovery rate, the expected proportion of false discoveries amongst the rejected hypotheses. The false discovery rate is a less stringent condition than the family wise error rate, so these methods are more powerful than the others.

Note that you can set n larger than length(p) which means the unobserved p-values are assumed to be greater than all the observed p for "bonferonni" and "holm" methods and equal to 1 for the other methods.

Value

A vector of corrected p-values (same length as p).

References

Benjamini, Y., and Hochberg, Y. (1995). Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society Series* B, **57**, 289–300.

Benjamini, Y., and Yekutieli, D. (2001). The control of the false discovery rate in multiple testing under dependency. *Annals of Statistics* **29**, 1165–1188.

Holm, S. (1979). A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, **6**, 65–70.

Hommel, G. (1988). A stagewise rejective multiple test procedure based on a modified Bonferroni test. *Biometrika*, **75**, 383–386.

Hochberg, Y. (1988). A sharper Bonferroni procedure for multiple tests of significance. *Biometrika*, **75**, 800–803.

Shaffer, J. P. (1995). Multiple hypothesis testing. *Annual Review of Psychology*, **46**, 561–576. (An excellent review of the area.)

Sarkar, S. (1998). Some probability inequalities for ordered MTP2 random variables: a proof of Simes conjecture. *Annals of Statistics*, **26**, 494–504.

Sarkar, S., and Chang, C. K. (1997). Simes' method for multiple hypothesis testing with positively dependent test statistics. *Journal of the American Statistical Association*, **92**, 1601–1608.

Wright, S. P. (1992). Adjusted P-values for simultaneous inference. *Biometrics*, **48**, 1005–1013. (Explains the adjusted P-value approach.)

See Also

`pairwise.*` functions such as `pairwise.t.test`.

Examples

```
set.seed(123)
x <- rnorm(50, m=c(rep(0,25),rep(3,25)))
p <- 2*pnorm( sort(-abs(x)))

round(p, 3)
round(p.adjust(p), 3)
round(p.adjust(p,"BH"), 3)

## or all of them at once (dropping the "fdr" alias):
p.adjust.M <- p.adjust.methods[p.adjust.methods != "fdr"]
p.adj <- sapply(p.adjust.M, function(meth) p.adjust(p, meth))
round(p.adj, 3)
## or a bit nicer:
noquote(apply(p.adj, 2, format.pval, digits = 3))

## and a graphic:
matplot(p, p.adj, ylab="p.adjust(p, meth)", type = "l", asp=1, lty=1:6,
        main = "P-value adjustments")
legend(.7,.6, p.adjust.M, col=1:6, lty=1:6)

## Can work with NA's:
pN <- p; iN <- c(46,47); pN[iN] <- NA
pN.a <- sapply(p.adjust.M, function(meth) p.adjust(pN, meth))
## The smallest 20 P-values all affected by the NA's :
round((pN.a / p.adj)[1:20, ] , 4)
```

`pairwise.prop.test` *Pairwise comparisons for proportions*

Description

Calculate pairwise comparisons between pairs of proportions with correction for multiple testing

Usage

```
pairwise.prop.test(x, n, p.adjust.method = p.adjust.methods, ...)
```

Arguments

`x`	Vector of counts of successes or a matrix with 2 columns giving the counts of successes and failures, respectively.
`n`	Vector of counts of trials; ignored if `x` is a matrix.
`p.adjust.method`	Method for adjusting p values (see `p.adjust`)
`...`	Additional arguments to pass to `prop.test`

Value

Object of class "pairwise.htest"

See Also

prop.test, p.adjust

Examples

```
smokers  <- c( 83, 90, 129, 70 )
patients <- c( 86, 93, 136, 82 )
pairwise.prop.test(smokers, patients)
```

---

pairwise.t.test *Pairwise t tests*

Description

Calculate pairwise comparisons between group levels with corrections for multiple testing

Usage

```
pairwise.t.test(x, g, p.adjust.method = p.adjust.methods,
                pool.sd = TRUE, ...)
```

Arguments

x	Response vector
g	Grouping vector or factor
p.adjust.method	Method for adjusting p values (see p.adjust)
pool.sd	Switch to allow/disallow the use of a pooled SD
...	Additional arguments to pass to t.test

Value

Object of class "pairwise.htest"

See Also

t.test, p.adjust

Examples

```
attach(airquality)
Month <- factor(Month, labels = month.abb[5:9])
pairwise.t.test(Ozone, Month)
pairwise.t.test(Ozone, Month, p.adj = "bonf")
pairwise.t.test(Ozone, Month, pool.sd = FALSE)
detach()
```

pairwise.table *Tabulate p values for pairwise comparisons*

Description

Creates table of p values for pairwise comparisons with corrections for multiple testing.

Usage

```
pairwise.table(compare.levels, level.names, p.adjust.method)
```

Arguments

compare.levels	Function to compute (raw) p value given indices i and j
level.names	Names of the group levels
p.adjust.method	Method for multiple testing adjustment

Details

Functions that do multiple group comparisons create separate compare.levels functions (assumed to be symmetrical in i and j) and passes them to this function.

Value

Table of p values in lower triangular form.

See Also

pairwise.t.test, et al.

pairwise.wilcox.test *Pairwise Wilcoxon rank sum tests*

Description

Calculate pairwise comparisons between group levels with corrections for multiple testing.

Usage

```
pairwise.wilcox.test(x, g, p.adjust.method = p.adjust.methods, ...)
```

Arguments

x	Response vector
g	Grouping vector or factor
p.adjust.method	Method for adjusting p values (see p.adjust)
...	Additional arguments to pass to wilcox.test.

Value

Object of class "pairwise.htest"

See Also

wilcox.test, p.adjust

Examples

```
attach(airquality)
Month <- factor(Month, labels = month.abb[5:9])
## These give warnings because of ties :
pairwise.wilcox.test(Ozone, Month)
pairwise.wilcox.test(Ozone, Month, p.adj = "bonf")
detach()
```

---

plot.acf *Plot Autocovariance and Autocorrelation Functions*

Description

Plot method for objects of class "acf".

Usage

```
## S3 method for class 'acf':
plot(x, ci = 0.95, type = "h", xlab = "Lag", ylab = NULL,
     ylim = NULL, main = NULL,
     ci.col = "blue", ci.type = c("white", "ma"),
     max.mfrow = 6, ask = Npgs > 1 && dev.interactive(),
     mar = if(nser > 2) c(3,2,2,0.8) else par("mar"),
     oma = if(nser > 2) c(1,1.2,1,1) else par("oma"),
     mgp = if(nser > 2) c(1.5,0.6,0) else par("mgp"),
     xpd = par("xpd"), cex.main = if(nser > 2) 1 else par("cex.main"),
     verbose = getOption("verbose"),
     ...)
```

Arguments

x	an object of class "acf".
ci	coverage probability for confidence interval. Plotting of the confidence interval is suppressed if ci is zero or negative.
type	the type of plot to be drawn, default to histogram like vertical lines.
xlab	the x label of the plot.
ylab	the y label of the plot.
ylim	numeric of length 2 giving the y limits for the plot.
main	overall title for the plot.
ci.col	colour to plot the confidence interval lines.

`ci.type`	should the confidence limits assume a white noise input or for lag k an MA($k-1$) input?
`max.mfrow`	positive integer; for multivariate `x` indicating how many rows and columns of plots should be put on one page, using `par(mfrow = c(m,m))`.
`ask`	logical; if `TRUE`, the user is asked before a new page is started.
`mar, oma, mgp, xpd, cex.main`	graphics parameters as in `par(*)`, by default adjusted to use smaller than default margins for multivariate `x` only.
`verbose`	logical. Should R report extra information on progress?
`...`	graphics parameters to be passed to the plotting routines.

Note

The confidence interval plotted in `plot.acf` is based on an *uncorrelated* series and should be treated with appropriate caution. Using `ci.type = "ma"` may be less potentially misleading.

See Also

`acf` which calls `plot.acf` by default.

Examples

```
z4  <- ts(matrix(rnorm(400), 100, 4), start=c(1961, 1), frequency=12)
z7  <- ts(matrix(rnorm(700), 100, 7), start=c(1961, 1), frequency=12)
acf(z4)
acf(z7, max.mfrow = 7)# squeeze on 1 page
acf(z7) # multi-page
```

`plot.density` *Plot Method for Kernel Density Estimation*

Description

The `plot` method for density objects.

Usage

```
## S3 method for class 'density':
plot(x, main = NULL, xlab = NULL, ylab = "Density", type = "l",
     zero.line = TRUE, ...)
```

Arguments

`x`	a "density" object.
`main, xlab, ylab, type`	plotting parameters with useful defaults.
`...`	further plotting parameters.
`zero.line`	logical; if `TRUE`, add a base line at $y = 0$

Value

None.

References

See Also

density.

---

plot.HoltWinters *Plot function for HoltWinters objects*

Description

Produces a chart of the original time series along with the fitted values. Optionally, predicted values (and their confidence bounds) can also be plotted.

Usage

```
## S3 method for class 'HoltWinters':
plot(x, predicted.values = NA, intervals = TRUE,
        separator = TRUE, col = 1, col.predicted = 2,
        col.intervals = 4, col.separator = 1, lty = 1,
        lty.predicted = 1, lty.intervals = 1, lty.separator = 3,
        ylab = "Observed / Fitted", main = "Holt-Winters filtering",
        ylim = NULL, ...)
```

Arguments

x	Object of class "HoltWinters"
predicted.values	Predicted values as returned by predict.HoltWinters
intervals	If TRUE, the prediction intervals are plotted (default).
separator	If TRUE, a separating line between fitted and predicted values is plotted (default).
col, lty	Color/line type of original data (default: black solid).
col.predicted, lty.predicted	Color/line type of fitted and predicted values (default: red solid).
col.intervals, lty.intervals	Color/line type of prediction intervals (default: blue solid).
col.separator, lty.separator	Color/line type of observed/predicted values separator (default: black dashed).
ylab	Label of the y-axis.
main	Main title.
ylim	Limits of the y-axis. If NULL, the range is chosen such that the plot contains the original series, the fitted values, and the predicted values if any.
...	Other graphics parametes.

Author(s)

David Meyer ⟨David.Meyer@wu-wien.ac.at⟩

References

C. C. Holt (1957) Forecasting seasonals and trends by exponentially weighted moving averages, ONR Research Memorandum, Carnigie Institute 52.

P. R. Winters (1960) Forecasting sales by exponentially weighted moving averages, *Management Science* **6**, 324–342.

See Also

`HoltWinters`, `predict.HoltWinters`

`plot.isoreg` *Plot Method for isoreg Objects*

Description

The `plot` and `lines` method for R objects of class `isoreg`.

Usage

```
## S3 method for class 'isoreg':
plot(x, plot.type = c("single", "row.wise", "col.wise"),
      main = paste("Isotonic regression", deparse(x$call)),
      main2 = "Cumulative Data and Convex Minorant",
      xlab = "x0", ylab = "x$y",
      par.fit = list(col = "red", cex = 1.5, pch = 13, lwd = 1.5),
      mar = if (both) 0.1 + c(3.5, 2.5, 1, 1) else par("mar"),
      mgp = if (both) c(1.6, 0.7, 0) else par("mgp"),
      grid = length(x$x) < 12, ...)
## S3 method for class 'isoreg':
lines(x, col = "red", lwd = 1.5,
       do.points = FALSE, cex = 1.5, pch = 13, ...)
```

Arguments

`x`	an `isoreg` object.
`plot.type`	character indicating which type of plot is desired. The first (default) only draws the data and the fit, where the others add a plot of the cumulative data and fit.
`main`	main title of plot, see `title`.
`main2`	title for second (cumulative) plot.
`xlab, ylab`	x- and y- axis annotation.
`par.fit`	a `list` of arguments (for `points` and `lines`) for drawing the fit.
`mar, mgp`	graphical parameters, see `par`, mainly for the case of two plots.

`grid` logical indicating if grid lines should be drawn. If true, `grid()` is used for the first plot, where as vertical lines are drawn at "touching" points for the cumulative plot.

`do.points` for `lines()`: logical indicating if the step points should be drawn as well (and as they are drawn in `plot()`).

`col, lwd, cex, pch` graphical arguments for `lines()`, where `cex` and `pch` are only used when `do.points` is `TRUE`.

`...` further arguments passed to and from methods.

See Also

`isoreg` for computation of `isoreg` objects.

Examples

```
example(isoreg) # for the examples there

plot(y3, main = "simple plot(.)  +  lines(<isoreg>)")
lines(ir3)

## 'same' plot as above, "proving" that only ranks of 'x' are important
plot(isoreg(2^(1:9), c(1,0,4,3,3,5,4,2,0)), plot.t = "row", log = "x")

plot(ir3, plot.type = "row", ylab = "y3")
plot(isoreg(y3 - 4), plot.t="r", ylab = "y3 - 4")
plot(ir4, plot.type = "ro",  ylab = "y4", xlab = "x = 1:n")

## experiment a bit with these (C-c C-j):
plot(isoreg(sample(9),  y3), plot.type="row")
plot(isoreg(sample(9),  y3), plot.type="col.wise")

plot(ir <- isoreg(sample(10), sample(10, replace = TRUE)), plot.t = "r")
```

---

plot.lm *Plot Diagnostics for an lm Object*

Description

Six plots (selectable by `which`) are currently available: a plot of residuals against fitted values, a Scale-Location plot of $\sqrt{|residuals|}$ against fitted values, a Normal Q-Q plot, a plot of Cook's distances versus row labels, a plot of residuals against leverages, and a plot of Cook's distances against leverage/(1-leverage). By default, the first three and `5` are provided.

Usage

```
## S3 method for class 'lm':
plot(x, which = c(1:3,5),
     caption = c("Residuals vs Fitted", "Normal Q-Q",
          "Scale-Location", "Cook's distance",
          "Residuals vs Leverage", "Cook's distance vs Leverage"),
     panel = if(add.smooth) panel.smooth else points,
```

```
    sub.caption = NULL, main = "",
    ask = prod(par("mfcol")) < length(which) && dev.interactive(),
    ...,
    id.n = 3, labels.id = names(residuals(x)), cex.id = 0.75,
    qqline = TRUE, cook.levels = c(0.5, 1.0),
    add.smooth = getOption("add.smooth"), label.pos = c(4,2))
```

Arguments

x	lm object, typically result of lm or glm.
which	if a subset of the plots is required, specify a subset of the numbers 1:6.
caption	captions to appear above the plots
panel	panel function. The useful alternative to points, panel.smooth can be chosen by add.smooth = TRUE.
sub.caption	common title—above figures if there are multiple; used as sub (s.title) otherwise. If NULL, as by default, a possible shortened version of deparse(x$call) is used.
main	title to each plot—in addition to the above caption.
ask	logical; if TRUE, the user is *ask*ed before each plot, see par(ask=.).
...	other parameters to be passed through to plotting functions.
id.n	number of points to be labelled in each plot, starting with the most extreme.
labels.id	vector of labels, from which the labels for extreme points will be chosen. NULL uses observation numbers.
cex.id	magnification of point labels.
qqline	logical indicating if a qqline() should be added to the normal Q-Q plot.
cook.levels	levels of Cook's distance at which to draw contours.
add.smooth	logical indicating if a smoother should be added to most plots; see also panel above.
label.pos	positioning of labels, for the left half and right half of the graph respectively, for plots 1-3.

Details

sub.caption—by default the function call—is shown as a subtitle (under the x-axis title) on each plot when plots are on separate pages, or as a subtitle in the outer margin (if any) when there are multiple plots per page.

The "Scale-Location" plot, also called "Spread-Location" or "S-L" plot, takes the square root of the absolute residuals in order to diminish skewness ($\sqrt{|E|}$ is much less skewed than $|E|$ for Gaussian zero-mean E).

The 'S-L', the Q-Q, and the Residual-Leverage plot, use *standardized* residuals which have identical variance (under the hypothesis). They are given as $R_i/(s \times \sqrt{1 - h_{ii}})$ where h_{ii} are the diagonal entries of the hat matrix, influence()$hat, see also hat.

The Residual-Leverage plot shows contours of equal Cook's distance, for values of cook.levels (by default 0.5 and 1) and omits cases with leverage one. If the leverages are constant (as is typically the case in a balanced aov situation) the plot uses factor level combinations instead of the leverages for the x-axis. (The factor levels are ordered by mean fitted value.)

In the Cook's distance vs leverage/(1-leverage) plot, contours of standardized residuals that are equal in magnitude are lines through the origin. The contour lines are labelled with the magnitudes.

Author(s)

John Maindonald and Martin Maechler.

References

Belsley, D. A., Kuh, E. and Welsch, R. E. (1980) *Regression Diagnostics.* New York: Wiley.

Cook, R. D. and Weisberg, S. (1982) *Residuals and Influence in Regression.* London: Chapman and Hall.

Firth, D. (1991) Generalized Linear Models. In Hinkley, D. V. and Reid, N. and Snell, E. J., eds: Pp. 55-82 in Statistical Theory and Modelling. In Honour of Sir David Cox, FRS. London: Chapman and Hall.

Hinkley, D. V. (1975) On power transformations to symmetry. *Biometrika* **62**, 101–111.

McCullagh, P. and Nelder, J. A. (1989) *Generalized Linear Models.* London: Chapman and Hall.

See Also

`termplot`, `lm.influence`, `cooks.distance`, `hatvalues`.

Examples

```
## Analysis of the life-cycle savings data
## given in Belsley, Kuh and Welsch.
plot(lm.SR <- lm(sr ~ pop15 + pop75 + dpi + ddpi, data = LifeCycleSavings))

## 4 plots on 1 page; allow room for printing model formula in outer margin:
par(mfrow = c(2, 2), oma = c(0, 0, 2, 0))
plot(lm.SR)
plot(lm.SR, id.n = NULL)               # no id's
plot(lm.SR, id.n = 5, labels.id = NULL)# 5 id numbers

## Was default in R <= 2.1.x: Cook's distances instead of Residual-Leverage plot
plot(lm.SR, which = 1:4)

## Fit a smooth curve, where applicable:
plot(lm.SR, panel = panel.smooth)
## Gives a smoother curve
plot(lm.SR, panel = function(x,y) panel.smooth(x, y, span = 1))

par(mfrow=c(2,1))# same oma as above
plot(lm.SR, which = 1:2, sub.caption = "Saving Rates, n=50, p=5")
```

---

plot.ppr *Plot Ridge Functions for Projection Pursuit Regression Fit*

Description

Plot ridge functions for projection pursuit regression fit.

Usage

```
## S3 method for class 'ppr':
plot(x, ask, type = "o", ...)
```

Arguments

x	A fit of class "ppr" as produced by a call to ppr.
ask	the graphics parameter ask: see par for details. If set to TRUE will ask between the plot of each cross-section.
type	the type of line to draw
...	further graphical parameters

Value

None

Side Effects

A series of plots are drawn on the current graphical device, one for each term in the fit.

See Also

ppr, par

Examples

```
with(rock, {
area1 <- area/10000; peri1 <- peri/10000
par(mfrow=c(3,2))# maybe: , pty="s")
rock.ppr <- ppr(log(perm) ~ area1 + peri1 + shape,
                data = rock, nterms = 2, max.terms = 5)
plot(rock.ppr, main="ppr(log(perm)~ ., nterms=2, max.terms=5)")
plot(update(rock.ppr, bass=5), main = "update(..., bass = 5)")
plot(update(rock.ppr, sm.method="gcv", gcvpen=2),
     main = "update(..., sm.method=\"gcv\", gcvpen=2)")
})
```

---

plot.profile.nls *Plot a profile.nls Object*

Description

Displays a series of plots of the profile t function and interpolated confidence intervals for the parameters in a nonlinear regression model that has been fit with nls and profiled with profile.nls.

Usage

```
## S3 method for class 'profile.nls':
plot(x, levels, conf= c(99, 95, 90, 80, 50)/100,
     nseg = 50, absVal =TRUE, ...)
```

Arguments

x	an object of class "profile.nls"
levels	levels, on the scale of the absolute value of a t statistic, at which to interpolate intervals. Usually conf is used instead of giving levels explicitly.
conf	a numeric vector of confidence levels for profile-based confidence intervals on the parameters. Defaults to c(0.99, 0.95, 0.90, 0.80, 0.50).
nseg	an integer value giving the number of segments to use in the spline interpolation of the profile t curves. Defaults to 50.
absVal	a logical value indicating whether or not the plots should be on the scale of the absolute value of the profile t. Defaults to TRUE.
...	other arguments to the plot function can be passed here.

Author(s)

Douglas M. Bates and Saikat DebRoy

References

Bates, D.M. and Watts, D.G. (1988), *Nonlinear Regression Analysis and Its Applications*, Wiley (chapter 6)

See Also

nls, profile, profile.nls

Examples

```
# obtain the fitted object
fm1 <- nls(demand ~ SSasympOrig( Time, A, lrc ), data = BOD)
# get the profile for the fitted model
pr1 <- profile( fm1 )
opar <- par(mfrow = c(2,2), oma = c(1.1, 0, 1.1, 0), las = 1)
plot(pr1, conf = c(95, 90, 80, 50)/100)
plot(pr1, conf = c(95, 90, 80, 50)/100, absVal = FALSE)
mtext("Confidence intervals based on the profile sum of squares",
      side = 3, outer = TRUE)
mtext("BOD data - confidence levels of 50%, 80%, 90% and 95%",
      side = 1, outer = TRUE)
par(opar)
```

---

plot.spec *Plotting Spectral Densities*

Description

Plotting method for objects of class "spec". For multivariate time series it plots the marginal spectra of the series or pairs plots of the coherency and phase of the cross-spectra.

Usage

```
## S3 method for class 'spec':
plot(x, add = FALSE, ci = 0.95, log = c("yes", "dB", "no"),
     xlab = "frequency", ylab = NULL, type = "l",
     ci.col = "blue", ci.lty = 3,
     main = NULL, sub = NULL,
     plot.type = c("marginal", "coherency", "phase"),
     ...)

plot.spec.phase(x, ci = 0.95,
                xlab = "frequency", ylab = "phase",
                ylim = c(-pi, pi), type = "l",
                main = NULL, ci.col = "blue", ci.lty = 3, ...)

plot.spec.coherency(x, ci = 0.95,
                    xlab = "frequency",
                    ylab = "squared coherency",
                    ylim = c(0, 1), type = "l",
                    main = NULL, ci.col = "blue", ci.lty = 3, ...)
```

Arguments

x	an object of class "spec".
add	logical. If TRUE, add to already existing plot. Only valid for plot.type = "marginal".
ci	coverage probability for confidence interval. Plotting of the confidence bar/limits is omitted unless ci is strictly positive.
log	If "dB", plot on log10 (decibel) scale (as S-PLUS), otherwise use conventional log scale or linear scale. Logical values are also accepted. The default is "yes" unless options(ts.S.compat = TRUE) has been set, when it is "dB". Only valid for plot.type = "marginal".
xlab	the x label of the plot.
ylab	the y label of the plot. If missing a suitable label will be constructed.
type	the type of plot to be drawn, defaults to lines.
ci.col	colour for plotting confidence bar or confidence intervals for coherency and phase.
ci.lty	line type for confidence intervals for coherency and phase.
main	overall title for the plot. If missing, a suitable title is constructed.
sub	a sub title for the plot. Only used for plot.type = "marginal". If missing, a description of the smoothing is used.
plot.type	For multivariate time series, the type of plot required. Only the first character is needed.
ylim, ...	Graphical parameters.

See Also

spectrum

`plot.stepfun` *Plot Step Functions*

Description

Method of the generic `plot` for `stepfun` objects and utility for plotting piecewise constant functions.

Usage

```
## S3 method for class 'stepfun':
plot(x, xval, xlim, ylim,
     xlab = "x", ylab = "f(x)", main = NULL,
     add = FALSE, verticals = TRUE, do.points = TRUE,
     pch = par("pch"),
     col.points = par("col"), cex.points = par("cex"),
     col.hor = par("col"), col.vert = par("col"),
     lty = par("lty"), lwd = par("lwd"), ...)

## S3 method for class 'stepfun':
lines(x, ...)
```

Arguments

`x`	an R object inheriting from `"stepfun"`.
`xval`	numeric vector of abscissa values at which to evaluate `x`. Defaults to `knots(x)` restricted to `xlim`.
`xlim,ylim`	numeric(2) each; range of `x` or `y` values to use. Both have sensible defaults.
`xlab,ylab`	labels of x and y axis.
`main`	main title.
`add`	logical; if `TRUE` only *add* to an existing plot.
`verticals`	logical; if `TRUE`, draw vertical lines at steps.
`do.points`	logical; if `true`, also draw points at the (`xlim` restricted) knot locations.
`pch`	character; point character if `do.points`.
`col.points`	character or integer code; color of points if `do.points`.
`cex.points`	numeric; character expansion factor if `do.points`.
`col.hor`	color of horizontal lines.
`col.vert`	color of vertical lines.
`lty, lwd`	line type and thickness for all lines.
`...`	further arguments of `plot(.)`, or if `(add)` `segments(.)`.

Value

A list with two components

`t`	abscissa (x) values, including the two outermost ones.
`y`	y values 'in between' the `t[]`.

Author(s)

Martin Maechler ⟨maechler@stat.math.ethz.ch⟩, 1990, 1993; ported to R, 1997.

See Also

`ecdf` for empirical distribution functions as special step functions, `approxfun` and `splinefun`.

Examples

```
y0 <- c(1,2,4,3)
sfun0  <- stepfun(1:3, y0, f = 0)
sfun.2 <- stepfun(1:3, y0, f = .2)
sfun1  <- stepfun(1:3, y0, right = TRUE)

tt <- seq(0,3, by=0.1)
op <- par(mfrow=c(2,2))
plot(sfun0); plot(sfun0, xval=tt, add=TRUE, col.h="bisque")
plot(sfun.2);plot(sfun.2,xval=tt, add=TRUE, col.h="orange")
plot(sfun1);lines(sfun1, xval=tt, col.h="coral")
##-- This is  revealing :
plot(sfun0, verticals= FALSE,
     main = "stepfun(x, y0, f=f)  for f = 0, .2, 1")
for(i in 1:3)
  lines(list(sfun0,sfun.2,stepfun(1:3,y0,f = 1))[[i]], col.h=i, col.v=i)
legend(2.5, 1.9, paste("f =", c(0,0.2,1)), col=1:3, lty=1, y.inter=1)
par(op)

# Extend and/or restrict 'viewport':
plot(sfun0, xlim = c(0,5), ylim = c(0, 3.5),
     main = "plot(stepfun(*), xlim= . , ylim = .)")

##-- this works too (automatic call to  ecdf(.)):
plot.stepfun(rt(50, df=3), col.vert = "gray20")
```

---

plot.ts *Plotting Time-Series Objects*

Description

Plotting method for objects inheriting from class `"ts"`.

Usage

```
## S3 method for class 'ts':
plot(x, y = NULL, plot.type = c("multiple", "single"),
        xy.labels, xy.lines, panel = lines, nc, yax.flip = FALSE,
        mar.multi = c(0, 5.1, 0, if(yax.flip) 5.1 else 2.1),
        oma.multi = c(6, 0, 5, 0), axes = TRUE, ...)

## S3 method for class 'ts':
lines(x, ...)
```

Arguments

x, y	time series objects, usually inheriting from class "ts".
plot.type	for multivariate time series, should the series by plotted separately (with a common time axis) or on a single plot?
xy.labels	logical, indicating if text() labels should be used for an x-y plot, *or* character, supplying a vector of labels to be used. The default is to label for up to 150 points, and not for more.
xy.lines	logical, indicating if lines should be drawn for an x-y plot. Defaults to the value of xy.labels if that is logical, otherwise to TRUE.
panel	a function(x, col, bg, pch, type, ...) which gives the action to be carried out in each panel of the display for plot.type="multiple". The default is lines.
nc	the number of columns to use when type="multiple". Defaults to 1 for up to 4 series, otherwise to 2.
yax.flip	logical indicating if the y-axis (ticks and numbering) should flip from side 2 (left) to 4 (right) from series to series when type="multiple".
mar.multi, oma.multi	the (default) par settings for plot.type="multiple". Modify with care!
axes	logical indicating if x- and y- axes should be drawn.
...	additional graphical arguments, see plot, plot.default and par.

Details

If y is missing, this function creates a time series plot, for multivariate series of one of two kinds depending on plot.type.

If y is present, both x and y must be univariate, and a "scatter" plot y ~ x will be drawn, enhanced by using text if xy.labels is TRUE or character, and lines if xy.lines is TRUE.

See Also

ts for basic time series construction and access functionality.

Examples

```
## Multivariate
z <- ts(matrix(rt(200 * 8, df = 3), 200, 8), start=c(1961, 1), frequency=12)
plot(z, yax.flip = TRUE)
plot(z, axes= FALSE, ann= FALSE, frame.plot= TRUE,
      mar.mult= c(0,0,0,0), oma.mult= c(1,1,5,1))
title("plot(ts(..), axes=FALSE, ann=FALSE, frame.plot=TRUE, mar..., oma...)")

z <- window(z[,1:3], end = c(1969,12))
plot(z, type = "b")    # multiple
plot(z, plot.type="single", lty=1:3, col=4:2)

## A phase plot:
plot(nhtemp, c(nhtemp[-1], NA), cex = .8, col="blue",
     main = "Lag plot of New Haven temperatures")
## a clearer way to do this would be
## Not run:
plot(nhtemp, lag(nhtemp, 1), cex = .8, col="blue",
```

```
     main = "Lag plot of New Haven temperatures")
## End(Not run)

## xy.lines and xy.labels are FALSE for large series:
plot(lag(sunspots, 1), sunspots, pch = ".")

SMI <- EuStockMarkets[, "SMI"]
plot(lag(SMI,  1), SMI, pch = ".")
plot(lag(SMI, 20), SMI, pch = ".", log = "xy",
     main = "4 weeks lagged SMI stocks -- log scale", xy.lines= TRUE)
```

`Poisson` *The Poisson Distribution*

Description

Density, distribution function, quantile function and random generation for the Poisson distribution with parameter `lambda`.

Usage

```
dpois(x, lambda, log = FALSE)
ppois(q, lambda, lower.tail = TRUE, log.p = FALSE)
qpois(p, lambda, lower.tail = TRUE, log.p = FALSE)
rpois(n, lambda)
```

Arguments

`x`	vector of (non-negative integer) quantiles.
`q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of random values to return.
`lambda`	vector of (non-negative) means.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

The Poisson distribution has density

$$p(x) = \frac{\lambda^x e^{-\lambda}}{x!}$$

for $x = 0, 1, 2, \ldots$. The mean and variance are $E(X) = Var(X) = \lambda$.

If an element of `x` is not integer, the result of `dpois` is zero, with a warning. $p(x)$ is computed using Loader's algorithm, see the reference in `dbinom`.

The quantile is left continuous: `qgeom(q, prob)` is the largest integer x such that $P(X \leq x) < q$.

Setting `lower.tail = FALSE` allows to get much more precise results when the default, `lower.tail = TRUE` would return 1, see the example below.

Value

`dpois` gives the (log) density, `ppois` gives the (log) distribution function, `qpois` gives the quantile function, and `rpois` generates random deviates.

Invalid `lambda` will result in return value `NaN`, with a warning.

Source

`dpois` uses C code contributed by Catherine Loader (see `dbinom`).

`ppois` uses `pgamma`.

`qpois` uses the Cornish–Fisher Expansion to include a skewness correction to a normal approximation, followed by a search.

`rpois` uses

Ahrens, J. H. and Dieter, U. (1982). Computer generation of Poisson deviates from modified normal distributions. *ACM Transactions on Mathematical Software*, **8**, 163–179.

See Also

`dbinom` for the binomial and `dnbinom` for the negative binomial distribution.

Examples

```
-log(dpois(0:7, lambda=1) * gamma(1+ 0:7)) # == 1
Ni <- rpois(50, lam= 4); table(factor(Ni, 0:max(Ni)))

1 - ppois(10*(15:25), lambda=100)               # becomes 0 (cancellation)
    ppois(10*(15:25), lambda=100, lower=FALSE)  # no cancellation

par(mfrow = c(2, 1))
x <- seq(-0.01, 5, 0.01)
plot(x, ppois(x, 1), type="s", ylab="F(x)", main="Poisson(1) CDF")
plot(x, pbinom(x, 100, 0.01),type="s", ylab="F(x)",
     main="Binomial(100, 0.01) CDF")
```

`poly` *Compute Orthogonal Polynomials*

Description

Returns or evaluates orthogonal polynomials of degree 1 to `degree` over the specified set of points `x`. These are all orthogonal to the constant polynomial of degree 0. Alternatively, evaluate raw polynomials.

Usage

```
poly(x, ..., degree = 1, coefs = NULL, raw = FALSE)
polym(..., degree = 1, raw = FALSE)

## S3 method for class 'poly':
predict(object, newdata, ...)
```

Arguments

x, newdata	a numeric vector at which to evaluate the polynomial. x can also be a matrix. Missing values are not allowed in x.
degree	the degree of the polynomial
coefs	for prediction, coefficients from a previous fit.
raw	if true, use raw and not orthogonal polynomials.
object	an object inheriting from class "poly", normally the result of a call to poly with a single vector argument.
...	poly, polym: further vectors. predict.poly: arguments to be passed to or from other methods.

Details

Although formally degree should be named (as it follows ...), an unnamed second argument of length 1 will be interpreted as the degree.

The orthogonal polynomial is summarized by the coefficients, which can be used to evaluate it via the three-term recursion given in Kennedy & Gentle (1980, pp. 343–4), and used in the "predict" part of the code.

Value

For poly with a single vector argument:
A matrix with rows corresponding to points in x and columns corresponding to the degree, with attributes "degree" specifying the degrees of the columns and (unless raw = TRUE) "coefs" which contains the centering and normalization constants used in constructing the orthogonal polynomials. The matrix has given class c("poly", "matrix").

Other cases of poly and polym, and predict.poly: a matrix.

Note

This routine is intended for statistical purposes such as contr.poly: it does not attempt to orthogonalize to machine accuracy.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

Kennedy, W. J. Jr and Gentle, J. E. (1980) *Statistical Computing* Marcel Dekker.

See Also

contr.poly.

cars for an example of polynomial regression.

Examples

```
(z <- poly(1:10, 3))
predict(z, seq(2, 4, 0.5))
poly(seq(4, 6, 0.5), 3, coefs = attr(z, "coefs"))

polym(1:4, c(1, 4:6), degree=3) # or just poly()
poly(cbind(1:4, c(1, 4:6)), degree=3)
```

power *Create a Power Link Object*

Description

Creates a link object based on the link function $\eta = \mu^{\lambda}$.

Usage

```
power(lambda = 1)
```

Arguments

lambda	a real number.

Details

If `lambda` is non-positive, it is taken as zero, and the log link is obtained. The default `lambda = 1` gives the identity link.

Value

A list with components `linkfun`, `linkinv`, `mu.eta`, and `valideta`. See `make.link` for information on their meaning.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S.* Wadsworth & Brooks/Cole.

See Also

`make.link`, `family`

To raise a number to a power, see `Arithmetic`.

To calculate the power of a test, see various functions in the **stats** package, e.g., `power.t.test`.

Examples

```
power()
quasi(link=power(1/3))[c("linkfun", "linkinv")]
```

power.anova.test *Power calculations for balanced one-way analysis of variance tests*

Description

Compute power of test or determine parameters to obtain target power.

Usage

```
power.anova.test(groups = NULL, n = NULL, between.var = NULL,
                 within.var = NULL, sig.level = 0.05, power = NULL)
```

Arguments

groups	Number of groups
n	Number of observations (per group)
between.var	Between group variance
within.var	Within group variance
sig.level	Significance level (Type I error probability)
power	Power of test (1 minus Type II error probability)

Details

Exactly one of the parameters groups, n, between.var, power, within.var, and sig.level must be passed as NULL, and that parameter is determined from the others. Notice that sig.level has non-NULL default so NULL must be explicitly passed if you want it computed.

Value

Object of class "power.htest", a list of the arguments (including the computed one) augmented with method and note elements.

Note

uniroot is used to solve power equation for unknowns, so you may see errors from it, notably about inability to bracket the root when invalid arguments are given.

Author(s)

Claus Ekstrøm

See Also

anova, lm, uniroot

Examples

```
power.anova.test(groups=4, n=5, between.var=1, within.var=3)
# Power = 0.3535594

power.anova.test(groups=4, between.var=1, within.var=3,
                 power=.80)
# n = 11.92613

## Assume we have prior knowledge of the group means:
groupmeans <- c(120, 130, 140, 150)
power.anova.test(groups = length(groupmeans),
                 between.var=var(groupmeans),
                 within.var=500, power=.90) # n = 15.18834
```

---

power.prop.test *Power calculations two sample test for proportions*

Description

Compute power of test, or determine parameters to obtain target power.

Usage

```
power.prop.test(n = NULL, p1 = NULL, p2 = NULL, sig.level = 0.05,
                power = NULL,
                alternative = c("two.sided", "one.sided"),
                strict = FALSE)
```

Arguments

n	Number of observations (per group)
p1	probability in one group
p2	probability in other group
sig.level	Significance level (Type I error probability)
power	Power of test (1 minus Type II error probability)
alternative	One- or two-sided test
strict	Use strict interpretation in two-sided case

Details

Exactly one of the parameters n, p1, p2, power, and sig.level must be passed as NULL, and that parameter is determined from the others. Notice that sig.level has a non-NULL default so NULL must be explicitly passed if you want it computed.

If strict = TRUE is used, the power will include the probability of rejection in the opposite direction of the true effect, in the two-sided case. Without this the power will be half the significance level if the true difference is zero.

Value

Object of class "power.htest", a list of the arguments (including the computed one) augmented with method and note elements.

Note

uniroot is used to solve power equation for unknowns, so you may see errors from it, notably about inability to bracket the root when invalid arguments are given. If one of them is computed p1 < p2 will hold, although this is not enforced when both are specified.

Author(s)

Peter Dalgaard. Based on previous work by Claus Ekstrøm

See Also

prop.test, uniroot

Examples

```
power.prop.test(n = 50, p1 = .50, p2 = .75)
power.prop.test(p1 = .50, p2 = .75, power = .90)
power.prop.test(n = 50, p1 = .5, power = .90)
```

---

power.t.test *Power calculations for one and two sample t tests*

Description

Compute power of test, or determine parameters to obtain target power.

Usage

```
power.t.test(n = NULL, delta = NULL, sd = 1, sig.level = 0.05,
             power = NULL,
             type = c("two.sample", "one.sample", "paired"),
             alternative = c("two.sided", "one.sided"),
             strict = FALSE)
```

Arguments

n	Number of observations (per group)
delta	True difference in means
sd	Standard deviation
sig.level	Significance level (Type I error probability)
power	Power of test (1 minus Type II error probability)
type	Type of t test
alternative	One- or two-sided test
strict	Use strict interpretation in two-sided case

Details

Exactly one of the parameters `n`, `delta`, `power`, `sd`, and `sig.level` must be passed as NULL, and that parameter is determined from the others. Notice that the last two have non-NULL defaults so NULL must be explicitly passed if you want to compute them.

If `strict = TRUE` is used, the power will include the probability of rejection in the opposite direction of the true effect, in the two-sided case. Without this the power will be half the significance level if the true difference is zero.

Value

Object of class `"power.htest"`, a list of the arguments (including the computed one) augmented with `method` and `note` elements.

Note

`uniroot` is used to solve power equation for unknowns, so you may see errors from it, notably about inability to bracket the root when invalid arguments are given.

Author(s)

Peter Dalgaard. Based on previous work by Claus Ekstrøm

See Also

`t.test`, `uniroot`

Examples

```
power.t.test(n = 20, delta = 1)
power.t.test(power = .90, delta = 1)
power.t.test(power = .90, delta = 1, alt = "one.sided")
```

`PP.test` *Phillips-Perron Test for Unit Roots*

Description

Computes the Phillips-Perron test for the null hypothesis that `x` has a unit root against a stationary alternative.

Usage

```
PP.test(x, lshort = TRUE)
```

Arguments

`x`	a numeric vector or univariate time series.
`lshort`	a logical indicating whether the short or long version of the truncation lag parameter is used.

Details

The general regression equation which incorporates a constant and a linear trend is used and the corrected t-statistic for a first order autoregressive coefficient equals one is computed. To estimate `sigma^2` the Newey-West estimator is used. If `lshort` is `TRUE`, then the truncation lag parameter is set to `trunc(4*(n/100)^0.25)`, otherwise `trunc(12*(n/100)^0.25)` is used. The *p*-values are interpolated from Table 4.2, page 103 of Banerjee *et al.* (1993).

Missing values are not handled.

Value

A list with class `"htest"` containing the following components:

`statistic`	the value of the test statistic.
`parameter`	the truncation lag parameter.
`p.value`	the *p*-value of the test.
`method`	a character string indicating what type of test was performed.
`data.name`	a character string giving the name of the data.

Author(s)

A. Trapletti

References

A. Banerjee, J. J. Dolado, J. W. Galbraith, and D. F. Hendry (1993) *Cointegration, Error Correction, and the Econometric Analysis of Non-Stationary Data*, Oxford University Press, Oxford.

P. Perron (1988) Trends and random walks in macroeconomic time series. *Journal of Economic Dynamics and Control* **12**, 297–332.

Examples

```
x <- rnorm(1000)
PP.test(x)
y <- cumsum(x) # has unit root
PP.test(y)
```

---

ppoints *Ordinates for Probability Plotting*

Description

Generates the sequence of "probability" points `(1:m - a)/(m + (1-a)-a)` where `m` is either `n`, if `length(n)==1`, or `length(n)`.

Usage

```
ppoints(n, a = ifelse(n <= 10, 3/8, 1/2))
```

Arguments

`n`	either the number of points generated or a vector of observations.
`a`	the offset fraction to be used; typically in $(0, 1)$.

Details

If $0 < a < 1$, the resulting values are within $(0, 1)$ (excluding boundaries). In any case, the resulting sequence is symmetric in $[0, 1]$, i.e., `p + rev(p) == 1`.

`ppoints()` is used in `qqplot` and `qqnorm` to generate the set of probabilities at which to evaluate the inverse distribution.

The choice of `a` follows the documentation of the function of the same name in Becker *et al* (1988), and appears to have been motivated by results from Blom (1958) on approximations to expect normal order statistics (see also `quantile`).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Blom, G. (1958) *Statistical Estimates and Transformed Beta Variables.* Wiley

See Also

`qqplot`, `qqnorm`.

Examples

```
ppoints(4) # the same as  ppoints(1:4)
ppoints(10)
ppoints(10, a=1/2)
```

---

ppr *Projection Pursuit Regression*

Description

Fit a projection pursuit regression model.

Usage

```
ppr(x, ...)

## S3 method for class 'formula':
ppr(formula, data, weights, subset, na.action,
    contrasts = NULL, ..., model = FALSE)

## Default S3 method:
ppr(x, y, weights = rep(1,n),
    ww = rep(1,q), nterms, max.terms = nterms, optlevel = 2,
    sm.method = c("supsmu", "spline", "gcvspline"),
    bass = 0, span = 0, df = 5, gcvpen = 1, ...)
```

Arguments

formula	a formula specifying one or more numeric response variables and the explanatory variables.
x	numeric matrix of explanatory variables. Rows represent observations, and columns represent variables. Missing values are not accepted.
y	numeric matrix of response variables. Rows represent observations, and columns represent variables. Missing values are not accepted.
nterms	number of terms to include in the final model.
data	a data frame (or similar: see model.frame) from which variables specified in formula are preferentially to be taken.
weights	a vector of weights w_i for each *case*.
ww	a vector of weights for each *response*, so the fit criterion is the sum over case i and responses j of w_i ww_j (y_ij - fit_ij)^2 divided by the sum of w_i.
subset	an index vector specifying the cases to be used in the training sample. (NOTE: If given, this argument must be named.)
na.action	a function to specify the action to be taken if NAs are found. The default action is given by getOption("na.action"). (NOTE: If given, this argument must be named.)
contrasts	the contrasts to be used when any factor explanatory variables are coded.
max.terms	maximum number of terms to choose from when building the model.
optlevel	integer from 0 to 3 which determines the thoroughness of an optimization routine in the SMART program. See the **Details** section.
sm.method	the method used for smoothing the ridge functions. The default is to use Friedman's super smoother supsmu. The alternatives are to use the smoothing spline code underlying smooth.spline, either with a specified (equivalent) degrees of freedom for each ridge functions, or to allow the smoothness to be chosen by GCV.
bass	super smoother bass tone control used with automatic span selection (see supsmu); the range of values is 0 to 10, with larger values resulting in increased smoothing.
span	super smoother span control (see supsmu). The default, 0, results in automatic span selection by local cross validation. span can also take a value in (0, 1].
df	if sm.method is "spline" specifies the smoothness of each ridge term via the requested equivalent degrees of freedom.
gcvpen	if sm.method is "gcvspline" this is the penalty used in the GCV selection for each degree of freedom used.
...	arguments to be passed to or from other methods.
model	logical. If true, the model frame is returned.

Details

The basic method is given by Friedman (1984), and is essentially the same code used by S-PLUS's ppreg. This code is extremely sensitive to the compiler used.

The algorithm first adds up to max.terms ridge terms one at a time; it will use less if it is unable to find a term to add that makes sufficient difference. It then removes the least "*important*" term at each step until nterms terms are left.

The levels of optimization (argument `optlevel`) differ in how thoroughly the models are refitted during this process. At level 0 the existing ridge terms are not refitted. At level 1 the projection directions are not refitted, but the ridge functions and the regression coefficients are. Levels 2 and 3 refit all the terms and are equivalent for one response; level 3 is more careful to re-balance the contributions from each regressor at each step and so is a little less likely to converge to a saddle point of the sum of squares criterion.

Value

A list with the following components, many of which are for use by the method functions.

`call`	the matched call
`p`	the number of explanatory variables (after any coding)
`q`	the number of response variables
`mu`	the argument `nterms`
`ml`	the argument `max.terms`
`gof`	the overall residual (weighted) sum of squares for the selected model
`gofn`	the overall residual (weighted) sum of squares against the number of terms, up to `max.terms`. Will be invalid (and zero) for less than `nterms`.
`df`	the argument `df`
`edf`	if `sm.method` is `"spline"` or `"gcvspline"` the equivalent number of degrees of freedom for each ridge term used.
`xnames`	the names of the explanatory variables
`ynames`	the names of the response variables
`alpha`	a matrix of the projection directions, with a column for each ridge term
`beta`	a matrix of the coefficients applied for each response to the ridge terms: the rows are the responses and the columns the ridge terms
`yb`	the weighted means of each response
`ys`	the overall scale factor used: internally the responses are divided by `ys` to have unit total weighted sum of squares.
`fitted.values`	the fitted values, as a matrix if `q > 1`.
`residuals`	the residuals, as a matrix if `q > 1`.
`smod`	internal work array, which includes the ridge functions evaluated at the training set points.
`model`	(only if `model=TRUE`) the model frame.

References

Friedman, J. H. and Stuetzle, W. (1981) Projection pursuit regression. *Journal of the American Statistical Association*, **76**, 817–823.

Friedman, J. H. (1984) SMART User's Guide. Laboratory for Computational Statistics, Stanford University Technical Report No. 1.

Venables, W. N. & Ripley, B. D. (2002) *Modern Applied Statistics with S*. Springer.

See Also

`plot.ppr`, `supsmu`, `smooth.spline`

Examples

```
# Note: your numerical values may differ
attach(rock)
area1 <- area/10000; peri1 <- peri/10000
rock.ppr <- ppr(log(perm) ~ area1 + peri1 + shape,
                data = rock, nterms = 2, max.terms = 5)
rock.ppr
# Call:
# ppr.formula(formula = log(perm) ~ area1 + peri1 + shape, data = rock,
#     nterms = 2, max.terms = 5)
#
# Goodness of fit:
#  2 terms  3 terms  4 terms  5 terms
# 8.737806 5.289517 4.745799 4.490378

summary(rock.ppr)
# .....  (same as above)
# .....
#
# Projection direction vectors:
#       term 1      term 2
# area1  0.34357179  0.37071027
# peri1 -0.93781471 -0.61923542
# shape  0.04961846  0.69218595
#
# Coefficients of ridge terms:
#    term 1    term 2
# 1.6079271 0.5460971

par(mfrow=c(3,2))# maybe: , pty="s")
plot(rock.ppr, main="ppr(log(perm)~ ., nterms=2, max.terms=5)")
plot(update(rock.ppr, bass=5), main = "update(..., bass = 5)")
plot(update(rock.ppr, sm.method="gcv", gcvpen=2),
     main = "update(..., sm.method=\"gcv\", gcvpen=2)")
cbind(perm=rock$perm, prediction=round(exp(predict(rock.ppr)), 1))
detach()
```

---

prcomp *Principal Components Analysis*

Description

Performs a principal components analysis on the given data matrix and returns the results as an object of class `prcomp`.

Usage

```
prcomp(x, ...)

## S3 method for class 'formula':
prcomp(formula, data = NULL, subset, na.action, ...)

## Default S3 method:
```

```
prcomp(x, retx = TRUE, center = TRUE, scale. = FALSE, tol = NULL, ...)

## S3 method for class 'prcomp':
predict(object, newdata, ...)
```

Arguments

formula	a formula with no response variable, referring only to numeric variables.
data	an optional data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).
subset	an optional vector used to select rows (observations) of the data matrix x.
na.action	a function which indicates what should happen when the data contain NAs. The default is set by the na.action setting of options, and is na.fail if that is unset. The "factory-fresh" default is na.omit.
...	arguments passed to or from other methods. If x is a formula one might specify scale. or tol.
x	a numeric or complex matrix (or data frame) which provides the data for the principal components analysis.
retx	a logical value indicating whether the rotated variables should be returned.
center	a logical value indicating whether the variables should be shifted to be zero centered. Alternately, a vector of length equal the number of columns of x can be supplied. The value is passed to scale.
scale.	a logical value indicating whether the variables should be scaled to have unit variance before the analysis takes place. The default is FALSE for consistency with S, but in general scaling is advisable. Alternatively, a vector of length equal the number of columns of x can be supplied. The value is passed to scale.
tol	a value indicating the magnitude below which components should be omitted. (Components are omitted if their standard deviations are less than or equal to tol times the standard deviation of the first component.) With the default null setting, no components are omitted. Other settings for tol could be tol = 0 or tol = sqrt(.Machine$double.eps), which would omit essentially constant components.
object	Object of class inheriting from "prcomp"
newdata	An optional data frame or matrix in which to look for variables with which to predict. If omitted, the scores are used. If the original fit used a formula or a data frame or a matrix with column names, newdata must contain columns with the same names. Otherwise it must contain the same number of columns, to be used in the same order.

Details

The calculation is done by a singular value decomposition of the (centered and possibly scaled) data matrix, not by using eigen on the covariance matrix. This is generally the preferred method for numerical accuracy. The print method for these objects prints the results in a nice format and the plot method produces a scree plot.

Note that scale = TRUE cannot be used if there are zero or constant (for center = TRUE) variables.

Value

prcomp returns a list with class "prcomp" containing the following components:

sdev	the standard deviations of the principal components (i.e., the square roots of the eigenvalues of the covariance/correlation matrix, though the calculation is actually done with the singular values of the data matrix).
rotation	the matrix of variable loadings (i.e., a matrix whose columns contain the eigenvectors). The function princomp returns this in the element loadings.
x	if retx is true the value of the rotated data (the centred (and scaled if requested) data multiplied by the rotation matrix) is returned. Hence, cov(x) is the diagonal matrix diag(sdev^2). For the formula method, napredict is applied to handle the treatment of values omitted by the na.action.
center, scale	the centering and scaling used, or FALSE.

Note

The signs of the columns of the rotation matrix are arbitrary, and so may differ between different programs for PCA, and even between different builds of R.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Mardia, K. V., J. T. Kent, and J. M. Bibby (1979) *Multivariate Analysis*, London: Academic Press.

Venables, W. N. and B. D. Ripley (2002) *Modern Applied Statistics with S*, Springer-Verlag.

See Also

biplot.prcomp, princomp, cor, cov, svd, eigen.

Examples

```
## the variances of the variables in the
## USArrests data vary by orders of magnitude, so scaling is appropriate
prcomp(USArrests)  # inappropriate
prcomp(USArrests, scale = TRUE)
prcomp(~ Murder + Assault + Rape, data = USArrests, scale = TRUE)
plot(prcomp(USArrests))
summary(prcomp(USArrests, scale = TRUE))
biplot(prcomp(USArrests, scale = TRUE))
```

---

predict *Model Predictions*

Description

predict is a generic function for predictions from the results of various model fitting functions. The function invokes particular *methods* which depend on the class of the first argument.

Usage

```
predict (object, ...)
```

Arguments

`object`	a model object for which prediction is desired.
`...`	additional arguments affecting the predictions produced.

Details

Most prediction methods which similar to fitting linear models have an argument `newdata` specifiying the first place to look for explanatory variables to be used for prediction. Some considerable attempts are made to match up the columns in `newdata` to those used for fitting, for example that they are of comparable types and that any factors have the same level set in the same order (or can be transformed to be so).

Time series prediction methods in package **stats** have an argument `n.ahead` specifying how many time steps ahead to predict.

Many methods have a logical argument `se.fit` saying if standard errors are to returned.

Value

The form of the value returned by `predict` depends on the class of its argument. See the documentation of the particular methods for details of what is produced by that method.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

See Also

`predict.glm`, `predict.lm`, `predict.loess`, `predict.nls`, `predict.poly`, `predict.princomp`, `predict.smooth.spline`.

For time-series prediction, `predict.ar`, `predict.Arima`, `predict.arima0`, `predict.HoltWinters`, `predict.StructTS`.

Examples

```
## All the "predict" methods found
## NB most of the methods in the standard packages are hidden.
for(fn in methods("predict"))
   try({
       f <- eval(substitute(getAnywhere(fn)$objs[[1]], list(fn = fn)))
       cat(fn, ":\n\t", deparse(args(f)), "\n")
       }, silent = TRUE)
```

`predict.Arima` *Forecast from ARIMA fits*

Description

Forecast from models fitted by `arima`.

Usage

```
## S3 method for class 'Arima':
predict(object, n.ahead = 1, newxreg = NULL,
        se.fit = TRUE, ...)
```

Arguments

`object`	The result of an `arima` fit.
`n.ahead`	The number of steps ahead for which prediction is required.
`newxreg`	New values of `xreg` to be used for prediction. Must have at least `n.ahead` rows.
`se.fit`	Logical: should standard errors of prediction be returned?
`...`	arguments passed to or from other methods.

Details

Finite-history prediction is used, via `KalmanForecast`. This is only statistically efficient if the MA part of the fit is invertible, so `predict.Arima` will give a warning for non-invertible MA models.

The standard errors of prediction exclude the uncertainty in the estimation of the ARMA model and the regression coefficients. According to Harvey (1993, pp. 58–9) the effect is small.

Value

A time series of predictions, or if `se.fit = TRUE`, a list with components `pred`, the predictions, and `se`, the estimated standard errors. Both components are time series.

References

Durbin, J. and Koopman, S. J. (2001) *Time Series Analysis by State Space Methods.* Oxford University Press.

Harvey, A. C. and McKenzie, C. R. (1982) Algorithm AS182. An algorithm for finite sample prediction from ARIMA processes. *Applied Statistics* **31**, 180–187.

Harvey, A. C. (1993) *Time Series Models*, 2nd Edition, Harvester Wheatsheaf, sections 3.3 and 4.4.

See Also

`arima`

Examples

```
predict(arima(lh, order = c(3,0,0)), n.ahead = 12)

(fit <- arima(USAccDeaths, order = c(0,1,1),
              seasonal = list(order=c(0,1,1))))
predict(fit, n.ahead = 6)
```

---

predict.glm *Predict Method for GLM Fits*

Description

Obtains predictions and optionally estimates standard errors of those predictions from a fitted generalized linear model object.

Usage

```
## S3 method for class 'glm':
predict(object, newdata = NULL,
            type = c("link", "response", "terms"),
            se.fit = FALSE, dispersion = NULL, terms = NULL,
            na.action = na.pass, ...)
```

Arguments

object	a fitted object of class inheriting from "glm".
newdata	optionally, a data frame in which to look for variables with which to predict. If omitted, the fitted linear predictors are used.
type	the type of prediction required. The default is on the scale of the linear predictors; the alternative "response" is on the scale of the response variable. Thus for a default binomial model the default predictions are of log-odds (probabilities on logit scale) and type = "response" gives the predicted probabilities. The "terms" option returns a matrix giving the fitted values of each term in the model formula on the linear predictor scale. The value of this argument can be abbreviated.
se.fit	logical switch indicating if standard errors are required.
dispersion	the dispersion of the GLM fit to be assumed in computing the standard errors. If omitted, that returned by summary applied to the object is used.
terms	with type="terms" by default all terms are returned. A character vector specifies which terms are to be returned
na.action	function determining what should be done with missing values in newdata. The default is to predict NA.
...	further arguments passed to or from other methods.

Details

If newdata is omitted the predictions are based on the data used for the fit. In that case how cases with missing values in the original fit is determined by the na.action argument of that fit. If na.action = na.omit omitted cases will not appear in the residuals, whereas if na.action = na.exclude they will appear (in predictions and standard errors), with residual value NA. See also napredict.

Value

If `se = FALSE`, a vector or matrix of predictions. If `se = TRUE`, a list with components

`fit`	Predictions
`se.fit`	Estimated standard errors
`residual.scale`	A scalar giving the square root of the dispersion used in computing the standard errors.

Note

Variables are first looked for in `newdata` and then searched for in the usual way (which will include the environment of the formula used in the fit). A warning will be given if the variables found are not of the same length as those in `newdata` if it was supplied.

See Also

`glm`, `SafePrediction`

Examples

```
## example from Venables and Ripley (2002, pp. 190-2.)
ldose <- rep(0:5, 2)
numdead <- c(1, 4, 9, 13, 18, 20, 0, 2, 6, 10, 12, 16)
sex <- factor(rep(c("M", "F"), c(6, 6)))
SF <- cbind(numdead, numalive=20-numdead)
budworm.lg <- glm(SF ~ sex*ldose, family=binomial)
summary(budworm.lg)

plot(c(1,32), c(0,1), type = "n", xlab = "dose",
     ylab = "prob", log = "x")
text(2^ldose, numdead/20, as.character(sex))
ld <- seq(0, 5, 0.1)
lines(2^ld, predict(budworm.lg, data.frame(ldose=ld,
   sex=factor(rep("M", length(ld)), levels=levels(sex))),
   type = "response"))
lines(2^ld, predict(budworm.lg, data.frame(ldose=ld,
   sex=factor(rep("F", length(ld)), levels=levels(sex))),
   type = "response"))
```

`predict.HoltWinters` *prediction function for fitted Holt-Winters models*

Description

Computes predictions and prediction intervals for models fitted by the Holt-Winters method.

Usage

```
## S3 method for class 'HoltWinters':
predict(object, n.ahead=1, prediction.interval = FALSE,
        level = 0.95, ...)
```

Arguments

object	An object of class HoltWinters.
n.ahead	Number of future periods to predict.
prediction.interval	logical. If TRUE, the lower and upper bounds of the corresponding prediction intervals are computed.
level	Confidence level for the prediction interval.
...	arguments passed to or from other methods.

Value

A time series of the predicted values. If prediction intervals are requested, a multiple time series is returned with columns fit, lwr and upr for the predicted values and the lower and upper bounds respectively.

Author(s)

David Meyer ⟨David.Meyer@wu-wien.ac.at⟩

References

C. C. Holt (1957) Forecasting seasonals and trends by exponentially weighted moving averages, ONR Research Memorandum, Carnigie Institute 52.

P. R. Winters (1960) Forecasting sales by exponentially weighted moving averages, *Management Science* **6**, 324–342.

See Also

HoltWinters

Examples

```
m <- HoltWinters(co2)
p <- predict(m, 50, prediction.interval = TRUE)
plot(m, p)
```

---

predict.lm *Predict method for Linear Model Fits*

Description

Predicted values based on linear model object.

Usage

```
## S3 method for class 'lm':
predict(object, newdata, se.fit = FALSE, scale = NULL, df = Inf,
        interval = c("none", "confidence", "prediction"),
        level = 0.95, type = c("response", "terms"),
        terms = NULL, na.action = na.pass, pred.var = res.var/weights,
        weights = 1, ...)
```

Arguments

object	Object of class inheriting from "lm"
newdata	An optional data frame in which to look for variables with which to predict. If omitted, the fitted values are used.
se.fit	A switch indicating if standard errors are required.
scale	Scale parameter for std.err. calculation
df	Degrees of freedom for scale
interval	Type of interval calculation.
level	Tolerance/confidence level
type	Type of prediction (response or model term).
terms	If type="terms", which terms (default is all terms)
na.action	function determining what should be done with missing values in newdata. The default is to predict NA.
pred.var	the variance(s) for future observations to be assumed for prediction intervals. See Details.
weights	variance weights for prediction. This can be a numeric vector or a one-sided model formula. In the latter case, it is interpreted as an expression evaluated in newdata
...	further arguments passed to or from other methods.

Details

predict.lm produces predicted values, obtained by evaluating the regression function in the frame newdata (which defaults to model.frame(object). If the logical se.fit is TRUE, standard errors of the predictions are calculated. If the numeric argument scale is set (with optional df), it is used as the residual standard deviation in the computation of the standard errors, otherwise this is extracted from the model fit. Setting intervals specifies computation of confidence or prediction (tolerance) intervals at the specified level, sometimes referred to as narrow vs. wide intervals.

If the fit is rank-deficient, some of the columns of the design matrix will have been dropped. Prediction from such a fit only makes sense if newdata is contained in the same subspace as the original data. That cannot be checked accurately, so a warning is issued.

If newdata is omitted the predictions are based on the data used for the fit. In that case how cases with missing values in the original fit is determined by the na.action argument of that fit. If na.action = na.omit omitted cases will not appear in the residuals, whereas if na.action = na.exclude they will appear (in predictions, standard errors or interval limits), with residual value NA. See also napredict.

The prediction intervals are for a single observation at each case in newdata (or by default, the data used for the fit) with error variance(s) pred.var. This can be a multiple of res.var, the estimated value of σ^2: the default is to assume that future observations have the same error variance as those used for fitting. If weights is supplied, the inverse of this is used as a scale factor. For a weighted fit, if the prediction is for the original data frame, weights defaults to the weights used for the model fit, with a warning since it might not be the intended result. If the fit was weighted and newdata is given, the default is to assume constant prediction variance, with a warning.

Value

`predict.lm` produces a vector of predictions or a matrix of predictions and bounds with column names `fit`, `lwr`, and `upr` if `interval` is set. If `se.fit` is `TRUE`, a list with the following components is returned:

`fit`	vector or matrix as above
`se.fit`	standard error of predicted means
`residual.scale`	residual standard deviations
`df`	degrees of freedom for residual

Note

Variables are first looked for in `newdata` and then searched for in the usual way (which will include the environment of the formula used in the fit). A warning will be given if the variables found are not of the same length as those in `newdata` if it was supplied.

Offsets specified by `offset` in the fit by `lm` will not be included in predictions, whereas those specified by an offset term in the formula will be.

Notice that prediction variances and prediction intervals always refer to *future* observations, possibly corresponding to the same predictors as used for the fit. The variance of the *residuals* will be smaller.

Strictly speaking, the formula used for prediction limits assumes that the degrees of freedom for the fit are the same as those for the residual variance. This may not be the case if `res.var` is not obtained from the fit.

See Also

The model fitting function `lm`, `predict`, `SafePrediction`

Examples

```
## Predictions
x <- rnorm(15)
y <- x + rnorm(15)
predict(lm(y ~ x))
new <- data.frame(x = seq(-3, 3, 0.5))
predict(lm(y ~ x), new, se.fit = TRUE)
pred.w.plim <- predict(lm(y ~ x), new, interval="prediction")
pred.w.clim <- predict(lm(y ~ x), new, interval="confidence")
matplot(new$x,cbind(pred.w.clim, pred.w.plim[,-1]),
        lty=c(1,2,2,3,3), type="l", ylab="predicted y")

## Prediction intervals, special cases
##  The first three of these throw warnings
w <- 1 + x^2
fit <- lm(y ~ x)
wfit <- lm(y ~ x, weights = w)
predict(fit, interval = "prediction")
predict(wfit, interval = "prediction")
predict(wfit, new, interval = "prediction")
predict(wfit, new, interval = "prediction", weights = (new$x)^2)
predict(wfit, new, interval = "prediction", weights = ~x^2)
```

`predict.loess` *Predict Loess Curve or Surface*

Description

Predictions from a `loess` fit, optionally with standard errors.

Usage

```
## S3 method for class 'loess':
predict(object, newdata = NULL, se = FALSE, ...)
```

Arguments

`object`	an object fitted by `loess`.
`newdata`	an optional data frame in which to look for variables with which to predict. If missing, the original data points are used.
`se`	should standard errors be computed?
`...`	arguments passed to or from other methods.

Details

The standard errors calculation is slower than prediction.

When the fit was made using `surface="interpolate"` (the default), `predict.loess` will not extrapolate – so points outside an axis-aligned hypercube enclosing the original data will have missing (`NA`) predictions and standard errors.

Value

If `se = FALSE`, a vector giving the prediction for each row of `newdata` (or the original data). If `se = TRUE`, a list containing components

`fit`	the predicted values.
`se`	an estimated standard error for each predicted value.
`residual.scale`	the estimated scale of the residuals used in computing the standard errors.
`df`	an estimate of the effective degrees of freedom used in estimating the residual scale, intended for use with t-based confidence intervals.

If `newdata` was the result of a call to `expand.grid`, the predictions (and s.e.'s if requested) will be an array of the appropriate dimensions.

Note

Variables are first looked for in `newdata` and then searched for in the usual way (which will include the environment of the formula used in the fit). A warning will be given if the variables found are not of the same length as those in `newdata` if it was supplied.

Author(s)

B.D. Ripley, based on the `cloess` package of Cleveland, Grosse and Shyu.

See Also

loess

Examples

```
cars.lo <- loess(dist ~ speed, cars)
predict(cars.lo, data.frame(speed=seq(5, 30, 1)), se=TRUE)
# to get extrapolation
cars.lo2 <- loess(dist ~ speed, cars,
  control=loess.control(surface="direct"))
predict(cars.lo2, data.frame(speed=seq(5, 30, 1)), se=TRUE)
```

---

predict.nls *Predicting from Nonlinear Least Squares Fits*

Description

predict.nls produces predicted values, obtained by evaluating the regression function in the frame newdata. If the logical se.fit is TRUE, standard errors of the predictions are calculated. If the numeric argument scale is set (with optional df), it is used as the residual standard deviation in the computation of the standard errors, otherwise this is extracted from the model fit. Setting intervals specifies computation of confidence or prediction (tolerance) intervals at the specified level.

At present se.fit and interval are ignored.

Usage

```
## S3 method for class 'nls':
predict(object, newdata , se.fit = FALSE, scale = NULL, df = Inf,
        interval = c("none", "confidence", "prediction"),
        level = 0.95, ...)
```

Arguments

object	An object that inherits from class nls.
newdata	A named list or data frame in which to look for variables with which to predict. If newdata is missing the fitted values at the original data points are returned.
se.fit	A logical value indicating if the standard errors of the predictions should be calculated. Defaults to FALSE. At present this argument is ignored.
scale	A numeric scalar. If it is set (with optional df), it is used as the residual standard deviation in the computation of the standard errors, otherwise this information is extracted from the model fit. At present this argument is ignored.
df	A positive numeric scalar giving the number of degrees of freedom for the scale estimate. At present this argument is ignored.
interval	A character string indicating if prediction intervals or a confidence interval on the mean responses are to be calculated. At present this argument is ignored.
level	A numeric scalar between 0 and 1 giving the confidence level for the intervals (if any) to be calculated. At present this argument is ignored.
...	Additional optional arguments. At present no optional arguments are used.

Value

`predict.nls` produces a vector of predictions. When implemented, `interval` will produce a matrix of predictions and bounds with column names `fit`, `lwr`, and `upr`. When implemented, if `se.fit` is `TRUE`, a list with the following components will be returned:

`fit`	vector or matrix as above
`se.fit`	standard error of predictions
`residual.scale`	residual standard deviations
`df`	degrees of freedom for residual

Note

Variables are first looked for in `newdata` and then searched for in the usual way (which will include the environment of the formula used in the fit). A warning will be given if the variables found are not of the same length as those in `newdata` if it was supplied.

See Also

The model fitting function `nls`, `predict`.

Examples

```
fm <- nls(demand ~ SSasympOrig(Time, A, lrc), data = BOD)
predict(fm)              # fitted values at observed times
## Form data plot and smooth line for the predictions
opar <- par(las = 1)
plot(demand ~ Time, data = BOD, col = 4,
     main = "BOD data and fitted first-order curve",
     xlim = c(0,7), ylim = c(0, 20) )
tt <- seq(0, 8, length = 101)
lines(tt, predict(fm, list(Time = tt)))
par(opar)
```

---

predict.smooth.spline

Predict from Smoothing Spline Fit

Description

Predict a smoothing spline fit at new points, return the derivative if desired. The predicted fit is linear beyond the original data.

Usage

```
## S3 method for class 'smooth.spline':
predict(object, x, deriv = 0, ...)
```

Arguments

`object`	a fit from `smooth.spline`.
`x`	the new values of x.
`deriv`	integer; the order of the derivative required.
`...`	further arguments passed to or from other methods.

Value

A list with components

`x`	The input `x`.
`y`	The fitted values or derivatives at `x`.

See Also

`smooth.spline`

Examples

```
attach(cars)
cars.spl <- smooth.spline(speed, dist, df=6.4)


## "Proof" that the derivatives are okay, by comparing with approximation
diff.quot <- function(x,y) {
  ## Difference quotient (central differences where available)
  n <- length(x); i1 <- 1:2; i2 <- (n-1):n
  c(diff(y[i1]) / diff(x[i1]), (y[-i1] - y[-i2]) / (x[-i1] - x[-i2]),
    diff(y[i2]) / diff(x[i2]))
}

xx <- unique(sort(c(seq(0,30, by = .2), kn <- unique(speed))))
i.kn <- match(kn, xx)# indices of knots within xx
op <- par(mfrow = c(2,2))
plot(speed, dist, xlim = range(xx), main = "Smooth.spline & derivatives")
lines(pp <- predict(cars.spl, xx), col = "red")
points(kn, pp$y[i.kn], pch = 3, col="dark red")
mtext("s(x)", col = "red")
for(d in 1:3){
  n <- length(pp$x)
  plot(pp$x, diff.quot(pp$x,pp$y), type = 'l', xlab="x", ylab="",
       col = "blue", col.main = "red",
       main= paste("s",paste(rep("'",d), collapse=""),"(x)", sep=""))
  mtext("Difference quotient approx.(last)", col = "blue")
  lines(pp <- predict(cars.spl, xx, deriv = d), col = "red")

  points(kn, pp$y[i.kn], pch = 3, col="dark red")
  abline(h=0, lty = 3, col = "gray")
}
detach(); par(op)
```

preplot *Pre-computations for a Plotting Objeect*

Description

Compute an object to be used for plots relating to the given model object.

Usage

```
preplot(object, ...)
```

Arguments

object	a fitted model object.
...	additional arguments for specific methods.

Details

Only the generic function is currently provided in base R, but some add-on packages have methods. Principally here for S compatibility.

Value

An object set up to make a plot that describes object.

princomp *Principal Components Analysis*

Description

princomp performs a principal components analysis on the given numeric data matrix and returns the results as an object of class princomp.

Usage

```
princomp(x, ...)

## S3 method for class 'formula':
princomp(formula, data = NULL, subset, na.action, ...)

## Default S3 method:
princomp(x, cor = FALSE, scores = TRUE, covmat = NULL,
         subset = rep(TRUE, nrow(as.matrix(x))), ...)

## S3 method for class 'princomp':
predict(object, newdata, ...)
```

Arguments

formula	a formula with no response variable, referring only to numeric variables.
data	an optional data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).
subset	an optional vector used to select rows (observations) of the data matrix x.
na.action	a function which indicates what should happen when the data contain NAs. The default is set by the na.action setting of options, and is na.fail if that is unset. The "factory-fresh" default is na.omit.
x	a numeric matrix or data frame which provides the data for the principal components analysis.
cor	a logical value indicating whether the calculation should use the correlation matrix or the covariance matrix. (The correlation matrix can only be used if there are no constant variables.)
scores	a logical value indicating whether the score on each principal component should be calculated.
covmat	a covariance matrix, or a covariance list as returned by cov.wt (and cov.mve or cov.mcd from package **MASS**). If supplied, this is used rather than the covariance matrix of x.
...	arguments passed to or from other methods. If x is a formula one might specify cor or scores.
object	Object of class inheriting from "princomp"
newdata	An optional data frame or matrix in which to look for variables with which to predict. If omitted, the scores are used. If the original fit used a formula or a data frame or a matrix with column names, newdata must contain columns with the same names. Otherwise it must contain the same number of columns, to be used in the same order.

Details

princomp is a generic function with "formula" and "default" methods.

The calculation is done using eigen on the correlation or covariance matrix, as determined by cor. This is done for compatibility with the S-PLUS result. A preferred method of calculation is to use svd on x, as is done in prcomp.

Note that the default calculation uses divisor N for the covariance matrix.

The print method for these objects prints the results in a nice format and the plot method produces a scree plot (screeplot). There is also a biplot method.

If x is a formula then the standard NA-handling is applied to the scores (if requested): see napredict.

princomp only handles so-called R-mode PCA, that is feature extraction of variables. If a data matrix is supplied (possibly via a formula) it is required that there are at least as many units as variables. For Q-mode PCA use prcomp.

Value

princomp returns a list with class "princomp" containing the following components:

sdev	the standard deviations of the principal components.

loadings	the matrix of variable loadings (i.e., a matrix whose columns contain the eigenvectors). This is of class "loadings": see loadings for its print method.
center	the means that were subtracted.
scale	the scalings applied to each variable.
n.obs	the number of observations.
scores	if scores = TRUE, the scores of the supplied data on the principal components. These are non-null only if x was supplied, and if covmat was also supplied if it was a covariance list. For the formula method, napredict is applied to handle the treatment of values omitted by the na.action.
call	the matched call.
na.action	If relevant.

Note

The signs of the columns of the loadings and scores are arbitrary, and so may differ between different programs for PCA, and even between different builds of R.

References

Mardia, K. V., J. T. Kent and J. M. Bibby (1979). *Multivariate Analysis*, London: Academic Press.

Venables, W. N. and B. D. Ripley (2002). *Modern Applied Statistics with S*, Springer-Verlag.

See Also

summary.princomp, screeplot, biplot.princomp, prcomp, cor, cov, eigen.

Examples

```
## The variances of the variables in the
## USArrests data vary by orders of magnitude, so scaling is appropriate
(pc.cr <- princomp(USArrests))  # inappropriate
princomp(USArrests, cor = TRUE) # =^= prcomp(USArrests, scale=TRUE)
## Similar, but different:
## The standard deviations differ by a factor of sqrt(49/50)

summary(pc.cr <- princomp(USArrests, cor = TRUE))
loadings(pc.cr)  ## note that blank entries are small but not zero
plot(pc.cr) # shows a screeplot.
biplot(pc.cr)

## Formula interface
princomp(~ ., data = USArrests, cor = TRUE)
# NA-handling
USArrests[1, 2] <- NA
pc.cr <- princomp(~ Murder + Assault + UrbanPop,
                  data = USArrests, na.action=na.exclude, cor = TRUE)
pc.cr$scores
```

print.power.htest *Print method for power calculation object*

Description

Print object of class "power.htest" in nice layout.

Usage

```
## S3 method for class 'power.htest':
print(x, ...)
```

Arguments

x	Object of class "power.htest".
...	further arguments to be passed to or from methods.

Details

A power.htest object is just a named list of numbers and character strings, supplemented with method and note elements. The method is displayed as a title, the note as a footnote, and the remaining elements are given in an aligned 'name = value' format.

Value

none

Author(s)

Peter Dalgaard

See Also

power.t.test, power.prop.test

print.ts *Printing Time-Series Objects*

Description

Print method for time series objects.

Usage

```
## S3 method for class 'ts':
print(x, calendar, ...)
```

Arguments

x	a time series object.
calendar	enable/disable the display of information about month names, quarter names or year when printing. The default is TRUE for a frequency of 4 or 12, FALSE otherwise.
...	additional arguments to print.

Details

This is the print methods for objects inheriting from class "ts".

See Also

print, ts.

Examples

```
print(ts(1:10, freq = 7, start = c(12, 2)), calendar = TRUE)
```

---

printCoefmat *Print Coefficient Matrices*

Description

Utility function to be used in "higher level" print methods, such as print.summary.lm, print.summary.glm and print.anova. The goal is to provide a flexible interface with smart defaults such that often, only x needs to be specified.

Usage

```
printCoefmat(x, digits=max(3, getOption("digits") - 2),
             signif.stars = getOption("show.signif.stars"),
             signif.legend = signif.stars,
             dig.tst = max(1, min(5, digits - 1)),
             cs.ind = 1:k, tst.ind = k + 1, zap.ind = integer(0),
             P.values = NULL,
             has.Pvalue = nc >= 4 &&
                          substr(colnames(x)[nc],1,3) == "Pr(",
             eps.Pvalue = .Machine$double.eps,
             na.print = "NA", ...)
```

Arguments

x	a numeric matrix like object, to be printed.
digits	minimum number of significant digits to be used for most numbers.
signif.stars	logical; if TRUE, P-values are additionally encoded visually as "significance stars" in order to help scanning of long coefficient tables. It defaults to the show.signif.stars slot of options.

`signif.legend` logical; if `TRUE`, a legend for the "significance stars" is printed provided `signif.stars=TRUE`.

`dig.tst` minimum number of significant digits for the test statistics, see `tst.ind`.

`cs.ind` indices (integer) of column numbers which are (like) **c**oefficients and **s**tandard errors to be formatted together.

`tst.ind` indices (integer) of column numbers for test statistics.

`zap.ind` indices (integer) of column numbers which should be formatted by `zapsmall`, i.e., by "zapping" values close to 0.

`P.values` logical or `NULL`; if `TRUE`, the last column of `x` is formatted by `format.pval` as P values. If `P.values = NULL`, the default, it is set to `TRUE` only if `options("show.coef.Pvalue")` is `TRUE` *and* `x` has at least 4 columns *and* the last column name of `x` starts with `"Pr("`.

`has.Pvalue` logical; if `TRUE`, the last column of `x` contains P values; in that case, it is printed if and only if `P.values` (above) is true.

`eps.Pvalue` number,..

`na.print` a character string to code `NA` values in printed output.

`...` further arguments for `print`.

Value

Invisibly returns its argument, `x`.

Author(s)

Martin Maechler

See Also

`print.summary.lm`, `format.pval`, `format`.

Examples

```
cmat <- cbind(rnorm(3, 10), sqrt(rchisq(3, 12)))
cmat <- cbind(cmat, cmat[,1]/cmat[,2])
cmat <- cbind(cmat, 2*pnorm(-cmat[,3]))
colnames(cmat) <- c("Estimate", "Std.Err", "Z value", "Pr(>z)")
printCoefmat(cmat[,1:3])
printCoefmat(cmat)
options(show.coef.Pvalues = FALSE)
printCoefmat(cmat, digits=2)
printCoefmat(cmat, digits=2, P.values = TRUE)
options(show.coef.Pvalues = TRUE)# revert
```

profile *Generic Function for Profiling Models*

Description

Investigates behavior of objective function near the solution represented by `fitted`.

See documentation on method functions for further details.

Usage

```
profile(fitted, ...)
```

Arguments

`fitted`	the original fitted model object.
`...`	additional parameters. See documentation on individual methods.

Value

A list with an element for each parameter being profiled. See the individual methods for further details.

See Also

`profile.nls`, `profile.glm` in package **MASS**, ...

For profiling code, see `Rprof`.

profile.nls *Method for Profiling nls Objects*

Description

Investigates the profilelog-likelihood function for a fitted model of class `"nls"`.

Usage

```
## S3 method for class 'nls':
profile(fitted, which = 1:npar, maxpts = 100, alphamax = 0.01,
        delta.t = cutoff/5, ...)
```

Arguments

`fitted`	the original fitted model object.
`which`	the original model parameters which should be profiled. This can be a numeric or character vector. By default, all non-linear parameters are profiled.
`maxpts`	maximum number of points to be used for profiling each parameter.
`alphamax`	maximum significance level allowed for the profile t-statistics.
`delta.t`	suggested change on the scale of the profile t-statistics. Default value chosen to allow profiling at about 10 parameter values.
`...`	further arguments passed to or from other methods.

Details

The profile t-statistics is defined as the square root of change in sum-of-squares divided by residual standard error with an appropriate sign.

Value

A list with an element for each parameter being profiled. The elements are data-frames with two variables

`par.vals`	a matrix of parameter values for each fitted model.
`tau`	the profile t-statistics.

Author(s)

Of the original version, Douglas M. Bates and Saikat DebRoy

References

Bates, D. M. and Watts, D. G. (1988), *Nonlinear Regression Analysis and Its Applications*, Wiley (chapter 6).

See Also

`nls`, `profile`, `plot.profile.nls`

Examples

```
# obtain the fitted object
fm1 <- nls(demand ~ SSasympOrig( Time, A, lrc ), data = BOD)
# get the profile for the fitted model
pr1 <- profile( fm1 )
# profiled values for the two parameters
pr1$A
pr1$lrc
```

`proj` *Projections of Models*

Description

`proj` returns a matrix or list of matrices giving the projections of the data onto the terms of a linear model. It is most frequently used for `aov` models.

Usage

```
proj(object, ...)

## S3 method for class 'aov':
proj(object, onedf = FALSE, unweighted.scale = FALSE, ...)

## S3 method for class 'aovlist':
proj(object, onedf = FALSE, unweighted.scale = FALSE, ...)
```

```
## Default S3 method:
proj(object, onedf = TRUE, ...)

## S3 method for class 'lm':
proj(object, onedf = FALSE, unweighted.scale = FALSE, ...)
```

Arguments

object	An object of class "lm" or a class inheriting from it, or an object with a similar structure including in particular components qr and effects.
onedf	A logical flag. If TRUE, a projection is returned for all the columns of the model matrix. If FALSE, the single-column projections are collapsed by terms of the model (as represented in the analysis of variance table).
unweighted.scale	If the fit producing object used weights, this determines if the projections correspond to weighted or unweighted observations.
...	Swallow and ignore any other arguments.

Details

A projection is given for each stratum of the object, so for aov models with an Error term the result is a list of projections.

Value

A projection matrix or (for multi-stratum objects) a list of projection matrices.

Each projection is a matrix with a row for each observations and either a column for each term (onedf = FALSE) or for each coefficient (onedf = TRUE). Projection matrices from the default method have orthogonal columns representing the projection of the response onto the column space of the Q matrix from the QR decomposition. The fitted values are the sum of the projections, and the sum of squares for each column is the reduction in sum of squares from fitting that column (after those to the left of it).

The methods for lm and aov models add a column to the projection matrix giving the residuals (the projection of the data onto the orthogonal complement of the model space).

Strictly, when onedf = FALSE the result is not a projection, but the columns represent sums of projections onto the columns of the model matrix corresponding to that term. In this case the matrix does not depend on the coding used.

Author(s)

The design was inspired by the S function of the same name described in Chambers *et al.* (1992).

References

Chambers, J. M., Freeny, A and Heiberger, R. M. (1992) *Analysis of variance; designed experiments.* Chapter 5 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

aov, lm, model.tables

Examples

```
N <- c(0,1,0,1,1,1,0,0,0,1,1,0,1,1,0,0,1,0,1,0,1,1,0,0)
P <- c(1,1,0,0,0,1,0,1,1,1,0,0,0,1,0,1,1,0,0,1,0,1,1,0)
K <- c(1,0,0,1,0,1,1,0,0,1,0,1,0,1,1,0,0,0,1,1,1,0,1,0)
yield <- c(49.5,62.8,46.8,57.0,59.8,58.5,55.5,56.0,62.8,55.8,69.5,
55.0, 62.0,48.8,45.5,44.2,52.0,51.5,49.8,48.8,57.2,59.0,53.2,56.0)

npk <- data.frame(block=gl(6,4), N=factor(N), P=factor(P),
                  K=factor(K), yield=yield)
npk.aov <- aov(yield ~ block + N*P*K, npk)
proj(npk.aov)

## as a test, not particularly sensible
options(contrasts=c("contr.helmert", "contr.treatment"))
npk.aovE <- aov(yield ~  N*P*K + Error(block), npk)
proj(npk.aovE)
```

---

prop.test *Test of Equal or Given Proportions*

Description

`prop.test` can be used for testing the null that the proportions (probabilities of success) in several groups are the same, or that they equal certain given values.

Usage

```
prop.test(x, n, p = NULL,
          alternative = c("two.sided", "less", "greater"),
          conf.level = 0.95, correct = TRUE)
```

Arguments

`x`	a vector of counts of successes or a matrix with 2 columns giving the counts of successes and failures, respectively.
`n`	a vector of counts of trials; ignored if `x` is a matrix.
`p`	a vector of probabilities of success. The length of `p` must be the same as the number of groups specified by `x`, and its elements must be greater than 0 and less than 1.
`alternative`	a character string specifying the alternative hypothesis, must be one of `"two.sided"` (default), `"greater"` or `"less"`. You can specify just the initial letter. Only used for testing the null that a single proportion equals a given value, or that two proportions are equal; ignored otherwise.
`conf.level`	confidence level of the returned confidence interval. Must be a single number between 0 and 1. Only used when testing the null that a single proportion equals a given value, or that two proportions are equal; ignored otherwise.
`correct`	a logical indicating whether Yates' continuity correction should be applied.

Details

Only groups with finite numbers of successes and failures are used. Counts of successes and failures must be nonnegative and hence not greater than the corresponding numbers of trials which must be positive. All finite counts should be integers.

If `p` is `NULL` and there is more than one group, the null tested is that the proportions in each group are the same. If there are two groups, the alternatives are that the probability of success in the first group is less than, not equal to, or greater than the probability of success in the second group, as specified by `alternative`. A confidence interval for the difference of proportions with confidence level as specified by `conf.level` and clipped to $[-1, 1]$ is returned. Continuity correction is used only if it does not exceed the difference of the sample proportions in absolute value. Otherwise, if there are more than 2 groups, the alternative is always `"two.sided"`, the returned confidence interval is `NULL`, and continuity correction is never used.

If there is only one group, then the null tested is that the underlying probability of success is `p`, or .5 if `p` is not given. The alternative is that the probability of success if less than, not equal to, or greater than `p` or 0.5, respectively, as specified by `alternative`. A confidence interval for the underlying proportion with confidence level as specified by `conf.level` and clipped to $[0, 1]$ is returned. Continuity correction is used only if it does not exceed the difference between sample and null proportions in absolute value. The confidence interval is computed by inverting the score test.

Finally, if `p` is given and there are more than 2 groups, the null tested is that the underlying probabilities of success are those given by `p`. The alternative is always `"two.sided"`, the returned confidence interval is `NULL`, and continuity correction is never used.

Value

A list with class `"htest"` containing the following components:

`statistic`	the value of Pearson's chi-squared test statistic.
`parameter`	the degrees of freedom of the approximate chi-squared distribution of the test statistic.
`p.value`	the p-value of the test.
`estimate`	a vector with the sample proportions `x/n`.
`conf.int`	a confidence interval for the true proportion if there is one group, or for the difference in proportions if there are 2 groups and `p` is not given, or `NULL` otherwise. In the cases where it is not `NULL`, the returned confidence interval has an asymptotic confidence level as specified by `conf.level`, and is appropriate to the specified alternative hypothesis.
`null.value`	the value of `p` if specified by the null, or `NULL` otherwise.
`alternative`	a character string describing the alternative.
`method`	a character string indicating the method used, and whether Yates' continuity correction was applied.
`data.name`	a character string giving the names of the data.

See Also

`binom.test` for an *exact* test of a binomial hypothesis.

Examples

```
heads <- rbinom(1, size=100, pr = .5)
prop.test(heads, 100)          # continuity correction TRUE by default
prop.test(heads, 100, correct = FALSE)

## Data from Fleiss (1981), p. 139.
## H0: The null hypothesis is that the four populations from which
##     the patients were drawn have the same true proportion of smokers.
## A:  The alternative is that this proportion is different in at
##     least one of the populations.

smokers  <- c( 83, 90, 129, 70 )
patients <- c( 86, 93, 136, 82 )
prop.test(smokers, patients)
```

---

prop.trend.test *Test for trend in proportions*

Description

Performs chi-squared test for trend in proportions, i.e., a test asymptotically optimal for local alternatives where the log odds vary in proportion with `score`. By default, `score` is chosen as the group numbers.

Usage

```
prop.trend.test(x, n, score = 1:length(x))
```

Arguments

`x`	Number of events
`n`	Number of trials
`score`	Group score

Value

An object of class `"htest"` with title, test statistic, p-value, etc.

Note

This really should get integrated with `prop.test`

Author(s)

Peter Dalgaard

See Also

`prop.test`

Examples

```
smokers  <- c( 83, 90, 129, 70 )
patients <- c( 86, 93, 136, 82 )
prop.test(smokers, patients)
prop.trend.test(smokers, patients)
prop.trend.test(smokers, patients,c(0,0,0,1))
```

---

qqnorm *Quantile-Quantile Plots*

Description

`qqnorm` is a generic function the default method of which produces a normal QQ plot of the values in `y`. `qqline` adds a line to a normal quantile-quantile plot which passes through the first and third quartiles.

`qqplot` produces a QQ plot of two datasets.

Graphical parameters may be given as arguments to `qqnorm`, `qqplot` and `qqline`.

Usage

```
qqnorm(y, ...)
## Default S3 method:
qqnorm(y, ylim, main = "Normal Q-Q Plot",
       xlab = "Theoretical Quantiles", ylab = "Sample Quantiles",
       plot.it = TRUE, datax = FALSE, ...)

qqline(y, datax = FALSE, ...)

qqplot(x, y, plot.it = TRUE, xlab = deparse(substitute(x)),
       ylab = deparse(substitute(y)), ...)
```

Arguments

`x`	The first sample for `qqplot`.
`y`	The second or only data sample.
`xlab, ylab, main`	plot labels. The `xlab` and `ylab` refer to the y and x axes respectively if `datax = TRUE`.
`plot.it`	logical. Should the result be plotted?
`datax`	logical. Should data values be on the x-axis?
`ylim, ...`	graphical parameters.

Value

For `qqnorm` and `qqplot`, a list with components

`x`	The x coordinates of the points that were/would be plotted
`y`	The original `y` vector, i.e., the corresponding y coordinates *including* `NA`*s*.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

ppoints, used by qqnorm to generate approximations to expected order statistics for a normal distribution.

Examples

```
y <- rt(200, df = 5)
qqnorm(y); qqline(y, col = 2)
qqplot(y, rt(300, df = 5))

qqnorm(precip, ylab = "Precipitation [in/yr] for 70 US cities")
```

---

quade.test *Quade Test*

Description

Performs a Quade test with unreplicated blocked data.

Usage

```
quade.test(y, ...)

## Default S3 method:
quade.test(y, groups, blocks, ...)

## S3 method for class 'formula':
quade.test(formula, data, subset, na.action, ...)
```

Arguments

y	either a numeric vector of data values, or a data matrix.
groups	a vector giving the group for the corresponding elements of y if this is a vector; ignored if y is a matrix. If not a factor object, it is coerced to one.
blocks	a vector giving the block for the corresponding elements of y if this is a vector; ignored if y is a matrix. If not a factor object, it is coerced to one.
formula	a formula of the form a ~ b | c, where a, b and c give the data values and corresponding groups and blocks, respectively.
data	an optional matrix or data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).
subset	an optional vector specifying a subset of observations to be used.
na.action	a function which indicates what should happen when the data contain NAs. Defaults to getOption("na.action").
...	further arguments to be passed to or from methods.

Details

`quade.test` can be used for analyzing unreplicated complete block designs (i.e., there is exactly one observation in `y` for each combination of levels of `groups` and `blocks`) where the normality assumption may be violated.

The null hypothesis is that apart from an effect of `blocks`, the location parameter of `y` is the same in each of the `groups`.

If `y` is a matrix, `groups` and `blocks` are obtained from the column and row indices, respectively. `NA`'s are not allowed in `groups` or `blocks`; if `y` contains `NA`'s, corresponding blocks are removed.

Value

A list with class `"htest"` containing the following components:

`statistic`	the value of Quade's F statistic.
`parameter`	a vector with the numerator and denominator degrees of freedom of the approximate F distribution of the test statistic.
`p.value`	the p-value of the test.
`method`	the character string `"Quade test"`.
`data.name`	a character string giving the names of the data.

References

D. Quade (1979), Using weighted rankings in the analysis of complete blocks with additive block effects. *Journal of the American Statistical Association*, **74**, 680–683.

William J. Conover (1999), *Practical nonparametric statistics*. New York: John Wiley & Sons. Pages 373–380.

See Also

`friedman.test`.

Examples

```
## Conover (1999, p. 375f):
## Numbers of five brands of a new hand lotion sold in seven stores
## during one week.
y <- matrix(c( 5,  4,  7, 10, 12,
               1,  3,  1,  0,  2,
              16, 12, 22, 22, 35,
               5,  4,  3,  5,  4,
              10,  9,  7, 13, 10,
              19, 18, 28, 37, 58,
              10,  7,  6,  8,  7),
            nr = 7, byrow = TRUE,
            dimnames =
            list(Store = as.character(1:7),
                 Brand = LETTERS[1:5]))
y
quade.test(y)
```

`quantile` *Sample Quantiles*

Description

The generic function `quantile` produces sample quantiles corresponding to the given probabilities. The smallest observation corresponds to a probability of 0 and the largest to a probability of 1.

Usage

```
quantile(x, ...)

## Default S3 method:
quantile(x, probs = seq(0, 1, 0.25), na.rm = FALSE,
         names = TRUE, type = 7, ...)
```

Arguments

`x`	numeric vectors whose sample quantiles are wanted. Missing values are ignored.
`probs`	numeric vector of probabilities with values in $[0, 1]$.
`na.rm`	logical; if true, any `NA` and `NaN`'s are removed from `x` before the quantiles are computed.
`names`	logical; if true, the result has a `names` attribute. Set to `FALSE` for speedup with many `probs`.
`type`	an integer between 1 and 9 selecting one of the nine quantile algorithms detailed below to be used.
`...`	further arguments passed to or from other methods.

Details

A vector of length `length(probs)` is returned; if `names = TRUE`, it has a `names` attribute.

`NA` and `NaN` values in `probs` are propagated to the result.

Types

`quantile` returns estimates of underlying distribution quantiles based on one or two order statistics from the supplied elements in `x` at probabilities in `probs`. One of the nine quantile algorithms discussed in Hyndman and Fan (1996), selected by `type`, is employed.

Sample quantiles of type i are defined by

$$Q_i(p) = (1 - \gamma)x_j + \gamma x_{j+1}$$

where $1 \leq i \leq 9$, $\frac{j-m}{n} \leq p < \frac{j-m+1}{n}$, x_j is the jth order statistic, n is the sample size, and m is a constant determined by the sample quantile type. Here γ depends on the fractional part of $g = np + m - j$.

For the continuous sample quantile types (4 through 9), the sample quantiles can be obtained by linear interpolation between the kth order statistic and $p(k)$:

$$p(k) = \frac{k - \alpha}{n - \alpha - \beta + 1}$$

where α and β are constants determined by the type. Further, $m = \alpha + p\,(1 - \alpha - \beta)$, and $\gamma = g$.

Discontinuous sample quantile types 1, 2, and 3

Type 1 Inverse of empirical distribution function.

Type 2 Similar to type 1 but with averaging at discontinuities.

Type 3 SAS definition: nearest even order statistic.

Continuous sample quantile types 4 through 9

Type 4 $p(k) = \frac{k}{n}$. That is, linear interpolation of the empirical cdf.

Type 5 $p(k) = \frac{k-0.5}{n}$. That is a piecewise linear function where the knots are the values midway through the steps of the empirical cdf. This is popular amongst hydrologists.

Type 6 $p(k) = \frac{k}{n+1}$. Thus $p(k) = \mathrm{E}[F(x_k)]$. This is used by Minitab and by SPSS.

Type 7 $p(k) = \frac{k-1}{n-1}$. In this case, $p(k) = \mathrm{mode}[F(x_k)]$. This is used by S.

Type 8 $p(k) = \frac{k-\frac{1}{3}}{n+\frac{1}{3}}$. Then $p(k) \approx \mathrm{median}[F(x_k)]$. The resulting quantile estimates are approximately median-unbiased regardless of the distribution of `x`.

Type 9 $p(k) = \frac{k-\frac{3}{8}}{n+\frac{1}{4}}$. The resulting quantile estimates are approximately unbiased for the expected order statistics if `x` is normally distributed.

Hyndman and Fan (1996) recommend type 8. The default method is type 7, as used by S and by R < 2.0.0.

Author(s)

of the version used in R >= 2.0.0, Ivan Frohne and Rob J Hyndman.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Hyndman, R. J. and Fan, Y. (1996) Sample quantiles in statistical packages, *American Statistician*, **50**, 361–365.

See Also

`ecdf` for empirical distributions of which `quantile` is the "inverse"; `boxplot.stats` and `fivenum` for computing "versions" of quartiles, etc.

Examples

```
quantile(x <- rnorm(1001))# Extremes & Quartiles by default
quantile(x,  probs=c(.1,.5,1,2,5,10,50, NA)/100)

### Compare different types
p <- c(0.1,0.5,1,2,5,10,50)/100
res <- matrix(as.numeric(NA), 9, 7)
for(type in 1:9) res[type, ] <- y <- quantile(x,  p, type=type)
dimnames(res) <- list(1:9, names(y))
round(res, 3)
```

`r2dtable` *Random 2-way Tables with Given Marginals*

Description

Generate random 2-way tables with given marginals using Patefield's algorithm.

Usage

```
r2dtable(n, r, c)
```

Arguments

`n`	a non-negative numeric giving the number of tables to be drawn.
`r`	a non-negative vector of length at least 2 giving the row totals, to be coerced to `integer`. Must sum to the same as `c`.
`c`	a non-negative vector of length at least 2 giving the column totals, to be coerced to `integer`.

Value

A list of length `n` containing the generated tables as its components.

References

Patefield, W. M. (1981) Algorithm AS159. An efficient method of generating r x c tables with given row and column totals. *Applied Statistics* **30**, 91–97.

Examples

```
## Fisher's Tea Drinker data.
TeaTasting <-
matrix(c(3, 1, 1, 3),
       nr = 2,
       dimnames = list(Guess = c("Milk", "Tea"),
                       Truth = c("Milk", "Tea")))
## Simulate permutation test for independence based on the maximum
## Pearson residuals (rather than their sum).
rowTotals <- rowSums(TeaTasting)
colTotals <- colSums(TeaTasting)
nOfCases <- sum(rowTotals)
expected <- outer(rowTotals, colTotals, "*") / nOfCases
maxSqResid <- function(x) max((x - expected) ^ 2 / expected)
simMaxSqResid <-
    sapply(r2dtable(1000, rowTotals, colTotals), maxSqResid)
sum(simMaxSqResid >= maxSqResid(TeaTasting)) / 1000
## Fisher's exact test gives p = 0.4857 ...
```

`read.ftable` *Manipulate Flat Contingency Tables*

Description

Read, write and coerce "flat" contingency tables.

Usage

```
read.ftable(file, sep = "", quote = "\"",
            row.var.names, col.vars, skip = 0)

write.ftable(x, file = "", quote = TRUE, append = FALSE,
             digits = getOption("digits"))

## S3 method for class 'ftable':
format(x, quote = TRUE, digits = getOption("digits"), ...)
```

Arguments

`file`	either a character string naming a file or a connection which the data are to be read from or written to. `""` indicates input from the console for reading and output to the console for writing.
`sep`	the field separator string. Values on each line of the file are separated by this string.
`quote`	a character string giving the set of quoting characters for `read.ftable`; to disable quoting altogether, use `quote=""`. For `write.table`, a logical indicating whether strings in the data will be surrounded by double quotes.
`row.var.names`	a character vector with the names of the row variables, in case these cannot be determined automatically.
`col.vars`	a list giving the names and levels of the column variables, in case these cannot be determined automatically.
`skip`	the number of lines of the data file to skip before beginning to read data.
`x`	an object of class `"ftable"`.
`append`	logical. If `TRUE` and `file` is the name of a file (and not a connection or `"|cmd"`), the output from `write.ftable` is appended to the file. If `FALSE`, the contents of `file` will be overwritten.
`digits`	an integer giving the number of significant digits to use for (the cell entries of) `x`.
`...`	further arguments to be passed to or from methods.

Details

`read.ftable` reads in a flat-like contingency table from a file. If the file contains the written representation of a flat table (more precisely, a header with all information on names and levels of column variables, followed by a line with the names of the row variables), no further arguments are needed. Similarly, flat tables with only one column variable the name of which is the only entry in the first line are handled automatically. Other variants can be dealt with by skipping all header

information using `skip`, and providing the names of the row variables and the names and levels of the column variable using `row.var.names` and `col.vars`, respectively. See the examples below.

Note that flat tables are characterized by their "ragged" display of row (and maybe also column) labels. If the full grid of levels of the row variables is given, one should instead use `read.table` to read in the data, and create the contingency table from this using `xtabs`.

`write.ftable` writes a flat table to a file, which is useful for generating "pretty" ASCII representations of contingency tables.

References

Agresti, A. (1990) *Categorical data analysis*. New York: Wiley.

See Also

`ftable` for more information on flat contingency tables.

Examples

```
## Agresti (1990), page 157, Table 5.8.
## Not in ftable standard format, but o.k.
file <- tempfile()
cat("             Intercourse\n",
    "Race  Gender     Yes  No\n",
    "White Male        43 134\n",
    "      Female      26 149\n",
    "Black Male        29  23\n",
    "      Female      22  36\n",
    file = file)
file.show(file)
ft <- read.ftable(file)
ft
unlink(file)

## Agresti (1990), page 297, Table 8.16.
## Almost o.k., but misses the name of the row variable.
file <- tempfile()
cat("                       \"Tonsil Size\"\n",
    "            \"Not Enl.\" \"Enl.\" \"Greatly Enl.\"\n",
    "Noncarriers       497     560           269\n",
    "Carriers           19      29            24\n",
    file = file)
file.show(file)
ft <- read.ftable(file, skip = 2,
                  row.var.names = "Status",
                  col.vars = list("Tonsil Size" =
                      c("Not Enl.", "Enl.", "Greatly Enl.")))
ft
unlink(file)

ft22 <- ftable(Titanic, row.vars = 2:1, col.vars = 4:3)
write.ftable(ft22, quote = FALSE)
```

`rect.hclust` *Draw Rectangles Around Hierarchical Clusters*

Description

Draws rectangles around the branches of a dendrogram highlighting the corresponding clusters. First the dendrogram is cut at a certain level, then a rectangle is drawn around selected branches.

Usage

```
rect.hclust(tree, k = NULL, which = NULL, x = NULL, h = NULL,
            border = 2, cluster = NULL)
```

Arguments

`tree`	an object of the type produced by `hclust`.
`k, h`	Scalar. Cut the dendrogram such that either exactly `k` clusters are produced or by cutting at height `h`.
`which, x`	A vector selecting the clusters around which a rectangle should be drawn. `which` seleccts clusters by number (from left to right in the tree), `x` selects clusters containing the respective horizontal coordinates. Default is `which = 1:k`.
`border`	Vector with border colors for the rectangles.
`cluster`	Optional vector with cluster memberships as returned by `cutree(hclust.obj, k = k)`, can be specified for efficiency if already computed.

Value

(Invisibly) returns a list where each element contains a vector of data points contained in the respective cluster.

See Also

`hclust`, `identify.hclust`.

Examples

```
hca <- hclust(dist(USArrests))
plot(hca)
rect.hclust(hca, k=3, border="red")
x <- rect.hclust(hca, h=50, which=c(2,7), border=3:4)
x
```

relevel *Reorder Levels of Factor*

Description

The levels of a factor are re-ordered so that the level specified by `ref` is first and the others are moved down. This is useful for `contr.treatment` contrasts which take the first level as the reference.

Usage

```
relevel(x, ref, ...)
```

Arguments

`x`	An unordered factor.
`ref`	The reference level.
`...`	Additional arguments for future methods.

Value

A factor of the same length as `x`.

See Also

`factor`, `contr.treatment`, `levels`, `reorder`.

Examples

```
warpbreaks$tension <- relevel(warpbreaks$tension, ref="M")
summary(lm(breaks ~ wool + tension, data=warpbreaks))
```

reorder.dendrogram *Reorder a Dendrogram*

Description

A method for the generic function `reorder`.

There are many different orderings of a dendrogram that are consistent with the structure imposed. This function takes a dendrogram and a vector of values and reorders the dendrogram in the order of the supplied vector, maintaining the constraints on the dendrogram.

Usage

```
## S3 method for class 'dendrogram':
reorder(x, wts, agglo.FUN = sum, ...)
```

Arguments

x	the (dendrogram) object to be reordered
wts	numeric weights (arbitrary values) for reordering.
agglo.FUN	a function for weights agglomeration, see below.
...	additional arguments

Details

Using the weights wts, the leaves of the dendrogram are reordered so as to be in an order as consistent as possible with the weights. At each node, the branches are ordered in increasing weights where the weight of a branch is defined as $f(w_j)$ where f is agglo.FUN and w_j is the weight of the j-th sub branch).

Value

A dendrogram where each node has a further attribute value with its corresponding weight.

Author(s)

R. Gentleman and M. Maechler

See Also

reorder.

rev.dendrogram which simply reverses the nodes' order; heatmap, cophenetic.

Examples

```
set.seed(123)
x <- rnorm(10)
hc <- hclust(dist(x))
dd <- as.dendrogram(hc)
dd.reorder <- reorder(dd, 10:1)
plot(dd, main = "random dendrogram `dd'")

op <- par(mfcol = 1:2)
plot(dd.reorder, main = "reorder(dd, 10:1)")
plot(reorder(dd,10:1, agglo.FUN= mean),
     main = "reorder(dd, 10:1, mean)")
par(op)
```

---

reorder.factor *Reorder Levels of a Factor*

Description

reorder is a generic function. Its "factor" method reorders the levels of a factor depending on values of a second variable, usually numeric.

Usage

```
reorder(x, ...)

## S3 method for class 'factor':
reorder(x, X, FUN = mean, ...,
        order = is.ordered(x))
```

Arguments

x	a factor (possibly ordered) whose levels will be reordered.
X	a vector of the same length as x, whose subset of values for each unique level of x determines the eventual order of that level.
FUN	a function whose first argument is a vector and returns a scalar, to be applied to each subset of X determined by the levels of x.
...	optional: extra arguments supplied to FUN
order	logical, whether return value will be an ordered factor rather than a factor.

Value

A factor or an ordered factor (depending on the value of order), with the order of the levels determined by FUN applied to X grouped by x. The levels are ordered such that the values returned by FUN are in increasing order.

Additionally, the values of FUN applied to the subsets of X (in the original order of the levels of x) is returned as the "scores" attribute.

Author(s)

Deepayan Sarkar ⟨deepayan@stat.wisc.edu⟩

See Also

reorder.dendrogram, levels, relevel.

Examples

```
bymedian <- with(InsectSprays, reorder(spray, count, median))
boxplot(count ~ bymedian, data = InsectSprays,
        xlab = "Type of spray", ylab = "Insect count",
        main = "InsectSprays data", varwidth = TRUE,
        col = "lightgray")
```

---

replications *Number of Replications of Terms*

Description

Returns a vector or a list of the number of replicates for each term in the formula.

Usage

```
replications(formula, data=NULL, na.action)
```

Arguments

formula	a formula or a terms object or a data frame.
data	a data frame used to find the objects in formula.
na.action	function for handling missing values. Defaults to a na.action attribute of data, then a setting of the option na.action, or na.fail if that is not set.

Details

If formula is a data frame and data is missing, formula is used for data with the formula ~ ..

Value

A vector or list with one entry for each term in the formula giving the number(s) of replications for each level. If all levels are balanced (have the same number of replications) the result is a vector, otherwise it is a list with a component for each terms, as a vector, matrix or array as required.

A test for balance is !is.list(replications(formula,data)).

Author(s)

The design was inspired by the S function of the same name described in Chambers *et al.* (1992).

References

Chambers, J. M., Freeny, A and Heiberger, R. M. (1992) *Analysis of variance; designed experiments.* Chapter 5 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

model.tables

Examples

```
## From Venables and Ripley (2002) p.165.
N <- c(0,1,0,1,1,1,0,0,0,1,1,0,1,1,0,0,1,0,1,0,1,1,0,0)
P <- c(1,1,0,0,0,1,0,1,1,1,0,0,0,1,0,1,1,0,0,1,0,1,1,0)
K <- c(1,0,0,1,0,1,1,0,0,1,0,1,0,1,1,0,0,0,1,1,1,0,1,0)
yield <- c(49.5,62.8,46.8,57.0,59.8,58.5,55.5,56.0,62.8,55.8,69.5,
55.0, 62.0,48.8,45.5,44.2,52.0,51.5,49.8,48.8,57.2,59.0,53.2,56.0)

npk <- data.frame(block=gl(6,4), N=factor(N), P=factor(P),
                  K=factor(K), yield=yield)
replications(~ . - yield, npk)
```

`reshape` *Reshape Grouped Data*

Description

This function reshapes a data frame between 'wide' format with repeated measurements in separate columns of the same record and 'long' format with the repeated measurements in separate records.

Usage

```
reshape(data, varying = NULL, v.names = NULL, timevar = "time",
        idvar = "id", ids = 1:NROW(data),
        times = seq(length = length(varying[[1]])),
        drop = NULL, direction, new.row.names = NULL,
        split = list(regexp="\\.", include=FALSE))
```

Arguments

`data`	a data frame
`varying`	names of sets of variables in the wide format that correspond to single variables in long format ('time-varying'). A list of vectors (or optionally a matrix for `direction="wide"`). See below for more details and options.
`v.names`	names of variables in the long format that correspond to multiple variables in the wide format.
`timevar`	the variable in long format that differentiates multiple records from the same group or individual.
`idvar`	Names of one or more variables in long format that identify multiple records from the same group/individual. These variables may also be present in wide format
`ids`	the values to use for a newly created `idvar` variable in long format.
`times`	the values to use for a newly created `timevar` variable in long format.
`drop`	a vector of names of variables to drop before reshaping
`direction`	character string, either `"wide"` to reshape to wide format, or `"long"` to reshape to long format.
`new.row.names`	logical; if `TRUE` and `direction="wide"`, create new row names in long format from the values of the id and time variables.
`split`	information for guessing the `varying`, `v.names`, and `times` arguments. See below for details.

Details

The arguments to this function are described in terms of longitudinal data, as that is the application motivating the functions. A 'wide' longitudinal dataset will have one record for each individual with some time-constant variables that occupy single columns and some time-varying variables that occupy a column for each time point. In 'long' format there will be multiple records for each individual, with some variables being constant across these records and others varying across the records. A 'long' format dataset also needs a 'time' variable identifying which time point each record comes from and an 'id' variable showing which records refer to the same person.

If the data frame resulted from a previous `reshape` then the operation can be reversed simply by `reshape(a)`. The `direction` argument is optional and the other arguments are stored as attributes on the data frame.

If `direction="long"` and no `varying` or `v.names` arguments are supplied it is assumed that all variables except `idvar` and `timevar` are time-varying. They are all expanded into multiple variables in wide format.

If `direction="wide"` the `varying` argument can be a vector of column names or column numbers (converted to column names). The function will attempt to guess the `v.names` and `times` from these names. The default is variable names like `x.1`, `x.2`,where `split=list(regexp="\.",include=FALSE)` to specifies to split at the dot and drop it from the name. To have alphabetic followed by numeric times use `split=list(regexp="[A-Za-z][0-9]",include=TRUE)`. This splits between the alphabetic and numeric parts of the name and does not drop the regular expression.

Value

The reshaped data frame with added attributes to simplify reshaping back to the original form.

See Also

`stack`, `aperm`

Examples

```
summary(Indometh)
wide <- reshape(Indometh, v.names="conc", idvar="Subject",
                timevar="time", direction="wide")
wide

reshape(wide, direction="long")
reshape(wide, idvar="Subject", varying=list(names(wide)[2:12]),
        v.names="conc", direction="long")

## times need not be numeric
df <- data.frame(id=rep(1:4,rep(2,4)),
                 visit=I(rep(c("Before","After"),4)),
                 x=rnorm(4), y=runif(4))
df
reshape(df, timevar="visit", idvar="id", direction="wide")
## warns that y is really varying
reshape(df, timevar="visit", idvar="id", direction="wide", v.names="x")

##  unbalanced 'long' data leads to NA fill in 'wide' form
df2 <- df[1:7,]
df2
reshape(df2, timevar="visit", idvar="id", direction="wide")

## Alternative regular expressions for guessing names
df3 <- data.frame(id=1:4, age=c(40,50,60,50), dose1=c(1,2,1,2),
                  dose2=c(2,1,2,1), dose4=c(3,3,3,3))
reshape(df3, direction="long", varying=3:5,
        split=list(regexp="[a-z][0-9]", include=TRUE))

## an example that isn't longitudinal data
state.x77 <- as.data.frame(state.x77)
```

```
long <- reshape(state.x77, idvar="state", ids=row.names(state.x77),
                times=names(state.x77), timevar="Characteristic",
                varying=list(names(state.x77)), direction="long")

reshape(long, direction="wide")

reshape(long, direction="wide", new.row.names=unique(long$state))

## multiple id variables
df3 <- data.frame(school=rep(1:3,each=4), class=rep(9:10,6),
                  time=rep(c(1,1,2,2),3),
score=rnorm(12))
wide <- reshape(df3, idvar=c("school","class"), direction="wide")
wide
## transform back
reshape(wide)
```

`residuals` *Extract Model Residuals*

Description

`residuals` is a generic function which extracts model residuals from objects returned by modeling functions.

The abbreviated form `resid` is an alias for `residuals`. It is intended to encourage users to access object components through an accessor function rather than by directly referencing an object slot.

All object classes which are returned by model fitting functions should provide a `residuals` method. (Note that the method is for '`residuals`' and not '`resid`'.)

Methods can make use of `naresid` methods to compensate for the omission of missing values. The default, `nls` and `smooth.spline` methods do.

Usage

```
residuals(object, ...)
resid(object, ...)
```

Arguments

`object`	an object for which the extraction of model residuals is meaningful.
`...`	other arguments.

Value

Residuals extracted from the object `object`.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

See Also

coefficients, fitted.values, glm, lm.

---

runmed *Running Medians – Robust Scatter Plot Smoothing*

Description

Compute running medians of odd span. This is the "most robust" scatter plot smoothing possible. For efficiency (and historical reason), you can use one of two different algorithms giving identical results.

Usage

```
runmed(x, k, endrule = c("median", "keep", "constant"),
       algorithm = NULL, print.level = 0)
```

Arguments

x	numeric vector, the "dependent" variable to be smoothed.
k	integer width of median window; must be odd. Turlach had a default of `k <- 1 + 2 * min((n-1)%/% 2, ceiling(0.1*n))`. Use `k = 3` for "minimal" robust smoothing eliminating isolated outliers.
endrule	character string indicating how the values at the beginning and the end (of the data) should be treated.
	"keep" keeps the first and last k_2 values at both ends, where k_2 is the half-bandwidth `k2 = k %/% 2`, i.e., `y[j] = x[j]` for $j \in \{1, \ldots, k_2; n - k_2 + 1, \ldots, n\}$;
	"constant" copies `median(y[1:k2])` to the first values and analogously for the last ones making the smoothed ends *constant*;
	"median" the default, smoothes the ends by using symmetrical medians of subsequently smaller bandwidth, but for the very first and last value where Tukey's robust end-point rule is applied, see smoothEnds.
algorithm	character string (partially matching "Turlach" or "Stuetzle") or the default NULL, specifying which algorithm should be applied. The default choice depends on `n = length(x)` and `k` where "Turlach" will be used for larger problems.
print.level	integer, indicating verboseness of algorithm; should rarely be changed by average users.

Details

Apart from the end values, the result `y = runmed(x, k)` simply has `y[j] = median(x[(j-k2):(j+k2)])` (k = 2*k2+1), computed very efficiently.

The two algorithms are internally entirely different:

"Turlach" is the Härdle-Steiger algorithm (see Ref.) as implemented by Berwin Turlach. A tree algorithm is used, ensuring performance $O(n \log k)$ where `n <- length(x)` which is asymptotically optimal.

"Stuetzle" is the (older) Stuetzle-Friedman implementation which makes use of median *updating* when one observation enters and one leaves the smoothing window. While this performs as $O(n \times k)$ which is slower asymptotically, it is considerably faster for small k or n.

Value

vector of smoothed values of the same length as `x` with an `attr`ibute `k` containing (the 'oddified') `k`.

Author(s)

Martin Maechler ⟨maechler@stat.math.ethz.ch⟩, based on Fortran code from Werner Stuetzle and S-plus and C code from Berwin Turlach.

References

Härdle, W. and Steiger, W. (1995) [Algorithm AS 296] Optimal median smoothing, *Applied Statistics* **44**, 258–264.

Jerome H. Friedman and Werner Stuetzle (1982) *Smoothing of Scatterplots*; Report, Dep. Statistics, Stanford U., Project Orion 003.

Martin Maechler (2003) Fast Running Medians: Finite Sample and Asymptotic Optimality; working paper available from the author.

See Also

`smoothEnds` which implements Tukey's end point rule and is called by default from `runmed(*, endrule = "median")`. `smooth` uses running medians of 3 for its compound smoothers.

Examples

```
example(nhtemp)
myNHT <- as.vector(nhtemp)
myNHT[20] <- 2 * nhtemp[20]
plot(myNHT, type="b", ylim = c(48,60), main = "Running Medians Example")
lines(runmed(myNHT, 7), col = "red")

## special: multiple y values for one x
plot(cars, main = "'cars' data and runmed(dist, 3)")
lines(cars, col = "light gray", type = "c")
with(cars, lines(speed, runmed(dist, k = 3), col = 2))

## nice quadratic with a few outliers
y <- ys <- (-20:20)^2
y [c(1,10,21,41)] <- c(150, 30, 400, 450)
all(y == runmed(y, 1)) # 1-neigborhood <==> interpolation
plot(y) ## lines(y, lwd=.1, col="light gray")
lines(lowess(seq(y),y, f = .3), col = "brown")
lines(runmed(y, 7), lwd=2, col = "blue")
lines(runmed(y,11), lwd=2, col = "red")

## Lowess is not robust
y <- ys ; y[21] <- 6666 ; x <- seq(y)
col <- c("black", "brown","blue")
plot(y, col=col[1])
lines(lowess(x,y, f = .3), col = col[2])
```

```
lines(runmed(y, 7),      lwd=2, col = col[3])
legend(length(y),max(y), c("data", "lowess(y, f = 0.3)", "runmed(y, 7)"),
       xjust = 1, col = col, lty = c(0, 1,1), pch = c(1,NA,NA))
```

`scatter.smooth` *Scatter Plot with Smooth Curve Fitted by Loess*

Description

Plot and add a smooth curve computed by `loess` to a scatter plot.

Usage

```
scatter.smooth(x, y = NULL, span = 2/3, degree = 1,
    family = c("symmetric", "gaussian"),
    xlab = NULL, ylab = NULL,
    ylim = range(y, prediction$y, na.rm = TRUE),
    evaluation = 50, ...)

loess.smooth(x, y, span = 2/3, degree = 1,
    family = c("symmetric", "gaussian"), evaluation = 50, ...)
```

Arguments

`x,y`	the `x` and `y` arguments provide the x and y coordinates for the plot. Any reasonable way of defining the coordinates is acceptable. See the function `xy.coords` for details.
`span`	smoothness parameter for `loess`.
`degree`	degree of local polynomial used.
`family`	if `"gaussian"` fitting is by least-squares, and if `family="symmetric"` a re-descending M estimator is used.
`xlab`	label for x axis.
`ylab`	label for y axis.
`ylim`	the y limits of the plot.
`evaluation`	number of points at which to evaluate the smooth curve.
`...`	graphical parameters.

Details

`loess.smooth` is an auxiliary function which evaluates the `loess` smooth at `evaluation` equally spaced points covering the range of `x`.

Value

For `scatter.smooth`, none.

For `loess.smooth`, a list with two components, `x` (the grid of evaluation points) and `y` (the smoothed values at the grid points).

See Also

loess

Examples

```
attach(cars)
scatter.smooth(speed, dist)
detach()
```

---

screeplot *Screeplot of PCA Results*

Description

screeplot plots the variances against the number of the principal component. This is also the plot method for class "princomp".

Usage

```
screeplot(x, npcs = min(10, length(x$sdev)),
          type = c("barplot", "lines"),
          main = deparse(substitute(x)), ...)
```

Arguments

x	an object of class "princomp", as from princomp().
npcs	the number of principal components to be plotted.
type	the type of plot.
main, ...	graphics parameters.

References

Mardia, K. V., J. T. Kent and J. M. Bibby (1979). *Multivariate Analysis*, London: Academic Press.

Venables, W. N. and B. D. Ripley (2002). *Modern Applied Statistics with S*, Springer-Verlag.

See Also

princomp.

Examples

```
## The variances of the variables in the
## USArrests data vary by orders of magnitude, so scaling is appropriate
(pc.cr <- princomp(USArrests, cor = TRUE))  # inappropriate
screeplot(pc.cr)

fit <- princomp(covmat=Harman74.cor)
screeplot(fit)
screeplot(fit, npcs=24, type="lines")
```

sd *Standard Deviation*

Description

This function computes the standard deviation of the values in `x`. If `na.rm` is `TRUE` then missing values are removed before computation proceeds. If `x` is a matrix or a data frame, a vector of the standard deviation of the columns is returned.

Usage

```
sd(x, na.rm = FALSE)
```

Arguments

`x`	a numeric vector, matrix or data frame.
`na.rm`	logical. Should missing values be removed?

See Also

`var` for its square, and `mad`, the most robust alternative.

Examples

```
sd(1:2) ^ 2
```

se.contrast *Standard Errors for Contrasts in Model Terms*

Description

Returns the standard errors for one or more contrasts in an `aov` object.

Usage

```
se.contrast(object, ...)
## S3 method for class 'aov':
se.contrast(object, contrast.obj,
            coef = contr.helmert(ncol(contrast))[, 1],
            data = NULL, ...)
```

Arguments

`object`	A suitable fit, usually from `aov`.
`contrast.obj`	The contrasts for which standard errors are requested. This can be specified via a list or via a matrix. A single contrast can be specified by a list of logical vectors giving the cells to be contrasted. Multiple contrasts should be specified by a matrix, each column of which is a numerical contrast vector (summing to zero).

coef	used when contrast.obj is a list; it should be a vector of the same length as the list with zero sum. The default value is the first Helmert contrast, which contrasts the first and second cell means specified by the list.
data	The data frame used to evaluate contrast.obj.
...	further arguments passed to or from other methods.

Details

Contrasts are usually used to test if certain means are significantly different; it can be easier to use se.contrast than compute them directly from the coefficients.

In multistratum models, the contrasts can appear in more than one stratum, in which case the standard errors are computed in the lowest stratum and adjusted for efficiencies and comparisons between strata. (See the comments in the note in the help for aov about using orthogonal contrasts.) Such standard errors are often conservative.

Suitable matrices for use with coef can be found by calling contrasts and indexing the columns by a factor.

Value

A vector giving the standard errors for each contrast.

See Also

contrasts, model.tables

Examples

```
## From Venables and Ripley (2002) p.165.
N <- c(0,1,0,1,1,1,0,0,0,1,1,0,1,1,0,0,1,0,1,0,1,1,0,0)
P <- c(1,1,0,0,0,1,0,1,1,1,0,0,0,1,0,1,1,0,0,1,0,1,1,0)
K <- c(1,0,0,1,0,1,1,1,0,0,1,0,1,0,1,1,0,0,0,1,1,1,0,1,0)
yield <- c(49.5,62.8,46.8,57.0,59.8,58.5,55.5,56.0,62.8,55.8,69.5,
55.0, 62.0,48.8,45.5,44.2,52.0,51.5,49.8,48.8,57.2,59.0,53.2,56.0)

npk <- data.frame(block = gl(6,4), N = factor(N), P = factor(P),
                  K = factor(K), yield = yield)
## Set suitable contrasts.
options(contrasts=c("contr.helmert", "contr.poly"))
npk.aov1 <- aov(yield ~ block + N + K, data=npk)
se.contrast(npk.aov1, list(N == "0", N == "1"), data=npk)
# or via a matrix
cont <- matrix(c(-1,1), 2, 1, dimnames=list(NULL, "N"))
se.contrast(npk.aov1, cont[N, , drop=FALSE]/12, data=npk)

## test a multi-stratum model
npk.aov2 <- aov(yield ~ N + K + Error(block/(N + K)), data=npk)
se.contrast(npk.aov2, list(N == "0", N == "1"))

## an example looking at an interaction contrast
## Dataset from R.E. Kirk (1995)
## 'Experimental Design: procedures for the behavioral sciences'
score <- c(12, 8,10, 6, 8, 4,10,12, 8, 6,10,14, 9, 7, 9, 5,11,12,
           7,13, 9, 9, 5,11, 8, 7, 3, 8,12,10,13,14,19, 9,16,14)
A <- gl(2, 18, labels=c("a1", "a2"))
```

```
B <- rep(gl(3, 6, labels=c("b1", "b2", "b3")), 2)
fit <- aov(score ~ A*B)
cont <- c(1, -1)[A] * c(1, -1, 0)[B]
sum(cont)       # 0
sum(cont*score) # value of the contrast
se.contrast(fit, as.matrix(cont))
(t.stat <- sum(cont*score)/se.contrast(fit, as.matrix(cont)))
summary(fit, split=list(B=1:2), expand.split = TRUE)
## t.stat^2 is the F value on the A:B: C1 line (with Helmert contrasts)
## Now look at all three interaction contrasts
cont <- c(1, -1)[A] * cbind(c(1, -1, 0), c(1, 0, -1), c(0, 1, -1))[B,]
se.contrast(fit, cont)  # same, due to balance.
rm(A,B,score)

## multi-stratum example where efficiencies play a role
example(eff.aovlist)
fit <- aov(Yield ~ A + B * C + Error(Block), data = aovdat)
cont1 <- c(-1, 1)[A]/32  # Helmert contrasts
cont2 <- c(-1, 1)[B] * c(-1, 1)[C]/32
cont <- cbind(A=cont1, BC=cont2)
colSums(cont*Yield) # values of the contrasts
se.contrast(fit, as.matrix(cont))
## Not run:
# comparison with lme
library(nlme)
fit2 <- lme(Yield ~ A + B*C, random = ~1 | Block, data = aovdat)
summary(fit2)$tTable # same estimates, similar (but smaller) se's.
## End(Not run)
```

`selfStart` *Construct Self-starting Nonlinear Models*

Description

Construct self-starting nonlinear models.

Usage

```
selfStart(model, initial, parameters, template)
```

Arguments

`model`	a function object defining a nonlinear model or a nonlinear formula object of the form `~expression`.
`initial`	a function object, taking three arguments: `mCall`, `data`, and `LHS`, representing, respectively, a matched call to the function `model`, a data frame in which to interpret the variables in `mCall`, and the expression from the left-hand side of the model formula in the call to `nls`. This function should return initial values for the parameters in `model`.
`parameters`	a character vector specifying the terms on the right hand side of `model` for which initial estimates should be calculated. Passed as the `namevec` argument to the `deriv` function.

`template` an optional prototype for the calling sequence of the returned object, passed as the `function.arg` argument to the `deriv` function. By default, a template is generated with the covariates in `model` coming first and the parameters in `model` coming last in the calling sequence.

Details

This function is generic; methods functions can be written to handle specific classes of objects.

Value

a function object of class `"selfStart"`, for the `formula` method obtained by applying `deriv` to the right hand side of the `model` formula. An `initial` attribute (defined by the `initial` argument) is added to the function to calculate starting estimates for the parameters in the model automatically.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

`nls`

Examples

```
## self-starting logistic model

SSlogis <- selfStart(~ Asym/(1 + exp((xmid - x)/scal)),
  function(mCall, data, LHS)
  {
    xy <- sortedXyData(mCall[["x"]], LHS, data)
    if(nrow(xy) < 4) {
      stop("Too few distinct x values to fit a logistic")
    }
    z <- xy[["y"]]
    if (min(z) <= 0) { z <- z + 0.05 * max(z) } # avoid zeroes
    z <- z/(1.05 * max(z))              # scale to within unit height
    xy[["z"]] <- log(z/(1 - z))         # logit transformation
    aux <- coef(lm(x ~ z, xy))
    parameters(xy) <- list(xmid = aux[1], scal = aux[2])
    pars <- as.vector(coef(nls(y ~ 1/(1 + exp((xmid - x)/scal)),
                               data = xy, algorithm = "plinear")))
    value <- c(pars[3], pars[1], pars[2])
    names(value) <- mCall[c("Asym", "xmid", "scal")]
    value
  }, c("Asym", "xmid", "scal"))

# 'first.order.log.model' is a function object defining a first order
# compartment model
# 'first.order.log.initial' is a function object which calculates initial
# values for the parameters in 'first.order.log.model'

# self-starting first order compartment model
## Not run:
SSfol <- selfStart(first.order.log.model, first.order.log.initial)
```

```
## End(Not run)
```

`setNames` *Set the Names in an Object*

Description

This is a convenience function that sets the names on an object and returns the object. It is most useful at the end of a function definition where one is creating the object to be returned and would prefer not to store it under a name just so the names can be assigned.

Usage

```
setNames(object, nm)
```

Arguments

`object`	an object for which a `names` attribute will be meaningful
`nm`	a character vector of names to assign to the object

Value

An object of the same sort as `object` with the new names assigned.

Author(s)

Douglas M. Bates and Saikat DebRoy

See Also

`clearNames`

Examples

```
setNames( 1:3, c("foo", "bar", "baz") )
# this is just a short form of
tmp <- 1:3
names(tmp) <-  c("foo", "bar", "baz")
tmp
```

`shapiro.test` *Shapiro-Wilk Normality Test*

Description

Performs the Shapiro-Wilk test of normality.

Usage

```
shapiro.test(x)
```

Arguments

`x`	a numeric vector of data values, the number of which must be between 3 and 5000. Missing values are allowed.

Value

A list with class `"htest"` containing the following components:

`statistic`	the value of the Shapiro-Wilk statistic.
`p.value`	the p-value for the test.
`method`	the character string `"Shapiro-Wilk normality test"`.
`data.name`	a character string giving the name(s) of the data.

References

Patrick Royston (1982) An Extension of Shapiro and Wilk's W Test for Normality to Large Samples. *Applied Statistics*, **31**, 115–124.

Patrick Royston (1982) Algorithm AS 181: The W Test for Normality. *Applied Statistics*, **31**, 176–180.

Patrick Royston (1995) A Remark on Algorithm AS 181: The W Test for Normality. *Applied Statistics*, **44**, 547–551.

See Also

`qqnorm` for producing a normal quantile-quantile plot.

Examples

```
shapiro.test(rnorm(100, mean = 5, sd = 3))
shapiro.test(runif(100, min = 2, max = 4))
```

SignRank *Distribution of the Wilcoxon Signed Rank Statistic*

Description

Density, distribution function, quantile function and random generation for the distribution of the Wilcoxon Signed Rank statistic obtained from a sample with size n.

Usage

```
dsignrank(x, n, log = FALSE)
psignrank(q, n, lower.tail = TRUE, log.p = FALSE)
qsignrank(p, n, lower.tail = TRUE, log.p = FALSE)
rsignrank(nn, n)
```

Arguments

x, q	vector of quantiles.
p	vector of probabilities.
nn	number of observations. If length(nn) > 1, the length is taken to be the number required.
n	number(s) of observations in the sample(s). A positive integer, or a vector of such integers.
log, log.p	logical; if TRUE, probabilities p are given as log(p).
lower.tail	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

This distribution is obtained as follows. Let x be a sample of size n from a continuous distribution symmetric about the origin. Then the Wilcoxon signed rank statistic is the sum of the ranks of the absolute values x[i] for which x[i] is positive. This statistic takes values between 0 and $n(n+1)/2$, and its mean and variance are $n(n+1)/4$ and $n(n+1)(2n+1)/24$, respectively.

If either of the first two arguments is a vector, the recycling rule is used to do the calculations for all combinations of the two up to the length of the longer vector.

Value

dsignrank gives the density, psignrank gives the distribution function, qsignrank gives the quantile function, and rsignrank generates random deviates.

Author(s)

Kurt Hornik

See Also

wilcox.test to calculate the statistic from data, find p values and so on.

dwilcox etc, for the distribution of *two-sample* Wilcoxon rank sum statistic.

Examples

```
par(mfrow=c(2,2))
for(n in c(4:5,10,40)) {
  x <- seq(0, n*(n+1)/2, length=501)
  plot(x, dsignrank(x,n=n), type='l', main=paste("dsignrank(x,n=",n,")"))
}
```

---

simulate *Simulate Responses*

Description

Simulate one or more response vectors from the theoretical distribution corresponding to a fitted model object.

Usage

```
simulate(object, nsim, seed, ...)
```

Arguments

object	an object representing a fitted model.
nsim	number of response vectors to simulate. Defaults to 1.
seed	an object specifying if and how the random number generator should be initialized ("seeded"). For the "lm" method, either NULL or an integer that will be used in a call to set.seed before simulating the response vectors. If set, the value is saved as the "seed" attribute of the returned value. The default, NULL will not change the random generator state, and return .Random.seed as "seed" attribute, see below.
...	additional optional arguments.

Details

This is a generic function with a method for lm objects. Consult the individual modeling functions for details on how to use this function.

Value

Typically, a list of length nsim of simulated response vectors. When appropriate the result can be a data frame (which is a special type of list).

For the "lm" method, the result is a data frame with an attribute "seed" containing the seed argument and as.list(RNGkind()) if seed was not NULL, or the value of .Random.seed before the simulation was started when seed was NULL as by default.

See Also

fitted.values and residuals for related methods; glm, lm for model fitting.

Examples

```
x <- 1:5
mod1 <- lm(c(1:3,7,6) ~ x)
S1 <- simulate(mod1, nsim = 4)
## repeat the simulation:
.Random.seed <- attr(S1, "seed")
identical(S1, simulate(mod1, nsim = 4))

S2 <- simulate(mod1, nsim = 200, seed = 101)
rowMeans(S2) # should be about
fitted(mod1)

## repeat identically:
(sseed <- attr(S2, "seed")) # seed; RNGkind as attribute
stopifnot(identical(S2, simulate(mod1, nsim = 200, seed = sseed)))

## To be sure about the proper RNGkind, e.g., after
RNGversion("1.0.0")
## first set the RNG kind, then simulate
do.call(RNGkind, attr(sseed, "kind"))
identical(S2, simulate(mod1, nsim = 200, seed = sseed))
```

`smooth` *Tukey's (Running Median) Smoothing*

Description

Tukey's smoothers, *3RS3R*, *3RSS*, *3R*, etc.

Usage

```
smooth(x, kind = c("3RS3R", "3RSS", "3RSR", "3R", "3", "S"),
       twiceit = FALSE, endrule = "Tukey", do.ends = FALSE)
```

Arguments

`x`	a vector or time series
`kind`	a character string indicating the kind of smoother required; defaults to `"3RS3R"`.
`twiceit`	logical, indicating if the result should be "twiced". Twicing a smoother $S(y)$ means $S(y) + S(y - S(y))$, i.e., adding smoothed residuals to the smoothed values. This decreases bias (increasing variance).
`endrule`	a character string indicating the rule for smoothing at the boundary. Either `"Tukey"` (default) or `"copy"`.
`do.ends`	logical, indicating if the 3-splitting of ties should also happen at the boundaries ("ends"). This is only used for `kind = "S"`.

Details

3 is Tukey's short notation for running medians of length **3**,
3R stands for **R**epeated *3* until convergence, and
S for **S**plitting of horizontal stretches of length 2 or 3.

Hence, *3RS3R* is a concatenation of 3R, S and 3R, *3RSS* similarly, whereas *3RSR* means first 3R and then (S and 3) **R**epeated until convergence – which can be bad.

Value

An object of class "tukeysmooth" (which has print and summary methods) and is a vector or time series containing the smoothed values with additional attributes.

Note

S and S-PLUS use a different (somewhat better) Tukey smoother in smooth(*). Note that there are other smoothing methods which provide rather better results. These were designed for hand calculations and may be used mainly for didactical purposes.

Since R version 1.2, smooth *does* really implement Tukey's end-point rule correctly (see argument endrule).

kind = "3RSR" has been the default till R-1.1, but it can have very bad properties, see the examples.

Note that repeated application of smooth(*) *does* smooth more, for the "3RS*" kinds.

References

Tukey, J. W. (1977). *Exploratory Data Analysis*, Reading Massachusetts: Addison-Wesley.

See Also

lowess; loess, supsmu and smooth.spline.

Examples

```
## see also   demo(smooth) !

x1 <- c(4, 1, 3, 6, 6, 4, 1, 6, 2, 4, 2) # very artificial
(x3R <- smooth(x1, "3R")) # 2 iterations of "3"
smooth(x3R, kind = "S")

sm.3RS <- function(x, ...)
   smooth(smooth(x, "3R", ...), "S", ...)

y <- c(1,1, 19:1)
plot(y, main = "misbehaviour of \"3RSR\"", col.main = 3)
lines(sm.3RS(y))
lines(smooth(y))
lines(smooth(y, "3RSR"), col = 3, lwd = 2)# the horror

x <- c(8:10,10, 0,0, 9,9)
plot(x, main = "breakdown of  3R  and  S  and hence  3RSS")
matlines(cbind(smooth(x,"3R"),smooth(x,"S"), smooth(x,"3RSS"),smooth(x)))

presidents[is.na(presidents)] <- 0 # silly
```

```
summary(sm3 <- smooth(presidents, "3R"))
summary(sm2 <- smooth(presidents,"3RSS"))
summary(sm  <- smooth(presidents))

all.equal(c(sm2),c(smooth(smooth(sm3, "S"), "S"))) # 3RSS  === 3R S S
all.equal(c(sm), c(smooth(smooth(sm3, "S"), "3R")))# 3RS3R === 3R S 3R

plot(presidents, main = "smooth(presidents0, *) :  3R and default 3RS3R")
lines(sm3,col = 3, lwd = 1.5)
lines(sm, col = 2, lwd = 1.25)
```

`smooth.spline` *Fit a Smoothing Spline*

Description

Fits a cubic smoothing spline to the supplied data.

Usage

```
smooth.spline(x, y = NULL, w = NULL, df, spar = NULL,
              cv = FALSE, all.knots = FALSE, nknots = NULL,
              keep.data = TRUE, df.offset = 0, penalty = 1,
              control.spar = list())
```

Arguments

`x`	a vector giving the values of the predictor variable, or a list or a two-column matrix specifying x and y.
`y`	responses. If `y` is missing, the responses are assumed to be specified by `x`.
`w`	optional vector of weights of the same length as `x`; defaults to all 1.
`df`	the desired equivalent number of degrees of freedom (trace of the smoother matrix).
`spar`	smoothing parameter, typically (but not necessarily) in $(0, 1]$. The coefficient λ of the integral of the squared second derivative in the fit (penalized log likelihood) criterion is a monotone function of `spar`, see the details below.
`cv`	ordinary (`TRUE`) or "generalized" cross-validation (GCV) when `FALSE`.
`all.knots`	if `TRUE`, all distinct points in `x` are used as knots. If `FALSE` (default), a subset of `x[]` is used, specifically `x[j]` where the `nknots` indices are evenly spaced in `1:n`, see also the next argument `nknots`.
`nknots`	integer giving the number of knots to use when `all.knots=FALSE`. Per default, this is less than n, the number of unique `x` values for $n > 49$.
`keep.data`	logical specifiying if the input data should be kept in the result. If `TRUE` (as per default), fitted values and residuals are available from the result.
`df.offset`	allows the degrees of freedom to be increased by `df.offset` in the GCV criterion.
`penalty`	the coefficient of the penalty for degrees of freedom in the GCV criterion.

`control.spar` optional list with named components controlling the root finding when the smoothing parameter `spar` is computed, i.e., missing or `NULL`, see below.
Note that this is partly *experimental* and may change with general spar computation improvements!

low: lower bound for `spar`; defaults to -1.5 (used to implicitly default to 0 in R versions earlier than 1.4).

high: upper bound for `spar`; defaults to +1.5.

tol: the absolute precision (**tol**erance) used; defaults to 1e-4 (formerly 1e-3).

eps: the relative precision used; defaults to 2e-8 (formerly 0.00244).

trace: logical indicating if iterations should be traced.

maxit: integer giving the maximal number of iterations; defaults to 500.

Note that `spar` is only searched for in the interval $[low, high]$.

Details

The `x` vector should contain at least four distinct values. *Distinct* here means "distinct after rounding to 6 significant digits", i.e., `x` will be transformed to `unique(sort(signif(x, 6)))`, and `y` and `w` are pooled accordingly.

The computational λ used (as a function of $s = spar$) is $\lambda = r * 256^{3s-1}$ where $r = tr(X'WX)/tr(\Sigma)$, Σ is the matrix given by $\Sigma_{ij} = \int B_i''(t)B_j''(t)dt$, X is given by $X_{ij} = B_j(x_i)$, W is the diagonal matrix of weights (scaled such that its trace is n, the original number of observations) and $B_k(.)$ is the k-th B-spline.

Note that with these definitions, $f_i = f(x_i)$, and the B-spline basis representation $f = Xc$ (i.e., c is the vector of spline coefficients), the penalized log likelihood is $L = (y - f)'W(y - f) + \lambda c'\Sigma c$, and hence c is the solution of the (ridge regression) $(X'WX + \lambda\Sigma)c = X'Wy$.

If `spar` is missing or `NULL`, the value of `df` is used to determine the degree of smoothing. If both are missing, leave-one-out cross-validation (ordinary or "generalized" as determined by `cv`) is used to determine λ. Note that from the above relation, `spar` is $s = s0 + 0.0601 * \log \lambda$, which is intentionally *different* from the S-plus implementation of `smooth.spline` (where `spar` is proportional to λ). In R's ($\log \lambda$) scale, it makes more sense to vary `spar` linearly.

Note however that currently the results may become very unreliable for `spar` values smaller than about -1 or -2. The same may happen for values larger than 2 or so. Don't think of setting `spar` or the controls `low` and `high` outside such a safe range, unless you know what you are doing!

The "generalized" cross-validation method will work correctly when there are duplicated points in `x`. However, it is ambiguous what leave-one-out cross-validation means with duplicated points, and the internal code uses an approximation that involves leaving out groups of duplicated points. `cv=TRUE` is best avoided in that case.

Value

An object of class `"smooth.spline"` with components

`x`	the *distinct* `x` values in increasing order, see the **Details** above.
`y`	the fitted values corresponding to `x`.
`w`	the weights used at the unique values of `x`.
`yin`	the y values used at the unique `y` values.
`data`	only if `keep.data = TRUE`: itself a `list` with components `x`, `y` and `w` of the same length. These are the original $(x_i, y_i, w_i), i = 1, \ldots, n$, values where `data$x` may have repeated values and hence be longer than the above `x` component; see details.

lev	leverages, the diagonal values of the smoother matrix.
cv.crit	cross-validation score, "generalized" or true, depending on cv.
pen.crit	penalized criterion
crit	the criterion value minimized in the underlying .Fortran routine 'sslvrg'.
df	equivalent degrees of freedom used. Note that (currently) this value may become quite unprecise when the true df is between and 1 and 2.
spar	the value of spar computed or given.
lambda	the value of λ corresponding to spar, see the details above.
iparms	named integer(3) vector where ..$ipars["iter"] gives number of spar computing iterations used.
fit	list for use by predict.smooth.spline, with components **knot:** the knot sequence (including the repeated boundary knots). **nk:** number of coefficients or number of "proper" knots plus 2. **coef:** coefficients for the spline basis used. **min, range:** numbers giving the corresponding quantities of x.
call	the matched call.

Note

The default all.knots = FALSE and nknots = NULL entails using only $O(n^{0.2})$ knots instead of n for $n > 49$. This cuts speed and memory requirements, but not drastically anymore since R version 1.5.1 where it is only $O(n_k) + O(n)$ where n_k is the number of knots. In this case where not all unique x values are used as knots, the result is not a smoothing spline in the strict sense, but very close unless a small smoothing parameter (or large df) is used.

Author(s)

R implementation by B. D. Ripley and Martin Maechler (spar/lambda, etc).

This function is based on code in the GAMFIT Fortran program by T. Hastie and R. Tibshirani (http://lib.stat.cmu.edu/general/), which makes use of spline code by Finbarr O'Sullivan. Its design parallels the smooth.spline function of Chambers & Hastie (1992).

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*, Wadsworth & Brooks/Cole.

Green, P. J. and Silverman, B. W. (1994) *Nonparametric Regression and Generalized Linear Models: A Roughness Penalty Approach.* Chapman and Hall.

Hastie, T. J. and Tibshirani, R. J. (1990) *Generalized Additive Models.* Chapman and Hall.

See Also

predict.smooth.spline for evaluating the spline and its derivatives.

Examples

```
attach(cars)
plot(speed, dist, main = "data(cars)  &  smoothing splines")
cars.spl <- smooth.spline(speed, dist)
(cars.spl)
## This example has duplicate points, so avoid cv=TRUE

lines(cars.spl, col = "blue")
lines(smooth.spline(speed, dist, df=10), lty=2, col = "red")
legend(5,120,c(paste("default [C.V.] => df =",round(cars.spl$df,1)),
               "s( * , df = 10)"), col = c("blue","red"), lty = 1:2,
       bg='bisque')
detach()

## Residual (Tukey Anscombe) plot:
plot(residuals(cars.spl) ~ fitted(cars.spl))
abline(h = 0, col="gray")

## consistency check:
stopifnot(all.equal(cars$dist,
                    fitted(cars.spl) + residuals(cars.spl)))

##-- artificial example
y18 <- c(1:3,5,4,7:3,2*(2:5),rep(10,4))
xx  <- seq(1,length(y18), len=201)
(s2  <- smooth.spline(y18)) # GCV
(s02 <- smooth.spline(y18, spar = 0.2))
plot(y18, main=deparse(s2$call), col.main=2)
lines(s2, col = "gray"); lines(predict(s2, xx), col = 2)
lines(predict(s02, xx), col = 3); mtext(deparse(s02$call), col = 3)

## The following shows the problematic behavior of 'spar' searching:
(s2  <- smooth.spline(y18, con = list(trace=TRUE,tol=1e-6, low= -1.5)))
(s2m <- smooth.spline(y18, cv = TRUE,
                      con = list(trace=TRUE,tol=1e-6, low= -1.5)))
## both above do quite similarly (Df = 8.5 +- 0.2)
```

---

smoothEnds *End Points Smoothing (for Running Medians)*

Description

Smooth end points of a vector `y` using subsequently smaller medians and Tukey's end point rule at the very end. (of odd span),

Usage

```
smoothEnds(y, k = 3)
```

Arguments

`y`	dependent variable to be smoothed (vector).
`k`	width of largest median window; must be odd.

Details

`smoothEnds` is used to only do the "end point smoothing", i.e., change at most the observations closer to the beginning/end than half the window `k`. The first and last value are computed using "*Tukey's end point rule*", i.e., `sm[1] = median(y[1], sm[2], 3*sm[2] - 2*sm[3])`.

Value

vector of smoothed values, the same length as `y`.

Author(s)

Martin Maechler

References

John W. Tukey (1977) *Exploratory Data Analysis*, Addison.

Velleman, P.F., and Hoaglin, D.C. (1981) *ABC of EDA (Applications, Basics, and Computing of Exploratory Data Analysis)*; Duxbury.

See Also

`runmed(*, end.rule = "median")` which calls `smoothEnds()`.

Examples

```
y <- ys <- (-20:20)^2
y [c(1,10,21,41)] <-  c(100, 30, 400, 470)
s7k <- runmed(y,7, end = "keep")
s7. <- runmed(y,7, end = "const")
s7m <- runmed(y,7)
col3 <- c("midnightblue","blue","steelblue")
plot(y, main = "Running Medians -- runmed(*, k=7, end.rule = X)")
lines(ys, col = "light gray")
matlines(cbind(s7k,s7.,s7m), lwd= 1.5, lty = 1, col = col3)
legend(1,470, paste("end.rule",c("keep","constant","median"),sep=" = "),
       col = col3, lwd = 1.5, lty = 1)

stopifnot(identical(s7m, smoothEnds(s7k, 7)))
```

`sortedXyData` *Create a sortedXyData object*

Description

This is a constructor function for the class of `sortedXyData` objects. These objects are mostly used in the `initial` function for a self-starting nonlinear regression model, which will be of the `selfStart` class.

Usage

```
sortedXyData(x, y, data)
```

Arguments

x	a numeric vector or an expression that will evaluate in data to a numeric vector
y	a numeric vector or an expression that will evaluate in data to a numeric vector
data	an optional data frame in which to evaluate expressions for x and y, if they are given as expressions

Value

A sortedXyData object. This is a data frame with exactly two numeric columns, named x and y. The rows are sorted so the x column is in increasing order. Duplicate x values are eliminated by averaging the corresponding y values.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

selfStart, NLSstClosestX, NLSstLfAsymptote, NLSstRtAsymptote

Examples

```
DNase.2 <- DNase[ DNase$Run == "2", ]
sortedXyData( expression(log(conc)), expression(density), DNase.2 )
```

---

spec.ar *Estimate Spectral Density of a Time Series from AR Fit*

Description

Fits an AR model to x (or uses the existing fit) and computes (and by default plots) the spectral density of the fitted model.

Usage

```
spec.ar(x, n.freq, order = NULL, plot = TRUE, na.action = na.fail,
        method = "yule-walker", ...)
```

Arguments

x	A univariate (not yet:or multivariate) time series or the result of a fit by ar.
n.freq	The number of points at which to plot.
order	The order of the AR model to be fitted. If omitted, the order is chosen by AIC.
plot	Plot the periodogram?
na.action	NA action function.
method	method for ar fit.
...	Graphical arguments passed to plot.spec.

Value

An object of class `"spec"`. The result is returned invisibly if `plot` is true.

Warning

Some authors, for example Thomson (1990), warn strongly that AR spectra can be misleading.

Note

The multivariate case is not yet implemented.

References

Thompson, D.J. (1990) Time series analysis of Holocene climate data. *Phil. Trans. Roy. Soc. A* **330**, 601–616.

Venables, W.N. and Ripley, B.D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer. (Especially page 402.)

See Also

`ar`, `spectrum`.

Examples

```
spec.ar(lh)

spec.ar(ldeaths)
spec.ar(ldeaths, method="burg")
```

---

spec.pgram *Estimate Spectral Density of a Time Series by a Smoothed Periodogram*

Description

`spec.pgram` calculates the periodogram using a fast Fourier transform, and optionally smooths the result with a series of modified Daniell smoothers (moving averages giving half weight to the end values).

Usage

```
spec.pgram(x, spans = NULL, kernel, taper = 0.1,
           pad = 0, fast = TRUE, demean = FALSE, detrend = TRUE,
           plot = TRUE, na.action = na.fail, ...)
```

Arguments

x	univariate or multivariate time series.
spans	vector of odd integers giving the widths of modified Daniell smoothers to be used to smooth the periodogram.
kernel	alternatively, a kernel smoother of class "tskernel".
taper	specifies theproportion of data to taper. A split cosine bell taper is applied to this proportion of the data at the beginning and end of the series.
pad	proportion of data to pad. Zeros are added to the end of the series to increase its length by the proportion pad.
fast	logical; if TRUE, pad the series to a highly composite length.
demean	logical. If TRUE, subtract the mean of the series.
detrend	logical. If TRUE, remove a linear trend from the series. This will also remove the mean.
plot	plot the periodogram?
na.action	NA action function.
...	graphical arguments passed to plot.spec.

Details

The raw periodogram is not a consistent estimator of the spectral density, but adjacent values are asymptotically independent. Hence a consistent estimator can be derived by smoothing the raw periodogram, assuming that the spectral density is smooth.

The series will be automatically padded with zeros until the series length is a highly composite number in order to help the Fast Fourier Transform. This is controlled by the fast and not the pad argument.

The periodogram at zero is in theory zero as the mean of the series is removed (but this may be affected by tapering): it is replaced by an interpolation of adjacent values during smoothing, and no value is returned for that frequency.

Value

A list object of class "spec" (see spectrum) with the following additional components:

kernel	The kernel argument, or the kernel constructed from spans.
df	The distribution of the spectral density estimate can be approximated by a (scaled) chi square distribution with df degrees of freedom.
bandwidth	The equivalent bandwidth of the kernel smoother as defined by Bloomfield (1976, page 201).
taper	The value of the taper argument.
pad	The value of the pad argument.
detrend	The value of the detrend argument.
demean	The value of the demean argument.

The result is returned invisibly if plot is true.

Author(s)

Originally Martyn Plummer; kernel smoothing by Adrian Trapletti, synthesis by B.D. Ripley

References

Bloomfield, P. (1976) *Fourier Analysis of Time Series: An Introduction.* Wiley.

Brockwell, P.J. and Davis, R.A. (1991) *Time Series: Theory and Methods.* Second edition. Springer.

Venables, W.N. and Ripley, B.D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer. (Especially pp. 392–7.)

See Also

`spectrum`, `spec.taper`, `plot.spec`, `fft`

Examples

```
## Examples from Venables & Ripley
spectrum(ldeaths)
spectrum(ldeaths, spans = c(3,5))
spectrum(ldeaths, spans = c(5,7))
spectrum(mdeaths, spans = c(3,3))
spectrum(fdeaths, spans = c(3,3))

## bivariate example
mfdeaths.spc <- spec.pgram(ts.union(mdeaths, fdeaths), spans = c(3,3))
# plots marginal spectra: now plot coherency and phase
plot(mfdeaths.spc, plot.type = "coherency")
plot(mfdeaths.spc, plot.type = "phase")

## now impose a lack of alignment
mfdeaths.spc <- spec.pgram(ts.intersect(mdeaths, lag(fdeaths, 4)),
   spans = c(3,3), plot = FALSE)
plot(mfdeaths.spc, plot.type = "coherency")
plot(mfdeaths.spc, plot.type = "phase")

stocks.spc <- spectrum(EuStockMarkets, kernel("daniell", c(30,50)),
                       plot = FALSE)
plot(stocks.spc, plot.type = "marginal") # the default type
plot(stocks.spc, plot.type = "coherency")
plot(stocks.spc, plot.type = "phase")

sales.spc <- spectrum(ts.union(BJsales, BJsales.lead),
                      kernel("modified.daniell", c(5,7)))
plot(sales.spc, plot.type = "coherency")
plot(sales.spc, plot.type = "phase")
```

`spec.taper` *Taper a Time Series by a Cosine Bell*

Description

Apply a cosine-bell taper to a time series.

Usage

```
spec.taper(x, p = 0.1)
```

Arguments

`x`	A univariate or multivariate time series
`p`	The proportion to be tapered at each end of the series, either a scalar (giving the proportion for all series) or a vector of the length of the number of series (giving the proportion for each series..

Details

The cosine-bell taper is applied to the first and last `p[i]` observations of time series `x[, i]`.

Value

A new time series object.

See Also

`spec.pgram`, `cpgram`

`spectrum` *Spectral Density Estimation*

Description

The `spectrum` function estimates the spectral density of a time series.

Usage

```
spectrum(x, ..., method = c("pgram", "ar"))
```

Arguments

`x`	A univariate or multivariate time series.
`method`	String specifying the method used to estimate the spectral density. Allowed methods are `"pgram"` (the default) and `"ar"`.
`...`	Further arguments to specific spec methods or `plot.spec`.

Details

`spectrum` is a wrapper function which calls the methods `spec.pgram` and `spec.ar`.

The spectrum here is defined with scaling `1/frequency(x)`, following S-PLUS. This makes the spectral density a density over the range `(-frequency(x)/2, +frequency(x)/2]`, whereas a more common scaling is 2π and range $(-0.5, 0.5]$ (e.g., Bloomfield) or 1 and range $(-\pi, \pi]$.

If available, a confidence interval will be plotted by `plot.spec`: this is asymmetric, and the width of the centre mark indicates the equivalent bandwidth.

Value

An object of class "spec", which is a list containing at least the following components:

freq	vector of frequencies at which the spectral density is estimated. (Possibly approximate Fourier frequencies.) The units are the reciprocal of cycles per unit time (and not per observation spacing): see Details below.
spec	Vector (for univariate series) or matrix (for multivariate series) of estimates of the spectral density at frequencies corresponding to freq.
coh	NULL for univariate series. For multivariate time series, a matrix containing the *squared* coherency between different series. Column $i + (j-1) * (j-2)/2$ of coh contains the squared coherency between columns i and j of x, where $i < j$.
phase	NULL for univariate series. For multivariate time series a matrix containing the cross-spectrum phase between different series. The format is the same as coh.
series	The name of the time series.
snames	For multivariate input, the names of the component series.
method	The method used to calculate the spectrum.

The result is returned invisibly if plot is true.

Note

The default plot for objects of class "spec" is quite complex, including an error bar and default title, subtitle and axis labels. The defaults can all be overridden by supplying the appropriate graphical parameters.

Author(s)

Martyn Plummer, B.D. Ripley

References

Bloomfield, P. (1976) *Fourier Analysis of Time Series: An Introduction.* Wiley.

Brockwell, P. J. and Davis, R. A. (1991) *Time Series: Theory and Methods.* Second edition. Springer.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S-PLUS.* Fourth edition. Springer. (Especially pages 392–7.)

See Also

spec.ar, spec.pgram; plot.spec.

Examples

```
## Examples from Venables & Ripley
## spec.pgram
par(mfrow=c(2,2))
spectrum(lh)
spectrum(lh, spans=3)
spectrum(lh, spans=c(3,3))
spectrum(lh, spans=c(3,5))
```

```
spectrum(ldeaths)
spectrum(ldeaths, spans=c(3,3))
spectrum(ldeaths, spans=c(3,5))
spectrum(ldeaths, spans=c(5,7))
spectrum(ldeaths, spans=c(5,7), log="dB", ci=0.8)

# for multivariate examples see the help for spec.pgram

## spec.ar
spectrum(lh, method="ar")
spectrum(ldeaths, method="ar")
```

`splinefun` *Interpolating Splines*

Description

Perform cubic spline interpolation of given data points, returning either a list of points obtained by the interpolation or a function performing the interpolation.

Usage

```
splinefun(x, y = NULL, method = "fmm", ties = mean)

spline(x, y = NULL, n = 3*length(x), method = "fmm",
       xmin = min(x), xmax = max(x), ties = mean)
```

Arguments

`x,y`	vectors giving the coordinates of the points to be interpolated. Alternatively a single plotting structure can be specified: see `xy.coords`.
`method`	specifies the type of spline to be used. Possible values are `"fmm"`, `"natural"` and `"periodic"`.
`n`	interpolation takes place at `n` equally spaced points spanning the interval [`xmin`, `xmax`].
`xmin`	left-hand endpoint of the interpolation interval.
`xmax`	right-hand endpoint of the interpolation interval.
`ties`	Handling of tied `x` values. Either a function with a single vector argument returning a single number result or the string `"ordered"`.

Details

The inputs can contain missing values which are deleted, so at least one complete `(x, y)` pair is required. If `method = "fmm"`, the spline used is that of Forsythe, Malcolm and Moler (an exact cubic is fitted through the four points at each end of the data, and this is used to determine the end conditions). Natural splines are used when `method = "natural"`, and periodic splines when `method = "periodic"`.

These interpolation splines can also be used for extrapolation, that is prediction at points outside the range of `x`. Extrapolation makes little sense for `method = "fmm"`; for natural splines it is linear using the slope of the interpolating curve at the nearest data point.

Value

`spline` returns a list containing components `x` and `y` which give the ordinates where interpolation took place and the interpolated values.

`splinefun` returns a function with formal arguments `x` and `deriv`, the latter defaulting to zero. This function can be used to evaluate the interpolating cubic spline (`deriv`=0), or its derivatives (`deriv`=1,2,3) at the points `x`, where the spline function interpolates the data points originally specified. This is often more useful than `spline`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Forsythe, G. E., Malcolm, M. A. and Moler, C. B. (1977) *Computer Methods for Mathematical Computations*.

See Also

`approx` and `approxfun` for constant and linear interpolation.

Package **splines**, especially `interpSpline` and `periodicSpline` for interpolation splines. That package also generates spline bases that can be used for regression splines.

`smooth.spline` for smoothing splines.

Examples

```
op <- par(mfrow = c(2,1), mgp = c(2,.8,0), mar = .1+c(3,3,3,1))
n <- 9
x <- 1:n
y <- rnorm(n)
plot(x, y, main = paste("spline[fun](.) through", n, "points"))
lines(spline(x, y))
lines(spline(x, y, n = 201), col = 2)

y <- (x-6)^2
plot(x, y, main = "spline(.) -- 3 methods")
lines(spline(x, y, n = 201), col = 2)
lines(spline(x, y, n = 201, method = "natural"), col = 3)
lines(spline(x, y, n = 201, method = "periodic"), col = 4)
legend(6,25, c("fmm","natural","periodic"), col=2:4, lty=1)

y <- sin((x-0.5)*pi)
f <- splinefun(x, y)
ls(envir = environment(f))
splinecoef <- get("z", envir = environment(f))
curve(f(x), 1, 10, col = "green", lwd = 1.5)
points(splinecoef, col = "purple", cex = 2)
curve(f(x, deriv=1), 1, 10, col = 2, lwd = 1.5)
curve(f(x, deriv=2), 1, 10, col = 2, lwd = 1.5, n = 401)
curve(f(x, deriv=3), 1, 10, col = 2, lwd = 1.5, n = 401)
par(op)

## An example with ties (non-unique  x values):
set.seed(1); x <- round(rnorm(30), 1); y <- sin(pi * x) + rnorm(30)/10
plot(x,y, main="spline(x,y)  when x has ties")
lines(spline(x,y, n= 201), col = 2)
```

```
## visualizes the non-unique ones:
tx <- table(x); mx <- as.numeric(names(tx[tx > 1]))
ry <- matrix(unlist(tapply(y, match(x,mx), range, simplify=FALSE)),
             ncol=2, byrow=TRUE)
segments(mx, ry[,1], mx, ry[,2], col = "blue", lwd = 2)
```

SSasymp *Asymptotic Regression Model*

Description

This `selfStart` model evaluates the asymptotic regression function and its gradient. It has an `initial` attribute that will evaluate initial estimates of the parameters `Asym`, `R0`, and `lrc` for a given set of data.

Usage

```
SSasymp(input, Asym, R0, lrc)
```

Arguments

`input`	a numeric vector of values at which to evaluate the model.
`Asym`	a numeric parameter representing the horizontal asymptote on the right side (very large values of `input`).
`R0`	a numeric parameter representing the response when `input` is zero.
`lrc`	a numeric parameter representing the natural logarithm of the rate constant.

Value

a numeric vector of the same length as `input`. It is the value of the expression `Asym+(R0-Asym)*exp(-exp(lrc)*input)`. If all of the arguments `Asym`, `R0`, and `lrc` are names of objects, the gradient matrix with respect to these names is attached as an attribute named `gradient`.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

`nls`, `selfStart`

Examples

```
Lob.329 <- Loblolly[ Loblolly$Seed == "329", ]
SSasymp( Lob.329$age, 100, -8.5, -3.2 )  # response only
Asym <- 100
resp0 <- -8.5
lrc <- -3.2
SSasymp( Lob.329$age, Asym, resp0, lrc ) # response and gradient
getInitial(height ~ SSasymp( age, Asym, resp0, lrc), data = Lob.329)
## Initial values are in fact the converged values
```

```
fm1 <- nls(height ~ SSasymp( age, Asym, resp0, lrc), data = Lob.329)
summary(fm1)
```

SSasympOff *Asymptotic Regression Model with an Offset*

Description

This selfStart model evaluates an alternative parameterization of the asymptotic regression function and the gradient with respect to those parameters. It has an initial attribute that creates initial estimates of the parameters Asym, lrc, and c0.

Usage

```
SSasympOff(input, Asym, lrc, c0)
```

Arguments

input	a numeric vector of values at which to evaluate the model.
Asym	a numeric parameter representing the horizontal asymptote on the right side (very large values of input).
lrc	a numeric parameter representing the natural logarithm of the rate constant.
c0	a numeric parameter representing the input for which the response is zero.

Value

a numeric vector of the same length as input. It is the value of the expression Asym*(1 - exp(-exp(lrc)*(input - c0))). If all of the arguments Asym, lrc, and c0 are names of objects, the gradient matrix with respect to these names is attached as an attribute named gradient.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

nls, selfStart

Examples

```
CO2.Qn1 <- CO2[CO2$Plant == "Qn1", ]
SSasympOff( CO2.Qn1$conc, 32, -4, 43 )  # response only
Asym <- 32; lrc <- -4; c0 <- 43
SSasympOff( CO2.Qn1$conc, Asym, lrc, c0 ) # response and gradient
getInitial(uptake ~ SSasymp( conc, Asym, lrc, c0), data = CO2.Qn1)
## Initial values are in fact the converged values
fm1 <- nls(uptake ~ SSasymp( conc, Asym, lrc, c0), data = CO2.Qn1)
summary(fm1)
```

SSasympOrig *Asymptotic Regression Model through the Origin*

Description

This selfStart model evaluates the asymptotic regression function through the origin and its gradient. It has an initial attribute that will evaluate initial estimates of the parameters Asym and lrc for a given set of data.

Usage

```
SSasympOrig(input, Asym, lrc)
```

Arguments

input	a numeric vector of values at which to evaluate the model.
Asym	a numeric parameter representing the horizontal asymptote.
lrc	a numeric parameter representing the natural logarithm of the rate constant.

Value

a numeric vector of the same length as input. It is the value of the expression Asym*(1 - exp(-exp(lrc)*input)). If all of the arguments Asym and lrc are names of objects, the gradient matrix with respect to these names is attached as an attribute named gradient.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

nls, selfStart

Examples

```
Lob.329 <- Loblolly[ Loblolly$Seed == "329", ]
SSasympOrig( Lob.329$age, 100, -3.2 )  # response only
Asym <- 100; lrc <- -3.2
SSasympOrig( Lob.329$age, Asym, lrc ) # response and gradient
getInitial(height ~ SSasympOrig(age, Asym, lrc), data = Lob.329)
## Initial values are in fact the converged values
fm1 <- nls(height ~ SSasympOrig( age, Asym, lrc), data = Lob.329)
summary(fm1)
```

SSbiexp *Biexponential model*

Description

This `selfStart` model evaluates the biexponential model function and its gradient. It has an `initial` attribute that creates initial estimates of the parameters `A1`, `lrc1`, `A2`, and `lrc2`.

Usage

```
SSbiexp(input, A1, lrc1, A2, lrc2)
```

Arguments

`input`	a numeric vector of values at which to evaluate the model.
`A1`	a numeric parameter representing the multiplier of the first exponential.
`lrc1`	a numeric parameter representing the natural logarithm of the rate constant of the first exponential.
`A2`	a numeric parameter representing the multiplier of the second exponential.
`lrc2`	a numeric parameter representing the natural logarithm of the rate constant of the second exponential.

Value

a numeric vector of the same length as `input`. It is the value of the expression `A1*exp(-exp(lrc1)*input)+A2*exp(-exp(lrc2)*input)`. If all of the arguments `A1`, `lrc1`, `A2`, and `lrc2` are names of objects, the gradient matrix with respect to these names is attached as an attribute named `gradient`.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

`nls`, `selfStart`

Examples

```
Indo.1 <- Indometh[Indometh$Subject == 1, ]
SSbiexp( Indo.1$time, 3, 1, 0.6, -1.3 )  # response only
A1 <- 3; lrc1 <- 1; A2 <- 0.6; lrc2 <- -1.3
SSbiexp( Indo.1$time, A1, lrc1, A2, lrc2 ) # response and gradient
getInitial(conc ~ SSbiexp(time, A1, lrc1, A2, lrc2), data = Indo.1)
## Initial values are in fact the converged values
fm1 <- nls(conc ~ SSbiexp(time, A1, lrc1, A2, lrc2), data = Indo.1)
summary(fm1)
```

SSD *SSD Matrix and Estimated Variance Matrix in Multivariate Models*

Description

Functions to compute matrix of residual sums of squares and products, or the estimated variance matrix for multivariate linear models.

Usage

```
# S3 method for class 'mlm'
SSD(object, ...)

# S3 methods for class 'SSD' and 'mlm'
estVar(object, ...)
```

Arguments

object	object of class "mlm", or "SSD" in the case of estVar.
...	Unused

Value

SSD() returns a list of class "SSD" containing the following components

SSD	The residual sums of squares and products matrix
df	Degrees of freedom
call	Copied from object

estVar returns a matrix with the estimated variances and covariances.

See Also

mauchley.test, anova.mlm

Examples

```
# Lifted from Baron+Li:
# "Notes on the use of R for psychology experiments and questionnaires"
# Maxwell and Delaney, p. 497
reacttime <- matrix(c(
420, 420, 480, 480, 600, 780,
420, 480, 480, 360, 480, 600,
480, 480, 540, 660, 780, 780,
420, 540, 540, 480, 780, 900,
540, 660, 540, 480, 660, 720,
360, 420, 360, 360, 480, 540,
480, 480, 600, 540, 720, 840,
480, 600, 660, 540, 720, 900,
540, 600, 540, 480, 720, 780,
480, 420, 540, 540, 660, 780),
ncol = 6, byrow = TRUE,
dimnames=list(subj=1:10,
```

```
                cond=c("deg0NA", "deg4NA", "deg8NA",
                       "deg0NP", "deg4NP", "deg8NP")))

mlmfit <- lm(reacttime~1)
SSD(mlmfit)
estVar(mlmfit)
```

`SSfol` *First-order Compartment Model*

Description

This `selfStart` model evaluates the first-order compartment function and its gradient. It has an `initial` attribute that creates initial estimates of the parameters `lKe`, `lKa`, and `lCl`.

Usage

```
SSfol(Dose, input, lKe, lKa, lCl)
```

Arguments

`Dose`	a numeric value representing the initial dose.
`input`	a numeric vector at which to evaluate the model.
`lKe`	a numeric parameter representing the natural logarithm of the elimination rate constant.
`lKa`	a numeric parameter representing the natural logarithm of the absorption rate constant.
`lCl`	a numeric parameter representing the natural logarithm of the clearance.

Value

a numeric vector of the same length as `input`, which is the value of the expression `Dose * exp(lKe+lKa-lCl) * (exp(-exp(lKe)*input)-exp(-exp(lKa)*input)) / (exp(lKa)-exp(lKe))`.

If all of the arguments `lKe`, `lKa`, and `lCl` are names of objects, the gradient matrix with respect to these names is attached as an attribute named `gradient`.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

`nls`, `selfStart`

Examples

```
Theoph.1 <- Theoph[ Theoph$Subject == 1, ]
SSfol( Theoph.1$Dose, Theoph.1$Time, -2.5, 0.5, -3 )  # response only
lKe <- -2.5; lKa <- 0.5; lCl <- -3
SSfol( Theoph.1$Dose, Theoph.1$Time, lKe, lKa, lCl ) # response and gradient
getInitial(conc ~ SSfol(Dose, Time, lKe, lKa, lCl), data = Theoph.1)
## Initial values are in fact the converged values
fm1 <- nls(conc ~ SSfol(Dose, Time, lKe, lKa, lCl), data = Theoph.1)
summary(fm1)
```

SSfpl *Four-parameter Logistic Model*

Description

This `selfStart` model evaluates the four-parameter logistic function and its gradient. It has an `initial` attribute that will evaluate initial estimates of the parameters `A`, `B`, `xmid`, and `scal` for a given set of data.

Usage

```
SSfpl(input, A, B, xmid, scal)
```

Arguments

`input`	a numeric vector of values at which to evaluate the model.
`A`	a numeric parameter representing the horizontal asymptote on the left side (very small values of `input`).
`B`	a numeric parameter representing the horizontal asymptote on the right side (very large values of `input`).
`xmid`	a numeric parameter representing the `input` value at the inflection point of the curve. The value of `SSfpl` will be midway between `A` and `B` at `xmid`.
`scal`	a numeric scale parameter on the `input` axis.

Value

a numeric vector of the same length as `input`. It is the value of the expression `A+(B-A)/(1+exp((xmid-input)/scal))`. If all of the arguments `A`, `B`, `xmid`, and `scal` are names of objects, the gradient matrix with respect to these names is attached as an attribute named `gradient`.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

`nls`, `selfStart`

Examples

```
Chick.1 <- ChickWeight[ChickWeight$Chick == 1, ]
SSfpl( Chick.1$Time, 13, 368, 14, 6 )  # response only
A <- 13; B <- 368; xmid <- 14; scal <- 6
SSfpl( Chick.1$Time, A, B, xmid, scal ) # response and gradient
getInitial(weight ~ SSfpl(Time, A, B, xmid, scal), data = Chick.1)
## Initial values are in fact the converged values
fm1 <- nls(weight ~ SSfpl(Time, A, B, xmid, scal), data = Chick.1)
summary(fm1)
```

SSgompertz *Gompertz Growth Model*

Description

This `selfStart` model evaluates the Gompertz growth model and its gradient. It has an `initial` attribute that creates initial estimates of the parameters `Asym`, `b2`, and `b3`.

Usage

```
SSgompertz(x, Asym, b2, b3)
```

Arguments

`x`	a numeric vector of values at which to evaluate the model.
`Asym`	a numeric parameter representing the asymptote.
`b2`	a numeric parameter related to the value of the function at `x = 0`
`b3`	a numeric parameter related to the scale the `x` axis.

Value

a numeric vector of the same length as `input`. It is the value of the expression `Asym*exp(-b2*b3^x)`. If all of the arguments `Asym`, `b2`, and `b3` are names of objects the gradient matrix with respect to these names is attached as an attribute named `gradient`.

Author(s)

Douglas Bates

See Also

`nls`, `selfStart`

Examples

```
DNase.1 <- subset(DNase, Run == 1)
SSlogis(log(DNase.1$conc), 4.5, 2.3, 0.7)  # response only
Asym <- 4.5; b2 <- 2.3; b3 <- 0.7
SSgompertz(log(DNase.1$conc), Asym, b2, b3 ) # response and gradient
getInitial(density ~ SSgompertz(log(conc), Asym, b2, b3),
           data = DNase.1)
## Initial values are in fact the converged values
fm1 <- nls(density ~ SSgompertz(log(conc), Asym, b2, b3),
           data = DNase.1)
summary(fm1)
```

SSlogis *Logistic Model*

Description

This `selfStart` model evaluates the logistic function and its gradient. It has an `initial` attribute that creates initial estimates of the parameters `Asym`, `xmid`, and `scal`.

Usage

```
SSlogis(input, Asym, xmid, scal)
```

Arguments

`input`	a numeric vector of values at which to evaluate the model.
`Asym`	a numeric parameter representing the asymptote.
`xmid`	a numeric parameter representing the `x` value at the inflection point of the curve. The value of `SSlogis` will be `Asym/2` at `xmid`.
`scal`	a numeric scale parameter on the `input` axis.

Value

a numeric vector of the same length as `input`. It is the value of the expression `Asym/(1+exp((xmid-input)/scal))`. If all of the arguments `Asym`, `xmid`, and `scal` are names of objects the gradient matrix with respect to these names is attached as an attribute named `gradient`.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

`nls`, `selfStart`

Examples

```
Chick.1 <- ChickWeight[ChickWeight$Chick == 1, ]
SSlogis( Chick.1$Time, 368, 14, 6 )  # response only
Asym <- 368; xmid <- 14; scal <- 6
SSlogis( Chick.1$Time, Asym, xmid, scal ) # response and gradient
getInitial(weight ~ SSlogis(Time, Asym, xmid, scal), data = Chick.1)
## Initial values are in fact the converged values
fm1 <- nls(weight ~ SSlogis(Time, Asym, xmid, scal), data = Chick.1)
summary(fm1)
```

SSmicmen *Michaelis-Menten Model*

Description

This `selfStart` model evaluates the Michaelis-Menten model and its gradient. It has an `initial` attribute that will evaluate initial estimates of the parameters `Vm` and `K`

Usage

```
SSmicmen(input, Vm, K)
```

Arguments

`input`	a numeric vector of values at which to evaluate the model.
`Vm`	a numeric parameter representing the maximum value of the response.
`K`	a numeric parameter representing the `input` value at which half the maximum response is attained. In the field of enzyme kinetics this is called the Michaelis parameter.

Value

a numeric vector of the same length as `input`. It is the value of the expression `Vm*input/(K+input)`. If both the arguments `Vm` and `K` are names of objects, the gradient matrix with respect to these names is attached as an attribute named `gradient`.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

`nls`, `selfStart`

Examples

```
PurTrt <- Puromycin[ Puromycin$state == "treated", ]
SSmicmen( PurTrt$conc, 200, 0.05 )  # response only
Vm <- 200; K <- 0.05
SSmicmen( PurTrt$conc, Vm, K ) # response and gradient
getInitial(rate ~ SSmicmen(conc, Vm, K), data = PurTrt)
## Initial values are in fact the converged values
fm1 <- nls(rate ~ SSmicmen(conc, Vm, K), data = PurTrt)
summary( fm1 )
## Alternative call using the subset argument
fm2 <- nls(rate ~ SSmicmen(conc, Vm, K), data = Puromycin,
           subset = state == "treated")
summary(fm2)
```

SSweibull *Weibull growth curve model*

Description

This `selfStart` model evaluates the Weibull model for growth curve data and its gradient. It has an `initial` attribute that will evaluate initial estimates of the parameters `Asym`, `Drop`, `lrc`, and `pwr` for a given set of data.

Usage

```
SSweibull(x, Asym, Drop, lrc, pwr)
```

Arguments

`x`	a numeric vector of values at which to evaluate the model.
`Asym`	a numeric parameter representing the horizontal asymptote on the right side (very small values of `x`).
`Drop`	a numeric parameter representing the change from `Asym` to the `y` intercept.
`lrc`	a numeric parameter representing the natural logarithm of the rate constant.
`pwr`	a numeric parameter representing the power to which `x` is raised.

Details

This model is a generalization of the `SSasymp` model in that it reduces to `SSasymp` when `pwr` is unity.

Value

a numeric vector of the same length as `x`. It is the value of the expression `Asym-Drop*exp(-exp(lrc)*x^pwr)`. If all of the arguments `Asym`, `Drop`, `lrc`, and `pwr` are names of objects, the gradient matrix with respect to these names is attached as an attribute named `gradient`.

Author(s)

Douglas Bates

References

Ratkowsky, David A. (1983), *Nonlinear Regression Modeling*, Dekker. (section 4.4.5)

See Also

nls, selfStart, SSasymp

Examples

```
Chick.6 <- subset(ChickWeight, (Chick == 6) & (Time > 0))
SSweibull(Chick.6$Time, 160, 115, -5.5, 2.5 )  # response only
Asym <- 160; Drop <- 115; lrc <- -5.5; pwr <- 2.5
SSweibull(Chick.6$Time, Asym, Drop, lrc, pwr)  # response and gradient
getInitial(weight ~ SSweibull(Time, Asym, Drop, lrc, pwr), data = Chick.6)
## Initial values are in fact the converged values
fm1 <- nls(weight ~ SSweibull(Time, Asym, Drop, lrc, pwr), data = Chick.6)
summary(fm1)
```

---

start *Encode the Terminal Times of Time Series*

Description

Extract and encode the times the first and last observations were taken. Provided only for compatibility with S version 2.

Usage

```
start(x, ...)
end(x, ...)
```

Arguments

x	a univariate or multivariate time-series, or a vector or matrix.
...	extra arguments for future methods.

Details

These are generic functions, which will use the tsp attribute of x if it exists. Their default methods decode the start time from the original time units, so that for a monthly series 1995.5 is represented as c(1995, 7). For a series of frequency f, time n+i/f is presented as c(n, i+1) (even for i = 0 and f = 1).

Warning

The representation used by start and end has no meaning unless the frequency is supplied.

See Also

ts, time, tsp.

stat.anova *GLM Anova Statistics*

Description

This is a utility function, used in `lm` and `glm` methods for `anova(..., test != NULL)` and should not be used by the average user.

Usage

```
stat.anova(table, test = c("Chisq", "F", "Cp"), scale, df.scale, n)
```

Arguments

`table`	numeric matrix as results from `anova.glm(..., test=NULL)`.
`test`	a character string, matching one of `"Chisq"`, `"F"` or `"Cp"`.
`scale`	a residual mean square or other scale estimate to be used as the denominator in an F test.
`df.scale`	degrees of freedom corresponding to `scale`.
`n`	number of observations.

Value

A matrix which is the original `table`, augmented by a column of test statistics, depending on the `test` argument.

References

Hastie, T. J. and Pregibon, D. (1992) *Generalized linear models.* Chapter 6 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`anova.lm`, `anova.glm`.

Examples

```
##-- Continued from '?glm':

print(ag <- anova(glm.D93))
stat.anova(ag$table, test = "Cp",
           scale = sum(resid(glm.D93, "pearson")^2)/4, df = 4, n = 9)
```

stats-deprecated *Deprecated Functions in Stats package*

Description

These functions are provided for compatibility with older versions of R only, and may be defunct as soon as the next release.

Details

There are currently no deprecated functions in this package.

See Also

Deprecated

step *Choose a model by AIC in a Stepwise Algorithm*

Description

Select a formula-based model by AIC.

Usage

```
step(object, scope, scale = 0,
     direction = c("both", "backward", "forward"),
     trace = 1, keep = NULL, steps = 1000, k = 2, ...)
```

Arguments

object	an object representing a model of an appropriate class (mainly "lm" and "glm"). This is used as the initial model in the stepwise search.
scope	defines the range of models examined in the stepwise search. This should be either a single formula, or a list containing components upper and lower, both formulae. See the details for how to specify the formulae and how they are used.
scale	used in the definition of the AIC statistic for selecting the models, currently only for lm, aov and glm models. The default value, 0, indicates the scale should be estimated: see extractAIC.
direction	the mode of stepwise search, can be one of "both", "backward", or "forward", with a default of "both". If the scope argument is missing the default for direction is "backward".
trace	if positive, information is printed during the running of step. Larger values may give more detailed information.
keep	a filter function whose input is a fitted model object and the associated AIC statistic, and whose output is arbitrary. Typically keep will select a subset of the components of the object and return them. The default is not to keep anything.

`steps`	the maximum number of steps to be considered. The default is 1000 (essentially as many as required). It is typically used to stop the process early.
`k`	the multiple of the number of degrees of freedom used for the penalty. Only `k = 2` gives the genuine AIC: `k = log(n)` is sometimes referred to as BIC or SBC.
`...`	any additional arguments to `extractAIC`.

Details

`step` uses `add1` and `drop1` repeatedly; it will work for any method for which they work, and that is determined by having a valid method for `extractAIC`. When the additive constant can be chosen so that AIC is equal to Mallows' C_p, this is done and the tables are labelled appropriately.

The set of models searched is determined by the `scope` argument. The right-hand-side of its `lower` component is always included in the model, and right-hand-side of the model is included in the `upper` component. If `scope` is a single formula, it specifes the `upper` component, and the `lower` model is empty. If `scope` is missing, the initial model is used as the `upper` model.

Models specified by `scope` can be templates to update `object` as used by `update.formula`. So using `.` in a `scope` formula means 'what is already there', with `.^2` indicating all interactions of existing terms.

There is a potential problem in using `glm` fits with a variable `scale`, as in that case the deviance is not simply related to the maximized log-likelihood. The `"glm"` method for function `extractAIC` makes the appropriate adjustment for a `gaussian` family, but may need to be amended for other cases. (The `binomial` and `poisson` families have fixed `scale` by default and do not correspond to a particular maximum-likelihood problem for variable `scale`.)

Value

the stepwise-selected model is returned, with up to two additional components. There is an `"anova"` component corresponding to the steps taken in the search, as well as a `"keep"` component if the `keep=` argument was supplied in the call. The `"Resid. Dev"` column of the analysis of deviance table refers to a constant minus twice the maximized log likelihood: it will be a deviance only in cases where a saturated model is well-defined (thus excluding `lm`, `aov` and `survreg` fits, for example).

Warning

The model fitting must apply the models to the same dataset. This may be a problem if there are missing values and R's default of `na.action = na.omit` is used. We suggest you remove the missing values first.

Note

This function differs considerably from the function in S, which uses a number of approximations and does not in general compute the correct AIC.

This is a minimal implementation. Use `stepAIC` in package **MASS** for a wider range of object classes.

Author(s)

B. D. Ripley: `step` is a slightly simplified version of `stepAIC` in package **MASS** (Venables & Ripley, 2002 and earlier editions).

The idea of a `step` function follows that described in Hastie & Pregibon (1992); but the implementation in R is more general.

References

Hastie, T. J. and Pregibon, D. (1992) *Generalized linear models.* Chapter 6 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* New York: Springer (4th ed).

See Also

`stepAIC` in **MASS**, `add1`, `drop1`

Examples

```
example(lm)
step(lm.D9)

summary(lm1 <- lm(Fertility ~ ., data = swiss))
slm1 <- step(lm1)
summary(slm1)
slm1$anova
```

---

stepfun *Step Function Class*

Description

Given the vectors $(x_1, \ldots, x_n)$ and $(y_0, y_1, \ldots, y_n)$ (one value more!), `stepfun(x,y,...)` returns an interpolating "step" function, say `fn`. I.e., $fn(t) = c_i$ (constant) for $t \in (x_i, x_{i+1})$ and at the abscissa values, if (by default) `right = FALSE`, $fn(x_i) = y_i$ and for `right = TRUE`, $fn(x_i) = y_{i-1}$, for $i = 1, \ldots, n$.

The value of the constant c_i above depends on the "continuity" parameter `f`. For the default, `right = FALSE, f = 0`, `fn` is a "cadlag" function, i.e., continuous at right, limit ("the point") at left. In general, c_i is interpolated in between the neighbouring y values, $c_i = (1 - f)y_i + f \cdot y_{i+1}$. Therefore, for non-0 values of `f`, `fn` may no longer be a proper step function, since it can be discontinuous from both sides, unless `right = TRUE, f = 1` which is right-continuous.

Usage

```
stepfun(x, y, f = as.numeric(right), ties = "ordered",
        right = FALSE)

is.stepfun(x)
knots(Fn, ...)
as.stepfun(x, ...)

## S3 method for class 'stepfun':
print(x, digits = getOption("digits") - 2, ...)

## S3 method for class 'stepfun':
summary(object, ...)
```

Arguments

`x`	numeric vector giving the knots or jump locations of the step function for `stepfun()`. For the other functions, `x` is as `object` below.
`y`	numeric vector one longer than `x`, giving the heights of the function values *between* the x values.
`f`	a number between 0 and 1, indicating how interpolation outside the given x values should happen. See `approxfun`.
`ties`	Handling of tied `x` values. Either a function or the string `"ordered"`. See `approxfun`.
`right`	logical, indicating if the intervals should be closed on the right (and open on the left) or vice versa.
`Fn, object`	an R object inheriting from `"stepfun"`.
`digits`	number of significant digits to use, see `print`.
`...`	potentially further arguments (required by the generic).

Value

A function of class `"stepfun"`, say `fn`. There are methods available for summarizing (`"summary(.)"`), representing (`"print(.)"`) and plotting (`"plot(.)"`, see `plot.stepfun`) `"stepfun"` objects.

The `environment` of `fn` contains all the information needed;

`"x","y"`	the original arguments
`"n"`	number of knots (x values)
`"f"`	continuity parameter
`"yleft", "yright"`	the function values *outside* the knots
`"method"`	(always `== "constant"`, from `approxfun(.)`).

The knots are also available via `knots(fn)`.

Author(s)

Martin Maechler, ⟨maechler@stat.math.ethz.ch⟩ with some basic code from Thomas Lumley.

See Also

`ecdf` for empirical distribution functions as special step functions and `plot.stepfun` for *plotting* step functions.

`approxfun` and `splinefun`.

Examples

```
y0 <- c(1,2,4,3)
sfun0  <- stepfun(1:3, y0, f = 0)
sfun.2 <- stepfun(1:3, y0, f = .2)
sfun1  <- stepfun(1:3, y0, f = 1)
sfun1c <- stepfun(1:3, y0, right=TRUE)# hence f=1
sfun0
summary(sfun0)
```

```
summary(sfun.2)

## look at the internal structure:
unclass(sfun0)
ls(envir = environment(sfun0))

x0 <- seq(0.5,3.5, by = 0.25)
rbind(x=x0, f.f0 = sfun0(x0), f.f02= sfun.2(x0),
      f.f1 = sfun1(x0), f.f1c = sfun1c(x0))
## Identities :
stopifnot(identical(y0[-1], sfun0 (1:3)),# right = FALSE
          identical(y0[-4], sfun1c(1:3)))# right = TRUE
```

`stl` *Seasonal Decomposition of Time Series by Loess*

Description

Decompose a time series into seasonal, trend and irregular components using `loess`, acronym STL.

Usage

```
stl(x, s.window, s.degree = 0,
    t.window = NULL, t.degree = 1,
    l.window = nextodd(period), l.degree = t.degree,
    s.jump = ceiling(s.window/10),
    t.jump = ceiling(t.window/10),
    l.jump = ceiling(l.window/10),
    robust = FALSE,
    inner = if(robust)  1 else 2,
    outer = if(robust) 15 else 0,
    na.action = na.fail)
```

Arguments

`x`	univariate time series to be decomposed. This should be an object of class `"ts"` with a frequency greater than one.
`s.window`	either the character string `"periodic"` or the span (in lags) of the loess window for seasonal extraction, which should be odd. This has no default.
`s.degree`	degree of locally-fitted polynomial in seasonal extraction. Should be zero or one.
`t.window`	the span (in lags) of the loess window for trend extraction, which should be odd. If `NULL`, the default, `nextodd(ceiling((1.5*period) / (1-(1.5/s.window))))`, is taken.
`t.degree`	degree of locally-fitted polynomial in trend extraction. Should be zero or one.
`l.window`	the span (in lags) of the loess window of the low-pass filter used for each subseries. Defaults to the smallest odd integer greater than or equal to `frequency(x)` which is recommended since it prevents competition between the trend and seasonal components. If not an odd integer its given value is increased to the next odd one.

`l.degree` degree of locally-fitted polynomial for the subseries low-pass filter. Must be 0 or 1.

`s.jump, t.jump, l.jump`
integers at least one to increase speed of the respective smoother. Linear interpolation happens between every `*.jump`th value.

`robust` logical indicating if robust fitting be used in the `loess` procedure.

`inner` integer; the number of 'inner' (backfitting) iterations; usually very few (2) iterations suffice.

`outer` integer; the number of 'outer' robustness iterations.

`na.action` action on missing values.

Details

The seasonal component is found by *loess* smoothing the seasonal sub-series (the series of all January values, ...); if `s.window = "periodic"` smoothing is effectively replaced by taking the mean. The seasonal values are removed, and the remainder smoothed to find the trend. The overall level is removed from the seasonal component and added to the trend component. This process is iterated a few times. The `remainder` component is the residuals from the seasonal plus trend fit.

Several methods for the resulting class `"stl"` objects, see, `plot.stl`.

Value

`stl` returns an object of class `"stl"` with components

`time.series` a multiple time series with columns `seasonal`, `trend` and `remainder`.

`weights` the final robust weights (all one if fitting is not done robustly).

`call` the matched call.

`win` integer (length 3 vector) with the spans used for the `"s"`, `"t"`, and `"l"` smoothers.

`deg` integer (length 3) vector with the polynomial degrees for these smoothers.

`jump` integer (length 3) vector with the "jumps" (skips) used for these smoothers.

`ni` number of **i**nner iterations

`no` number of **o**uter robustness iterations

Note

This is similar to but not identical to the `stl` function in S-PLUS. The `remainder` component given by S-PLUS is the sum of the `trend` and `remainder` series from this function.

Author(s)

B.D. Ripley; Fortran code by Cleveland *et al.* (1990) from 'netlib'.

References

R. B. Cleveland, W. S. Cleveland, J.E. McRae, and I. Terpenning (1990) STL: A Seasonal-Trend Decomposition Procedure Based on Loess. *Journal of Official Statistics*, **6**, 3–73.

See Also

`plot.stl` for `stl` methods; `loess` in package **stats** (which is not actually used in `stl`).

Examples

```
plot(stl(nottem, "per"))
plot(stl(nottem, s.win = 4, t.win = 50, t.jump = 1))

plot(stllc <- stl(log(co2), s.window=21))
summary(stllc)
## linear trend, strict period.
plot(stl(log(co2), s.window="per", t.window=1000))

## Two STL plotted side by side :
        stmd <- stl(mdeaths, s.w = "per") # un-robust
summary(stmR <- stl(mdeaths, s.w = "per", robust = TRUE))
op <- par(mar = c(0, 4, 0, 3), oma = c(5, 0, 4, 0), mfcol = c(4, 2))
plot(stmd, set.pars=NULL, labels = NULL,
     main = "stl(mdeaths, s.w = \"per\",  robust = FALSE / TRUE )")
plot(stmR, set.pars=NULL)
# mark the 'outliers' :
(iO <- which(stmR $ weights  < 1e-8)) # 10 were considered outliers
sts <- stmR$time.series
points(time(sts)[iO], .8* sts[,"remainder"][iO], pch = 4, col = "red")
par(op)# reset
```

`stlmethods` *Methods for STL Objects*

Description

Methods for objects of class `stl`, typically the result of `stl`. The `plot` method does a multiple figure plot with some flexibility.

There are also (non-visible) `print` and `summary` methods.

Usage

```
## S3 method for class 'stl':
plot(x, labels = colnames(X),
     set.pars = list(mar = c(0, 6, 0, 6), oma = c(6, 0, 4, 0),
                     tck = -0.01, mfrow = c(nplot, 1)),
     main = NULL, range.bars = TRUE, ..., col.range = "light gray")
```

Arguments

`x`	`stl` object.
`labels`	character of length 4 giving the names of the component time-series.
`set.pars`	settings for `par(.)` when setting up the plot.
`main`	plot main title.
`range.bars`	logical indicating if each plot should have a bar at its right side which are of equal heights in user coordinates.
`...`	further arguments passed to or from other methods.
`col.range`	colour to be used for the range bars, if plotted. Note this appears after `...` and so cannot be abbreviated.

See Also

`plot.ts` and `stl`, particularly for examples.

`StructTS` *Fit Structural Time Series*

Description

Fit a structural model for a time series by maximum likelihood.

Usage

```
StructTS(x, type = c("level", "trend", "BSM"), init = NULL,
         fixed = NULL, optim.control = NULL)
```

Arguments

`x`	a univariate numeric time series. Missing values are allowed.
`type`	the class of structural model. If omitted, a BSM is used for a time series with `frequency(x) > 1`, and a local trend model otherwise.
`init`	initial values of the variance parameters.
`fixed`	optional numeric vector of the same length as the total number of parameters. If supplied, only `NA` entries in `fixed` will be varied. Probably most useful for setting variances to zero.
`optim.control`	List of control parameters for `optim`. Method `"L-BFGS-B"` is used.

Details

Structural time series models are (linear Gaussian) state-space models for (univariate) time series based on a decomposition of the series into a number of components. They are specified by a set of error variances, some of which may be zero.

The simplest model is the *local level* model specified by `type = "level"`. This has an underlying level μ_t which evolves by

$$\mu_{t+1} = \mu_t + \xi_t, \qquad \xi_t \sim N(0, \sigma_\xi^2)$$

The observations are

$$x_t = \mu_t + \epsilon_t, \qquad \epsilon_t \sim N(0, \sigma_\epsilon^2)$$

There are two parameters, σ_ξ^2 and σ_ϵ^2. It is an ARIMA(0,1,1) model, but with restrictions on the parameter set.

The *local linear trend model*, `type = "trend"`, has the same measurement equation, but with a time-varying slope in the dynamics for μ_t, given by

$$\mu_{t+1} = \mu_t + \nu_t + \xi_t, \qquad \xi_t \sim N(0, \sigma_\xi^2)$$

$$\nu_{t+1} = \nu_t + \zeta_t, \qquad \zeta_t \sim N(0, \sigma_\zeta^2)$$

with three variance parameters. It is not uncommon to find $\sigma_\zeta^2 = 0$ (which reduces to the local level model) or $\sigma_\xi^2 = 0$, which ensures a smooth trend. This is a restricted ARIMA(0,2,2) model.

The *basic structural model*, `type = "BSM"`, is a local trend model with an additional seasonal component. Thus the measurement equation is

$$x_t = \mu_t + \gamma_t + \epsilon_t, \qquad \epsilon_t \sim N(0, \sigma_\epsilon^2)$$

where γ_t is a seasonal component with dynamics

$$\gamma_{t+1} = -\gamma_t + \cdots + \gamma_{t-s+2} + \omega_t, \qquad \omega_t \sim N(0, \sigma_\omega^2)$$

The boundary case $\sigma_\omega^2 = 0$ corresponds to a deterministic (but arbitrary) seasonal pattern. (This is sometimes known as the 'dummy variable' version of the BSM.)

Value

A list of class `"StructTS"` with components:

`coef`	the estimated variances of the components.
`loglik`	the maximized log-likelihood. Note that as all these models are non-stationary this includes a diffuse prior for some observations and hence is not comparable with `arima` nor different types of structural models.
`data`	the time series `x`.
`residuals`	the standardized residuals.
`fitted`	a multiple time series with one component for the level, slope and seasonal components, estimated contemporaneously (that is at time t and not at the end of the series).
`call`	the matched call.
`series`	the name of the series `x`.
`code`	the `convergence` code returned by `optim`.
`model, model0`	Lists representing the Kalman Filter used in the fitting. See `KalmanLike`. `model0` is the initial state of the filter, `model` its final state.
`xtsp`	the `tsp` attributes of `x`.

Note

Optimization of structural models is a lot harder than many of the references admit. For example, the `AirPassengers` data are considered in Brockwell & Davis (1996): their solution appears to be a local maximum, but nowhere near as good a fit as that produced by `StructTS`. It is quite common to find fits with one or more variances zero, and this can include σ_ϵ^2.

References

Brockwell, P. J. & Davis, R. A. (1996). *Introduction to Time Series and Forecasting*. Springer, New York. Sections 8.2 and 8.5.

Durbin, J. and Koopman, S. J. (2001) *Time Series Analysis by State Space Methods.* Oxford University Press.

Harvey, A. C. (1989) *Forecasting, Structural Time Series Models and the Kalman Filter*. Cambridge University Press.

Harvey, A. C. (1993) *Time Series Models*. 2nd Edition, Harvester Wheatsheaf.

See Also

KalmanLike, tsSmooth.

Examples

```
## see also JohnsonJohnson, Nile and AirPassengers

trees <- window(treering, start=0)
(fit <- StructTS(trees, type = "level"))
plot(trees)
lines(fitted(fit), col = "green")
tsdiag(fit)

(fit <- StructTS(log10(UKgas), type = "BSM"))
par(mfrow = c(4, 1))
plot(log10(UKgas))
plot(cbind(fitted(fit), resids=resid(fit)), main = "UK gas consumption")

## keep some parameters fixed; trace optimizer:
StructTS(log10(UKgas), type = "BSM", fixed = c(0.1,0.001,NA,NA),
         optim.control = list(trace=TRUE))
```

---

summary.aov *Summarize an Analysis of Variance Model*

Description

Summarize an analysis of variance model.

Usage

```
## S3 method for class 'aov':
summary(object, intercept = FALSE, split,
        expand.split = TRUE, keep.zero.df = TRUE, ...)

## S3 method for class 'aovlist':
summary(object, ...)
```

Arguments

object	An object of class "aov" or "aovlist".
intercept	logical: should intercept terms be included?
split	an optional named list, with names corresponding to terms in the model. Each component is itself a list with integer components giving contrasts whose contributions are to be summed.
expand.split	logical: should the split apply also to interactions involving the factor?
keep.zero.df	logical: should terms with no degrees of freedom be included?
...	Arguments to be passed to or from other methods, for summary.aovlist including those for summary.aov.

Value

An object of class `c("summary.aov", "listof")` or `"summary.aovlist"` respectively.

For a fits with a single stratum the result will be a list of ANOVA tables, one for each response (even if there is only one response): the tables are of class `"anova"` inheriting from class `"data.frame"`. They have columns `"Df"`, `"Sum Sq"`, `"Mean Sq"`, as well as `"F value"` and `"Pr(>F)"` if there are non-zero residual degrees of freedom. There is a row for each term in the model, plus one for `"Residuals"` if there are any.

For multistratum fits the return value is a list of such summaries, one for each stratum.

Note

The use of `expand.split = TRUE` is little tested: it is always possible to set it to `FALSE` and specify exactly all the splits required.

See Also

`aov`, `summary`, `model.tables`, `TukeyHSD`

Examples

```
## From Venables and Ripley (2002) p.165.
N <- c(0,1,0,1,1,1,0,0,0,1,1,0,1,1,0,0,1,0,1,0,1,1,0,0)
P <- c(1,1,0,0,0,1,0,1,1,1,0,0,0,1,0,1,1,0,0,1,0,1,1,0)
K <- c(1,0,0,1,0,1,1,0,0,1,0,1,0,1,1,0,0,0,1,1,1,0,1,0)
yield <- c(49.5,62.8,46.8,57.0,59.8,58.5,55.5,56.0,62.8,55.8,69.5,55.0,
           62.0,48.8,45.5,44.2,52.0,51.5,49.8,48.8,57.2,59.0,53.2,56.0)
npk <- data.frame(block=gl(6,4), N=factor(N), P=factor(P),
                  K=factor(K), yield=yield)

( npk.aov <- aov(yield ~ block + N*P*K, npk) )
summary(npk.aov)
coefficients(npk.aov)

# Cochran and Cox (1957, p.164)
# 3x3 factorial with ordered factors, each is average of 12.
CC <- data.frame(
    y = c(449, 413, 326, 409, 358, 291, 341, 278, 312)/12,
    P = ordered(gl(3, 3)), N = ordered(gl(3, 1, 9))
)
CC.aov <- aov(y ~ N * P, data = CC , weights = rep(12, 9))
summary(CC.aov)

# Split both main effects into linear and quadratic parts.
summary(CC.aov, split = list(N = list(L = 1, Q = 2),
                             P = list(L = 1, Q = 2)))

# Split only the interaction
summary(CC.aov, split = list("N:P" = list(L.L = 1, Q = 2:4)))

# split on just one var
summary(CC.aov, split = list(P = list(lin = 1, quad = 2)))
summary(CC.aov, split = list(P = list(lin = 1, quad = 2)),
        expand.split=FALSE)
```

summary.glm *Summarizing Generalized Linear Model Fits*

Description

These functions are all methods for class glm or summary.glm objects.

Usage

```
## S3 method for class 'glm':
summary(object, dispersion = NULL, correlation = FALSE,
        symbolic.cor = FALSE, ...)

## S3 method for class 'summary.glm':
print(x, digits = max(3, getOption("digits") - 3),
      symbolic.cor = x$symbolic.cor,
      signif.stars = getOption("show.signif.stars"), ...)
```

Arguments

object	an object of class "glm", usually, a result of a call to glm.
x	an object of class "summary.glm", usually, a result of a call to summary.glm.
dispersion	the dispersion parameter for the family used. Either a single numerical value or NULL (the default), when it is inferred from object (see Details).
correlation	logical; if TRUE, the correlation matrix of the estimated parameters is returned and printed.
digits	the number of significant digits to use when printing.
symbolic.cor	logical. If TRUE, print the correlations in a symbolic form (see symnum) rather than as numbers.
signif.stars	logical. If TRUE, "significance stars" are printed for each coefficient.
...	further arguments passed to or from other methods.

Details

print.summary.glm tries to be smart about formatting the coefficients, standard errors, etc. and additionally gives "significance stars" if signif.stars is TRUE. The coefficients component of the result gives the estimated coefficients and their estimated standard errors, together with their ratio. This third column is labelled t ratio if the dispersion is estimated, and z ratio if the dispersion is known (or fixed by the family). A fourth column gives the two-tailed p-value corresponding to the t or z ratio based on a Student t or Normal reference distribution. (It is possible that the dispersion is not known and there are no residual degrees of freedom from which to estimate it. In that case the estimate is NaN.)

Aliased coefficients are omitted in the returned object but restored by the print method.

Correlations are printed to two decimal places (or symbolically): to see the actual correlations print summary(object)$correlation directly.

The dispersion of a GLM is not used in the fitting process, but it is needed to find standard errors. If dispersion is not supplied or NULL, the dispersion is taken as 1 for the binomial and

Poisson families, and otherwise estimated by the residual Chisquared statistic (calculated from cases with non-zero weights) divided by the residual degrees of freedom.

summary can be used with Gaussian glm fits to handle the case of a linear regression with known error variance, something not handled by summary.lm.

Value

summary.glm returns an object of class "summary.glm", a list with components

call	the component from object.
family	the component from object.
deviance	the component from object.
contrasts	the component from object.
df.residual	the component from object.
null.deviance	the component from object.
df.null	the component from object.
deviance.resid	the deviance residuals: see residuals.glm.
coefficients	the matrix of coefficients, standard errors, z-values and p-values. Aliased coefficients are omitted.
aliased	named logical vector showing if the original coefficients are aliased.
dispersion	either the supplied argument or the inferred/estimated dispersion if the latter is NULL.
df	a 3-vector of the rank of the model and the number of residual degrees of freedom, plus number of non-aliased coefficients.
cov.unscaled	the unscaled (dispersion = 1) estimated covariance matrix of the estimated coefficients.
cov.scaled	ditto, scaled by dispersion.
correlation	(only if correlation is true.) The estimated correlations of the estimated coefficients.
symbolic.cor	(only if correlation is true.) The value of the argument symbolic.cor.

See Also

glm, summary.

Examples

```
## --- Continuing the Example from  '?glm':

summary(glm.D93)
```

summary.lm *Summarizing Linear Model Fits*

Description

summary method for class "lm".

Usage

```
## S3 method for class 'lm':
summary(object, correlation = FALSE, symbolic.cor = FALSE, ...)

## S3 method for class 'summary.lm':
print(x, digits = max(3, getOption("digits") - 3),
      symbolic.cor = x$symbolic.cor,
      signif.stars = getOption("show.signif.stars"), ...)
```

Arguments

object	an object of class "lm", usually, a result of a call to lm.
x	an object of class "summary.lm", usually, a result of a call to summary.lm.
correlation	logical; if TRUE, the correlation matrix of the estimated parameters is returned and printed.
digits	the number of significant digits to use when printing.
symbolic.cor	logical. If TRUE, print the correlations in a symbolic form (see symnum) rather than as numbers.
signif.stars	logical. If TRUE, "significance stars" are printed for each coefficient.
...	further arguments passed to or from other methods.

Details

print.summary.lm tries to be smart about formatting the coefficients, standard errors, etc. and additionally gives "significance stars" if signif.stars is TRUE.

Correlations are printed to two decimal places (or symbolically): to see the actual correlations print summary(object)$correlation directly.

Value

The function summary.lm computes and returns a list of summary statistics of the fitted linear model given in object, using the components (list elements) "call" and "terms" from its argument, plus

residuals	the *weighted* residuals, the usual residuals rescaled by the square root of the weights specified in the call to lm.
coefficients	a $p \times 4$ matrix with columns for the estimated coefficient, its standard error, t-statistic and corresponding (two-sided) p-value. Aliased coefficients are omitted.
aliased	named logical vector showing if the original coefficients are aliased.

`sigma` the square root of the estimated variance of the random error

$$\hat{\sigma}^2 = \frac{1}{n-p}\sum_i w_i R_i^2,$$

where R_i is the i-th residual, `residuals[i]`.

`df` degrees of freedom, a 3-vector $(p, n-p, p*)$, the last being the number of non-aliased coefficients.

`fstatistic` (for models including non-intercept terms) a 3-vector with the value of the F-statistic with its numerator and denominator degrees of freedom.

`r.squared` R^2, the "fraction of variance explained by the model",

$$R^2 = 1 - \frac{\sum_i R_i^2}{\sum_i (y_i - y^*)^2},$$

where y^* is the mean of y_i if there is an intercept and zero otherwise.

`adj.r.squared` the above R^2 statistic "*adjusted*", penalizing for higher p.

`cov.unscaled` a $p \times p$ matrix of (unscaled) covariances of the $\hat{\beta}_j$, $j = 1, \ldots, p$.

`correlation` the correlation matrix corresponding to the above `cov.unscaled`, if `correlation = TRUE` is specified.

`symbolic.cor` (only if `correlation` is true.) The value of the argument `symbolic.cor`.

`na.action` from `object`, if present there.

See Also

The model fitting function `lm`, `summary`.

Function `coef` will extract the matrix of coefficients with standard errors, t-statistics and p-values.

Examples

```
##-- Continuing the  lm(.) example:
coef(lm.D90)# the bare coefficients
sld90 <- summary(lm.D90 <- lm(weight ~ group -1))# omitting intercept
sld90
coef(sld90)# much more
```

---

summary.manova *Summary Method for Multivariate Analysis of Variance*

Description

A `summary` method for class `"manova"`.

Usage

```
## S3 method for class 'manova':
summary(object,
        test = c("Pillai", "Wilks", "Hotelling-Lawley", "Roy"),
        intercept = FALSE, ...)
```

Arguments

object	An object of class "manova" or an aov object with multiple responses.
test	The name of the test statistic to be used. Partial matching is used so the name can be abbreviated.
intercept	logical. If TRUE, the intercept term is included in the table.
...	further arguments passed to or from other methods.

Details

The summary.manova method uses a multivariate test statistic for the summary table. Wilks' statistic is most popular in the literature, but the default Pillai–Bartlett statistic is recommended by Hand and Taylor (1987).

The table gives a transformation of the test statistic which has approximately an F distribution. The approximations used follow S-PLUS and SAS (the latter apart from some cases of the Hotelling–Lawley statistic), but many other distributional approximations exist: see Anderson (1984) and Krzanowski and Marriott (1994) for further references. All four approximate F statistics are the same when the term being tested has one degree of freedom, but in other cases that for the Roy statistic is an upper bound.

Value

A list with components

SS	A named list of sums of squares and product matrices.
Eigenvalues	A matrix of eigenvalues.
stats	A matrix of the statistics, approximate F value, degrees of freedom and P value.

References

Anderson, T. W. (1994) *An Introduction to Multivariate Statistical Analysis.* Wiley.

Hand, D. J. and Taylor, C. C. (1987) *Multivariate Analysis of Variance and Repeated Measures.* Chapman and Hall.

Krzanowski, W. J. (1988) *Principles of Multivariate Analysis. A User's Perspective.* Oxford.

Krzanowski, W. J. and Marriott, F. H. C. (1994) *Multivariate Analysis. Part I: Distributions, Ordination and Inference.* Edward Arnold.

See Also

manova, aov

Examples

```
## Example on producing plastic film from Krzanowski (1998, p. 381)
tear <- c(6.5, 6.2, 5.8, 6.5, 6.5, 6.9, 7.2, 6.9, 6.1, 6.3,
          6.7, 6.6, 7.2, 7.1, 6.8, 7.1, 7.0, 7.2, 7.5, 7.6)
gloss <- c(9.5, 9.9, 9.6, 9.6, 9.2, 9.1, 10.0, 9.9, 9.5, 9.4,
           9.1, 9.3, 8.3, 8.4, 8.5, 9.2, 8.8, 9.7, 10.1, 9.2)
opacity <- c(4.4, 6.4, 3.0, 4.1, 0.8, 5.7, 2.0, 3.9, 1.9, 5.7,
             2.8, 4.1, 3.8, 1.6, 3.4, 8.4, 5.2, 6.9, 2.7, 1.9)
Y <- cbind(tear, gloss, opacity)
rate <- factor(gl(2,10), labels=c("Low", "High"))
```

```
additive <- factor(gl(2, 5, len=20), labels=c("Low", "High"))

fit <- manova(Y ~ rate * additive)
summary.aov(fit)           # univariate ANOVA tables
summary(fit, test="Wilks") # ANOVA table of Wilks' lambda
summary(fit)               # same F statistics as single-df terms
```

`summary.nls` *Summarizing Non-Linear Least-Squares Model Fits*

Description

`summary` method for class `"nls"`.

Usage

```
## S3 method for class 'nls':
summary(object, correlation = FALSE, symbolic.cor = FALSE, ...)

## S3 method for class 'summary.nls':
print(x, digits = max(3, getOption("digits") - 3),
      symbolic.cor = x$symbolic.cor,
      signif.stars = getOption("show.signif.stars"), ...)
```

Arguments

`object`	an object of class `"nls"`.
`x`	an object of class `"summary.nls"`, usually the result of a call to `summary.nls`.
`correlation`	logical; if `TRUE`, the correlation matrix of the estimated parameters is returned and printed.
`digits`	the number of significant digits to use when printing.
`symbolic.cor`	logical. If `TRUE`, print the correlations in a symbolic form (see `symnum`) rather than as numbers.
`signif.stars`	logical. If `TRUE`, "significance stars" are printed for each coefficient.
`...`	further arguments passed to or from other methods.

Details

The distribution theory used to find the distribution of the standard errors and of the residual standard error (for t ratios) is based on linearization and is approximate, maybe very approximate.

`print.summary.nls` tries to be smart about formatting the coefficients, standard errors, etc. and additionally gives "significance stars" if `signif.stars` is `TRUE`.

Correlations are printed to two decimal places (or symbolically): to see the actual correlations print `summary(object)$correlation` directly.

Value

The function `summary.nls` computes and returns a list of summary statistics of the fitted model given in `object`, using the component `"formula"` from its argument, plus

`residuals` the *weighted* residuals, the usual residuals rescaled by the square root of the weights specified in the call to `nls`.

`coefficients` a $p \times 4$ matrix with columns for the estimated coefficient, its standard error, t-statistic and corresponding (two-sided) p-value.

`sigma` the square root of the estimated variance of the random error

$$\hat{\sigma}^2 = \frac{1}{n-p} \sum_i R_i^2,$$

where R_i is the i-th weighted residual.

`df` degrees of freedom, a 2-vector $(p, n-p)$. (Here and elsewhere n omits observations with zero weights.)

`cov.unscaled` a $p \times p$ matrix of (unscaled) covariances of the parameter estimates.

`correlation` the correlation matrix corresponding to the above `cov.unscaled`, if `correlation = TRUE` is specified and there are a non-zero number of residual degrees of freedom.

`symbolic.cor` (only if `correlation` is true.) The value of the argument `symbolic.cor`.

See Also

The model fitting function `nls`, `summary`.

Function `coef` will extract the matrix of coefficients with standard errors, t-statistics and p-values.

`summary.princomp` *Summary method for Principal Components Analysis*

Description

The `summary` method for class `"princomp"`.

Usage

```
## S3 method for class 'princomp':
summary(object, loadings = FALSE, cutoff = 0.1, ...)

## S3 method for class 'summary.princomp':
print(x, digits = 3, loadings = x$print.loadings,
      cutoff = x$cutoff, ...)
```

Arguments

`object` an object of class `"princomp"`, as from `princomp()`.

`loadings` logical. Should loadings be included?

`cutoff` numeric. Loadings below this cutoff in absolute value are shown as blank in the output.

`x` an object of class "summary.princomp".

`digits` the number of significant digits to be used in listing loadings.

`...` arguments to be passed to or from other methods.

Value

`object` with additional components `cutoff` and `print.loadings`.

See Also

`princomp`

Examples

```
summary(pc.cr <- princomp(USArrests, cor=TRUE))
print(summary(princomp(USArrests, cor=TRUE),
              loadings = TRUE, cutoff = 0.2), digits = 2)
```

---

supsmu *Friedman's SuperSmoother*

Description

Smooth the (x, y) values by Friedman's "super smoother".

Usage

```
supsmu(x, y, wt, span = "cv", periodic = FALSE, bass = 0)
```

Arguments

`x`	x values for smoothing
`y`	y values for smoothing
`wt`	case weights, by default all equal
`span`	the fraction of the observations in the span of the running lines smoother, or `"cv"` to choose this by leave-one-out cross-validation.
`periodic`	if `TRUE`, the x values are assumed to be in `[0, 1]` and of period 1.
`bass`	controls the smoothness of the fitted curve. Values of up to 10 indicate increasing smoothness.

Details

`supsmu` is a running lines smoother which chooses between three spans for the lines. The running lines smoothers are symmetric, with `k/2` data points each side of the predicted point, and values of `k` as `0.5 * n`, `0.2 * n` and `0.05 * n`, where `n` is the number of data points. If `span` is specified, a single smoother with span `span * n` is used.

The best of the three smoothers is chosen by cross-validation for each prediction. The best spans are then smoothed by a running lines smoother and the final prediction chosen by linear interpolation.

The FORTRAN code says: "For small samples (`n < 40`) or if there are substantial serial correlations between observations close in x-value, then a prespecified fixed span smoother (`span > 0`) should be used. Reasonable span values are 0.2 to 0.4."

Value

A list with components

`x`	the input values in increasing order with duplicates removed.
`y`	the corresponding y values on the fitted curve.

References

Friedman, J. H. (1984) SMART User's Guide. Laboratory for Computational Statistics, Stanford University Technical Report No. 1.

Friedman, J. H. (1984) A variable span scatterplot smoother. Laboratory for Computational Statistics, Stanford University Technical Report No. 5.

See Also

`ppr`

Examples

```
with(cars, {
    plot(speed, dist)
    lines(supsmu(speed, dist))
    lines(supsmu(speed, dist, bass = 7), lty = 2)
    })
```

`symnum` *Symbolic Number Coding*

Description

Symbolically encode a given numeric or logical vector or array. Particularly useful for "visualization" of structured matrices, e.g., correlation, sparse, or logical ones.

Usage

```
symnum(x, cutpoints = c(0.3, 0.6, 0.8, 0.9, 0.95),
       symbols= if(numeric.x) c(" ", ".", ",", "+", "*", "B") else c(".", "|"),
       legend = length(symbols) >= 3,
       na = "?", eps = 1e-5, numeric.x = is.numeric(x),
       corr = missing(cutpoints) && numeric.x,
       show.max = if(corr) "1", show.min = NULL,
       abbr.colnames = has.colnames,
       lower.triangular = corr && is.numeric(x) && is.matrix(x),
       diag.lower.tri   = corr && !is.null(show.max))
```

Arguments

x	numeric or logical vector or array.
cutpoints	numeric vector whose values cutpoints[j] $= c_j$ (*after* augmentation, see corr below) are used for intervals.
symbols	character vector, one shorter than (the *augmented*, see corr below) cutpoints. symbols[j]$= s_j$ are used as "code" for the (half open) interval $(c_j, c_{j+1}]$. When numeric.x is FALSE, i.e., by default when argument x is logical, the default is c(" ",".","|") (graphical 0 / 1 s).
legend	logical indicating if a "legend" attribute is desired.
na	character or logical. How NAs are coded. If na == FALSE, NAs are coded invisibly, *including* the "legend" attribute below, which otherwise mentions NA coding.
eps	absolute precision to be used at left and right boundary.
numeric.x	logical indicating if x should be treated as numbers, otherwise as logical.
corr	logical. If TRUE, x contains correlations. The cutpoints are augmented by 0 and 1 and abs(x) is coded.
show.max	if TRUE, or of mode character, the maximal cutpoint is coded especially.
show.min	if TRUE, or of mode character, the minimal cutpoint is coded especially.
abbr.colnames	logical, integer or NULL indicating how column names should be abbreviated (if they are); if NULL (or FALSE and x has no column names), the column names will all be empty, i.e., ""; otherwise if abbr.colnames is false, they are left unchanged. If TRUE or integer, existing column names will be abbreviated to abbreviate(*, minlength = abbr.colnames).
lower.triangular	logical. If TRUE and x is a matrix, only the *lower triangular* part of the matrix is coded as non-blank.
diag.lower.tri	logical. If lower.triangular *and* this are TRUE, the *diagonal* part of the matrix is shown.

Value

An atomic character object of class noquote and the same dimensions as x.

If legend is TRUE (as by default when there are more than two classes), the result has an attribute "legend" containing a legend of the returned character codes, in the form

$$c_1 s_1 c_2 s_2 \ldots s_n c_{n+1}$$

where c_j = cutpoints[j] and s_j = symbols[j].

Note

The optional (mostly logical) arguments all try to use smart defaults. Specifying them explicitly may lead to considerably improved output in many cases.

Author(s)

Martin Maechler ⟨maechler@stat.math.ethz.ch⟩

See Also

as.character; image

Examples

```
ii <- 0:8; names(ii) <- ii
symnum(ii, cut= 2*(0:4), sym = c(".", "-", "+", "$"))
symnum(ii, cut= 2*(0:4), sym = c(".", "-", "+", "$"), show.max=TRUE)

symnum(1:12 %% 3 == 0)# --> "|" = TRUE, "." = FALSE  for logical

## Pascal's Triangle modulo 2 -- odd and even numbers:
N <- 38
pascal <- t(sapply(0:N, function(n) round(choose(n, 0:N - (N-n)%/%2))))
rownames(pascal) <- rep("", 1+N) # <-- to improve "graphic"
symnum(pascal %% 2, symbols = c(" ", "A"), numeric = FALSE)

##-- Symbolic correlation matrices:
symnum(cor(attitude), diag = FALSE)
symnum(cor(attitude), abbr.= NULL)
symnum(cor(attitude), abbr.= FALSE)
symnum(cor(attitude), abbr.= 2)

symnum(cor(rbind(1, rnorm(25), rnorm(25)^2)))
symnum(cor(matrix(rexp(30, 1), 5, 18))) # <<-- PATTERN ! --
symnum(cm1 <- cor(matrix(rnorm(90) ,  5, 18))) # < White Noise SMALL n
symnum(cm1, diag=FALSE)
symnum(cm2 <- cor(matrix(rnorm(900), 50, 18))) # < White Noise "BIG" n
symnum(cm2, lower=FALSE)

## NA's:
Cm <- cor(matrix(rnorm(60),  10, 6)); Cm[c(3,6), 2] <- NA
symnum(Cm, show.max=NULL)

## Graphical P-values (aka "significance stars"):
pval <- rev(sort(c(outer(1:6, 10^-(1:3)))))
symp <- symnum(pval, corr=FALSE,
               cutpoints = c(0,  .001,.01,.05, .1, 1),
               symbols = c("***","**","*",".","," "))
noquote(cbind(P.val = format(pval), Signif= symp))
```

---

t.test *Student's t-Test*

Description

Performs one and two sample t-tests on vectors of data.

Usage

```
t.test(x, ...)

## Default S3 method:
t.test(x, y = NULL,
```

```
        alternative = c("two.sided", "less", "greater"),
        mu = 0, paired = FALSE, var.equal = FALSE,
        conf.level = 0.95, ...)

## S3 method for class 'formula':
t.test(formula, data, subset, na.action, ...)
```

Arguments

x	a (non-empty) numeric vector of data values.
y	an optional (non-empty) numeric vector of data values.
alternative	a character string specifying the alternative hypothesis, must be one of "two.sided" (default), "greater" or "less". You can specify just the initial letter.
mu	a number indicating the true value of the mean (or difference in means if you are performing a two sample test).
paired	a logical indicating whether you want a paired t-test.
var.equal	a logical variable indicating whether to treat the two variances as being equal. If TRUE then the pooled variance is used to estimate the variance otherwise the Welch (or Satterthwaite) approximation to the degrees of freedom is used.
conf.level	confidence level of the interval.
formula	a formula of the form lhs ~ rhs where lhs is a numeric variable giving the data values and rhs a factor with two levels giving the corresponding groups.
data	an optional matrix or data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).
subset	an optional vector specifying a subset of observations to be used.
na.action	a function which indicates what should happen when the data contain NAs. Defaults to getOption("na.action").
...	further arguments to be passed to or from methods.

Details

The formula interface is only applicable for the 2-sample tests.

alternative = "greater" is the alternative that x has a larger mean than y.

If paired is TRUE then both x and y must be specified and they must be the same length. Missing values are removed (in pairs if paired is TRUE). If var.equal is TRUE then the pooled estimate of the variance is used. By default, if var.equal is FALSE then the variance is estimated separately for both groups and the Welch modification to the degrees of freedom is used.

If the input data are effectively constant (compared to the larger of the two means) an error is generated.

Value

A list with class "htest" containing the following components:

statistic	the value of the t-statistic.
parameter	the degrees of freedom for the t-statistic.
p.value	the p-value for the test.

`conf.int`	a confidence interval for the mean appropriate to the specified alternative hypothesis.
`estimate`	the estimated mean or difference in means depending on whether it was a one-sample test or a two-sample test.
`null.value`	the specified hypothesized value of the mean or mean difference depending on whether it was a one-sample test or a two-sample test.
`alternative`	a character string describing the alternative hypothesis.
`method`	a character string indicating what type of t-test was performed.
`data.name`	a character string giving the name(s) of the data.

See Also

`prop.test`

Examples

```
t.test(1:10,y=c(7:20))      # P = .00001855
t.test(1:10,y=c(7:20, 200)) # P = .1245    -- NOT significant anymore

## Classical example: Student's sleep data
plot(extra ~ group, data = sleep)
## Traditional interface
with(sleep, t.test(extra[group == 1], extra[group == 2]))
## Formula interface
t.test(extra ~ group, data = sleep)
```

`TDist` *The Student t Distribution*

Description

Density, distribution function, quantile function and random generation for the t distribution with `df` degrees of freedom (and optional noncentrality parameter `ncp`).

Usage

```
dt(x, df, ncp = 0, log = FALSE)
pt(q, df, ncp = 0, lower.tail = TRUE, log.p = FALSE)
qt(p, df, ncp = 0, lower.tail = TRUE, log.p = FALSE)
rt(n, df, ncp = 0)
```

Arguments

`x, q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`df`	degrees of freedom (> 0, maybe non-integer). `df = Inf` is allowed. For `qt` only values of at least one are currently supported.

`ncp`	non-centrality parameter δ; currently for `pt()` and `dt()`, only for `abs(ncp) <= 37.62`.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \le x]$, otherwise, $P[X > x]$.

Details

The t distribution with `df` $= \nu$ degrees of freedom has density

$$f(x) = \frac{\Gamma((\nu+1)/2)}{\sqrt{\pi\nu}\,\Gamma(\nu/2)}(1 + x^2/\nu)^{-(\nu+1)/2}$$

for all real x. It has mean 0 (for $\nu > 1$) and variance $\frac{\nu}{\nu-2}$ (for $\nu > 2$).

The general *non-central* t with parameters (ν, δ) = `(df, ncp)` is defined as the distribution of $T_\nu(\delta) := \frac{U+\delta}{\chi_\nu/\sqrt{\nu}}$ where U and χ_ν are independent random variables, $U \sim \mathcal{N}(0,1)$, and χ^2_ν is chi-squared, see Chisquare.

The most used applications are power calculations for t-tests:
Let $T = \frac{\bar{X}-\mu_0}{S/\sqrt{n}}$ where $\bar{X}$ is the `mean` and S the sample standard deviation (`sd`) of $X_1, X_2, \ldots, X_n$ which are i.i.d. $N(\mu, \sigma^2)$. Then T is distributed as non-centrally t with `df`$= n - 1$ degrees of freedom and **n**on-**c**entrality **p**arameter `ncp`$= (\mu - \mu_0)\sqrt{n}/\sigma$.

Value

`dt` gives the density, `pt` gives the distribution function, `qt` gives the quantile function, and `rt` generates random deviates.

Invalid arguments will result in return value `NaN`, with a warning.

Source

The central `dt` is computed via an accurate formula provided by Catherine Loader (see the reference in `dbinom`).

For the non-central case of `dt`, contributed by Claus Ekstrøm based on the relationship (for $x \neq 0$) to the cumulative distribution.

For the central case of `pt`, a normal approximation in the tails, otherwise via `pbeta`.

For the non-central case of `pt` based on a C translation of

Lenth, R. V. (1989). *Algorithm AS 243* — Cumulative distribution function of the non-central t distribution, *Applied Statistics* **38**, 185–189.

For central `qt`, a C translation of

Hill, G. W. (1970) Algorithm 396: Student's t-quantiles. *Communications of the ACM*, **13(10)**, 619–620.

altered to take account of

Hill, G. W. (1981) Remark on Algorithm 396, *ACM Transactions on Mathematical Software*, **7**, 250–1.

The non-central case is done by inversion.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (Except non-central versions.)

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, volume 2, chapters 28 and 31. Wiley, New York.

See Also

df for the F distribution.

Examples

```
1 - pt(1:5, df = 1)
qt(.975, df = c(1:10,20,50,100,1000))

tt <- seq(0,10, len=21)
ncp <- seq(0,6, len=31)
ptn <- outer(tt,ncp, function(t,d) pt(t, df = 3, ncp=d))
image(tt,ncp,ptn, zlim=c(0,1),main=t.tit <- "Non-central t - Probabilities")
persp(tt,ncp,ptn, zlim=0:1, r=2, phi=20, theta=200, main=t.tit,
      xlab = "t", ylab = "noncentrality parameter", zlab = "Pr(T <= t)")

plot(function(x) dt(x, df = 3, ncp = 2), -3, 11, ylim = c(0, 0.32),
     main="Non-central t - Density", yaxs="i")
```

---

termplot *Plot regression terms*

Description

Plots regression terms against their predictors, optionally with standard errors and partial residuals added.

Usage

```
termplot(model, data=NULL, envir=environment(formula(model)),
         partial.resid=FALSE, rug=FALSE,
         terms=NULL, se=FALSE, xlabs=NULL, ylabs=NULL, main = NULL,
         col.term = 2, lwd.term = 1.5,
         col.se = "orange", lty.se = 2, lwd.se = 1,
         col.res = "gray", cex = 1, pch = par("pch"),
         col.smth = "darkred", lty.smth = 2, span.smth = 2/3,
         ask = interactive() && nb.fig < n.tms &&
               .Device != "postscript",
         use.factor.levels = TRUE, smooth = NULL, ...)
```

Arguments

model	fitted model object
data	data frame in which variables in model can be found
envir	environment in which variables in model can be found
partial.resid	logical; should partial residuals be plotted?
rug	add rugplots (jittered 1-d histograms) to the axes?
terms	which terms to plot (default NULL means all terms)
se	plot pointwise standard errors?
xlabs	vector of labels for the x axes

`ylabs` vector of labels for the y axes

`main` logical, or vector of main titles; if `TRUE`, the model's call is taken as main title, `NULL` or `FALSE` mean no titles.

`col.term, lwd.term` color and line width for the "term curve", see `lines`.

`col.se, lty.se, lwd.se` color, line type and line width for the "twice-standard-error curve" when `se = TRUE`.

`col.res, cex, pch` color, plotting character expansion and type for partial residuals, when `partial.resid = TRUE`, see `points`.

`ask` logical; if `TRUE`, the user is *ask*ed before each plot, see `par(ask=.)`.

`use.factor.levels` Should x-axis ticks use factor levels or numbers for factor terms?

`smooth` `NULL` or a function with the same arguments as `panel.smooth` to draw a smooth through the partial residuals for non-factor terms

`lty.smth,col.smth, span.smth` Passed to `smooth`

`...` other graphical parameters

Details

The model object must have a `predict` method that accepts `type=terms`, eg `glm` in the **base** package, `coxph` and `survreg` in the **survival** package.

For the `partial.resid=TRUE` option it must have a `residuals` method that accepts `type="partial"`, which `lm` and `glm` do.

The `data` argument should rarely be needed, but in some cases `termplot` may be unable to reconstruct the original data frame. Using `na.action=na.exclude` makes these problems less likely.

Nothing sensible happens for interaction terms.

See Also

For (generalized) linear models, `plot.lm` and `predict.glm`.

Examples

```
had.splines <- "package:splines" %in% search()
if(!had.splines) rs <- require(splines)
x <- 1:100
z <- factor(rep(LETTERS[1:4],25))
y <- rnorm(100,sin(x/10)+as.numeric(z))
model <- glm(y ~ ns(x,6) + z)

par(mfrow=c(2,2)) ## 2 x 2 plots for same model :
termplot(model, main = paste("termplot( ", deparse(model$call)," ...)"))
termplot(model, rug=TRUE)
termplot(model, partial=TRUE, se = TRUE, main = TRUE)
termplot(model, partial=TRUE, smooth=panel.smooth,span.smth=1/4)
if(!had.splines && rs) detach("package:splines")
```

terms *Model Terms*

Description

The function terms is a generic function which can be used to extract *terms* objects from various kinds of R data objects.

Usage

```
terms(x, ...)
```

Arguments

x	object used to select a method to dispatch.
...	further arguments passed to or from other methods.

Details

There are methods for classes "aovlist", and "terms" "formula" (see terms.formula): the default method just extracts the terms component of the object (if any).

There are print and labels methods for class "terms": the latter prints the term labels (see terms.object).

Value

An object of class c("terms", "formula") which contains the *terms* representation of a symbolic model. See terms.object for its structure.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical models.* Chapter 2 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

terms.object, terms.formula, lm, glm, formula.

terms.formula *Construct a terms Object from a Formula*

Description

This function takes a formula and some optional arguments and constructs a terms object. The terms object can then be used to construct a model.matrix.

Usage

```
## S3 method for class 'formula':
terms(x, specials = NULL, abb = NULL, data = NULL, neg.out = TRUE,
      keep.order = FALSE, simplify = FALSE, ...,
      allowDotAsName = FALSE)
```

Arguments

x	a formula.
specials	which functions in the formula should be marked as special in the terms object.
abb	Not implemented in R.
data	a data frame from which the meaning of the special symbol . can be inferred. It is unused if there is no . in the formula.
neg.out	Not implemented in R.
keep.order	a logical value indicating whether the terms should keep their positions. If FALSE the terms are reordered so that main effects come first, followed by the interactions, all second-order, all third-order and so on. Effects of a given order are kept in the order specified.
simplify	should the formula be expanded and simplified, the pre-1.7.0 behaviour?
...	further arguments passed to or from other methods.
allowDotAsName	normally . in a formula refers to the remaining variables contained in data. Exceptionally, . can be treated as a name for non-standard uses of formulae.

Details

Not all of the options work in the same way that they do in S and not all are implemented.

Value

A terms.object object is returned. The object itself is the re-ordered (unless keep.order = TRUE) formula. In all cases variables within an interaction term in the formula are re-ordered by the ordering of the "variables" attribute, which is the order in which the variables occur in the formula.

See Also

terms, terms.object

---

terms.object *Description of Terms Objects*

Description

An object of class terms holds information about a model. Usually the model was specified in terms of a formula and that formula was used to determine the terms object.

Value

The object itself is simply the formula supplied to the call of `terms.formula`. The object has a number of attributes and they are used to construct the model frame:

`factors`	A matrix of variables by terms showing which variables appear in which terms. The entries are 0 if the variable does not occur in the term, 1 if it does occur and should be coded by contrasts, and 2 if it occurs and should be coded via dummy variables for all levels (as when an intercept or lower-order term is missing). If there are no terms other than an intercept and offsets, this is `numeric(0)`.
`term.labels`	A character vector containing the labels for each of the terms in the model, except for offsets. Non-syntactic names will be quoted by backticks. Note that these are after possible re-ordering (unless argument `keep.order` was false).
`variables`	A call to `list` of the variables in the model.
`intercept`	Either 0, indicating no intercept is to be fit, or 1 indicating that an intercept is to be fit.
`order`	A vector of the same length as `term.labels` indicating the order of interaction for each term.
`response`	The index of the variable (in variables) of the response (the left hand side of the formula). Zero, if there is no response.
`offset`	If the model contains `offset` terms there is an `offset` attribute indicating which variable(s) are offsets
`specials`	If a `specials` argument was given to `terms.formula` there is a `specials` attribute, a list of vectors (one for each specified special function) giving numeric indices of the arguments of the list returned as the `variables` attribute which contain these special functions.
`dataClasses`	optional. A named character vector giving the classes (as given by `.MFclass`) of the variables used in a fit.

The object has class `c("terms", "formula")`.

Note

These objects are different from those found in S. In particular there is no `formula` attribute, instead the object is itself a formula. Thus, the mode of a terms object is different as well.

Examples of the `specials` argument can be seen in the `aov` and `coxph` functions.

See Also

`terms`, `formula`.

Examples

```
## use of specials (as used for gam() in packages mgcv and gam)
(tf <- terms(y ~ x + x:z + s(x), specials = "s"))
## Note that the "factors" attribute has variables as row names
## and term labels as column names, both as character vectors.
attr(tf, "specials")    # index 's' variable(s)
rownames(attr(tf, "factors"))[attr(tf, "specials")$s]

## we can keep the order by
terms(y ~ x + x:z + s(x), specials = "s", keep.order = TRUE)
```

`time` *Sampling Times of Time Series*

Description

`time` creates the vector of times at which a time series was sampled.

`cycle` gives the positions in the cycle of each observation.

`frequency` returns the number of samples per unit time and `deltat` the time interval between observations (see `ts`).

Usage

```
time(x, ...)
## Default S3 method:
time(x, offset=0, ...)

cycle(x, ...)
frequency(x, ...)
deltat(x, ...)
```

Arguments

`x`	a univariate or multivariate time-series, or a vector or matrix.
`offset`	can be used to indicate when sampling took place in the time unit. `0` (the default) indicates the start of the unit, `0.5` the middle and `1` the end of the interval.
`...`	extra arguments for future methods.

Details

These are all generic functions, which will use the `tsp` attribute of `x` if it exists. `time` and `cycle` have methods for class `ts` that coerce the result to that class.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`ts`, `start`, `tsp`, `window`.

`date` for clock time, `system.time` for CPU usage.

Examples

```
cycle(presidents)
# a simple series plot: c() makes the x and y arguments into vectors
plot(c(time(presidents)), c(presidents), type="l")
```

toeplitz *Form Symmetric Toeplitz Matrix*

Description

Forms a symmetric Toeplitz matrix given its first row.

Usage

```
toeplitz(x)
```

Arguments

x	the first row to form the Toeplitz matrix.

Value

The Toeplitz matrix.

Author(s)

A. Trapletti

Examples

```
x <- 1:5
toeplitz (x)
```

ts *Time-Series Objects*

Description

The function `ts` is used to create time-series objects.

`as.ts` and `is.ts` coerce an object to a time-series and test whether an object is a time series.

Usage

```
ts(data = NA, start = 1, end = numeric(0), frequency = 1,
   deltat = 1, ts.eps = getOption("ts.eps"), class = , names = )
as.ts(x, ...)
is.ts(x)
```

Arguments

data	a numeric vector or matrix of the observed time-series values. A data frame will be coerced to a numeric matrix via data.matrix.
start	the time of the first observation. Either a single number or a vector of two integers, which specify a natural time unit and a (1-based) number of samples into the time unit. See the examples for the use of the second form.
end	the time of the last observation, specified in the same way as start.
frequency	the number of observations per unit of time.
deltat	the fraction of the sampling period between successive observations; e.g., 1/12 for monthly data. Only one of frequency or deltat should be provided.
ts.eps	time series comparison tolerance. Frequencies are considered equal if their absolute difference is less than ts.eps.
class	class to be given to the result, or none if NULL or "none". The default is "ts" for a single series, c("mts", "ts") for multiple series.
names	a character vector of names for the series in a multiple series: defaults to the colnames of data, or Series 1, Series 2, ...
x	an arbitrary R object.
...	arguments passed to methods (unused for the default method).

Details

The function ts is used to create time-series objects. These are vector or matrices with class of "ts" (and additional attributes) which represent data which has been sampled at equispaced points in time. In the matrix case, each column of the matrix data is assumed to contain a single (univariate) time series. Time series must have at least one observation, and although they need not be numeric there is very limited support for non-numeric series.

Class "ts" has a number of methods. In particular arithmetic will attempt to align time axes, and subsetting to extract subsets of series can be used (e.g., EuStockMarkets[, "DAX"]). However, subsetting the first (or only) dimension will return a matrix or vector, as will matrix subsetting. Subassignment can be used to replace values but not to extend a series (see window). There is a method for t that transposes the series as a matrix (a one-column matrix if a vector) and hence returns a result that does not inherit from class "ts".

The value of argument frequency is used when the series is sampled an integral number of times in each unit time interval. For example, one could use a value of 7 for frequency when the data are sampled daily, and the natural time period is a week, or 12 when the data are sampled monthly and the natural time period is a year. Values of 4 and 12 are assumed in (e.g.) print methods to imply a quarterly and monthly series respectively.

as.ts is generic. Its default method will use the tsp attribute of the object if it has one to set the start and end times and frequency.

is.ts tests if an object is a time series. It is generic: you can write methods to handle specific classes of objects, see InternalMethods.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

tsp, frequency, start, end, time, window; print.ts, the print method for time series objects; plot.ts, the plot method for time series objects.

Examples

```
ts(1:10, frequency = 4, start = c(1959, 2)) # 2nd Quarter of 1959
print( ts(1:10, freq = 7, start = c(12, 2)), calendar = TRUE) # print.ts(.)
## Using July 1954 as start date:
gnp <- ts(cumsum(1 + round(rnorm(100), 2)),
          start = c(1954, 7), frequency = 12)
plot(gnp) # using 'plot.ts' for time-series plot

## Multivariate
z <- ts(matrix(rnorm(300), 100, 3), start=c(1961, 1), frequency=12)
class(z)
plot(z)
plot(z, plot.type="single", lty=1:3)

## A phase plot:
plot(nhtemp, c(nhtemp[-1], NA), cex = .8, col="blue",
     main = "Lag plot of New Haven temperatures")
## a clearer way to do this would be
## Not run:
plot(nhtemp, lag(nhtemp, 1), cex = .8, col="blue",
     main = "Lag plot of New Haven temperatures")
## End(Not run)
```

---

ts-methods *Methods for Time Series Objects*

Description

Methods for objects of class "ts", typically the result of ts.

Usage

```
## S3 method for class 'ts':
diff(x, lag = 1, differences = 1, ...)

## S3 method for class 'ts':
na.omit(object, ...)
```

Arguments

x	an object of class "ts" containing the values to be differenced.
lag	an integer indicating which lag to use.
differences	an integer indicating the order of the difference.
object	a univariate or multivariate time series.
...	further arguments to be passed to or from methods.

Details

The `na.omit` method omits initial and final segments with missing values in one or more of the series. 'Internal' missing values will lead to failure.

Value

For the `na.omit` method, a time series without missing values. The class of `object` will be preserved.

See Also

`diff`; `na.omit`, `na.fail`, `na.contiguous`.

`ts.plot` *Plot Multiple Time Series*

Description

Plot several time series on a common plot. Unlike `plot.ts` the series can have a different time bases, but they should have the same frequency.

Usage

```
ts.plot(..., gpars = list())
```

Arguments

`...`	one or more univariate or multivariate time series.
`gpars`	list of named graphics parameters to be passed to the plotting functions. Those commonly used can be supplied directly in `...`.

Value

None.

Note

Although this can be used for a single time series, `plot` is easier to use and is preferred.

See Also

`plot.ts`

Examples

```
ts.plot(ldeaths, mdeaths, fdeaths,
        gpars=list(xlab="year", ylab="deaths", lty=c(1:3)))
```

`ts.union` *Bind Two or More Time Series*

Description

Bind time series which have a common frequency. `ts.union` pads with `NA`s to the total time coverage, `ts.intersect` restricts to the time covered by all the series.

Usage

```
ts.intersect(..., dframe = FALSE)
ts.union(..., dframe = FALSE)
```

Arguments

`...`	two or more univariate or multivariate time series, or objects which can coerced to time series.
`dframe`	logical; if `TRUE` return the result as a data frame.

Details

As a special case, `...` can contain vectors or matrices of the same length as the combined time series of the time series present, as well as those of a single row.

Value

A time series object if `dframe` is `FALSE`, otherwise a data frame.

See Also

`cbind`.

Examples

```
ts.union(mdeaths, fdeaths)
cbind(mdeaths, fdeaths) # same as the previous line
ts.intersect(window(mdeaths, 1976), window(fdeaths, 1974, 1978))

sales1 <- ts.union(BJsales, lead = BJsales.lead)
ts.intersect(sales1, lead3 = lag(BJsales.lead, -3))
```

`tsdiag` *Diagnostic Plots for Time-Series Fits*

Description

A generic function to plot time-series diagnostics.

Usage

```
tsdiag(object, gof.lag, ...)
```

Arguments

`object`	a fitted time-series model
`gof.lag`	the maximum number of lags for a Portmanteau goodness-of-fit test
`...`	further arguments to be passed to particular methods

Details

This is a generic function. It will generally plot the residuals, often standadized, the autocorrelation function of the residuals, and the p-values of a Portmanteau test for all lags up to `gof.lag`.

The methods for `arima` and `StructTS` objects plots residuals scaled by the estimate of their (individual) variance, and use the Ljung–Box version of the portmanteau test.

Value

None. Diagnostics are plotted.

See Also

`arima`, `StructTS`, `Box.test`

Examples

```
## Not run: fit <- arima(lh, c(1,0,0))
tsdiag(fit)

## see also examples(arima)

(fit <- StructTS(log10(JohnsonJohnson), type="BSM"))
tsdiag(fit)
## End(Not run)
```

tsp *Tsp Attribute of Time-Series-like Objects*

Description

`tsp` returns the `tsp` attribute (or `NULL`). It is included for compatibility with S version 2. `tsp<-` sets the `tsp` attribute. `hasTsp` ensures `x` has a `tsp` attribute, by adding one if needed.

Usage

```
tsp(x)
tsp(x) <- value
hasTsp(x)
```

Arguments

`x`	a vector or matrix or univariate or multivariate time-series.
`value`	a numeric vector of length 3 or `NULL`.

Details

The `tsp` attribute was previously described here as `c(start(x), end(x), frequency(x))`, but this is incorrect. It gives the start time *in time units*, the end time and the frequency.

Assignments are checked for consistency.

Assigning `NULL` which removes the `tsp` attribute *and* any `"ts"` (or `"mts"`) class of `x`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`ts`, `time`, `start`.

tsSmooth *Use Fixed-Interval Smoothing on Time Series*

Description

Performs fixed-interval smoothing on a univariate time series via a state-space model. Fixed-interval smoothing gives the best estimate of the state at each time point based on the whole observed series.

Usage

```
tsSmooth(object, ...)
```

Arguments

`object`	a time-series fit. Currently only class "`StructTS`" is supported
`...`	possible arguments for future methods.

Value

A time series, with as many dimensions as the state space and results at each time point of the original series. (For seasonal models, only the current seasonal component is returned.)

Author(s)

B. D. Ripley

References

Durbin, J. and Koopman, S. J. (2001) *Time Series Analysis by State Space Methods.* Oxford University Press.

See Also

`KalmanSmooth`, `StructTS`.

For examples consult `AirPassengers`, `JohnsonJohnson` and `Nile`.

`Tukey` *The Studentized Range Distribution*

Description

Functions on the distribution of the studentized range, R/s, where R is the range of a standard normal sample of size n and s^2 is independently distributed as chi-squared with df degrees of freedom, see `pchisq`.

Usage

```
ptukey(q, nmeans, df, nranges = 1, lower.tail = TRUE, log.p = FALSE)
qtukey(p, nmeans, df, nranges = 1, lower.tail = TRUE, log.p = FALSE)
```

Arguments

`q`	vector of quantiles.
`p`	vector of probabilities.
`nmeans`	sample size for range (same for each group).
`df`	degrees of freedom for s (see below).
`nranges`	number of *groups* whose **maximum** range is considered.
`log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \le x]$, otherwise, $P[X > x]$.

Details

If n_g =nranges is greater than one, R is the *maximum* of n_g groups of nmeans observations each.

Value

ptukey gives the distribution function and qtukey its inverse, the quantile function.

Note

A Legendre 16-point formula is used for the integral of ptukey. The computations are relatively expensive, especially for qtukey which uses a simple secant method for finding the inverse of ptukey. qtukey will be accurate to the 4th decimal place.

References

Copenhaver, Margaret Diponzio and Holland, Burt S. (1988) Multiple comparisons of simple effects in the two-way analysis of variance with fixed effects. *Journal of Statistical Computation and Simulation*, **30**, 1–15.

See Also

pnorm and qnorm for the corresponding functions for the normal distribution.

Examples

```
if(interactive())
  curve(ptukey(x, nm=6, df=5), from=-1, to=8, n=101)
(ptt <- ptukey(0:10, 2, df= 5))
(qtt <- qtukey(.95, 2, df= 2:11))
## The precision may be not much more than about 8 digits:
summary(abs(.95 - ptukey(qtt,2, df = 2:11)))
```

TukeyHSD *Compute Tukey Honest Significant Differences*

Description

Create a set of confidence intervals on the differences between the means of the levels of a factor with the specified family-wise probability of coverage. The intervals are based on the Studentized range statistic, Tukey's 'Honest Significant Difference' method. There is a plot method.

Usage

```
TukeyHSD(x, which, ordered = FALSE, conf.level = 0.95, ...)
```

Arguments

x	A fitted model object, usually an aov fit.
which	A character vector listing terms in the fitted model for which the intervals should be calculated. Defaults to all the terms.
ordered	A logical value indicating if the levels of the factor should be ordered according to increasing average in the sample before taking differences. If ordered is true then the calculated differences in the means will all be positive. The significant differences will be those for which the lwr end point is positive.
conf.level	A numeric value between zero and one giving the family-wise confidence level to use.
...	Optional additional arguments. None are used at present.

Details

When comparing the means for the levels of a factor in an analysis of variance, a simple comparison using t-tests will inflate the probability of declaring a significant difference when it is not in fact present. This because the intervals are calculated with a given coverage probability for each interval but the interpretation of the coverage is usually with respect to the entire family of intervals.

John Tukey introduced intervals based on the range of the sample means rather than the individual differences. The intervals returned by this function are based on this Studentized range statistics.

Technically the intervals constructed in this way would only apply to balanced designs where there are the same number of observations made at each level of the factor. This function incorporates an adjustment for sample size that produces sensible intervals for mildly unbalanced designs.

If which specifies non-factor terms these will be dropped with a warning: if no terms are left this is a an error.

Value

A list with one component for each term requested in which. Each component is a matrix with columns diff giving the difference in the observed means, lwr giving the lower end point of the interval, upr giving the upper end point and p adj giving the p-value after adjustment for the multiple comparisons.

Author(s)

Douglas Bates

References

Miller, R. G. (1981) *Simultaneous Statistical Inference*. Springer.

Yandell, B. S. (1997) *Practical Data Analysis for Designed Experiments*. Chapman & Hall.

See Also

aov, qtukey, model.tables, simint

Examples

```
summary(fm1 <- aov(breaks ~ wool + tension, data = warpbreaks))
TukeyHSD(fm1, "tension", ordered = TRUE)
plot(TukeyHSD(fm1, "tension"))
```

`Uniform` *The Uniform Distribution*

Description

These functions provide information about the uniform distribution on the interval from `min` to `max`. `dunif` gives the density, `punif` gives the distribution function `qunif` gives the quantile function and `runif` generates random deviates.

Usage

```
dunif(x, min=0, max=1, log = FALSE)
punif(q, min=0, max=1, lower.tail = TRUE, log.p = FALSE)
qunif(p, min=0, max=1, lower.tail = TRUE, log.p = FALSE)
runif(n, min=0, max=1)
```

Arguments

`x,q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`min,max`	lower and upper limits of the distribution. Must be finite.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

If `min` or `max` are not specified they assume the default values of `0` and `1` respectively.

The uniform distribution has density

$$f(x) = \frac{1}{max - min}$$

for $min \leq x \leq max$.

For the case of $u := min == max$, the limit case of $X \equiv u$ is assumed, although there is no density in that case and `dunif` will return `NaN` (the error condition).

`runif` will not generate either of the extreme values unless `max = min` or `max-min` is small compared to `min`, and in particular not for the default arguments.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`.Random.seed` about random number generation, `rnorm`, etc for other distributions.

Examples

```
u <- runif(20)

## The following relations always hold :
punif(u) == u
dunif(u) == 1

var(runif(10000))#- ~ = 1/12 = .08333
```

---

uniroot *One Dimensional Root (Zero) Finding*

Description

The function `uniroot` searches the interval from `lower` to `upper` for a root (i.e., zero) of the function `f` with respect to its first argument.

Usage

```
uniroot(f, interval, lower = min(interval), upper = max(interval),
        tol = .Machine$double.eps^0.25, maxiter = 1000, ...)
```

Arguments

`f`	the function for which the root is sought.
`interval`	a vector containing the end-points of the interval to be searched for the root.
`lower`	the lower end point of the interval to be searched.
`upper`	the upper end point of the interval to be searched.
`tol`	the desired accuracy (convergence tolerance).
`maxiter`	the maximum number of iterations.
`...`	additional named or unnamed arguments to be passed to `f` (but beware of partial matching to other arguments).

Details

Either `interval` or both `lower` and `upper` must be specified. The function uses Fortran subroutine '"zeroin"' (from Netlib) based on algorithms given in the reference below.

Convergence is declared either if `f(x) == 0` or the change in `x` for one step of the algorithm is less than `tol` (plus an allowance for representation error in `x`).

If the algorithm does not converge in `maxiter` steps, a warning is printed and the current approximation is returned.

`f` will be called as `f(x, ...)` for a numeric value of *x*.

Value

A list with four components: `root` and `f.root` give the location of the root and the value of the function evaluated at that point. `iter` and `estim.prec` give the number of iterations used and an approximate estimated precision for `root`.

Source

Based on 'zeroin.c' in http://www.netlib.org/c/brent.shar.

References

Brent, R. (1973) *Algorithms for Minimization without Derivatives.* Englewood Cliffs, NJ: Prentice-Hall.

See Also

`polyroot` for all complex roots of a polynomial; `optimize`, `nlm`.

Examples

```
f <- function (x,a) x - a
str(xmin <- uniroot(f, c(0, 1), tol = 0.0001, a = 1/3))
str(uniroot(function(x) x*(x^2-1) + .5, low = -2, up = 2, tol = 0.0001),
    dig = 10)
str(uniroot(function(x) x*(x^2-1) + .5, low = -2, up = 2, tol = 1e-10 ),
    dig = 10)

## Find the smallest value x for which exp(x) > 0 (numerically):
r <- uniroot(function(x) 1e80*exp(x)-1e-300, c(-1000,0), tol = 1e-15)
str(r, digits= 15) ##> around -745, depending on the platform.

exp(r$r)        # = 0, but not for r$r * 0.999...
minexp <- r$r * (1 - 10*.Machine$double.eps)
exp(minexp)     # typically denormalized
```

`update` *Update and Re-fit a Model Call*

Description

`update` will update and (by default) re-fit a model. It does this by extracting the call stored in the object, updating the call and (by default) evaluating that call. Sometimes it is useful to call `update` with only one argument, for example if the data frame has been corrected.

Usage

```
update(object, ...)

## Default S3 method:
update(object, formula., ..., evaluate = TRUE)
```

Arguments

`object`	An existing fit from a model function such as `lm`, `glm` and many others.
`formula.`	Changes to the formula – see `update.formula` for details.
`...`	Additional arguments to the call, or arguments with changed values. Use `name=NULL` to remove the argument `name`.
`evaluate`	If true evaluate the new call else return the call.

Value

If `evaluate = TRUE` the fitted object, otherwise the updated call.

References

Chambers, J. M. (1992) *Linear models.* Chapter 4 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`update.formula`

Examples

```
oldcon <- options(contrasts = c("contr.treatment", "contr.poly"))
## Annette Dobson (1990) "An Introduction to Generalized Linear Models".
## Page 9: Plant Weight Data.
ctl <- c(4.17,5.58,5.18,6.11,4.50,4.61,5.17,4.53,5.33,5.14)
trt <- c(4.81,4.17,4.41,3.59,5.87,3.83,6.03,4.89,4.32,4.69)
group <- gl(2, 10, 20, labels = c("Ctl", "Trt"))
weight <- c(ctl, trt)
lm.D9 <- lm(weight ~ group)
lm.D9
summary(lm.D90 <- update(lm.D9, . ~ . - 1))
options(contrasts = c("contr.helmert", "contr.poly"))
update(lm.D9)
options(oldcon)
```

`update.formula` *Model Updating*

Description

`update.formula` is used to update model formulae. This typically involves adding or dropping terms, but updates can be more general.

Usage

```
## S3 method for class 'formula':
update(old, new, ...)
```

Arguments

`old`	a model formula to be updated.
`new`	a formula giving a template which specifies how to update.
`...`	further arguments passed to or from other methods.

Details

The function works by first identifying the *left-hand side* and *right-hand side* of the `old` formula. It then examines the `new` formula and substitutes the *lhs* of the `old` formula for any occurence of "`.`" on the left of `new`, and substitutes the *rhs* of the `old` formula for any occurence of "`.`" on the right of `new`.

Value

The updated formula is returned.

See Also

`terms`, `model.matrix`.

Examples

```
update(y ~ x,     ~ . + x2) #> y ~ x + x2
update(y ~ x, log(.) ~ . ) #> log(y) ~ x
```

`var.test` *F Test to Compare Two Variances*

Description

Performs an F test to compare the variances of two samples from normal populations.

Usage

```
var.test(x, ...)

## Default S3 method:
var.test(x, y, ratio = 1,
         alternative = c("two.sided", "less", "greater"),
         conf.level = 0.95, ...)

## S3 method for class 'formula':
var.test(formula, data, subset, na.action, ...)
```

Arguments

`x, y`	numeric vectors of data values, or fitted linear model objects (inheriting from class `"lm"`).
`ratio`	the hypothesized ratio of the population variances of `x` and `y`.
`alternative`	a character string specifying the alternative hypothesis, must be one of `"two.sided"` (default), `"greater"` or `"less"`. You can specify just the initial letter.
`conf.level`	confidence level for the returned confidence interval.
`formula`	a formula of the form `lhs ~ rhs` where `lhs` is a numeric variable giving the data values and `rhs` a factor with two levels giving the corresponding groups.
`data`	an optional matrix or data frame (or similar: see `model.frame`) containing the variables in the formula `formula`. By default the variables are taken from `environment(formula)`.
`subset`	an optional vector specifying a subset of observations to be used.
`na.action`	a function which indicates what should happen when the data contain `NA`s. Defaults to `getOption("na.action")`.
`...`	further arguments to be passed to or from methods.

Details

The null hypothesis is that the ratio of the variances of the populations from which `x` and `y` were drawn, or in the data to which the linear models `x` and `y` were fitted, is equal to `ratio`.

Value

A list with class `"htest"` containing the following components:

`statistic`	the value of the F test statistic.
`parameter`	the degrees of the freedom of the F distribtion of the test statistic.
`p.value`	the p-value of the test.
`conf.int`	a confidence interval for the ratio of the population variances.
`estimate`	the ratio of the sample variances of `x` and `y`.
`null.value`	the ratio of population variances under the null.
`alternative`	a character string describing the alternative hypothesis.
`method`	the character string `"F test to compare two variances"`.
`data.name`	a character string giving the names of the data.

See Also

`bartlett.test` for testing homogeneity of variances in more than two samples from normal distributions; `ansari.test` and `mood.test` for two rank based (nonparametric) two-sample tests for difference in scale.

Examples

```
x <- rnorm(50, mean = 0, sd = 2)
y <- rnorm(30, mean = 1, sd = 1)
var.test(x, y)                  # Do x and y have the same variance?
var.test(lm(x ~ 1), lm(y ~ 1))  # The same.
```

`varimax` *Rotation Methods for Factor Analysis*

Description

These functions 'rotate' loading matrices in factor analysis.

Usage

```
varimax(x, normalize = TRUE, eps = 1e-5)
promax(x, m = 4)
```

Arguments

`x`	A loadings matrix, with p rows and $k < p$ columns
`m`	The power used the target for `promax`. Values of 2 to 4 are recommended.
`normalize`	logical. Should Kaiser normalization be performed? If so the rows of `x` are re-scaled to unit length before rotation, and scaled back afterwards.
`eps`	The tolerance for stopping: the relative change in the sum of singular values.

Details

These seek a 'rotation' of the factors `x %*% T` that aims to clarify the structure of the loadings matrix. The matrix `T` is a rotation (possibly with reflection) for `varimax`, but a general linear transformation for `promax`, with the variance of the factors being preserved.

Value

A list with components

`loadings`	The 'rotated' loadings matrix, `x %*% rotmat`, of class `"loadings"`.
`rotmat`	The 'rotation' matrix.

References

Hendrickson, A. E. and White, P. O. (1964) Promax: a quick method for rotation to orthogonal oblique structure. *British Journal of Statistical Psychology*, **17**, 65–70.

Horst, P. (1965) *Factor Analysis of Data Matrices.* Holt, Rinehart and Winston. Chapter 10.

Kaiser, H. F. (1958) The varimax criterion for analytic rotation in factor analysis. *Psychometrika* **23**, 187–200.

Lawley, D. N. and Maxwell, A. E. (1971) *Factor Analysis as a Statistical Method*. Second edition. Butterworths.

See Also

`factanal`, `Harman74.cor`.

Examples

```
## varimax with normalize = TRUE is the default
fa <- factanal( ~., 2, data = swiss)
varimax(loadings(fa), normalize = FALSE)
promax(loadings(fa))
```

---

vcov *Calculate Variance-Covariance Matrix for a Fitted Model Object*

Description

Returns the variance-covariance matrix of the main parameters of a fitted model object.

Usage

```
vcov(object, ...)
```

Arguments

`object`	a fitted model object.
`...`	additional arguments for method functions. For the `glm` method this can be used to pass a `dispersion` parameter.

Details

This is a generic function. Functions with names beginning in `vcov.` will be methods for this function. Classes with methods for this function include: `lm`, `mlm`, `glm`, `nls`, `lme`, `gls`, `coxph` and `survreg` (the last two in package **survival**).

Value

A matrix of the estimated covariances between the parameter estimates in the linear or non-linear predictor of the model.

`Weibull` *The Weibull Distribution*

Description

Density, distribution function, quantile function and random generation for the Weibull distribution with parameters `shape` and `scale`.

Usage

```
dweibull(x, shape, scale = 1, log = FALSE)
pweibull(q, shape, scale = 1, lower.tail = TRUE, log.p = FALSE)
qweibull(p, shape, scale = 1, lower.tail = TRUE, log.p = FALSE)
rweibull(n, shape, scale = 1)
```

Arguments

`x, q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`shape, scale`	shape and scale parameters, the latter defaulting to 1.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

The Weibull distribution with `shape` parameter a and `scale` parameter σ has density given by

$$f(x) = (a/\sigma)(x/\sigma)^{a-1} \exp(-(x/\sigma)^a)$$

for $x \geq 0$. The cumulative distribution function is $F(x) = 1 - \exp(-(x/\sigma)^a)$ on $x \geq 0$, the mean is $E(X) = \sigma\Gamma(1 + 1/a)$, and the $Var(X) = \sigma^2(\Gamma(1 + 2/a) - (\Gamma(1 + 1/a))^2)$.

Value

`dweibull` gives the density, `pweibull` gives the distribution function, `qweibull` gives the quantile function, and `rweibull` generates random deviates.

Invalid arguments will result in return value `NaN`, with a warning.

Note

The cumulative hazard $H(t) = -\log(1 - F(t))$ is `-pweibull(t, a, b, lower = FALSE, log = TRUE)` which is just $H(t) = (t/b)^a$.

Source

`[dpq]weibull` are calculated directly from the definitions. `rweibull` uses inversion.

References

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, volume 1, chapter 21. Wiley, New York.

See Also

The Exponential is a special case of the Weibull distribution.

Examples

```
x <- c(0,rlnorm(50))
all.equal(dweibull(x, shape = 1), dexp(x))
all.equal(pweibull(x, shape = 1, scale = pi), pexp(x, rate = 1/pi))
## Cumulative hazard H():
all.equal(pweibull(x, 2.5, pi, lower=FALSE, log=TRUE), -(x/pi)^2.5,
          tol = 1e-15)
all.equal(qweibull(x/11, shape = 1, scale = pi), qexp(x/11, rate = 1/pi))
```

`weighted.mean` *Weighted Arithmetic Mean*

Description

Compute a weighted mean of a numeric vector.

Usage

```
weighted.mean(x, w, na.rm = FALSE)
```

Arguments

`x`	a numeric vector containing the values whose mean is to be computed.
`w`	a vector of weights the same length as `x` giving the weights to use for each element of `x`.
`na.rm`	a logical value indicating whether `NA` values in `x` should be stripped before the computation proceeds.

Details

If `w` is missing then all elements of `x` are given the same weight.

Missing values in `w` are not handled.

See Also

mean

Examples

```
## GPA from Siegel 1994
wt <- c(5,  5,  4,  1)/15
x <- c(3.7,3.3,3.5,2.8)
xm <- weighted.mean(x,wt)
```

---

weighted.residuals *Compute Weighted Residuals*

Description

Computed weighted residuals from a linear model fit.

Usage

```
weighted.residuals(obj, drop0 = TRUE)
```

Arguments

obj	R object, typically of class lm or glm.
drop0	logical. If TRUE, drop all cases with weights == 0.

Details

Weighted residuals are based on the deviance residuals, which for a lm fit are the raw residuals R_i multiplied by $\sqrt{w_i}$, where w_i are the weights as specified in lm's call.

Dropping cases with weights zero is compatible with influence and related functions.

Value

Numeric vector of length n', where n' is the number of of non-0 weights (drop0 = TRUE) or the number of observations, otherwise.

See Also

residuals, lm.influence, etc.

Examples

```
example("lm")
all.equal(weighted.residuals(lm.D9),
          residuals(lm.D9))
x <- 1:10
w <- 0:9
y <- rnorm(x)
weighted.residuals(lmxy <- lm(y ~ x, weights = w))
weighted.residuals(lmxy, drop0 = FALSE)
```

`wilcox.test` *Wilcoxon Rank Sum and Signed Rank Tests*

Description

Performs one and two sample Wilcoxon tests on vectors of data; the latter is also known as 'Mann-Whitney' test.

Usage

```
wilcox.test(x, ...)

## Default S3 method:
wilcox.test(x, y = NULL,
            alternative = c("two.sided", "less", "greater"),
            mu = 0, paired = FALSE, exact = NULL, correct = TRUE,
            conf.int = FALSE, conf.level = 0.95, ...)

## S3 method for class 'formula':
wilcox.test(formula, data, subset, na.action, ...)
```

Arguments

`x`	numeric vector of data values. Non-finite (e.g. infinite or missing) values will be omitted.
`y`	an optional numeric vector of data values.
`alternative`	a character string specifying the alternative hypothesis, must be one of `"two.sided"` (default), `"greater"` or `"less"`. You can specify just the initial letter.
`mu`	a number specifying an optional parameter used to form the null hypothesis. See Details.
`paired`	a logical indicating whether you want a paired test.
`exact`	a logical indicating whether an exact p-value should be computed.
`correct`	a logical indicating whether to apply continuity correction in the normal approximation for the p-value.
`conf.int`	a logical indicating whether a confidence interval should be computed.
`conf.level`	confidence level of the interval.
`formula`	a formula of the form `lhs ~ rhs` where `lhs` is a numeric variable giving the data values and `rhs` a factor with two levels giving the corresponding groups.
`data`	an optional matrix or data frame (or similar: see `model.frame`) containing the variables in the formula `formula`. By default the variables are taken from `environment(formula)`.
`subset`	an optional vector specifying a subset of observations to be used.
`na.action`	a function which indicates what should happen when the data contain `NA`s. Defaults to `getOption("na.action")`.
`...`	further arguments to be passed to or from methods.

Details

The formula interface is only applicable for the 2-sample tests.

If only `x` is given, or if both `x` and `y` are given and `paired` is `TRUE`, a Wilcoxon signed rank test of the null that the distribution of `x` (in the one sample case) or of `x - y` (in the paired two sample case) is symmetric about `mu` is performed.

Otherwise, if both `x` and `y` are given and `paired` is `FALSE`, a Wilcoxon rank sum test (equivalent to the Mann-Whitney test: see the Note) is carried out. In this case, the null hypothesis is that the distributions of `x` and `y` differ by a location shift of `mu` and the alternative is that they differ by some other location shift (and the one-sided alternative `"greater"` is that `x` is shifted to the right of `y`).

By default (if `exact` is not specified), an exact p-value is computed if the samples contain less than 50 finite values and there are no ties. Otherwise, a normal approximation is used.

Optionally (if argument `conf.int` is true), a nonparametric confidence interval and an estimator for the pseudomedian (one-sample case) or for the difference of the location parameters `x-y` is computed. (The pseudomedian of a distribution F is the median of the distribution of $(u + v)/2$, where u and v are independent, each with distribution F. If F is symmetric, then the pseudomedian and median coincide. See Hollander & Wolfe (1973), page 34.) If exact p-values are available, an exact confidence interval is obtained by the algorithm described in Bauer (1972), and the Hodges-Lehmann estimator is employed. Otherwise, the returned confidence interval and point estimate are based on normal approximations.

With small samples it may not be possible to achieve very high confidence interval coverages. If this happens a warning will be given and an interval with lower coverage will be substituted.

Value

A list with class `"htest"` containing the following components:

`statistic`	the value of the test statistic with a name describing it.
`parameter`	the parameter(s) for the exact distribution of the test statistic.
`p.value`	the p-value for the test.
`null.value`	the location parameter `mu`.
`alternative`	a character string describing the alternative hypothesis.
`method`	the type of test applied.
`data.name`	a character string giving the names of the data.
`conf.int`	a confidence interval for the location parameter. (Only present if argument `conf.int = TRUE`.)
`estimate`	an estimate of the location parameter. (Only present if argument `conf.int = TRUE`.)

Warning

This function can use large amounts of memory and stack (and even crash R if the stack limit is exceeded) if `exact = TRUE` and one sample is large (several thousands or more).

Note

The literature is not unanimous about the definitions of the Wilcoxon rank sum and Mann-Whitney tests. The two most common definitions correspond to the sum of the ranks of the first sample with the minimum value subtracted or not: R subtracts and S-PLUS does not, giving a value which is

larger by $m(m+1)/2$ for a first sample of size m. (It seems Wilcoxon's original paper used the unadjusted sum of the ranks but subsequent tables subtracted the minimum.)

R's value can also be computed as the number of all pairs `(x[i], y[j])` for which `y[j]` is not greater than `x[i]`, the most common definition of the Mann-Whitney test.

References

David F. Bauer (1972), Constructing confidence sets using rank statistics. *Journal of the American Statistical Association* **67**, 687–690.

Myles Hollander & Douglas A. Wolfe (1973), *Nonparametric statistical inference*. New York: John Wiley & Sons. Pages 27–33 (one-sample), 68–75 (two-sample).
Or second edition (1999).

See Also

`psignrank`, `pwilcox`.

`wilcox.exact` in **exactRankTests** covers much of the same ground, but also produces exact p-values in the presence of ties.

`wilcox_test` in package **coin** for exact and approximate *conditional* p-values for the Wilcoxon tests.

`kruskal.test` for testing homogeneity in location parameters in the case of two or more samples; `t.test` for an alternative under normality assumptions [or large samples]

Examples

```
## One-sample test.
## Hollander & Wolfe (1973), 29f.
## Hamilton depression scale factor measurements in 9 patients with
##  mixed anxiety and depression, taken at the first (x) and second
##  (y) visit after initiation of a therapy (administration of a
##  tranquilizer).
x <- c(1.83,  0.50,  1.62,  2.48, 1.68, 1.88, 1.55, 3.06, 1.30)
y <- c(0.878, 0.647, 0.598, 2.05, 1.06, 1.29, 1.06, 3.14, 1.29)
wilcox.test(x, y, paired = TRUE, alternative = "greater")
wilcox.test(y - x, alternative = "less")    # The same.
wilcox.test(y - x, alternative = "less",
            exact = FALSE, correct = FALSE) # H&W large sample
                                            # approximation

## Two-sample test.
## Hollander & Wolfe (1973), 69f.
## Permeability constants of the human chorioamnion (a placental
##  membrane) at term (x) and between 12 to 26 weeks gestational
##  age (y).  The alternative of interest is greater permeability
##  of the human chorioamnion for the term pregnancy.
x <- c(0.80, 0.83, 1.89, 1.04, 1.45, 1.38, 1.91, 1.64, 0.73, 1.46)
y <- c(1.15, 0.88, 0.90, 0.74, 1.21)
wilcox.test(x, y, alternative = "g")        # greater
wilcox.test(x, y, alternative = "greater",
            exact = FALSE, correct = FALSE) # H&W large sample
                                            # approximation

wilcox.test(rnorm(10), rnorm(10, 2), conf.int = TRUE)
```

```
## Formula interface.
boxplot(Ozone ~ Month, data = airquality)
wilcox.test(Ozone ~ Month, data = airquality,
            subset = Month %in% c(5, 8))
```

`Wilcoxon` *Distribution of the Wilcoxon Rank Sum Statistic*

Description

Density, distribution function, quantile function and random generation for the distribution of the Wilcoxon rank sum statistic obtained from samples with size `m` and `n`, respectively.

Usage

```
dwilcox(x, m, n, log = FALSE)
pwilcox(q, m, n, lower.tail = TRUE, log.p = FALSE)
qwilcox(p, m, n, lower.tail = TRUE, log.p = FALSE)
rwilcox(nn, m, n)
```

Arguments

`x, q`	vector of quantiles.
`p`	vector of probabilities.
`nn`	number of observations. If `length(nn) > 1`, the length is taken to be the number required.
`m, n`	numbers of observations in the first and second sample, respectively. Can be vectors of positive integers.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

This distribution is obtained as follows. Let `x` and `y` be two random, independent samples of size `m` and `n`. Then the Wilcoxon rank sum statistic is the number of all pairs `(x[i], y[j])` for which `y[j]` is not greater than `x[i]`. This statistic takes values between `0` and `m * n`, and its mean and variance are `m * n / 2` and `m * n * (m + n + 1) / 12`, respectively.

If any of the first three arguments are vectors, the recycling rule is used to do the calculations for all combinations of the three up to the length of the longest vector.

Value

`dwilcox` gives the density, `pwilcox` gives the distribution function, `qwilcox` gives the quantile function, and `rwilcox` generates random deviates.

Warning

These functions can use large amounts of memory and stack (and even crash R if the stack limit is exceeded and stack-checking is not in place) if one sample is large (several thousands or more).

Note

S-PLUS uses a different (but equivalent) definition of the Wilcoxon statistic: see `wilcox.test` for details.

Author(s)

Kurt Hornik

Source

These are calculated via recursion, based on `cwilcox(k, m, n)`, the number of choices with statistic `k` from samples of size `m` and `n`, which is itself calculated recursively and the results cached. Then `dwilcox` and `pwilcox` sum appropriate values of `cwilcox`, and `qwilcox` is based on inversion.

`rwilcox` generates a random permutation of ranks and evaluates the statistic.

See Also

`wilcox.test` to calculate the statistic from data, find p values and so on.

`dsignrank` etc, for the distribution of the *one-sample* Wilcoxon signed rank statistic.

Examples

```
x <- -1:(4*6 + 1)
fx <- dwilcox(x, 4, 6)
Fx <- pwilcox(x, 4, 6)

layout(rbind(1,2),width=1,heights=c(3,2))
plot(x, fx,type='h', col="violet",
     main= "Probabilities (density) of Wilcoxon-Statist.(n=6,m=4)")
plot(x, Fx,type="s", col="blue",
     main= "Distribution of Wilcoxon-Statist.(n=6,m=4)")
abline(h=0:1, col="gray20",lty=2)
layout(1)# set back

N <- 200
hist(U <- rwilcox(N, m=4,n=6), breaks=0:25 - 1/2, border="red", col="pink",
     sub = paste("N =",N))
mtext("N * f(x),  f() = true \"density\"", side=3, col="blue")
 lines(x, N*fx, type='h', col='blue', lwd=2)
points(x, N*fx, cex=2)

## Better is a Quantile-Quantile Plot
qqplot(U, qw <- qwilcox((1:N - 1/2)/N, m=4,n=6),
       main = paste("Q-Q-Plot of empirical and theoretical quantiles",
                    "Wilcoxon Statistic,  (m=4, n=6)",sep="\n"))
n <- as.numeric(names(print(tU <- table(U))))
text(n+.2, n+.5, labels=tU, col="red")
```

window *Time Windows*

Description

window is a generic function which extracts the subset of the object x observed between the times start and end. If a frequency is specified, the series is then re-sampled at the new frequency.

Usage

```
window(x, ...)
## S3 method for class 'ts':
window(x, ...)
## Default S3 method:
window(x, start = NULL, end = NULL,
       frequency = NULL, deltat = NULL, extend = FALSE, ...)

window(x, ...) <- value
## S3 method for class 'ts':
window(x, start, end, frequency, deltat, ...) <- value
```

Arguments

x	a time-series (or other object if not replacing values).
start	the start time of the period of interest.
end	the end time of the period of interest.
frequency, deltat	the new frequency can be specified by either (or both if they are consistent).
extend	logical. If true, the start and end values are allowed to extend the series. If false, attempts to extend the series give a warning and are ignored.
...	further arguments passed to or from other methods.
value	replacement values.

Details

The start and end times can be specified as for ts. If there is no observation at the new start or end, the immediately following (start) or preceding (end) observation time is used.

The replacement function has a method for ts objects, and is allowed to extend the series (with a warning). There is no default method.

Value

The value depends on the method. window.default will return a vector or matrix with an appropriate tsp attribute.

window.ts differs from window.default only in ensuring the result is a ts object.

If extend = TRUE the series will be padded with NAs if needed.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`time`, `ts`.

Examples

```
window(presidents, 1960, c(1969,4)) # values in the 1960's
window(presidents, deltat=1)  # All Qtr1s
window(presidents, start=c(1945,3), deltat=1)  # All Qtr3s
window(presidents, 1944, c(1979,2), extend=TRUE)

pres <- window(presidents, 1945, c(1949,4)) # values in the 1940's
window(pres, 1945.25, 1945.50) <- c(60, 70)
window(pres, 1944, 1944.75) <- 0 # will generate a warning
window(pres, c(1945,4), c(1949,4), freq=1) <- 85:89
pres
```

---

xtabs *Cross Tabulation*

Description

Create a contingency table from cross-classifying factors, usually contained in a data frame, using a formula interface.

Usage

```
xtabs(formula = ~., data = parent.frame(), subset, na.action,
      exclude = c(NA, NaN), drop.unused.levels = FALSE)
```

Arguments

`formula`	a formula object with the cross-classifying variables, separated by +, on the right hand side. Interactions are not allowed. On the left hand side, one may optionally give a vector or a matrix of counts; in the latter case, the columns are interpreted as corresponding to the levels of a variable. This is useful if the data has already been tabulated, see the examples below.
`data`	an optional matrix or data frame (or similar: see `model.frame`) containing the variables in the formula `formula`. By default the variables are taken from `environment(formula)`.
`subset`	an optional vector specifying a subset of observations to be used.
`na.action`	a function which indicates what should happen when the data contain `NA`s.
`exclude`	a vector of values to be excluded when forming the set of levels of the classifying factors.
`drop.unused.levels`	a logical indicating whether to drop unused levels in the classifying factors. If this is `FALSE` and there are unused levels, the table will contain zero marginals, and a subsequent chi-squared test for independence of the factors will not work.

Details

There is a `summary` method for contingency table objects created by `table` or `xtabs`, which gives basic information and performs a chi-squared test for independence of factors (note that the function `chisq.test` currently only handles 2-d tables).

If a left hand side is given in `formula`, its entries are simply summed over the cells corresponding to the right hand side; this also works if the lhs does not give counts.

Value

A contingency table in array representation of class `c("xtabs", "table")`, with a `"call"` attribute storing the matched call.

See Also

`table` for "traditional" cross-tabulation, and `as.data.frame.table` which is the inverse operation of `xtabs` (see the `DF` example below).

Examples

```
## 'esoph' has the frequencies of cases and controls for all levels of
## the variables 'agegp', 'alcgp', and 'tobgp'.
xtabs(cbind(ncases, ncontrols) ~ ., data = esoph)
## Output is not really helpful ... flat tables are better:
ftable(xtabs(cbind(ncases, ncontrols) ~ ., data = esoph))
## In particular if we have fewer factors ...
ftable(xtabs(cbind(ncases, ncontrols) ~ agegp, data = esoph))

## This is already a contingency table in array form.
DF <- as.data.frame(UCBAdmissions)
## Now 'DF' is a data frame with a grid of the factors and the counts
## in variable 'Freq'.
DF
## Nice for taking margins ...
xtabs(Freq ~ Gender + Admit, DF)
## And for testing independence ...
summary(xtabs(Freq ~ ., DF))

## Create a nice display for the warp break data.
warpbreaks$replicate <- rep(1:9, len = 54)
ftable(xtabs(breaks ~ wool + tension + replicate, data = warpbreaks))
```

Chapter 8

The `tools` package

`tools-package` *Tools for Package Development*

Description

Tools for package development, administration and documentation

Details

This package contains tools for manipulating R packages and their documentation.

For a complete list of functions, use `library(help="tools")`.

Author(s)

Kurt Hornik and Friedrich Leisch

Maintainer: R Core Team ⟨R-core@r-project.org⟩

`buildVignettes` *List and Build Package Vignettes*

Description

Run `Sweave` and `texi2dvi` on all vignettes of a package.

Usage

```
buildVignettes(package, dir, lib.loc = NULL, quiet = TRUE)
pkgVignettes(package, dir, lib.loc = NULL)
```

Arguments

package	a character string naming an installed package. If given, Sweave files are searched in subdirectory doc.
dir	a character string specifying the path to a package's root source directory. This subdirectory inst/doc is searched for Sweave files.
lib.loc	a character vector of directory names of R libraries, or NULL. The default value of NULL corresponds to all libraries currently known. The specified library trees are used to to search for package.
quiet	logical. Run Sweave and texi2dvi in quiet mode.

Value

buildVignettes is called for its side effect of creating the PDF versions of all vignettes.

pkgVignettes returns an object of class "pkgVignettes".

---

charsets *Conversion Tables between Character Sets*

Description

charset_to_Unicode is a matrix of Unicode points with columns for the common 8-bit encodings.

Adobe_glyphs is a dataframe which gives Adobe glyph names for Unicode points. It has two character columns, "adobe" and "unicode" (a 4-digit hex representation).

Usage

```
charset_to_Unicode

Adobe_glyphs
```

Details

charset_to_Unicode is an integer matrix of class c("noquote", "noquote") so prints in hexadecimal. The mappings are those used by libiconv: there are differences in the way quotes and minus/hyphen are mapped between sources (and the postscript encoding files use a different mapping).

Adobe_glyphs include all the Adobe glyph names which correspond to single Unicode characters. It is sorted by Unicode point and within a point alphabetically on the glyph(there can be more than one name for a Unicode point). The data are in the file 'R_HOME/share/encodings/Adobe_glyphlist'.

Source

http://partners.adobe.com/public/developer/en/opentype/glyphlist.txt

Examples

```
## find Adobe names for ISOLatin2 chars.
latin2 <- charset_to_Unicode[, "ISOLatin2"]
aUnicode <- as.numeric(paste("0x", Adobe_glyphs$unicode, sep=""))
keep <- aUnicode %in% latin2
aUnicode <- aUnicode[keep]
aAdobe <- Adobe_glyphs[keep, 1]
## first match
aLatin2 <- aAdobe[match(latin2, aUnicode)]
## all matches
bLatin2 <- lapply(1:256, function(x) aAdobe[aUnicode == latin2[x]])
format(bLatin2, justify="none")
```

---

checkFF *Check Foreign Function Calls*

Description

Performs checks on calls to compiled code from R code. Currently only whether the interface functions such as `.C` and `.Fortran` are called with a "`NativeSymbolInfo`" first argument or with argument `PACKAGE` specified, which is highly recommended to avoid name clashes in foreign function calls.

Usage

```
checkFF(package, dir, file, lib.loc = NULL,
        verbose = getOption("verbose"))
```

Arguments

`package`	a character string naming an installed package. If given, the installed R code of the package is checked.
`dir`	a character string specifying the path to a package's root source directory. This should contain the subdirectory `R` (for R code). Only used if `package` is not given.
`file`	the name of a file containing R code to be checked. Used if neither `package` nor `dir` are given.
`lib.loc`	a character vector of directory names of R libraries, or `NULL`. The default value of `NULL` corresponds to all libraries currently known. The specified library trees are used to to search for `package`.
`verbose`	a logical. If `TRUE`, additional diagnostics are printed (and the result is returned invisibly).

Details

Note that we can only check if the `name` argument is a symbol or a character string, not what class of object the symbol resolves to at run-time.

If the package has a namespace and *if* that contains a `useDynLib` directive, calls in top-level functions in the package are not reported as their symbols will be preferentially looked up in the DLL named in the first `useDynLib` directive.

Value

An object of class "checkFF", which currently is a list of the (parsed) foreign function calls with a character first argument and no PACKAGE argument.

There is a print method to display the information contained in such objects.

Warning

This function is still experimental. Both name and interface might change in future versions.

See Also

.C, .Fortran; Foreign.

Examples

```
checkFF(package = "stats", verbose = TRUE)
```

---

checkMD5sums *Check and Create MD5 Checksum Files*

Description

checkMD5sums checks the files against a file MD5.

Usage

```
checkMD5sums(pkg, dir)
```

Arguments

pkg	the name of an installed package
dir	the path to the top-level directory of an installed package.

Details

The file 'MD5' which is created is in a format which can be checked by md5sum -c MD5 if a suitable command-line version of md5sum is available. (One is supplied in the bundle at http://www.murdoch-sutherland.com/Rtools/tools.zip.)

If dir is missing, an installed package of name pkg is searched for.

The private function tools:::.installMD5sums is used to create MD5 files in the Windows build.

Value

checkMD5sums returns a logical, NA if there is no MD5 file to be checked.

See Also

md5sum

checkTnF *Check R Packages or Code for T/F*

Description

Checks the specified R package or code file for occurrences of `T` or `F`, and gathers the expression containing these. This is useful as in R `T` and `F` are just variables which are set to the logicals `TRUE` and `FALSE` by default, but are not reserved words and hence can be overwritten by the user. Hence, one should always use `TRUE` and `FALSE` for the logicals.

Usage

```
checkTnF(package, dir, file, lib.loc = NULL)
```

Arguments

`package`	a character string naming an installed package. If given, the installed R code and the examples in the documentation files of the package are checked. R code installed as an image file cannot be checked.
`dir`	a character string specifying the path to a package's root source directory. This must contain the subdirectory 'R' (for R code), and should also contain 'man' (for documentation). Only used if `package` is not given. If used, the R code files and the examples in the documentation files are checked.
`file`	the name of a file containing R code to be checked. Used if neither `package` nor `dir` are given.
`lib.loc`	a character vector of directory names of R libraries, or `NULL`. The default value of `NULL` corresponds to all libraries currently known. The specified library trees are used to to search for `package`.

Value

An object of class `"checkTnF"` which is a list containing, for each file where occurences of `T` or `F` were found, a list with the expressions containing these occurrences. The names of the list are the corresponding file names.

There is a `print` method for nicely displaying the information contained in such objects.

Warning

This function is still experimental. Both name and interface might change in future versions.

checkVignettes *Check Package Vignettes*

Description

Check all Sweave files of a package by running Sweave and/or Stangle on them. All R source code files found after the tangling step are sourceed to check whether all code can be executed without errors.

Usage

```
checkVignettes(package, dir, lib.loc = NULL, tangle = TRUE,
               weave = TRUE, workdir = c("tmp", "src", "cur"),
               keepfiles = FALSE)
```

Arguments

package	a character string naming an installed package. If given, Sweave files are searched in subdirectory doc.
dir	a character string specifying the path to a package's root source directory. This subdirectory inst/doc is searched for Sweave files.
lib.loc	a character vector of directory names of R libraries, or NULL. The default value of NULL corresponds to all libraries currently known. The specified library trees are used to to search for package.
tangle	Perform a tangle and source the extraced code?
weave	Perform a weave?
workdir	Directory used as working directory while checking the vignettes. If "tmp" then a temporary directory is created, this is the default. If "src" then the directory containing the vignettes itself is used, if "cur" then the current working directory of R is used.
keepfiles	Delete file in temporary directory? This option is ignored when workdir!="tmp".

Value

An object of class "checkVignettes" which is a list with the error messages found during the tangle and weave steps. There is a print method for nicely displaying the information contained in such objects.

codoc *Check Code/Documentation Consistency*

Description

Find inconsistencies between actual and documented "structure" of R objects in a package. codoc compares names and optionally also corresponding positions and default values of the arguments of functions. codocClasses and codocData compare slot names of S4 classes and variable names of data sets, respectively.

Usage

```
codoc(package, dir, lib.loc = NULL,
      use.values = NULL, verbose = getOption("verbose"))
codocClasses(package, lib.loc = NULL)
codocData(package, lib.loc = NULL)
```

Arguments

package	a character string naming an installed package.
dir	a character string specifying the path to a package's root source directory. This must contain the subdirectories 'man' with R documentation sources (in Rd format) and 'R' with R code. Only used if package is not given.
lib.loc	a character vector of directory names of R libraries, or NULL. The default value of NULL corresponds to all libraries currently known. The specified library trees are used to to search for package.
use.values	if FALSE, do not use function default values when comparing code and docs. Otherwise, compare *all* default values if TRUE, and only the ones documented in the usage otherwise (default).
verbose	a logical. If TRUE, additional diagnostics are printed.

Details

The purpose of codoc is to check whether the documented usage of function objects agrees with their formal arguments as defined in the R code. This is not always straightforward, in particular as the usage information for methods to generic functions often employs the name of the generic rather than the method.

The following algorithm is used. If an installed package is used, it is loaded (unless it is the base package), after possibly detaching an already loaded version of the package. Otherwise, if the sources are used, the R code files of the package are collected and sourced in a new environment. Then, the usage sections of the Rd files are extracted and parsed "as much as possible" to give the formals documented. For interpreted functions in the code environment, the formals are compared between code and documentation according to the values of the argument use.values. Synopsis sections are used if present; their occurrence is reported if verbose is true.

If a package has a namespace both exported and unexported objects are checked, as well as registered S3 methods. (In the unlikely event of differences the order is exported objects in the package, registered S3 methods and finally objects in the namespace and only the first found is checked.)

Currently, the R documentation format has no high-level markup for the basic "structure" of classes and data sets (similar to the usage sections for function synopses). Variable names for data frames in documentation objects obtained by suitably editing "shells" created by prompt are recognized by codocData and used provided that the documentation object is for a single data frame (i.e., only has one alias). codocClasses analogously handles slot names for classes in documentation objects obtained by editing shells created by promptClass.

Help files named pkg_name-defunct.Rd for the appropriate pkg_name are checked more loosely, as they may have undocumented arguments.

Value

codoc returns an object of class "codoc". Currently, this is a list which, for each Rd object in the package where an inconsistency was found, contains an element with a list of the mismatches

(which in turn are lists with elements `code` and `docs`, giving the corresponding arguments obtained from the function's code and documented usage).

`codocClasses` and `codocData` return objects of class `"codocClasses"` and `"codocData"`, respectively, with a structure similar to class `"codoc"`.

There are `print` methods for nicely displaying the information contained in such objects.

Warning

Both `codocClasses` and `codocData` are still experimental. Names, interfaces and values might change in future versions.

Note

The default for `use.values` has been changed from `FALSE` to `NULL`, for R versions 1.9.0 and later.

See Also

`undoc`, `QC`

---

delimMatch *Delimited Pattern Matching*

Description

Match delimited substrings in a character vector, with proper nesting.

Usage

```
delimMatch(x, delim = c("{", "}"), syntax = "Rd")
```

Arguments

`x`	a character vector.
`delim`	a character vector of length 2 giving the start and end delimiters. Future versions might allow for arbitrary regular expressions.
`syntax`	currently, always the string '`"Rd"`' indicating Rd syntax (i.e., '`%`' starts a comment extending till the end of the line, and '`\`' escapes). Future versions might know about other syntaxes, perhaps via "syntax tables" allowing to flexibly specify comment, escape, and quote characters.

Value

An integer vector of the same length as `x` giving the starting position (in characters) of the first match, or -1 if there is none, with attribute `"match.length"` giving the length (in characters) of the matched text (or -1 for no match).

See Also

`regexpr` for "simple" pattern matching.

Examples

```
x <- c("\value{foo}", "function(bar)")
delimMatch(x)
delimMatch(x, c("(", ")"))
```

---

encoded_text_to_latex

Translate non-ASCII Text to LaTeX Escapes

Description

Translate non-ASCII characters in text to LaTeX escape sequences.

Usage

```
encoded_text_to_latex(x, encoding = c("latin1", "latin2", "latin9",
                                      "UTF-8", "utf8"))
```

Arguments

x	a character vector.
encoding	the encoding to be assumed. "latin9" is officially ISO-8859-15 or Latin-9, but known as latin9 to LaTeX's inputenc package.

Details

Non-ASCII characters in x are replaced by an appropriate LaTeX escape sequence, or ? if there is no appropriate sequence.

Even if there is an appropriate sequence, it may not be supported by the font in use. Hyphen is mapped to
-.

Value

A character vector of the same length as x.

See Also

iconv

Examples

```
x <- "fa\xE7ile"
encoded_text_to_latex(x, "latin1")
## Not run:
## create a tex file to show the upper half of 8-bit charsets
x <- rawToChar(as.raw(160:255), multiple=TRUE)
(x <- matrix(x, ncol=16, byrow=TRUE))
xx <- x
xx[] <- encoded_text_to_latex(x, "latin1") # or latin2 or latin9
xx <- apply(xx, 1, paste, collapse="&")
con <- file("test-encoding.tex", "w")
```

```
header <- c(
"\documentclass{article}",
"\usepackage[T1]{fontenc}",
"\usepackage{Rd}",
"\begin{document}",
"\HeaderA{test}{}{test}",
"\begin{Details}\relax",
"\Tabular{ccccccccccccccc}{")
trailer <- c("}", "\end{Details}", "\end{document}")
writeLines(header, con)
writeLines(paste(xx, "\\", sep=""), con)
writeLines(trailer, con)
close(con)
## and some UTF_8 chars
x <- intToUtf8(as.integer(
    c(160:383,0x0192,0x02C6,0x02C7,0x02CA,0x02D8,
      0x02D9, 0x02DD, 0x200C, 0x2018, 0x2019, 0x201C,
      0x201D, 0x2020, 0x2022, 0x2026, 0x20AC)),
               multiple=TRUE)
x <- matrix(x, ncol=16, byrow=TRUE)
xx <- x
xx[] <- encoded_text_to_latex(x, "UTF-8")
xx <- apply(xx, 1, paste, collapse="&")
con <- file("test-utf8.tex", "w")
writeLines(header, con)
writeLines(paste(xx, "\\", sep=""), con)
writeLines(trailer, con)
close(con)
## End(Not run)
```

`fileutils` *File Utilities*

Description

Utilities for testing and listing files, and manipulating file paths.

Usage

```
file_path_as_absolute(x)
file_path_sans_ext(x)
file_test(op, x, y)
list_files_with_exts(dir, exts, all.files = FALSE, full.names = TRUE)
list_files_with_type(dir, type, all.files = FALSE, full.names = TRUE,
                     OS_subdirs = .OStype())
```

Arguments

`x,y`	character vectors giving file paths.
`op`	a character string specifying the test to be performed. Unary tests (only `x` is used) are `"-f"` (existence and not being a directory) and `"-d"` (existence and directory); binary tests are `"-nt"` (newer than, using the modification dates) and `"-ot"`.

`dir`	a character string with the path name to a directory.
`exts`	a character vector of possible file extensions.
`all.files`	a logical. If `FALSE` (default), only visible files are considered; if `TRUE`, all files are used.
`full.names`	a logical indicating whether the full paths of the files found are returned (default), or just the file names.
`type`	a character string giving the "type" of the files to be listed, as characterized by their extensions. Currently, possible values are `"code"` (R code), `"data"` (data sets), `"demo"` (demos), `"docs"` (R documentation), and `"vignette"` (vignettes).
`OS_subdirs`	a character vector with the names of OS-specific subdirectories to possibly include in the listing of R code and documentation files. By default, the value of the environment variable `R_OSTYPE`, or if this is empty, the value of `.Platform$OS.type`, is used.

Details

`file_path_as_absolute` turns a possibly relative file path absolute, performing tilde expansion if necessary. Currently, only a single existing path can be given.

`file_path_sans_ext` returns the file paths without extensions. (Only purely alphanumeric extensions are recognized.)

`file_test` performs shell-style file tests. Note that `file.exists` only tests for existence (`test -e` on some systems) but not for not being a directory.

`list_files_with_exts` returns the paths or names of the files in directory `dir` with extension matching one of the elements of `exts`. Note that by default, full paths are returned, and that only visible files are used.

`list_files_with_type` returns the paths of the files in `dir` of the given "type", as determined by the extensions recognized by R. When listing R code and documentation files, files in OS-specific subdirectories are included if present according to the value of `OS_subdirs`. Note that by default, full paths are returned, and that only visible files are used.

See Also

`file.path`, `file.info`, `list.files`

Examples

```
dir <- file.path(R.home(), "library", "stats")
file_test("-d", dir)
file_test("-nt", file.path(dir, "R"), file.path(dir, "demo"))
list_files_with_exts(file.path(dir, "demo"), "R")
list_files_with_type(file.path(dir, "demo"), "demo") # the same
file_path_sans_ext(list.files(file.path(R.home(), "modules")))
```

getDepList *Functions to Retrieve Dependency Information*

Description

Given a dependency matrix, will create a DependsList object for that package which will include the dependencies for that matrix, which ones are installed, which unresolved dependencies were found online, which unresolved dependencies were not found online, and any R dependencies.

Usage

```
getDepList(depMtrx, instPkgs, recursive = TRUE, local = TRUE,
           reduce = TRUE, lib.loc = NULL)

pkgDepends(pkg, recursive = TRUE, local = TRUE, reduce = TRUE,
           lib.loc = NULL)
```

Arguments

depMtrx	A dependency matrix as from package.dependencies
pkg	The name of the package
instPkgs	A matrix specifying all packages installed on the local system, as from installed.packages
recursive	Whether or not to include indirect dependencies
local	Whether or not to search only locally
reduce	Whether or not to collapse all sets of dependencies to a minimal value
lib.loc	What libraries to use when looking for installed packages. NULL indicates all library directories in the user's .libPaths().

Details

The function pkgDepends is a convenience function which wraps getDepList and takes as input a package name. It will then query installed.packages and also generate a dependency matrix, calling getDepList with this information and returning the result.

These functions will retrieve information about the dependencies of the matrix, resulting in a DependsList object. This is a list with four elements:

Depends A vector of the dependencies for this package.

Installed A vector of the dependencies which have been satisfied by the currently installed packages.

Found A list representing the dependencies which are not in Installed but were found online. This list has element names which are the URLs for the repositories in which packages were found and the elements themselves are vectors of package names which were found in the respective repositories. If local=TRUE, the Found element will always be empty.

R Any R version dependencies.

If `recursive` is `TRUE`, any package that is specified as a dependency will in turn have its dependencies included (and so on), these are known as indirect dependencies. If `recursive` is `FALSE`, only the dependencies directly stated by the package will be used.

If `local` is `TRUE`, the system will only look at the user's local install and not online to find unresolved dependencies.

If `reduce` is `TRUE`, the system will collapse the fields in the `DependsList` object such that a minimal set of dependencies are specified (for instance if there was ('foo', 'foo (>= 1.0.0)', 'foo (>= 1.3.0)'), it would only return 'foo (>= 1.3.0)').

Value

An object of class `DependsList`

Author(s)

Jeff Gentry

See Also

`installFoundDepends`

Examples

```
pkgDepends("tools", local = FALSE)
```

---

installFoundDepends

A function to install unresolved dependencies

Description

This function will take the `Found` element of a `pkgDependsList` object and attempt to install all of the listed packages from the specified repositories.

Usage

```
installFoundDepends(depPkgList, ...)
```

Arguments

`depPkgList`	A `Found` element from a `pkgDependsList` object
`...`	Arguments to pass on to `install.packages`

Details

This function takes as input the `Found` list from a `pkgDependsList` object. This list will have element names being URLs corresponding to repositories and the elements will be vectors of package names. For each element, `install.packages` is called for that URL to install all packages listed in the vector.

Author(s)

Jeff Gentry

See Also

pkgDepends, install.packages

Examples

```
## Set up a temporary directory to install packages to
tmp <- tempfile()
dir.create(tmp)

pDL <- pkgDepends("tools",local=FALSE)
installFoundDepends(pDL$Found, destdir=tmp)
```

---

makeLazyLoading *Lazy Loading of Packages*

Description

Tools for Lazy Loading of Packages from a Database.

Usage

```
makeLazyLoading(package, lib.loc = NULL, compress = TRUE,
                keep.source = getOption("keep.source.pkgs"))
```

Arguments

package	package name string
lib.loc	library trees, as in library
keep.source	logical; should sources be kept when saving from source
compress	logical; whether to compress entries on the database.

Details

A tool to set up packages for lazy loading from a database. For packages other than base you can use makeLazyLoading(package) to convert them to use lazy loading.

Author(s)

Luke Tierney and Brian Ripley

Examples

```
  # set up package "splines" for lazy loading -- already done
  ## Not run:
    tools:::makeLazyLoading("splines")

## End(Not run)
```

`md5sum` *Compute MD5 Checksums*

Description

Compute the 32-byte MD5 checksums of one or more files.

Usage

```
md5sum(files)
```

Arguments

`files`	character. The paths of file(s) to be check-summed.

Value

A character vector of the same length as `files`, with names equal to `files`. The elements will be `NA` for non-existent or unreadable files, otherwise a 32-character string of hexadecimal digits.

On Windows all files are read in binary mode (as the `md5sum` utilities there do): on other OSes the files are read in the default way.

See Also

`checkMD5sums`

Examples

```
md5sum(dir(R.home(), pattern="^COPY", full.names=TRUE))
```

`package.dependencies`
Check Package Dependencies

Description

Parses and checks the dependencies of a package against the currently installed version of R [and other packages].

Usage

```
package.dependencies(x, check = FALSE,
                     depLevel = c("Depends", "Imports", "Suggests"))
```

Arguments

`x`	A matrix of package descriptions as returned by `available.packages`.
`check`	If `TRUE`, return logical vector of check results. If `FALSE`, return parsed list of dependencies.
`depLevel`	Whether to look for `Depends` or `Suggests` level dependencies.

Details

Currently we only check if the package conforms with the currently running version of R. In the future we might add checks for inter-package dependencies.

See Also

update.packages

QC *QC Checks for R Code and/or Documentation*

Description

Functions for performing various quality checks.

Usage

```
checkDocFiles(package, dir, lib.loc = NULL)
checkDocStyle(package, dir, lib.loc = NULL)
checkReplaceFuns(package, dir, lib.loc = NULL)
checkS3methods(package, dir, lib.loc = NULL)
```

Arguments

package	a character string naming an installed package.
dir	a character string specifying the path to a package's root source directory. This should contain the subdirectories R (for R code) and 'man' with R documentation sources (in Rd format). Only used if package is not given.
lib.loc	a character vector of directory names of R libraries, or NULL. The default value of NULL corresponds to all libraries currently known. The specified library trees are used to to search for package.

Details

checkDocFiles checks, for all Rd files in a package, whether all arguments shown in the usage sections of the Rd file are documented in its arguments section. It also reports duplicated entries in the arguments section, and "over-documented" arguments which are given in the arguments section but not in the usage. Note that the match is for the usage section and not a possibly existing synopsis section, as the usage is what gets displayed.

checkDocStyle investigates how (S3) methods are shown in the usages of the Rd files in a package. It reports the methods shown by their full name rather than using the Rd \method markup for indicating S3 methods. Earlier versions of R also reported about methods shown along with their generic, which typically caused problems for the documentation of the primary argument in the generic and its methods. With \method now being expanded in a way that class information is preserved, "joint" documentation is no longer necessarily a problem. (The corresponding information is still contained in the object returned by checkDocStyle.)

checkReplaceFuns checks whether replacement functions or S3/S4 replacement methods in the package R code have their final argument named value.

checkS3methods checks whether all S3 methods defined in the package R code have all arguments of the corresponding generic, with positional arguments of the generics in the same positions

for the method. As an exception, the first argument of a formula method *may* be called `formula` even if this is not the name used by the generic. The rules when `...` is involved are subtle: see the source code. Functions recognized as S3 generics are those with a call to `UseMethod` in their body, internal S3 generics (see `InternalMethods`), and S3 group generics (see `Math`). Possible dispatch under a different name is not taken into account. The generics are sought first in the given package, then in the **base** package and (currently) the packages **graphics**, **stats**, and **utils** added in R 1.9.0 by splitting the former **base**, and, if an installed package is tested, also in the loaded namespaces/packages listed in the package's 'DESCRIPTION' Depends field.

If using an installed package, the checks needing access to all R objects of the package will load the package (unless it is the **base** package), after possibly detaching an already loaded version of the package.

Value

The functions return objects of class the same as the respective function names containing the information about problems detected. There is a `print` method for nicely displaying the information contained in such objects.

Warning

These functions are still experimental. Names, interfaces and values might change in future versions.

`Rdindex` *Generate Index from Rd Files*

Description

Print a 2-column index table with "names" and titles from given R documentation files to a given output file or connection. The titles are nicely formatted between two column positions (typically 25 and 72, respectively).

Usage

```
Rdindex(RdFiles, outFile = "", type = NULL,
        width = 0.9 * getOption("width"), indent = NULL)
```

Arguments

`RdFiles`	a character vector specifying the Rd files to be used for creating the index, either by giving the paths to the files, or the path to a single directory with the sources of a package.
`outFile`	a connection, or a character string naming the output file to print to. `""` (the default) indicates output is to the console.
`type`	a character string giving the documentation type of the Rd files to be included in the index, or `NULL` (the default). The type of an Rd file is typically specified via the `\docType` tag; if `type` is `"data"`, Rd files whose *only* keyword is `datasets` are included as well.
`width`	a positive integer giving the target column for wrapping lines in the output.
`indent`	a positive integer specifying the indentation of the second column. Must not be greater than `width/2`, and defaults to `width/3`.

Details

If a name is not a valid alias, the first alias (or the empty string if there is none) is used instead.

`Rdutils` *Rd Utilities*

Description

Utilities for computing on the information in Rd objects.

Usage

```
Rd_db(package, dir, lib.loc = NULL)
Rd_parse(file, text = NULL)
```

Arguments

`package`	a character string naming an installed package.
`dir`	a character string specifying the path to a package's root source directory. This should contain the subdirectory 'man' with R documentation sources (in Rd format). Only used if `package` is not given.
`lib.loc`	a character vector of directory names of R libraries, or `NULL`. The default value of `NULL` corresponds to all libraries currently known. The specified library trees are used to to search for `package`.
`file`	a connection, or a character string giving the name of a file or a URL to read documentation in Rd format from.
`text`	character vector with documentation in Rd format. Elements are treated as if they were lines of a file.

Details

`Rd_db` builds a simple "data base" of all Rd sources in a package, as a list of character vectors with the lines of the Rd files in the package. This is particularly useful for working on installed packages, where the individual Rd files in the sources are no longer available.

`Rd_parse` is a simple top-level Rd parser/analyzer. It returns a list with components

`meta` a list containing the Rd metadata (aliases, concepts, keywords, and documentation type);

`data` a data frame with the names (`tags`) and corresponding text (`vals`) of the top-level sections in the R documentation object;

`rest` top-level text not accounted for (currently, silently discarded by Rdconv, and hence usually the indication of a problem).

Note that at least for the time being, only the top-level structure is analyzed.

Warning

These functions are still experimental. Names, interfaces and values might change in future versions.

Examples

```
## Build the Rd db for the (installed) base package.
db <- Rd_db("base")
## Run Rd_parse on all entries in the Rd db.
db <- lapply(db, function(txt) Rd_parse(text = txt))
## Extract the metadata.
meta <- lapply(db, "[[", "meta")

## Keyword metadata per Rd file.
keywords <- lapply(meta, "[[", "keywords")
## Tabulate the keyword entries.
kw_table <- sort(table(unlist(keywords)))
## The 5 most frequent ones:
rev(kw_table)[1 : 5]
## The "most informative" ones:
kw_table[kw_table == 1]

## Concept metadata per Rd file.
concepts <- lapply(meta, "[[", "concepts")
## How many files already have \concept metadata?
sum(sapply(concepts, length) > 0)
## How many concept entries altogether?
length(unlist(concepts))
```

---

read.00Index *Read 00Index-style Files*

Description

Read item/description information from 00Index-style files. Such files are description lists rendered in tabular form, and currently used for the 'INDEX' and 'demo/00Index' files of add-on packages.

Usage

```
read.00Index(file)
```

Arguments

file	the name of a file to read data values from. If the specified file is "", then input is taken from the keyboard (in this case input can be terminated by a blank line). Alternatively, file can be a connection, which will be opened if necessary, and if so closed at the end of the function call.

Value

a character matrix with 2 columns named "Item" and "Description" which hold the items and descriptions.

See Also

formatDL for the inverse operation of creating a 00Index-style file from items and their descriptions.

texi2dvi *Compile LaTeX Files*

Description

Run latex and bibtex until all cross-references are resolved and create either a dvi or PDF file.

Usage

```
texi2dvi(file, pdf = FALSE, clean = FALSE, quiet = TRUE,
         texi2dvi = getOption("texi2dvi"))
```

Arguments

file	character. Name of TeX source file.
pdf	logical. If TRUE, a PDF file is produced insted of the default dvi file (texi2dvi command line option '--pdf').
clean	logical. If TRUE, all auxiliary files are removed (texi2dvi command line option '--clean'). Does not work on some platforms.
quiet	logical. No output unless an error occurs.
texi2dvi	character (or NULL). Script or program used to compile a TeX file to dvi or PDF, respectively. If set to NULL, the 'texi2dvi' script in R's 'bin' directory is used (if it exists), otherwise it is assumed that texi2dvi is in the search path.

Details

Some TeX installations do not have 'texi2dvi.exe'. If 'texify.exe' is present, then it can be used instead: set options(texi2dvi="texify.exe") or to the full path of the program.

Author(s)

Achim Zeileis

tools-deprecated *Deprecated Objects in Package tools*

Description

The functions or variables listed here are provided for compatibility with older versions of R only, and may be defunct as soon as of the next release.

Usage

See Also

Deprecated, Defunct

undoc *Find Undocumented Objects*

Description

Finds the objects in a package which are undocumented, in the sense that they are visible to the user (or data objects or S4 classes provided by the package), but no documentation entry exists.

Usage

```
undoc(package, dir, lib.loc = NULL)
```

Arguments

package	a character string naming an installed package.
dir	a character string specifying the path to a package's root source directory. This must contain the subdirectory 'man' with R documentation sources (in Rd format), and at least one of the 'R' or 'data' subdirectories with R code or data objects, respectively.
lib.loc	a character vector of directory names of R libraries, or NULL. The default value of NULL corresponds to all libraries currently known. The specified library trees are used to to search for package.

Details

This function is useful for package maintainers mostly. In principle, *all* user level R objects should be documented; note however that the precise rules for documenting methods of generic functions are still under discussion.

Value

An object of class "undoc" which is a list of character vectors containing the names of the undocumented objects split according to documentation type. This representation is still experimental, and might change in future versions.

There is a print method for nicely displaying the information contained in such objects.

See Also

codoc, QC

Examples

```
undoc("tools")                  # Undocumented objects in 'tools'
```

`vignetteDepends` *Retrieve Dependency Information for a Vignette*

Description

Given a vignette name, will create a DependsList object that reports information about the packages the vignette depends on.

Usage

```
vignetteDepends(vignette, recursive = TRUE, reduce = TRUE,
                local = TRUE, lib.loc = NULL)
```

Arguments

`vignette`	The path to the vignette source
`recursive`	Whether or not to include indirect dependencies
`reduce`	Whether or not to collapse all sets of dependencies to a minimal value
`local`	Whether or not to search only locally
`lib.loc`	What libraries to search in locally

Details

If `recursive` is `TRUE`, any package that is specified as a dependency will in turn have its dependencies included (and so on), these are known as indirect dependencies. If recursive is `FALSE`, only the dependencies directly named by the vignette will be used.

If `local` is `TRUE`, the system will only look at the user's local machine and not online to find dependencies.

If `reduce` is `TRUE`, the system will collapse the fields in the `DependsList` to the minimal set of dependencies (for instance if the dependencies were ('foo', 'foo (>= 1.0.0)', 'foo (>= 1.3.0)'), the return value would be 'foo (>= 1.3.0)').

Value

An object of class `DependsList`

Author(s)

Jeff Gentry

See Also

`pkgDepends`

Examples

```
gridEx <- system.file("doc", "grid.Snw", package = "grid")
vignetteDepends(gridEx)
```

`write_PACKAGES` *Generate PACKAGES files*

Description

Generate 'PACKAGES' and 'PACKAGES.gz' files for a repository of source or Mac/Windows binary packages.

Usage

```
write_PACKAGES(dir, fields = NULL,
               type = c("source", "mac.binary", "win.binary"),
               verbose = FALSE)
```

Arguments

`dir`	Character vector describing the location of the repository (directory including source or binary packages) to generate the 'PACKAGES' and 'PACKAGES.gz' files from and write them to.
`fields`	a character vector giving the fields to be used in the 'PACKAGES' and 'PACKAGES.gz' files in addition to the default ones, or `NULL` (default). The default corresponds to the fields needed by `available.packages`: "Package", "Bundle", "Priority", "Version", "Depends", "Suggests", "Imports" and "Contains".
`type`	Type of packages: currently source '.tar.gz' archives, and Mac or Windows binary ('.tgz' or '.zip', respectively) packages are supported. Defaults to `"win.binary"` on Windows and to `"source"` otherwise.
`verbose`	logical. Should packages be listed as they are processed?

Details

`type = "win.binary"` uses `unz` connections to read all 'DESCRIPTION' files contained in the (zipped) binary packages for Windows in the given directory `dir`, and builds 'PACKAGES' and 'PACKAGES.gz' files from these information.

Value

Invisibly returns the number of packages described in the resulting 'PACKAGES' and 'PACKAGES.gz' files. If 0, no packages were found and no files were written.

Note

Processing '.tar.gz' archives to extract the 'DESCRIPTION' files is quite slow.

This function can be useful on other OSes to prepare a repository to be accessed by Windows machines, so `type = "win.binary"` should work on all OSes.

Author(s)

Uwe Ligges and R-core.

See Also

See read.dcf and write.dcf for reading 'DESCRIPTION' files and writing the 'PACKAGES' and 'PACKAGES.gz' files.

Examples

```
## Not run:
write_PACKAGES("c:/myFolder/myRepository")  # on Windows
write_PACKAGES("/pub/RWin/bin/windows/contrib/2.1",
               type="win.binary")  # on Linux
## End(Not run)
```

---

xgettext *Extract Translatable Messages from R Files in a Package*

Description

For each file in the 'R' directory (including system-specific subdirectories) of a package, extract the unique arguments passed to stop, warning, message, gettext and gettextf, or to ngettext.

Usage

```
xgettext(dir, verbose = FALSE, asCall = TRUE)

xngettext(dir, verbose = FALSE)

xgettext2pot(dir, potFile)
```

Arguments

dir	the directory of a source package.
verbose	logical: should each file be listed as it is processed?
asCall	logical: if TRUE each argument is returned whole, otherwise the strings within each argument are extracted.
potFile	name of po template file to be produced. Defaults to "R-pkg.pot" where pkg is the basename of dir.

Details

Leading and trailing white space (space, tab and linefeed) is removed for calls to gettext, gettextf, stop, warning, and message, as it is by the internal code that passes strings for translation.

We look to see if these functions were called with domain = NA and if so omit the call if asCall = TRUE: note that the call might contain a call to gettext which would be visible if asCall = FALSE.

xgettext2pot calls xgettext and then xngettext, and writes a PO template file for use with the **GNU Gettext** tools. This ensures that the strings for simple translation are unique in the file (as **GNU Gettext** requires), but does not do so for ngettext calls (and the rules are not stated in the Gettext manual).

If applied to the base package, this also looks in the .R files in 'R_HOME/share/R'.

Value

For `xgettext`, a list of objects of class `"xgettext"` (which has a `print` method), one per source file that potentially contains translatable strings

For `nxgettext`, a list of objects of class `"xngettext"`, which are themselves lists of length-2 character strings.

Examples

```
## Not run:
## in a source-directory build of R:
xgettext(file.path(R.home(), "src", "library", "splines"))
## End(Not run)
```

Chapter 9

The `utils` package

`utils-package` *The R Utils Package*

Description

R utility functions

Details

This package contains a collection of utility functions.

For a complete list, use `library(help="utils")`.

Author(s)

R Development Core Team and contributors worldwide

Maintainer: R Core Team ⟨R-core@r-project.org⟩

`alarm` *Alert the user*

Description

Gives an audible or visual signal to the user.

Usage

```
alarm()
```

Details

`alarm()` works by sending a `"\a"` character to the console. On most platforms this will ring a bell, beep, or give some other signal to the user (unless standard output has been redirected).

Value

No useful value is returned.

Examples

```
alarm()
```

---

apropos *Find Objects by (Partial) Name*

Description

`apropos` returns a character vector giving the names of all objects in the search list matching `what`.

`find` is a different user interface to the same task as `apropos`.

Usage

```
apropos(what, where = FALSE, mode = "any")

find(what, mode = "any", numeric. = FALSE, simple.words = TRUE)
```

Arguments

`what`	name of an object, or regular expression to match against
`where, numeric.`	a logical indicating whether positions in the search list should also be returned
`mode`	character; if not `"any"`, only objects whose `mode` equals `mode` are searched.
`simple.words`	logical; if `TRUE`, the `what` argument is only searched as whole word.

Details

If `mode != "any"` only those objects which are of mode `mode` are considered. If `where` is `TRUE`, the positions in the search list are returned as the names attribute.

`find` is a different user interface for the same task as `apropos`. However, by default (`simple.words == TRUE`), only full words are searched with `grep(fixed = TRUE)`.

Author(s)

Kurt Hornik and Martin Maechler (May 1997).

See Also

`glob2rx` to convert wildcard patterns to regular expressions.

`objects` for listing objects from one place, `help.search` for searching the help system, `search` for the search path.

Examples

```
## Not run: apropos("lm")
apropos(ls)
apropos("lq")

cor <- 1:pi
find(cor)        #> ".GlobalEnv"   "package:stats"
find(cor, num=TRUE) # numbers with these names
find(cor, num=TRUE, mode="function")# only the second one
rm(cor)

## Not run: apropos(".", mode="list") # a long list

# need a DOUBLE backslash '\\' (in case you don't see it anymore)
apropos("\\[")

## Not run: # everything
length(apropos("."))

# those starting with 'pr'
apropos("^pr")

# the 1-letter things
apropos("^.$")
# the 1-2-letter things
apropos("^..?$")
# the 2-to-4 letter things
apropos("^.{2,4}$")

# the 8-and-more letter things
apropos("^.{8,}$")
table(nchar(apropos("^.{8,}$")))
## End(Not run)
```

BATCH *Batch Execution of R*

Description

Run R non-interactively with input from `infile` and send output (stdout/stderr) to another file.

Usage

```
R CMD BATCH [options] infile [outfile]
```

Arguments

`infile`	the name of a file with R code to be executed.
`options`	a list of R command line options, e.g., for setting the amount of memory available and controlling the load/save process. If `infile` starts with a '`-`', use '`--`' as the final option. The default options are '`--restore --save --no-readline`'.
`outfile`	the name of a file to which to write output. If not given, the name used is that of `infile`, with a possible '.R' extension stripped, and '.Rout' appended.

Details

Use `R CMD BATCH --help` to be reminded of the usage.

By default, the input commands are printed along with the output. To suppress this behavior, add `options(echo = FALSE)` at the beginning of `infile`.

The `infile` can have end of line marked by LF or CRLF (but not just CR), and files with an incomplete last line (missing end of line (EOL) mark) are processed correctly.

Additional options can be set by the environment variable `R_BATCH_OPTIONS`: these come after '`--restore --save --no-readline`' and before any options given on the command line.

Note

Unlike `Splus BATCH`, this does not run the R process in the background. In most shells, `R CMD BATCH [options] infile [outfile] &` will do so.

Report bugs to ⟨r-bugs@r-project.org⟩.

`browseEnv` *Browse Objects in Environment*

Description

The `browseEnv` function opens a browser with list of objects currently in `sys.frame()` environment.

Usage

```
browseEnv(envir = .GlobalEnv, pattern,
          excludepatt = "^last\\.warning",
          html = .Platform$OS.type != "mac",
          expanded = TRUE, properties = NULL,
          main = NULL, debugMe = FALSE)
```

Arguments

`envir`	an `environment` the objects of which are to be browsed.
`pattern`	a regular expression for object subselection is passed to the internal `ls()` call.
`excludepatt`	a regular expression for *dropping* objects with matching names.
`html`	is used on non Macintosh machines to display the workspace on a HTML page in your favorite browser.
`expanded`	whether to show one level of recursion. It can be useful to switch it to `FALSE` if your workspace is large. This option is ignored if `html` is set to `FALSE`.
`properties`	a named list of global properties (of the objects chosen) to be showed in the browser; when `NULL` (as per default), user, date, and machine information is used.
`main`	a title string to be used in the browser; when `NULL` (as per default) a title is constructed.
`debugMe`	logical switch; if true, some diagnostic output is produced.

Details

Very experimental code. Only allows one level of recursion into object structures. The HTML version is not dynamic.

It can be generalized. See sources ('..../library/base/R/databrowser.R') for details.

`wsbrowser()` is currently just an internally used function; its argument list will certainly change.

Most probably, this should rather work through using the 'tkWidget' package (from `www.Bioconductor.org`).

See Also

`str`, `ls`.

Examples

```
if(interactive()) {
   ## create some interesting objects :
   ofa <- ordered(4:1)
   ex1 <- expression(1+ 0:9)
   ex3 <- expression(u,v, 1+ 0:9)
   example(factor, echo = FALSE)
   example(table, echo = FALSE)
   example(ftable, echo = FALSE)
   example(lm, echo = FALSE)
   example(str, echo = FALSE)

   ## and browse them:
   browseEnv()

   ## a (simple) function's environment:
   af12 <- approxfun(1:2, 1:2, method = "const")
   browseEnv(envir = environment(af12))
 }
```

`browseURL` *Load URL into a WWW Browser*

Description

Load a given URL into a WWW browser.

Usage

```
browseURL(url, browser = getOption("browser"))
```

Arguments

`url`	a non-empty character string giving the URL to be loaded.
`browser`	a non-empty character string giving the name of the program to be used as hypertext browser. It should be in the PATH, or a full path specified.

Details

The default browser is set by option "browser", in turn set by the environment variable R_BROWSER which is by default set in file 'R_HOME/etc/Renviron' to a choice made manually or automatically when R was configured. (See Startup for where to override that default value.)

If browser supports remote control and R knows how to perform it, the URL is opened in any already running browser or a new one if necessary. This mechanism currently is available for browsers which support the "-remote openURL(...)" interface (which includes Netscape 4.x, 6.2.x (but not 6.0/1), 7.1, Opera 5/6, Mozilla >= 0.9.5 and Mozilla Firefox), Galeon, KDE konqueror (via kfmclient) and the GNOME interface to Mozilla. Netscape 7.0 and Opera 7 behave slightly differently, and you will need to open them first. Note that the type of browser is determined from its name, so this mechanism will only be used if the browser is installed under its canonical name.

Because "-remote" will use any browser displaying on the X server (whatever machine it is running on), the remote control mechanism is only used if DISPLAY points to the local host. This may not allow displaying more than one URL at a time from a remote host.

It is the caller's responsibility to encode url if necessary (see URLencode). This can be tricky for file URLs, where the format accepted can depend on the browser and OS.

Examples

```
## Not run:
## for KDE users who want to open files in a new tab
option(browser="kfmclient newTab")
browseURL("http://www.r-project.org")
## End(Not run)
```

---

bug.report *Send a Bug Report*

Description

Invokes an editor to write a bug report and optionally mail it to the automated r-bugs repository at ⟨r-bugs@r-project.org⟩. Some standard information on the current version and configuration of R are included automatically.

Usage

```
bug.report(subject = "",
           ccaddress = Sys.getenv("USER"),
           method = getOption("mailer"),
           address = "r-bugs@r-project.org",
           file = "R.bug.report")
```

Arguments

subject	Subject of the email. Please do not use single quotes (') in the subject! File separate bug reports for multiple bugs
ccaddress	Optional email address for copies (default is current user). Use ccaddress = FALSE for no copies.

method	Submission method, one of "mailx", "gnudoit", "none", or "ess".
address	Recipient's email address.
file	File to use for setting up the email (or storing it when method is "none" or sending mail fails).

Details

Currently direct submission of bug reports works only on Unix systems. If the submission method is "mailx", then the default editor is used to write the bug report. Which editor is used can be controlled using options, type getOption("editor") to see what editor is currently defined. Please use the help pages of the respective editor for details of usage. After saving the bug report (in the temporary file opened) and exiting the editor the report is mailed using a Unix command line mail utility such as mailx. A copy of the mail is sent to the current user.

If method is "gnudoit", then an emacs mail buffer is opened and used for sending the email.

If method is "none" or NULL (and in every case on Windows systems), then only an editor is opened to help writing the bug report. The report can then be copied to your favorite email program and be sent to the r-bugs list.

If method is "ess" the body of the mail is simply sent to stdout.

Value

Nothing useful.

When is there a bug?

If R executes an illegal instruction, or dies with an operating system error message that indicates a problem in the program (as opposed to something like "disk full"), then it is certainly a bug.

Taking forever to complete a command can be a bug, but you must make certain that it was really R's fault. Some commands simply take a long time. If the input was such that you KNOW it should have been processed quickly, report a bug. If you don't know whether the command should take a long time, find out by looking in the manual or by asking for assistance.

If a command you are familiar with causes an R error message in a case where its usual definition ought to be reasonable, it is probably a bug. If a command does the wrong thing, that is a bug. But be sure you know for certain what it ought to have done. If you aren't familiar with the command, or don't know for certain how the command is supposed to work, then it might actually be working right. Rather than jumping to conclusions, show the problem to someone who knows for certain.

Finally, a command's intended definition may not be best for statistical analysis. This is a very important sort of problem, but it is also a matter of judgment. Also, it is easy to come to such a conclusion out of ignorance of some of the existing features. It is probably best not to complain about such a problem until you have checked the documentation in the usual ways, feel confident that you understand it, and know for certain that what you want is not available. The mailing list r-devel@r-project.org is a better place for discussions of this sort than the bug list.

If you are not sure what the command is supposed to do after a careful reading of the manual this indicates a bug in the manual. The manual's job is to make everything clear. It is just as important to report documentation bugs as program bugs.

If the online argument list of a function disagrees with the manual, one of them must be wrong, so report the bug.

How to report a bug

When you decide that there is a bug, it is important to report it and to report it in a way which is useful. What is most useful is an exact description of what commands you type, from when you start R until the problem happens. Always include the version of R, machine, and operating system that you are using; type `version` in R to print this. To help us keep track of which bugs have been fixed and which are still open please send a separate report for each bug.

The most important principle in reporting a bug is to report FACTS, not hypotheses or categorizations. It is always easier to report the facts, but people seem to prefer to strain to posit explanations and report them instead. If the explanations are based on guesses about how R is implemented, they will be useless; we will have to try to figure out what the facts must have been to lead to such speculations. Sometimes this is impossible. But in any case, it is unnecessary work for us.

For example, suppose that on a data set which you know to be quite large the command `data.frame(x, y, z, monday, tuesday)` never returns. Do not report that `data.frame()` fails for large data sets. Perhaps it fails when a variable name is a day of the week. If this is so then when we got your report we would try out the `data.frame()` command on a large data set, probably with no day of the week variable name, and not see any problem. There is no way in the world that we could guess that we should try a day of the week variable name.

Or perhaps the command fails because the last command you used was a `[` method that had a bug causing R's internal data structures to be corrupted and making the `data.frame()` command fail from then on. This is why we need to know what other commands you have typed (or read from your startup file).

It is very useful to try and find simple examples that produce apparently the same bug, and somewhat useful to find simple examples that might be expected to produce the bug but actually do not. If you want to debug the problem and find exactly what caused it, that is wonderful. You should still report the facts as well as any explanations or solutions.

Invoking R with the '`--vanilla`' option may help in isolating a bug. This ensures that the site profile and saved data files are not read.

A bug report can be generated using the `bug.report()` function. This automatically includes the version information and sends the bug to the correct address. Alternatively the bug report can be emailed to ⟨r-bugs@r-project.org⟩ or submitted to the Web page at `http://bugs.r-project.org`.

Bug reports on **contributed packages** should be sent to the package maintainer rather than to r-bugs.

Author(s)

This help page is adapted from the Emacs manual and the R FAQ

See Also

R FAQ, also `sessionInfo()` from which you may add to the bug report.

`capture.output` *Send output to a character string or file*

Description

Evaluates its arguments with the output being returned as a character string or sent to a file. Related to `sink` in the same way that `with` is related to `attach`.

Usage

```
capture.output(..., file = NULL, append = FALSE)
```

Arguments

`...`	Expressions to be evaluated
`file`	A file name or a connection, or `NULL` to return the output as a string. If the connection is not open it will be opened and then closed on exit.
`append`	Append or overwrite the file?

Value

A character string, or `NULL` if a `file` argument was supplied.

See Also

`sink`, `textConnection`

Examples

```
require(stats)
glmout <- capture.output(example(glm))
glmout[1:5]
capture.output(1+1, 2+2)
capture.output({1+1; 2+2})
## Not run:
## on Unix with enscript available
ps <- pipe("enscript -o tempout.ps","w")
capture.output(example(glm), file=ps)
close(ps)
## End(Not run)
```

`chooseCRANmirror` *Select a CRAN Mirror*

Description

Interact with the user to choose a CRAN mirror.

Usage

```
chooseCRANmirror(graphics = getOption("menu.graphics"))
```

Arguments

`graphics`	Logical. If true and **tcltk** and an X server are available, use a Tk widget, or if under the AQUA interface use a MacOS X widget, otherwise use `menu`.

Details

A list of mirrors is stored in file 'R_HOME/doc/CRAN_mirrors.csv', but first an on-line list of current mirrors is consulted, and the file copy used only if the on-line list is inaccessible.

This function was originally written to support a Windows GUI menu item, but is also called by `contrib.url` if it finds the initial dummy value of `options("repos")`.

Value

None. This function is invoked for its side effect of updating `options("repos")`

See Also

`setRepositories`, `contrib.url`.

`citation` *Citing R and R Packages in Publications*

Description

How to cite R and R packages in publications.

Usage

```
citation(package = "base", lib.loc = NULL)
## S3 method for class 'citation':
toBibtex(object, ...)
## S3 method for class 'citationList':
toBibtex(object, ...)
```

Arguments

`package`	a character string with the name of a single package. An error occurs if more than one package name is given.
`lib.loc`	a character vector with path names of R libraries, or `NULL`. The default value of `NULL` corresponds to all libraries currently known. If the default is used, the loaded packages are searched before the libraries.
`object`	return object of `citation`.
`...`	currently not used.

Details

The R core development team and the very active community of package authors have invested a lot of time and effort in creating R as it is today. Please give credit where credit is due and cite R and R packages when you use them for data anlysis.

Execute function `citation()` for information on how to cite the base R system in publications. If the name of a non-base package is given, the function either returns the information contained in the `CITATION` file of the package or auto-generates citation information. In the latter case the package 'DESCRIPTION' file is parsed, the resulting citation object may be arbitrarily bad, but is quite useful (at least as a starting point) in most cases.

If only one reference is given, the print method shows both a text version and a BibTeX entry for it, if a package has more than one reference then only the text versions are shown. The BibTeX versions can be obtained using function `toBibtex` (see the examples below).

Value

An object of class `"citationList"`.

See Also

`citEntry`

Examples

```
## the basic R reference
citation()

## references for a package -- might not have these installed
if(nchar(system.file(package="lattice"))) citation("lattice")
if(nchar(system.file(package="foreign"))) citation("foreign")

## extract the bibtex entry from the return value
x <- citation()
toBibtex(x)
```

`citEntry` *Writing Package CITATION Files*

Description

The 'CITATION' file of R packages contains an annotated list of references that should be used for citing the packages.

Usage

```
citEntry(entry, textVersion, header = NULL, footer = NULL, ...)
citHeader(...)
citFooter(...)
readCitationFile(file)
```

Arguments

`entry`	a character string with a BibTeX entry type
`textVersion`	a character string with a text representation of the reference
`header`	a character string with optional header text
`footer`	a character string with optional footer text
`file`	a file name
`...`	see details below

Details

The 'CITATION' file of an R package should be placed in the 'inst' subdirectory of the package source. The file is an R source file and may contain arbitrary R commands including conditionals and computations. The file is `source()`ed by the R parser in a temporary environment and all resulting objects of class `"citation"` (the return value of `citEntry`) are collected.

Typically the file will contain zero or more calls to `citHeader`, then one or more calls to `citEntry`, and finally zero or more calls to `citFooter`. `citHeader` and `citFooter` are simply wrappers to `paste`, and their `...` argument is passed on to `paste` as is.

Value

`citEntry` returns an object of class `"citation"`, `readCitationFile` returns an object of class `"citationList"`.

Entry Types

`citEntry` creates `"citation"` objects, which are modeled after BibTeX entries. The entry should be a valid BibTeX entry type, e.g.,

article: An article from a journal or magazine.

book: A book with an explicit publisher.

inbook: A part of a book, which may be a chapter (or section or whatever) and/or a range of pages.

incollection: A part of a book having its own title.

inproceedings: An article in a conference proceedings.

manual: Technical documentation like a software manual.

mastersthesis: A Master's thesis.

misc: Use this type when nothing else fits.

phdthesis: A PhD thesis.

proceedings: The proceedings of a conference.

techreport: A report published by a school or other institution, usually numbered within a series.

unpublished: A document having an author and title, but not formally published.

Entry Fields

The `...` argument of `citEntry` can be any number of BibTeX fields, including

address: The address of the publisher or other type of institution.

author: The name(s) of the author(s), either as a character string in the format described in the LaTeX book, or a `personList` object.

booktitle: Title of a book, part of which is being cited.

chapter: A chapter (or section or whatever) number.

editor: Name(s) of editor(s), same format as `author`.

institution: The publishing institution of a technical report.

journal: A journal name.

note: Any additional information that can help the reader. The first word should be capitalized.

number: The number of a journal, magazine, technical report, or of a work in a series.

pages: One or more page numbers or range of numbers.

publisher: The publisher's name.

school: The name of the school where a thesis was written.

series: The name of a series or set of books.

title: The work's title.

volume: The volume of a journal or multi-volume book.

year: The year of publication.

Examples

```
basecit <- system.file("CITATION", package="base")
source(basecit, echo=TRUE)
readCitationFile(basecit)
```

---

close.socket *Close a Socket*

Description

Closes the socket and frees the space in the file descriptor table. The port may not be freed immediately.

Usage

```
close.socket(socket, ...)
```

Arguments

socket	A socket object
...	further arguments passed to or from other methods.

Value

logical indicating success or failure

Author(s)

Thomas Lumley

See Also

make.socket, read.socket

combn *Generate All Combinations of n Elements, Taken m at a Time*

Description

Generate all combinations of the elements of x taken m at a time. If x is a positive integer, returns all combinations of the elements of seq(x) taken m at a time. If argument FUN is not NULL, applies a function given by the argument to each point. If simplify is FALSE, returns a list; otherwise returns an array, typically a matrix. ... are passed unchanged to the FUN function, if specified.

Usage

```
combn(x, m, FUN = NULL, simplify = TRUE, ...)
```

Arguments

x	vector source for combinations, or integer n for x <- seq(n).
m	number of elements to choose.
FUN	function to be applied to each combination; default NULL means the identity, i.e., to return the combination (vector of length m).
simplify	logical indicating if the result should be simplified to an array (typically a matrix); if FALSE, the function returns a list. Note that when simplify = TRUE as by default, the dimension of the result is simply determined from FUN(emph{<1st combination>}), for efficiency reasons. This will badly fail if FUN(u) is not of constant length.
...	optionally, further arguments to FUN.

Value

a list or array (in nondegenerate cases), see the simplify argument above.

Author(s)

Scott Chasalow wrote the original in 1994 for S; R package **combinat** and documentation by Vince Carey ⟨stvjc@channing.harvard.edu⟩; small changes by the R core team, notably to return an array in all cases of simplify = TRUE, e.g., for combn(5,5).

References

Nijenhuis, A. and Wilf, H.S. (1978) *Combinatorial Algorithms for Computers and Calculators*; Academic Press, NY.

See Also

choose for fast computation of the *number* of combinations. expand.grid for creating a data frame from all combinations of factors or vectors.

Examples

```
combn(letters[1:4], 2)
(m <- combn(10, 5, min))    # minimum value in each combination
mm <- combn(15, 6, function(x) matrix(x, 2,3))
stopifnot(round(choose(10,5)) == length(m),
          c(2,3, round(choose(15,6))) == dim(mm))

## Different way of encoding points:
combn(c(1,1,1,1,2,2,2,3,3,4), 3, tabulate, nbins = 4)

## Compute support points and (scaled) probabilities for a
## Multivariate-Hypergeometric(n = 3, N = c(4,3,2,1)) p.f.:
# table.mat(t(combn(c(1,1,1,1,2,2,2,3,3,4), 3, tabulate,nbins=4)))
```

---

compareVersion *Compare Two Package Version Numbers*

Description

Compare two package version numbers to see which is later.

Usage

```
compareVersion(a, b)
```

Arguments

a, b	Character strings representing package version numbers.

Details

R package version numbers are of the form `x.y-z` for integers `x`, `y` and `z`, with components after `x` optionally missing (in which case the version number is older than those with the components present).

Value

`0` if the numbers are equal, `-1` if `b` is later and `1` if `a` is later (analogous to the C function `strcmp`).

See Also

`package_version`, `library`, `packageStatus`.

Examples

```
compareVersion("1.0", "1.0-1")
compareVersion("7.2-0","7.1-12")
```

COMPILE *Compile Files for Use with R*

Description

Compile given source files so that they can subsequently be collected into a shared library using `R CMD SHLIB` and be loaded into R using `dyn.load()`.

Usage

```
R CMD COMPILE [options] srcfiles
```

Arguments

`srcfiles`	A list of the names of source files to be compiled. Currently, C, C++ and FORTRAN are supported; the corresponding files should have the extensions '.c', '.cc' (or '.cpp' or '.C'), and '.f', respectively.
`options`	A list of compile-relevant settings, such as special values for `CFLAGS` or `FFLAGS`, or for obtaining information about usage and version of the utility.

Details

Note that Ratfor is not supported. If you have Ratfor source code, you need to convert it to FORTRAN. On many Solaris systems mixing Ratfor and FORTRAN code will work.

Note

Some binary distributions of R have `COMPILE` in a separate bundle, e.g. an `R-devel` RPM.

See Also

`SHLIB`, `dyn.load`; the section on "Customizing compilation under Unix" in "R Administration and Installation" (see the 'doc/manual' subdirectory of the R source tree).

data *Data Sets*

Description

Loads specified data sets, or list the available data sets.

Usage

```
data(..., list = character(0), package = NULL, lib.loc = NULL,
     verbose = getOption("verbose"), envir = .GlobalEnv)
```

Arguments

`...`	a sequence of names or literal character strings.
`list`	a character vector.
`package`	a character vector giving the package(s) to look in for data sets, or `NULL`. By default, all packages in the search path are used, then the 'data' subdirectory (if present) of the current working directory.
`lib.loc`	a character vector of directory names of R libraries, or `NULL`. The default value of `NULL` corresponds to all libraries currently known.
`verbose`	a logical. If `TRUE`, additional diagnostics are printed.
`envir`	the environment where the data should be loaded.

Details

Currently, four formats of data files are supported:

1. files ending '.R' or '.r' are `source()` d in, with the R working directory changed temporarily to the directory containing the respective file.
2. files ending '.RData' or '.rda' are `load()` ed.
3. files ending '.tab', '.txt' or '.TXT' are read using `read.table(..., header = TRUE)`, and hence result in a data frame.
4. files ending '.csv' or '.CSV' are read using `read.table(..., header = TRUE, sep = ";")`, and also result in a data frame.

If more than one matching file name is found, the first on this list is used.

The data sets to be loaded can be specified as a sequence of names or character strings, or as the character vector `list`, or as both.

For each given data set, the first two types ('.R' or '.r', and '.RData' or '.rda' files) can create several variables in the load environment, which might all be named differently from the data set. The second two ('.tab', '.txt', or '.TXT', and '.csv' or '.CSV' files) will always result in the creation of a single variable with the same name as the data set.

If no data sets are specified, `data` lists the available data sets. It looks for a new-style data index in the 'Meta' or, if this is not found, an old-style '00Index' file in the 'data' directory of each specified package, and uses these files to prepare a listing. If there is a 'data' area but no index, available data files for loading are computed and included in the listing, and a warning is given: such packages are incomplete. The information about available data sets is returned in an object of class `"packageIQR"`. The structure of this class is experimental. Where the datasets have a different name from the argument that should be used to retrieve them the index will have an entry like `beaver1 (beavers)` which tells us that dataset `beaver1` can be retrieved by the call `data(beaver)`.

If `lib.loc` and `package` are both `NULL` (the default), the data sets are searched for in all the currently loaded packages then in the 'data' directory (if any) of the current working directory.

If `lib.loc = NULL` but `package` is specified as a character vector, the specified package(s) are searched for first amongst loaded packages and then in the default library/ies (see `.libPaths`).

If `lib.loc` *is* specified (and not `NULL`), packages are searched for in the specified library/ies, even if they are already loaded from another library.

To just look in the 'data' directory of the current working directory, set `package = character(0)` (and `lib.loc = NULL`, the default).

Value

a character vector of all data sets specified, or information about all available data sets in an object of class "packageIQR" if none were specified.

Note

The data files can be many small files. On some file systems it is desirable to save space, and the files in the 'data' directory of an installed package can be zipped up as a zip archive 'Rdata.zip'. You will need to provide a single-column file 'filelist' of file names in that directory.

One can take advantage of the search order and the fact that a '.R' file will change directory. If raw data are stored in 'mydata.txt' then one can set up 'mydata.R' to read 'mydata.txt' and preprocess it, e.g., using transform. For instance one can convert numeric vectors to factors with the appropriate labels. Thus, the '.R' file can effectively contain a metadata specification for the plaintext formats.

See Also

help for obtaining documentation on data sets, save for *creating* the second ('.rda') kind of data, typically the most efficient one.

Examples

```
require(utils)
data()                        # list all available data sets
try(data(package = "rpart") )# list the data sets in the rpart package
data(USArrests, "VADeaths")  # load the data sets 'USArrests' and 'VADeaths'
help(USArrests)               # give information on data set 'USArrests'
```

---

dataentry *Spreadsheet Interface for Entering Data*

Description

A spreadsheet-like editor for entering or editing data.

Usage

```
data.entry(..., Modes = NULL, Names = NULL)
dataentry(data, modes)
de(..., Modes = list(), Names = NULL)
```

Arguments

...	A list of variables: currently these should be numeric or character vectors or list containing such vectors.
Modes	The modes to be used for the variables.
Names	The names to be used for the variables.
data	A list of numeric and/or character vectors.
modes	A list of length up to that of data giving the modes of (some of) the variables. list() is allowed.

Details

The data entry editor is only available on some platforms and GUIs. Where available it provides a means to visually edit a matrix or a collection of variables (including a data frame) as described in the "Notes" section.

`data.entry` has side effects, any changes made in the spreadsheet are reflected in the variables. The functions `de`, `de.ncols`, `de.setup` and `de.restore` are designed to help achieve these side effects. If the user passes in a matrix, `X` say, then the matrix is broken into columns before `dataentry` is called. Then on return the columns are collected and glued back together and the result assigned to the variable `X`. If you don't want this behaviour use dataentry directly.

The primitive function is `dataentry`. It takes a list of vectors of possibly different lengths and modes (the second argument) and opens a spreadsheet with these variables being the columns. The columns of the dataentry window are returned as vectors in a list when the spreadsheet is closed.

`de.ncols` counts the number of columns which are supplied as arguments to `data.entry`. It attempts to count columns in lists, matrices and vectors. `de.setup` sets things up so that on return the columns can be regrouped and reassigned to the correct name. This is handled by `de.restore`.

Value

`de` and `dataentry` return the edited value of their arguments. `data.entry` invisibly returns a vector of variable names but its main value is its side effect of assigning new version of those variables in the user's workspace.

Resources

The data entry window responds to X resources of class `R_dataentry`. Resources `foreground`, `background` and `geometry` are utilized.

Note

The details of interface to the data grid may differ by platform and GUI. The following description applies to the X11-based implementation under Unix.

You can navigate around the grid using the cursor keys or by clicking with the (left) mouse button on any cell. The active cell is highlighted by thickening the surrounding rectangle. Moving to the right or down will scroll the grid as needed: there is no constraint to the rows or columns currently in use.

There are alternative ways to navigate using the keys. Return and (keypad) Enter and LineFeed all move down. Tab moves right and Shift-Tab move left. Home moves to the top left.

PageDown or Control-F moves down a page, and PageUp or Control-B up by a page. End will show the last used column and the last few rows used (in any column).

Using any other key starts an editing process on the currently selected cell: moving away from that cell enters the edited value whereas Esc cancels the edit and restores the previous value. When the editing process starts the cell is cleared. In numerical columns (the default) only letters making up a valid number (including `-.eE`) are accepted, and entering an invalid edited value (such as blank) enters `NA` in that cell. The last entered value can be deleted using the BackSpace or Del(ete) key. Only a limited number of characters (currently 29) can be entered in a cell, and if necessary only the start or end of the string will be displayed, with the omissions indicated by > or <. (The start is shown except when editing.)

Entering a value in a cell further down a column than the last used cell extends the variable and fills the gap (if any) by `NA`s (not shown on screen).

The column names can only be selected by clicking in them. This gives a popup menu to select the column type (currently Real (numeric) or Character) or to change the name. Changing the type converts the current contents of the column (and converting from Character to Real may generate NAs.) If changing the name is selected the header cell becomes editable (and is cleared). As with all cells, the value is entered by moving away from the cell by clicking elsewhere or by any of the keys for moving down (only).

New columns are created by entering values in them (and not by just assigning a new name). The mode of the column is auto-detected from the first value entered: if this is a valid number it gives a numeric column. Unused columns are ignored, so adding data in `var5` to a three-column grid adds one extra variable, not two.

The `Copy` button copies the currently selected cell: `paste` copies the last copied value to the current cell, and right-clicking selects a cell *and* copies in the value. Initially the value is blank, and attempts to paste a blank value will have no effect.

Control-L will refresh the display, recalculating field widths to fit the current entries.

In the default mode the column widths are chosen to fit the contents of each column, with a default of 10 characters for empty columns. you can specify fixed column widths by setting option `de.cellwidth` to the required fixed width (in characters). (set it to zero to return to variable widths). The displayed width of any field is limited to 600 pixels (and by the window width).

See Also

`vi`, `edit`: `edit` uses `dataentry` to edit data frames.

Examples

```
# call data entry with variables x and y
## Not run: data.entry(x,y)
```

---

debugger *Post-Mortem Debugging*

Description

Functions to dump the evaluation environments (frames) and to examine dumped frames.

Usage

```
dump.frames(dumpto = "last.dump", to.file = FALSE)
debugger(dump = last.dump)
```

Arguments

`dumpto`	a character string. The name of the object or file to dump to.
`to.file`	logical. Should the dump be to an R object or to a file?
`dump`	An R dump object created by `dump.frames`.

Details

To use post-mortem debugging, set the option `error` to be a call to `dump.frames`. By default this dumps to an R object `"last.dump"` in the workspace, but it can be set to dump to a file (as dump of the object produced by a call to `save`). The dumped object contain the call stack, the active environments and the last error message as returned by `geterrmessage`.

When dumping to file, `dumpto` gives the name of the dumped object and the file name has `.rda` appended.

A dump object of class `"dump.frames"` can be examined by calling `debugger`. This will give the error message and a list of environments from which to select repeatedly. When an environment is selected, it is copied and the `browser` called from within the copy.

If `dump.frames` is installed as the error handler, execution will continue even in non-interactive sessions. See the examples for how to dump and then quit.

Value

None.

Note

Functions such as `sys.parent` and `environment` applied to closures will not work correctly inside `debugger`.

If the error occurred when computing the default value of a formal argument the debugger will report "recursive default argument reference" when trying to examine that environment.

Of course post-mortem debugging will not work if R is too damaged to produce and save the dump, for example if it has run out of workspace.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`options` for setting `error` options; `recover` is an interactive debugger working similarly to `debugger` but directly after the error occurs.

Examples

```
## Not run:
options(error=quote(dump.frames("testdump", TRUE)))

f <- function() {
    g <- function() stop("test dump.frames")
    g()
}
f()   # will generate a dump on file "testdump.rda"
options(error=NULL)

## possibly in another R session
load("testdump.rda")
debugger(testdump)
Available environments had calls:
1: f()
```

```
2: g()
3: stop("test dump.frames")

Enter an environment number, or 0 to exit
Selection: 1
Browsing in the environment with call:
f()
Called from: debugger.look(ind)
Browse[1]> ls()
[1] "g"
Browse[1]> g
function() stop("test dump.frames")
<environment: 759818>
Browse[1]>
Available environments had calls:
1: f()
2: g()
3: stop("test dump.frames")

Enter an environment number, or 0 to exit
Selection: 0

## A possible setting for non-interactive sessions
options(error=quote({dump.frames(to.file=TRUE); q()}))
## End(Not run)
```

---

demo *Demonstrations of R Functionality*

Description

demo is a user-friendly interface to running some demonstration R scripts. demo() gives the list of available topics.

Usage

```
demo(topic, package = NULL, lib.loc = NULL,
     character.only = FALSE, verbose = getOption("verbose"))
```

Arguments

topic	the topic which should be demonstrated, given as a name or literal character string, or a character string, depending on whether character.only is FALSE (default) or TRUE. If omitted, the list of available topics is displayed.
package	a character vector giving the packages to look into for demos, or NULL. By default, all packages in the search path are used.
lib.loc	a character vector of directory names of R libraries, or NULL. The default value of NULL corresponds to all libraries currently known. If the default is used, the loaded packages are searched before the libraries.
character.only	logical; if TRUE, use topic as character string.
verbose	a logical. If TRUE, additional diagnostics are printed.

Details

If no topics are given, demo lists the available demos. The corresponding information is returned in an object of class "packageIQR". The structure of this class is experimental. In earlier versions of R, an empty character vector was returned along with listing available demos.

See Also

source which is called by demo.

Examples

```
demo() # for attached packages

## All available demos:
demo(package = .packages(all.available = TRUE))

demo(lm.glm, package="stats")
## Not run:
 ch <- "scoping"
 demo(ch, character = TRUE)
## End(Not run)
```

---

download.file *Download File from the Internet*

Description

This function can be used to download a file from the Internet.

Usage

```
download.file(url, destfile, method, quiet = FALSE, mode = "w",
              cacheOK = TRUE)
```

Arguments

url	A character string naming the URL of a resource to be downloaded.
destfile	A character string with the name where the downloaded file is saved. Tilde-expansion is performed.
method	Method to be used for downloading files. Currently download methods "internal", "wget" and "lynx" are available, and there is a value "auto": see Details. The method can also be set through the option "download.file.method": see options().
quiet	If TRUE, suppress status messages (if any).
mode	character. The mode with which to write the file. Useful values are "w", "wb" (binary), "a" (append) and "ab". Only used for the "internal" method.
cacheOK	logical. Is a server-side cached value acceptable? Implemented for the "internal" and "wget" methods.

Details

The function `download.file` can be used to download a single file as described by `url` from the internet and store it in `destfile`. The `url` must start with a scheme such as `"http://"`, `"ftp://"` or `"file://"`.

If `method = "auto"` is chosen (the default), the internal method is chosen for `"file://"` URLs, and for the others provided `capabilities("http/ftp")` is true (which it almost always is). Otherwise methods `"wget"` and `"lynx"` are tried in turn.

`cacheOK = FALSE` is useful for `"http://"` URLs, and will attempt to get a copy directly from the site rather than from an intermediate cache. (Not all platforms support it.) It is used by `available.packages`.

The remaining details apply to method `"internal"` only.

See `url` for how `"file://"` URLs are interpreted, especially on Windows. This function does decode encoded URLs.

The timeout for many parts of the transfer can be set by the option `timeout` which defaults to 60 seconds.

The level of detail provided during transfer can be set by the `quiet` argument and the `internet.info` option. The details depend on the platform and scheme, but setting `internet.info` to 0 gives all available details, including all server responses. Using 2 (the default) gives only serious messages, and 3 or more suppresses all messages.

A progress bar tracks the transfer. If the file length is known, an equals represents 2% of the transfer completed: otherwise a dot represents 10Kb.

Method `"wget"` can be used with proxy firewalls which require user/password authentication if proper values are stored in the configuration file for `wget`.

Value

An (invisible) integer code, `0` for success and non-zero for failure. For the `"wget"` and `"lynx"` methods this is the status code returned by the external program. The `"internal"` method can return `1`, but will in most cases throw an error.

Setting Proxies

This applies to the internal code only.

Proxies can be specified via environment variables. Setting `"no_proxy"` stops any proxy being tried. Otherwise the setting of `"http_proxy"` or `"ftp_proxy"` (or failing that, the all upper-case version) is consulted and if non-empty used as a proxy site. For FTP transfers, the username and password on the proxy can be specified by `"ftp_proxy_user"` and `"ftp_proxy_password"`. The form of `"http_proxy"` should be `"http://proxy.dom.com/"` or `"http://proxy.dom.com:8080/"` where the port defaults to `80` and the trailing slash may be omitted. For `"ftp_proxy"` use the form `"ftp://proxy.dom.com:3128/"` where the default port is `21`. These environment variables must be set before the download code is first used: they cannot be altered later by calling `Sys.putenv`.

Usernames and passwords can be set for HTTP proxy transfers via environment variable `http_proxy_user` in the form `user:passwd`. Alternatively, `"http_proxy"` can be of the form `"http://user:pass@proxy.dom.com:8080/"` for compatibility with `wget`. Only the HTTP/1.0 basic authentication scheme is supported.

Note

Methods "wget" and "lynx" are for historical compatibility. They will block all other activity on the R process.

For methods "wget" and "lynx" a system call is made to the tool given by method, and the respective program must be installed on your system and be in the search path for executables.

See Also

options to set the timeout and internet.info options.

url for a finer-grained way to read data from URLs.

url.show, available.packages, download.packages for applications

---

edit *Invoke a Text Editor*

Description

Invoke a text editor on an R object.

Usage

```
## Default S3 method:
edit(name = NULL, file = "", title = NULL,
     editor = getOption("editor"), ...)

vi(name = NULL, file = "")
emacs(name = NULL, file = "")
pico(name = NULL, file = "")
xemacs(name = NULL, file = "")
xedit(name = NULL, file = "")
```

Arguments

name	a named object that you want to edit. If name is missing then the file specified by file is opened for editing.
file	a string naming the file to write the edited version to.
title	a display name for the object being edited.
editor	a string naming the text editor you want to use. On Unix the default is set from the environment variables EDITOR or VISUAL if either is set, otherwise vi is used. On Windows it defaults to notepad.
...	further arguments to be passed to or from methods.

Details

edit invokes the text editor specified by editor with the object name to be edited. It is a generic function, currently with a default method and one for data frames and matrices.

data.entry can be used to edit data, and is used by edit to edit matrices and data frames on systems for which data.entry is available.

It is important to realize that edit does not change the object called name. Instead, a copy of name is made and it is that copy which is changed. Should you want the changes to apply to the object name you must assign the result of edit to name. (Try fix if you want to make permanent changes to an object.)

In the form edit(name), edit deparses name into a temporary file and invokes the editor editor on this file. Quitting from the editor causes file to be parsed and that value returned. Should an error occur in parsing, possibly due to incorrect syntax, no value is returned. Calling edit(), with no arguments, will result in the temporary file being reopened for further editing.

Currently only the internal editor in Windows makes use of the title option; it displays the given name in the window header.

Note

The functions vi, emacs, pico, xemacs, xedit rely on the corresponding editor being available and being on the path. This is system-dependent.

See Also

edit.data.frame, data.entry, fix.

Examples

```
## Not run:
# use xedit on the function mean and assign the changes
mean <- edit(mean, editor = "xedit")

# use vi on mean and write the result to file mean.out
vi(mean, file = "mean.out")
## End(Not run)
```

---

edit.data.frame *Edit Data Frames and Matrices*

Description

Use data editor on data frame or matrix contents.

Usage

```
## S3 method for class 'data.frame':
edit(name, factor.mode = c("character", "numeric"),
     edit.row.names = any(row.names(name) != 1:nrow(name)), ...)

## S3 method for class 'matrix':
edit(name, edit.row.names = !is.null(dn[[1]]), ...)
```

Arguments

name	A data frame or matrix.
factor.mode	How to handle factors (as integers or using character levels) in a data frame.
edit.row.names	logical. Show the row names be displayed as a separate editable column?
...	further arguments passed to or from other methods.

Details

At present, this only works on simple data frames containing numeric, logical or character vectors and factors. Factors are represented in the spreadsheet as either numeric vectors (which is more suitable for data entry) or character vectors (better for browsing). After editing, vectors are padded with NA to have the same length and factor attributes are restored. The set of factor levels can not be changed by editing in numeric mode; invalid levels are changed to NA and a warning is issued. If new factor levels are introduced in character mode, they are added at the end of the list of levels in the order in which they encountered.

It is possible to use the data-editor's facilities to select the mode of columns to swap between numerical and factor columns in a data frame. Changing any column in a numerical matrix to character will cause the result to be coerced to a character matrix. Changing the mode of logical columns is not supported.

The columns are coerced on input to numeric unless logical, character or factor (which may well not be what you want).

For a data frame, the row names will be taken from the original object if `edit.row.names = FALSE` and the number of rows is unchanged, and from the edited output if `edit.row.names = TRUE` and there are no duplicates. (If the `row.names` column is incomplete, it is extended by entries like `row223`.) In all other cases the row names are replaced by `seq(length=nrows)`.

For a matrix, colnames will be added (of the form `col7`) if needed. The rownames will be taken from the original object if `edit.row.names = FALSE` and the number of rows is unchanged (otherwise NULL), and from the edited output if `edit.row.names = TRUE`. (If the `row.names` column is incomplete, it is extended by entries like `row223`.)

Value

The edited data frame.

Note

`fix(dataframe)` works for in-place editing by calling this function.

If the data editor is not available, a dump of the object is presented for editing using the default method of `edit`.

At present the data editor is limited to 65535 rows.

Author(s)

Peter Dalgaard

See Also

`data.entry`, `edit`

Examples

```
## Not run:
edit(InsectSprays)
edit(InsectSprays, factor.mode="numeric")
## End(Not run)
```

`example` *Run an Examples Section from the Online Help*

Description

Run all the R code from the **Examples** part of R's online help topic `topic` with two possible exceptions, `dontrun` and `dontshow`, see Details below.

Usage

```
example(topic, package = NULL, lib.loc = NULL,
        local = FALSE, echo = TRUE, verbose = getOption("verbose"),
        setRNG = FALSE, ask = getOption("example.ask"),
        prompt.echo = paste(abbreviate(topic, 6),"> ", sep=""))
```

Arguments

`topic`	name or literal character string: the online `help` topic the examples of which should be run.
`package`	a character vector giving the package names to look into for example code, or `NULL`. By default, all packages in the search path are used.
`lib.loc`	a character vector of directory names of R libraries, or `NULL`. The default value of `NULL` corresponds to all libraries currently known. If the default is used, the loaded packages are searched before the libraries.
`local`	logical: if `TRUE` evaluate locally, if `FALSE` evaluate in the workspace.
`echo`	logical; if `TRUE`, show the R input when sourcing.
`verbose`	logical; if `TRUE`, show even more when running example code.
`setRNG`	logical or expression; if not `FALSE`, the random number generator state is saved, then initialized to a specified state, the example is run and the (saved) state is restored. `setRNG = TRUE` sets the same state as `R CMD check` does for running a package's examples. This is currently equivalent to `setRNG = {RNGkind("default", "default"); set.seed(1)}`.
`ask`	logical (or `"default"`) indicating if `par(ask=TRUE)` should be called before graphical output happens from the example code. The value `"default"` (the factory-fresh default) means to ask if `echo == TRUE` and the graphics device appears to be interactive. This parameter applies both to any currently opened device and to any devices opened by the example code.
`prompt.echo`	character; gives the prompt to be used if `echo = TRUE`.

Details

If `lib.loc` is not specified, the packages are searched for amongst those already loaded, then in the specified libraries. If `lib.loc` is specified, they are searched for only in the specified libraries, even if they are already loaded from another library.

An attempt is made to load the package before running the examples, but this will not replace a package loaded from another location.

If `local=TRUE` objects are not created in the workspace and so not available for examination after `example` completes: on the other hand they cannot clobber objects of the same name in the workspace.

As detailed in the manual *Writing* R *Extensions*, the author of the help page can markup parts of the examples for two exception rules

`dontrun` encloses code that should not be run.

`dontshow` encloses code that is invisible on help pages, but will be run both by the package checking tools, and the `example()` function. This was previously `testonly`, and that form is still accepted.

If the examples file contains non-ASCII characters the encoding used will matter. If in a UTF-8 locale `example` first tries UTF-8 and then Latin-1. (This can be overridden by setting the encoding in the `.Rd` file.)

Value

The value of the last evaluated expression.

Note

The examples can be many small files. On some file systems it is desirable to save space, and the files in the 'R-ex' directory of an installed package can be zipped up as a zip archive 'Rex.zip'.

Author(s)

Martin Maechler and others

See Also

`demo`

Examples

```
example(InsectSprays)
## force use of the standard package 'stats':
example("smooth", package="stats", lib.loc=.Library)

## set RNG *before* example as when R CMD check is run:

r1 <- example(quantile, setRNG = TRUE)
x1 <- rnorm(1)
u <- runif(1)
## identical random numbers
r2 <- example(quantile, setRNG = TRUE)
x2 <- rnorm(1)
stopifnot(identical(r1, r2))
## but x1 and x2 differ since the RNG state from before example()
```

```
## differs and is restored!
x1; x2
```

`file.edit` *Edit One or More Files*

Description

Edit one or more files in a text editor.

Usage

```
file.edit(..., title = file, editor = getOption("editor"))
```

Arguments

`...`	one or more character vectors containing the names of the files to be edited.
`title`	the title to use in the editor; defaults to the filename.
`editor`	the text editor to be used.

Details

The behaviour of this function is very system dependent. Currently files can be opened only one at a time on Unix; on Windows, the internal editor allows multiple files to be opened, but has a limit of 50 simultaneous edit windows.

The `title` argument is used for the window caption in Windows, and is ignored on other platforms.

See Also

`files`, `file.show`, `edit`, `fix`,

Examples

```
## Not run:
# open two R scripts for editing
file.edit("script1.R", "script2.R")
## End(Not run)
```

`fix` *Fix an Object*

Description

`fix` invokes `edit` on `x` and then assigns the new (edited) version of `x` in the user's workspace.

Usage

```
fix(x, ...)
```

Arguments

`x`	the name of an R object, as a name or a character string.
`...`	arguments to pass to editor: see `edit`.

Details

The name supplied as `x` need not exist as an R object, in which case a function with no arguments and an empty body is supplied for editing.

See Also

`edit`, `edit.data.frame`

Examples

```
## Not run:
 ## Assume 'my.fun' is a user defined function :
 fix(my.fun)
 ## now my.fun is changed
 ## Also,
 fix(my.data.frame) # calls up data editor
 fix(my.data.frame, factor.mode="char") # use of ...
## End(Not run)
```

`flush.console` *Flush Output to A Console*

Description

This does nothing except on console-based versions of R. On the Mac OS X and Windows GUIs, it ensures that the display of output in the console is current, even if output buffering is on.

Usage

```
flush.console()
```

getAnywhere *Retrieve an R Object, Including from a Namespace*

Description

These functions locates all objects with name matching its argument, whether visible on the search path, registered as an S3 method or in a namespace but not exported. `getAnywhere()` returns the objects and `argsAnywhere()` returns the arguments of any objects that are functions.

Usage

```
getAnywhere(x)
argsAnywhere(x)
```

Arguments

`x`	a character string or name.

Details

The function looks at all loaded namespaces, whether or not they are associated with a package on the search list.

Where functions are found as an S3 method, an attempt is made to find which namespace registered them. This may not be correct, especially if a namespace is unloaded.

Value

For `getAnywhere()` an object of class `"getAnywhere"`. This is a list with components

`name`	the name searched for.
`objs`	a list of objects found
`where`	a character vector explaining where the object(s) were found
`visible`	logical: is the object visible
`dups`	logical: is the object identical to one earlier in the list.

Normally the structure will be hidden by the `print` method. There is a `[` method to extract one or more of the objects found.

For `argsAnywhere()` one or more argument lists as returned by `args`.

See Also

`get`, `getFromNamespace`, `args`

Examples

```
getAnywhere("format.dist")
getAnywhere("simpleLoess") # not exported from stats
argsAnywhere(format.dist)
```

`getFromNamespace` *Utility functions for Developing Namespaces*

Description

Utility functions to access and replace the non-exported functions in a namespace, for use in developing packages with namespaces.

Usage

```
getFromNamespace(x, ns, pos = -1, envir = as.environment(pos))
assignInNamespace(x, value, ns, pos = -1, envir = as.environment(pos))
fixInNamespace(x, ns, pos = -1, envir = as.environment(pos), ...)
```

Arguments

`x`	an object name (given as a character string).
`value`	an R object.
`ns`	a namespace, or character string giving the namespace.
`pos`	where to look for the object: see `get`.
`envir`	an alternative way to specify an environment to look in.
`...`	arguments to pass to the editor: see `edit`.

Details

The namespace can be specified in several ways. Using, for example, `ns = "stats"` is the most direct, but a loaded package with a namespace can be specified via any of the methods used for `get`: `ns` can also be the environment printed as `<namespace:foo>`.

`getFromNamespace` is similar to (but predates) the `:::` operator, but is more flexible in how the namespace is specified.

`fixInNamespace` invokes `edit` on the object named `x` and assigns the revised object in place of the original object. For compatibility with `fix`, `x` can be unquoted.

Value

`getFromNamespace` returns the object found (or gives an error).

`assignInNamespace` and `fixInNamespace` are invoked for their side effect of changing the object in the namespace.

Note

`assignInNamespace` and `fixInNamespace` change the copy in the namespace, but not any copies already exported from the namespace, in particular an object of that name in the package (if already attached) and any copies already imported into other namespaces. They are really intended to be used *only* for objects which are not exported from the namespace. They do attempt to alter a copy registered as an S3 method if one is found.

See Also

`get`, `fix`, `getS3method`

Examples

```
getFromNamespace("findGeneric", "utils")
## Not run:
fixInNamespace("predict.ppr", "stats")
stats:::predict.ppr
getS3method("predict", "ppr")
## alternatively
fixInNamespace("predict.ppr", pos = 3)
fixInNamespace("predict.ppr", pos = "package:stats")
## End(Not run)
```

---

getS3method *Get An S3 Method*

Description

Get a method for an S3 generic, possibly from a namespace.

Usage

```
getS3method(f, class, optional = FALSE)
```

Arguments

f	character: name of the generic.
class	character: name of the class.
optional	logical: should failure to find the generic or a method be allowed?

Details

S3 methods may be hidden in packages with namespaces, and will not then be found by `get`: this function can retrieve such functions, primarily for debugging purposes.

Value

The function found, or `NULL` if no function is found and `optional = TRUE`.

See Also

`methods`, `get`

Examples

```
require(stats)
exists("predict.ppr") # false
getS3method("predict", "ppr")
```

glob2rx *Change Wildcard or Globbing Pattern into Regular Expression*

Description

Change *wildcard* aka *globbing* (or "ls" like) pattern into the corresponding regular expression (regexp).

Usage

```
glob2rx(pattern, trim.head = FALSE, trim.tail = TRUE)
```

Arguments

pattern	character vector
trim.head	logical specifying if leading "^.*" should be trimmed from the result.
trim.tail	logical specifying if trailing ".*$" should be trimmed from the result.

Details

This takes a wildcard as used by most shells and returns an equivalent regular expression. ? is mapped to . (match a single character), * to .* (match any string, including an empty one), and the pattern is anchored (it must start at the beginning and end at the end). Optionally, the resulting regexp is simplified.

Value

A character vector of the same length as the input pattern where each "wild card" is translated to the corresponding regular expression.

Author(s)

Martin Maechler, Unix/sed based version, 1991; current: 2004

See Also

regexp about regular expression, sub, etc about substitutions using regexps.

Examples

```
stopifnot(glob2rx("abc.*") == "^abc\\.",
          glob2rx("a?b.*") == "^a.b\\.",
          glob2rx("a?b.*", trim.tail=FALSE) == "^a.b\\..*$",
          glob2rx("*.doc") == "^.*\\.doc$",
          glob2rx("*.doc", trim.head=TRUE) == "\\.doc$",
          glob2rx("*.t*")  == "^.*\\.t",
          glob2rx("*.t??") == "^.*\\.t..$"
)
```

`head` *Return the First or Last Part of an Object*

Description

Returns the first or last parts of a vector, matrix, table, data frame or function. Since `head()` and `tail()` are generic functions, they may also have been extended to other classes.

Usage

```
head(x, ...)
## Default S3 method:
head(x, n = 6, ...)
## S3 method for class 'data.frame':
head(x, n = 6, ...)
## S3 method for class 'matrix':
head(x, n = 6, ...)
## S3 method for class 'ftable':
head(x, n = 6, ...)
## S3 method for class 'table':
head(x, n = 6, ...)
## S3 method for class 'function':
head(x, n = 6, ...)

tail(x, ...)
## Default S3 method:
tail(x, n = 6, ...)
## S3 method for class 'data.frame':
tail(x, n = 6, ...)
## S3 method for class 'matrix':
tail(x, n = 6, addrownums = TRUE, ...)
## S3 method for class 'ftable':
tail(x, n = 6, addrownums = FALSE, ...)
## S3 method for class 'table':
tail(x, n = 6, addrownums = TRUE, ...)
## S3 method for class 'function':
tail(x, n = 6, ...)
```

Arguments

`x`	an object
`n`	a single integer. If positive, size for the resulting object: number of elements for a vector (including lists), rows for a matrix or data frame or lines for a function. If negative, all but the `n` last/first number of elements of `x`.
`addrownums`	if there are no row names, create them from the row numbers.
`...`	arguments to be passed to or from other methods.

Details

For matrices, 2-dim tables and data frames, `head()` (`tail()`) returns the first (last) `n` rows when `n > 0` or all but the last (first) `n` rows when `n < 0`. `head.matrix()` and `tail.matrix()` are exported. For functions, the lines of the deparsed function are returned as character strings.

If a matrix has no row names, then `tail()` will add row names of the form `"[n,]"` to the result, so that it looks similar to the last lines of `x` when printed. Setting `addrownums = FALSE` suppresses this behaviour.

Value

An object (usually) like `x` but generally smaller. For `ftable` objects `x`, a transformed `format(x)`.

Author(s)

Patrick Burns, improved and corrected by R-Core. Negative argument added by Vincent Goulet.

Examples

```
head(letters)
head(letters, n = -6)

head(freeny.x, n = 10)
head(freeny.y)

tail(letters)
tail(letters, n = -6)

tail(freeny.x)
tail(freeny.y)

tail(library)

head(ftable(Titanic))
```

---

help *Documentation*

Description

These functions provide access to documentation. Documentation on a topic with name `name` (typically, an R object or a data set) can be printed with either `help(name)` or `?name`.

Usage

```
help(topic, offline = FALSE, package = NULL,
     lib.loc = NULL, verbose = getOption("verbose"),
     try.all.packages = getOption("help.try.all.packages"),
     chmhelp = getOption("chmhelp"),
     htmlhelp = getOption("htmlhelp"),
     pager = getOption("pager"))
```

```
?topic

type?topic
```

Arguments

topic	usually, the name on which documentation is sought. The name may be quoted or unquoted (but note that if topic is the name of a variable containing a character string documentation is provided for the name, not for the character string). The topic argument may also be a function call, to ask for documentation on a corresponding method. See the section on method documentation. See Details for what happens if this is omitted.
offline	a logical indicating whether documentation should be displayed on-line to the screen (the default) or hardcopy of it should be produced.
package	a name or character vector giving the packages to look into for documentation, or NULL. By default, all packages in the search path are used.
lib.loc	a character vector of directory names of R libraries, or NULL. The default value of NULL corresponds to all libraries currently known. If the default is used, the loaded packages are searched before the libraries.
verbose	logical; if TRUE, the file name is reported.
try.all.packages	logical; see Note.
chmhelp	logical (or NULL). Only relevant under Windows. If TRUE the Compiled HTML version of the help on the topic will be shown in a help viewer.
htmlhelp	logical (or NULL). If TRUE (which is the default after help.start has been called), the HTML version of the help on the topic will be shown in the browser specified by options("browser"). See browseURL for details of the browsers that are supported. Where possible an existing browser window is re-used.
pager	the pager to be used for file.show.
type	the special type of documentation to use for this topic; for example, if the type is class, documentation is provided for the class with name topic. The function topicName returns the actual name used in this case. See the section on method documentation for the uses of type to get help on formal methods.

Details

topic is not optional: if it is omitted R will give a hint as to suitable topics if a package is specified, to available packages if lib.loc only is specified, and help on help itself if nothing is specified. (In all cases this will be text help.)

In the case of unary and binary operators and control-flow special forms (including if, for and function), the topic may need to be quoted.

If multiple help files matching topic are found, in interactive use a menu is presented for the user to choose one: otherwise the first on the search path is used. (The menu will be a graphical menu if possible if getOption("menu.graphics") is true, the default.)

If offline is TRUE, hardcopy of the documentation is produced by running the LaTeX version of the help page through latex (note that LaTeX 2e is needed) and dvips. Depending on your dvips configuration, hardcopy will be sent to the printer or saved in a file. If the programs are

in non-standard locations and hence were not found at compile time, you can either set the options `latexcmd` and `dvipscmd`, or the environment variables `R_LATEXCMD` and `R_DVIPSCMD` appropriately. The appearance of the output can be customized through a file 'Rhelp.cfg' somewhere in your LaTeX search path. The appearance of the output can be customized through a file 'Rhelp.cfg' somewhere in your LaTeX search path.

If LaTeX versions of help pages were not built at the installation of the package, the `print` method will ask if conversion with `R CMD Rdconv` (which requires Perl) should be attempted.

S4 Method Documentation

The authors of formal ('S4') methods can provide documentation on specific methods, as well as overall documentation on the methods of a particular function. The `"?"` operator allows access to this documentation in three ways.

The expression `methods ? f` will look for the overall documentation methods for the function `f`. Currently, this means the documentation file containing the alias `f-methods`.

There are two different ways to look for documentation on a particular method. The first is to supply the `topic` argument in the form of a function call, omitting the `type` argument. The effect is to look for documentation on the method that would be used if this function call were actually evaluated. See the examples below. If the function is not a generic (no S4 methods are defined for it), the help reverts to documentation on the function name.

The `"?"` operator can also be called with `type` supplied as `"method"`; in this case also, the `topic` argument is a function call, but the arguments are now interpreted as specifying the class of the argument, not the actual expression that will appear in a real call to the function. See the examples below.

The first approach will be tedious if the actual call involves complicated expressions, and may be slow if the arguments take a long time to evaluate. The second approach avoids these difficulties, but you do have to know what the classes of the actual arguments will be when they are evaluated.

Both approaches make use of any inherited methods; the signature of the method to be looked up is found by using `selectMethod` (see the documentation for `getMethod`).

Note

Unless `lib.loc` is specified explicitly, the loaded packages are searched before those in the specified libraries. This ensures that if a library is loaded from a library not in the known library trees, then the help from the loaded library is used. If `lib.loc` is specified explicitly, the loaded packages are *not* searched.

If this search fails and argument `try.all.packages` is `TRUE` and neither `packages` nor `lib.loc` is specified, then all the packages in the known library trees are searched for help on `topic` and a list of (any) packages where help may be found is printed (but no help is shown). **N.B.** searching all packages can be slow.

The help files can be many small files. On some file systems it is desirable to save space, and the text files in the 'help' directory of an installed package can be zipped up as a zip archive 'Rhelp.zip'. Ensure that file 'AnIndex' remains un-zipped. Similarly, all the files in the 'latex' directory can be zipped to 'Rhelp.zip'.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`help.search()` for finding help pages on a "vague" topic; `help.start()` which opens the HTML version of the R help pages; `library()` for listing available packages and the user-level objects they contain; `data()` for listing available data sets; `methods()`.

See `prompt()` to get a prototype for writing `help` pages of private packages.

Examples

```
help()
help(help)              # the same

help(lapply)
?lapply                 # the same

help("for")             # or ?"for", but the quotes are needed
?"+"

help(package="splines") # get help even when package is not loaded

data()                  # list all available data sets
?women                  # information about data set "women"

topi <- "women"
## Not run: help(topi) ##--> Error: No documentation for 'topi'

try(help("bs", try.all.packages=FALSE)) # reports not found (an error)
help("bs", try.all.packages=TRUE) # reports can be found in package 'splines'

## Not run:
require(methods)
## define a S4 generic function and some methods
combo <- function(x, y) c(x, y)
setGeneric("combo")
setMethod("combo", c("numeric", "numeric"), function(x, y) x+y)

## assume we have written some documentation for combo, and its methods ....

?combo  ## produces the function documentation

methods?combo  ## looks for the overall methods documentation

method?combo("numeric", "numeric")  ## documentation for the method above

?combo(1:10, rnorm(10))  ## ... the same method, selected according to
                         ## the arguments (one integer, the other numeric)

?combo(1:10, letters)    ## documentation for the default method
## End(Not run)
```

`help.search` *Search the Help System*

Description

Allows for searching the help system for documentation matching a given character string in the (file) name, alias, title, concept or keyword entries (or any combination thereof), using either fuzzy matching or regular expression matching. Names and titles of the matched help entries are displayed nicely formatted.

Usage

```
help.search(pattern, fields = c("alias", "concept", "title"),
            apropos, keyword, whatis, ignore.case = TRUE,
            package = NULL, lib.loc = NULL,
            help.db = getOption("help.db"),
            verbose = getOption("verbose"),
            rebuild = FALSE, agrep = NULL)
```

Arguments

`pattern`	a character string to be matched in the specified fields. If this is given, the arguments `apropos`, `keyword`, and `whatis` are ignored.
`fields`	a character vector specifying the fields of the help data bases to be searched. The entries must be abbreviations of `"name"`, `"title"`, `"alias"`, `"concept"`, and `"keyword"`, corresponding to the help page's (file) name, its title, the topics and concepts it provides documentation for, and the keywords it can be classified to.
`apropos`	a character string to be matched in the help page topics and title.
`keyword`	a character string to be matched in the help page 'keywords'. 'Keywords' are really categories: the standard categories are listed in file 'RHOME/doc/KEYWORDS' (see also the example) and some package writers have defined their own. If `keyword` is specified, `agrep` defaults to `FALSE`.
`whatis`	a character string to be matched in the help page topics.
`ignore.case`	a logical. If `TRUE`, case is ignored during matching; if `FALSE`, pattern matching is case sensitive.
`package`	a character vector with the names of packages to search through, or `NULL` in which case *all* available packages in the library trees specified by `lib.loc` are searched.
`lib.loc`	a character vector describing the location of R library trees to search through, or `NULL`. The default value of `NULL` corresponds to all libraries currently known.
`help.db`	a character string giving the file path to a previously built and saved help data base, or `NULL`.
`verbose`	logical; if `TRUE`, the search process is traced.
`rebuild`	a logical indicating whether the help data base should be rebuilt. This will be done automatically if `lib.loc` or the search path is changed, or if `package` is used and a value is not found.
`agrep`	if `NULL` (the default unless `keyword` is used) and the character string to be matched consists of alphanumeric characters, whitespace or a dash only, approximate (fuzzy) matching via `agrep` is used unless the string has fewer than 5 characters; otherwise, it is taken to contain a regular expression to be matched via `grep`. If `FALSE`, approximate matching is not used. Otherwise, one can give a numeric or a list specifying the maximal distance for the approximate match, see argument `max.distance` in the documentation for `agrep`.

Details

Upon installation of a package, a pre-built help.search index is serialized as 'hsearch.rds' in the 'Meta' directory. These files are used to create the data base.

The arguments `apropos` and `whatis` play a role similar to the Unix commands with the same names.

If possible, the help data base is saved in memory or (if memory limits have been set: see `mem.limits`) to a file in the session temporary directory for use by subsequent calls in the session.

Note that currently the aliases in the matching help files are not displayed.

Value

The results are returned in a list object of class `"hsearch"`, which has a print method for nicely formatting the results of the query. This mechanism is experimental, and may change in future versions of R.

The internal format of the class is undocumented and subject to change.

See Also

`help`; `help.start` for starting the hypertext (currently HTML) version of R's online documentation, which offers a similar search mechanism.

`RSiteSearch` to access an on-line search of R resources.

`apropos` uses regexps and has nice examples.

Examples

```
help.search("linear models")    # In case you forgot how to fit linear
                                # models
help.search("non-existent topic")
## Not run:
help.search("print")            # All help pages with topics or title
                                # matching 'print'
help.search(apropos = "print")  # The same

help.search(keyword = "hplot")  # All help pages documenting high-level
                                # plots.
file.show(file.path(R.home(), "doc", "KEYWORDS"))  # show all keywords

## Help pages with documented topics starting with 'try'.
help.search("\\btry", fields = "alias")
## Do not use '^' or '$' when matching aliases or keywords
## (unless all packages were installed using R 1.7 or newer).
## End(Not run)
```

`help.start` *Hypertext Documentation*

Description

Start the hypertext (currently HTML) version of R's online documentation.

Usage

```
help.start(gui = "irrelevant", browser = getOption("browser"),
           remote = NULL)
```

Arguments

gui	just for compatibility with S-PLUS.
browser	the name of the program to be used as hypertext browser. It should be in the PATH, or a full path specified.
remote	A character giving a valid URL for the '$R_HOME' directory on a remote location.

Details

All the packages in the known library trees are linked to directory '.R' in the per-session temporary directory. The links are re-made each time `help.start` is run, which should be done after packages are installed, updated or removed.

If the browser given by the `browser` argument is different from the default browser as specified by `options("browser")`, the default is changed to the given browser so that it gets used for all future help requests.

Note

There is a Java-based search facility available from the HTML page that `help.start` brings up. Should this not work, please consult the 'R Installation and Administration' manual which is linked from that page.

See Also

`help()` for on- and off-line help in ASCII/Editor or PostScript format.

`browseURL` for how the help file is displayed.

`RSiteSearch` to access an on-line search of R resources.

Examples

```
## Not run:
help.start()
## End(Not run)
```

---

index.search *Search Indices for Help Files*

Description

Used to search the indices for help files, possibly under aliases.

Usage

```
index.search(topic, path, file="AnIndex", type = "help")
```

Arguments

topic	The keyword to be searched for in the indices.
path	The path(s) to the packages to be searched.
file	The index file to be searched. Normally '"AnIndex"'.
type	The type of file required.

Details

For each package in path, examine the file file in directory 'type', and look up the matching file stem for topic topic, if any.

Value

A character vector of matching files, as if they are in directory type of the corresponding package. In the special cases of type = "html", "R-ex" and "latex" the file extensions ".html", ".R" and ".tex" are added.

See Also

help, example

INSTALL *Install Add-on Packages*

Description

Utility for installing add-on packages.

Usage

```
R CMD INSTALL [options] [-l lib] pkgs
```

Arguments

pkgs	A space-separated list with the path names of the packages to be installed.
lib	the path name of the R library tree to install to.
options	a space-separated list of options through which in particular the process for building the help files can be controlled. Options should only be given once. Use R CMD INSTALL --help for the current list of options.

Details

This will stop at the first error, so if you want all the pkgs to be tried, call this via a shell for or foreach loop.

If used as R CMD INSTALL pkgs without explicitly specifying lib, packages are installed into the library tree rooted at the first directory given in the environment variable R_LIBS if this is set and non-null, and to the default library tree (which is rooted at '$R_HOME/library') otherwise.

To install into the library tree lib, use R CMD INSTALL -l lib pkgs.

Both `lib` and the elements of `pkgs` may be absolute or relative path names of directories. `pkgs` may also contain names of package/bundle archive files of the form '`pkg_version.tar.gz`' as obtained from CRAN: these are then extracted in a temporary directory. Finally, binary package/bundle archive files (as created by `R CMD build --binary` can be supplied.

Some package sources contain a '`configure`' script that can be passed arguments or variables via the option '`--configure-args`' and '`--configure-vars`', respectively, if necessary. The latter is useful in particular if libraries or header files needed for the package are in non-system directories. In this case, one can use the configure variables `LIBS` and `CPPFLAGS` to specify these locations (and set these via '`--configure-vars`'), see section "Configuration variables" in "R Installation and Administration" for more information. (If these are used more than once on the command line, only the last instance is used.) One can bypass the configure mechanism using the option '`--no-configure`'.

If '`--no-docs`' is given, no help files are built. Options '`--no-text`', '`--no-html`', and '`--no-latex`' suppress creating the text, HTML, and LaTeX versions, respectively. The default is to build help files in all three versions.

If the attempt to install the package fails, leftovers are removed. If the package was already installed, the old version is restored. This happens either if a command encounters an error or if the install is interrupted from the keyboard: after cleaning up the script terminates.

Use `R CMD INSTALL --help` for more usage information.

Packages using the methods package

Packages that require the methods package, and that use functions such as `setMethod` or `setClass`, should be installed either (preferably) using lazy-loading or by creating a binary image: use the flags `LazyLoad` and `SaveImage` in the '`DESCRIPTION`' file to ensure this.

See Also

`REMOVE` and `library` for information on using several library trees; `update.packages` for automatic update of packages using the internet (or other R level installation of packages, such as by `install.packages`);

the section on "Add-on packages" in "R Installation and Administration" and the chapter on "Creating R packages" in "Writing R Extensions" (see the '`doc/manual`' subdirectory of the R source tree).

`installed.packages` *Find Installed Packages*

Description

Find (or retrieve) details of all packages installed in the specified libraries.

Usage

```
installed.packages(lib.loc = NULL, priority = NULL,
                   noCache = FALSE, fields = NULL)
```

Arguments

`lib.loc`	character vector describing the location of R library trees to search through.
`priority`	character vector or `NULL` (default). If non-null, used to select packages; `"high"` is equivalent to `c("base", "recommended")`. To select all packages without an assigned priority use `priority = "NA"`.
`noCache`	Do not use cached information.
`fields`	a character vector giving the fields to extract from each package's `DESCRIPTION` file in addition to the default ones, or `NULL` (default). Unavailable fields result in `NA` values.

Details

`installed.packages` scans the 'DESCRIPTION' files of each package found along `lib.loc` and returns a matrix of package names, library paths and version numbers.

Note: this works with package names, not bundle names, and for versioned installs with the name under which the package is installed, in the style `mypkg_1.3-7`.

The information found is cached (by library) for the R session and specified `fields` argument, and updated only if the top-level library directory has been altered, for example by installing or removing a package. If the cached information becomes confused, it can be refreshed by running `installed.packages(noCache = TRUE)`.

Value

A matrix with one row per package, row names the package names and column names `"Package"`, `"LibPath"`, `"Version"`, `"Priority"`, `"Bundle"`, `"Contains"`, `"Depends"`, `"Suggests"`, `"Imports"` and `"Built"` (the R version the package was built under). Additional columns can be specified using the `fields` argument.

See Also

`update.packages`, `INSTALL`, `REMOVE`.

Examples

```
str(ip <- installed.packages(priority = "high"))
ip[, c(1,3:5)]
plic <- installed.packages(priority = "high", fields="License")
## what licenses are there:
table( plic[,"License"] )
```

LINK *Create Executable Programs*

Description

Front-end for creating executable programs.

Usage

```
R CMD LINK [options] linkcmd
```

Arguments

linkcmd	a list of commands to link together suitable object files (include library objects) to create the executable program.
options	further options to control the linking, or for obtaining information about usage and version.

Details

The linker front-end is useful in particular when linking against the R shared library, in which case linkcmd must contain -lR but need not specify its library path.

Currently only works if the C compiler is used for linking, and no C++ code is used.

Use R CMD LINK --help for more usage information.

Note

Some binary distributions of R have LINK in a separate bundle, e.g. an R-devel RPM.

---

localeToCharset *Select a Suitable Encoding Name from a Locale Name*

Description

This functions aims to find a suitable coding for the locale named, by default the current locale, and if it is a UTF-8 locale a suitable single-byte encoding.

Usage

```
localeToCharset(locale = Sys.getlocale("LC_CTYPE"))
```

Arguments

locale	character string naming a locale.

Details

The operation differs by OS. Locale names are normaly like es_MX.iso88591. If final component indicates an encoding and it is not utf8 we just need to look up the equivalent encoding name. Otherwise, the language (here es) is used to choose a primary or fallback encoding.

In the C locale the answer will be "ASCII".

Value

A character vector naming an encoding and possibly a fallback single-encoding, NA if unknown.

Note

The encoding names are those used by libiconv, and ought also to work with glibc but maybe not with commercial Unixen.

See Also

Sys.getlocale, iconv.

Examples

```
localeToCharset()
```

`ls.str` *List Objects and their Structure*

Description

`ls.str` and `lsf.str` are "variations" of `ls` applying `str()` to each matched name, see section 'Value'.

Usage

```
ls.str(pos = 1, pattern, ...,  envir = as.environment(pos),
       mode = "any")

lsf.str(pos = 1, ..., envir = as.environment(pos))

## S3 method for class 'ls_str':
print(x, max.level = 1, give.attr = FALSE, ...)
```

Arguments

`pos`	integer indicating `search` path position.
`pattern`	a regular expression passed to `ls`. Only names matching `pattern` are considered.
`max.level`	maximal level of nesting which is applied for displaying nested structures, e.g., a list containing sub lists. Default 0: Display all nesting levels.
`give.attr`	logical; if `TRUE` (default), show attributes as sub structures.
`envir`	environment to use, see `ls`.
`mode`	character specifying the `mode` of objects to consider. Passed to `exists` and `get`.
`x`	an object of class `"ls_str"`.
`...`	further arguments to pass. and `lsf.str` passes them to `ls.str` which passes them on to `ls`. The (non-exported) print method `print.ls_str` passes them to `str`.

Value

`ls.str` and `lsf.str` return an object of class `"ls_str"`, basically the character vector of matching names (functions only for `lsf.str`), similarly to `ls`, with a `print()` method that calls `str()` on each object.

Author(s)

Martin Maechler

See Also

str, summary, args.

Examples

```
lsf.str()#- how do the functions look like which I am using?
ls.str(mode = "list") #- what are the structured objects I have defined?

## create a few objects
example(glm, echo = FALSE)
ll <- as.list(LETTERS)
print(ls.str(), max.level = 0)# don't show details

## which base functions have "file" in their name ?
lsf.str(pos = length(search()), pattern = "file")
```

---

make.packages.html *Update HTML documentation files*

Description

Functions to re-create the HTML documentation files to reflect all installed packages.

Usage

```
make.packages.html(lib.loc = .libPaths())
```

Arguments

lib.loc	character vector. List of libraries to be included.

Details

This sets up the links from packages in libraries to the '.R' subdirectory of the per-session directory (see tempdir) and then creates the 'packages.html' and 'index.txt' files to point to those links.

If a package is available in more than one library tree, all the copies are linked, after the first with suffix .1 etc.

Value

Logical, whether the function succeeded in recreating the files.

See Also

help.start

`make.socket` *Create a Socket Connection*

Description

With `server = FALSE` attempts to open a client socket to the specified port and host. With `server = TRUE` listens on the specified port for a connection and then returns a server socket. It is a good idea to use `on.exit` to ensure that a socket is closed, as you only get 64 of them.

Usage

```
make.socket(host = "localhost", port, fail = TRUE, server = FALSE)
```

Arguments

`host`	name of remote host
`port`	port to connect to/listen on
`fail`	failure to connect is an error?
`server`	a server socket?

Value

An object of class `"socket"`.

`socket`	socket number. This is for internal use
`port`	port number of the connection
`host`	name of remote computer

Warning

I don't know if the connecting host name returned when `server = TRUE` can be trusted. I suspect not.

Author(s)

Thomas Lumley

References

Adapted from Luke Tierney's code for `XLISP-Stat`, in turn based on code from Robbins and Robbins "Practical UNIX Programming"

See Also

`close.socket`, `read.socket`

Examples

```
daytime <- function(host = "localhost"){
    a <- make.socket(host, 13)
    on.exit(close.socket(a))
    read.socket(a)
}
## Offical time (UTC) from US Naval Observatory
## Not run: daytime("tick.usno.navy.mil")
```

---

memory.size *Report on Memory Allocation*

Description

`memory.size` and `memory.limit` are used to manage the total memory allocation on Windows. On other platforms these are stubs which report infinity with a warning.

Usage

```
memory.size(max = FALSE)

memory.limit(size = NA)
```

Arguments

`max`	logical. If true the maximum amount of memory obtained from the OS is reported, otherwise the amount currently in use.
`size`	numeric. If `NA` report the memory size, otherwise request a new limit, in Mb.

Details

To restrict memory usage on a Unix-alike use the facilities of the shell used to launch R, e.g. `limit` or `ulimit`.

Value

Size in bytes: always `Inf`.

See Also

Memory-limits for other limits.

menu *Menu Interaction Function*

Description

menu presents the user with a menu of choices labelled from 1 to the number of choices. To exit without choosing an item one can select '0'.

Usage

```
menu(choices, graphics = FALSE, title = "")
```

Arguments

choices	a character vector of choices
graphics	a logical indicating whether a graphics menu should be used if available.
title	a character string to be used as the title of the menu. NULL is also accepted.

Details

If graphics = TRUE and a windowing system is available (Windows, MacOS X or X11 *via* Tcl/Tk) a listbox widget is used, otherwise a text menu.

Ten or fewer items will be displayed in a single column, more in multiple columns if possible within the current display width.

No title is displayed if title is NULL or "".

Value

The number corresponding to the selected item, or 0 if no choice was made.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

select.list, which is used to implement the graphical menu, and allows multiple selections.

Examples

```
## Not run:
switch(menu(c("List letters", "List LETTERS")) + 1,
       cat("Nothing done\n"), letters, LETTERS)
## End(Not run)
```

`methods` *List Methods for S3 Generic Functions or Classes*

Description

List all available methods for an S3 generic function, or all methods for a class.

Usage

```
methods(generic.function, class)
```

Arguments

`generic.function`	a generic function, or a character string naming a generic function.
`class`	a symbol or character string naming a class: only used if `generic.function` is not supplied.

Details

Function `methods` can be used to find out about the methods for a particular generic function or class. The functions listed are those which *are named like methods* and may not actually be methods (known exceptions are discarded in the code). Note that the listed methods may not be user-visible objects, but often help will be available for them.

If `class` is used, we check that a matching generic can be found for each user-visible object named. If `generic.function` is given, there is a warning if it appears not to be a generic function. (The check for being generic used can be fooled.)

Value

An object of class `"MethodsFunction"`, a character vector of function names with an `"info"` attribute. There is a `print` method which marks with an asterisk any methods which are not visible: such functions can be examined by `getS3method` or `getAnywhere`.

The `"info"` attribute is a data frame, currently with a logical column, `visible` and a factor column `from` (indicating where the methods were found).

Note

This scheme is called *S3* (S version 3). For new projects, it is recommended to use the more flexible and robust *S4* scheme provided in the **methods** package. Functions can have both S3 and S4 methods, and function `showMethods` will list the S4 methods (possibly none).

The original `methods` function was written by Martin Maechler.

References

Chambers, J. M. (1992) *Classes and methods: object-oriented programming in S.* Appendix A of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`S3Methods`, `class`, `getS3method`

Examples

```
methods(summary)
methods(class = "aov")
methods("[[")    # uses C-internal dispatching
methods("$")
methods("$<-")   # replacement function
methods("+")     # binary operator
methods("Math")  # group generic
methods("axis")  # looks like it has methods, but not generic
## Not run:
methods(print)   # over 100
## End(Not run)
```

---

mirrorAdmin *Managing Repository Mirrors*

Description

Functions helping to maintain CRAN, some of them may also be useful for administrators of other repository networks.

Usage

```
mirror2html(mirrors = NULL, file = "mirrors.html",
  head = "mirrors-head.html", foot = "mirrors-foot.html")
checkCRAN(method)
```

Arguments

mirrors	A data frame, by default the CRAN list of mirrors is used.
file	A connection object or a character string.
head	Name of optional header file.
foot	Name of optional footer file.
method	Download method, see download.file.

Details

mirror2html creates the HTML file for the CRAN list of mirrors and invisibly returns the HTML text.

checkCRAN performs a sanity checks on all CRAN mirrors.

modifyList *Recursively Modify Elements of a List*

Description

Modifies a possibly nested list recursively by changing a subset of elements at each level to match a second list.

Usage

```
modifyList(x, val)
```

Arguments

x	a named list, possibly empty.
val	a named list with components to replace corresponding components in x.

Value

A modified version of x, with the modifications determined as follows (here, list elements are identified by their names). Elements in val which are missing from x are added to x. For elements that are common to both but are not both lists themselves, the component in x is replaced by the one in val. For common elements that are both lists, x[[name]] is replaced by modifyList(x[[name]], val[[name]]).

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

Examples

```
foo <- list(a = 1, b = list(c = "a", d = FALSE))
bar <- modifyList(foo, list(e = 2, b = list(d = TRUE)))
str(foo)
str(bar)
```

normalizePath *Express File Paths in Canonical Form*

Description

Convert file paths to canonical form for the platform, to display them in a user-understandable form.

Usage

```
normalizePath(path)
```

Arguments

path	character vector of file paths.

Details

Where the platform supports it this turns paths into absolute paths in their canonical form (no `./`, `../` nor symbolic links).

If the path is not a real path the result is undefined but will most likely be the corresponding input element.

Value

A character vector.

Examples

```
cat(normalizePath(c(R.home(), tempdir())), sep = "\n")
```

---

nsl *Look up the IP Address by Hostname*

Description

Interface to `gethostbyname`.

Usage

```
nsl(hostname)
```

Arguments

`hostname`	the name of the host.

Value

The IP address, as a character string, or `NULL` if the call fails.

Note

This was included as a test of internet connectivity, to fail if the node running R is not connected. It will also return `NULL` if BSD networking is not supported, including the header file 'arpa/inet.h'.

Examples

```
## Not run: nsl("www.r-project.org")
```

`object.size` *Report the Space Allocated for an Object*

Description

Provides an estimate of the memory that is being used to store an R object.

Usage

```
object.size(x)
```

Arguments

`x`	An R object.

Details

Exactly which parts of the memory allocation should be attributed to which object is not clear-cut. This function merely provides a rough indication: it should be reasonably accurate for atomic vectors, but does not detect if elements of a list are shared, for example. (Sharing amongst elements of a character vector is taken into account.)

The calculation is of the size of the object, and excludes the space needed to store its name in the symbol table.

Associated space (e.g. the enviromment of a function and what the pointer in a `EXTPTRSXP` points to) is not included in the calculation.

Object sizes are larger on 64-bit platforms than 32-bit ones, but will very likely be the same on different platforms with the same word length and pointer size.

Value

An estimate of the memory allocation attributable to the object, in bytes.

See Also

`Memory-limits` for the design limitations on object size.

Examples

```
object.size(letters)
object.size(ls)
## find the 10 largest objects in the base package
z <- sapply(ls("package:base"), function(x)
            object.size(get(x, envir = baseenv())))
as.matrix(rev(sort(z))[1:10])
```

`package.skeleton` *Create a Skeleton for a New Source Package*

Description

`package.skeleton` automates some of the setup for a new source package. It creates directories, saves functions and data to appropriate places, and creates skeleton help files and a 'Read-and-delete-me' file describing further steps in packaging.

Usage

```
package.skeleton(name = "anRpackage", list, environment = .GlobalEnv,
                 path = ".", force = FALSE, namespace = FALSE,
                 code_files = character())
```

Arguments

`name`	character string: the directory name for your package.
`list`	character vector naming the R objects to put in the package.
`environment`	if `list` is omitted and `code_files` is empty, the contents of this environment are packaged.
`path`	path to put the package directory in.
`force`	If `FALSE` will not overwrite an existing directory.
`namespace`	a logical indicating whether to add a name space for the package.
`code_files`	a character vector with the paths to R code files to build the package around.

Details

The package sources are placed in subdirectory `name` of `path`.

This tries to create filenames valid for all OSes known to run R. Invalid characters are replaced by '_', invalid names are preceded by 'zz', and finally the converted names are made unique by `make.unique(sep = "_")`. This can be done for code and help files but not data files (which are looked for by name). Also, the code and help files should have names starting with an ASCII letter or digit, and this is checked and if necessary `z` prepended.

If `code_files` are given, these are copied over to the package code directory (so that non-function code objects are not converted to data sets), and the corresponding help files are generated.

When you are done, delete the 'Read-and-delete-me' file, as it should not be distributed.

Value

used for its side-effects.

References

Read the *Writing R Extensions* manual for more details.

Once you have created a *source* package you need to install it: see the *R Installation and Administration* manual, `INSTALL` and `install.packages`.

See Also

prompt

Examples

```
## two functions and two "data sets" :
f <- function(x,y) x+y
g <- function(x,y) x-y
d <- data.frame(a=1, b=2)
e <- rnorm(1000)

package.skeleton(list=c("f","g","d","e"), name="mypkg")
```

---

packageDescription *Package Description*

Description

Parses and returns the 'DESCRIPTION' file of a package.

Usage

```
packageDescription(pkg, lib.loc = NULL, fields = NULL, drop = TRUE,
                   encoding = "")
```

Arguments

pkg	a character string with the package name.
lib.loc	a character vector of directory names of R libraries, or NULL. The default value of NULL corresponds to all libraries currently known. If the default is used, the loaded packages are searched before the libraries.
fields	a character vector giving the tags of fields to return (if other fields occur in the file they are ignored).
drop	If TRUE and the length of fields is 1, then a single character string with the value of the respective field is returned instead of an object of class "packageDescription".
encoding	If there is an Encoding field, to what encoding should re-encodong be attempted? If NA, no re-encoding. The other values are as used by iconv, so the default "" indicates the encoding of the current locale.

Details

A package will not be 'found' unless it has a 'DESCRIPTION' file which contains a valid Version field. Different warnings are given when no package directory is found and when there is a suitable directory but no valid 'DESCRIPTION' file.

Value

If a 'DESCRIPTION' file for the given package is found and can successfully be read, `packageDescription` returns an object of class `"packageDescription"`, which is a named list with the values of the (given) fields as elements and the tags as names, unless `drop = TRUE`.

If parsing the 'DESCRIPTION' file was not successful, it returns a named list of `NA`s with the field tags as names if `fields` is not null, and `NA` otherwise.

See Also

`read.dcf`

Examples

```
packageDescription("stats")
packageDescription("stats", fields = c("Package", "Version"))

packageDescription("stats", fields = "Version")
packageDescription("stats", fields = "Version", drop = FALSE)
```

---

packageStatus *Package Management Tools*

Description

Summarize information about installed packages and packages available at various repositories, and automatically upgrade outdated packages.

Usage

```
packageStatus(lib.loc = NULL, repositories = NULL, method,
              type = getOption("pkgType"))

## S3 method for class 'packageStatus':
summary(object, ...)

## S3 method for class 'packageStatus':
update(object, lib.loc = levels(object$inst$LibPath),
       repositories = levels(object$avail$Repository), ...)

## S3 method for class 'packageStatus':
upgrade(object, ask = TRUE, ...)
```

Arguments

`lib.loc`	a character vector describing the location of R library trees to search through, or `NULL`. The default value of `NULL` corresponds to all libraries currently known.
`repositories`	a character vector of URLs describing the location of R package repositories on the Internet or on the local machine.
`method`	Download method, see `download.file`.

type	type of package distribution: see install.packages.
object	an object of class "packageStatus" as returned by packageStatus.
ask	if TRUE, the user is prompted which packages should be upgraded and which not.
...	currently not used.

Details

The URLs in repositories should be full paths to the appropriate contrib sections of the repositories. The default is contrib.url(getOption("repos")).

There are print and summary methods for the "packageStatus" objects: the print method gives a brief tabular summary and the summary method prints the results.

The update method updates the "packageStatus" object. The upgrade method is similar to update.packages: it offers to install the current versions of those packages which are not currently up-to-date.

Value

An object of class "packageStatus". This is a list with two components

inst	a data frame with columns as the *matrix* returned by installed.packages plus "Status", a factor with levels c("ok", "upgrade"). Only the newest version of each package is reported, in the first repository in which it appears.
avail	a data frame with columns as the *matrix* returned by available.packages plus "Status", a factor with levels c("installed", "not installed", "unavailable")..

See Also

installed.packages, available.packages

Examples

```
## Not run:
x <- packageStatus()
print(x)
summary(x)
upgrade(x)
x <- update(x)
print(x)
## End(Not run)
```

page *Invoke a Pager on an R Object*

Description

Displays a representation of the object named by x in a pager *via* file.show.

Usage

```
page(x, method = c("dput", "print"), ...)
```

Arguments

x	An R object, or a character string naming an object.
method	The default method is to dump the object *via* dput. An alternative is to use print and capture the output to be shown in the pager.
...	additional arguments for dput, print or file.show (such as title).

Details

If x is a length-one character vector, it is used as the name of an object to look up in the environment from which page is called. All other objects are displayed directly.

A default value of title is passed to file.show if one is not supplied in ....

See Also

file.show, edit, fix.

To go to a new page when graphing, see frame.

Examples

```
## Not run:
## four ways to look at the code of 'page'
page(page)              # as an object
page("page")            # a character string
v <- "page"; page(v)    # a length-one character vector
page(utils::page)       # a call
## End(Not run)
```

person *Person Names and Contact Information*

Description

A class and utility methods for holding information about persons like name and email address.

Usage

```
person(first = "", last = "", middle = "", email = "")
personList(...)
as.person(x)
as.personList(x)

## S3 method for class 'person':
as.character(x, ...)
## S3 method for class 'personList':
as.character(x, ...)

## S3 method for class 'person':
toBibtex(object, ...)
## S3 method for class 'personList':
toBibtex(object, ...)
```

Arguments

`first`	character string, first name
`middle`	character string, middle name(s)
`last`	character string, last name
`email`	character string, email address
`...`	for `personList` an arbitrary number of `person` objects
`x`	a character string or an object of class `person` or `personList`
`object`	an object of class `person` or `personList`

Examples

```
## create a person object directly
p1 <- person("Karl", "Pearson", email = "pearson@stats.heaven")
p1

## convert a string
p2 <- as.person("Ronald Aylmer Fisher")
p2

## create one object holding both
p <- personList(p1, p2)
ps <- as.character(p)
ps
as.personList(ps)

## convert to BibTeX author field
toBibtex(p)
```

Description

Utilities for checking whether the sources of an R add-on package work correctly, and for building a source or binary package from them.

Usage

```
R CMD build [options] pkgdirs
R CMD check [options] pkgdirs
```

Arguments

`pkgdirs`	a list of names of directories with sources of R add-on packages.
`options`	further options to control the processing, or for obtaining information about usage and version of the utility.

Details

`R CMD check` checks R add-on packages from their sources, performing a wide variety of diagnostic checks.

`R CMD build` builds R source or binary packages from their sources. It will create index files in the sources if necessary, so it is often helpful to run `build` before `check`.

Use `R CMD foo --help` to obtain usage information on utility `foo`.

Several of the options to `build --binary` are passed to `INSTALL` so consult its help for the details.

See Also

The sections on "Checking and building packages" and "Processing Rd format" in "Writing R Extensions" (see the 'doc/manual' subdirectory of the R source tree).

`INSTALL` is called by `build --binary`.

`prompt` *Produce Prototype of an R Documentation File*

Description

Facilitate the constructing of files documenting R objects.

Usage

```
prompt(object, filename = NULL, name = NULL, ...)

## Default S3 method:
prompt(object, filename = NULL, name = NULL,
       force.function = FALSE, ...)

## S3 method for class 'data.frame':
prompt(object, filename = NULL, name = NULL, ...)
```

Arguments

`object`	an R object, typically a function for the default method.
`filename`	usually, a connection or a character string giving the name of the file to which the documentation shell should be written. The default corresponds to a file whose name is `name` followed by `".Rd"`. Can also be `NA` (see below).
`name`	a character string specifying the name of the object.
`force.function`	a logical. If `TRUE`, treat `object` as function in any case.
`...`	further arguments passed to or from other methods.

Details

Unless `filename` is `NA`, a documentation shell for `object` is written to the file specified by `filename`, and a message about this is given. For function objects, this shell contains the proper function and argument names. R documentation files thus created still need to be edited and moved into the 'man' subdirectory of the package containing the object to be documented.

If `filename` is `NA`, a list-style representation of the documentation shell is created and returned. Writing the shell to a file amounts to `cat(unlist(x), file = filename, sep = "\n")`, where `x` is the list-style representation.

When `prompt` is used in `for` loops or scripts, the explicit `name` specification will be useful.

Value

If `filename` is `NA`, a list-style representation of the documentation shell. Otherwise, the name of the file written to is returned invisibly.

Warning

The default filename may not be a valid filename under limited file systems (e.g. those on Windows).

Currently, calling `prompt` on a non-function object assumes that the object is in fact a data set and hence documents it as such. This may change in future versions of R. Use `promptData` to create documentation skeletons for data sets.

Note

The documentation file produced by `prompt.data.frame` does not have the same format as many of the data frame documentation files in the **base** package. We are trying to settle on a preferred format for the documentation.

Author(s)

Douglas Bates for `prompt.data.frame`

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

promptData, help and the chapter on "Writing R documentation" in "Writing R Extensions" (see the 'doc/manual' subdirectory of the R source tree).

For creation of many help pages (for a package), see package.skeleton.

To prompt the user for input, see readline.

Examples

```
require(graphics)
prompt(plot.default)
prompt(interactive, force.function = TRUE)
unlink("plot.default.Rd")
unlink("interactive.Rd")

prompt(women) # data.frame
unlink("women.Rd")

prompt(sunspots) # non-data.frame data
unlink("sunspots.Rd")
```

---

promptData *Generate a Shell for Documentation of Data Sets*

Description

Generates a shell of documentation for a data set.

Usage

```
promptData(object, filename = NULL, name = NULL)
```

Arguments

object	an R object to be documented as a data set.
filename	usually, a connection or a character string giving the name of the file to which the documentation shell should be written. The default corresponds to a file whose name is name followed by ".Rd". Can also be NA (see below).
name	a character string specifying the name of the object.

Details

Unless filename is NA, a documentation shell for object is written to the file specified by filename, and a message about this is given.

If filename is NA, a list-style representation of the documentation shell is created and returned. Writing the shell to a file amounts to cat(unlist(x), file = filename, sep = "\n"), where x is the list-style representation.

Currently, only data frames are handled explicitly by the code.

Value

If `filename` is `NA`, a list-style representation of the documentation shell. Otherwise, the name of the file written to is returned invisibly.

Warning

This function is still experimental. Both interface and value might change in future versions. In particular, it may be preferable to use a character string naming the data set and optionally a specification of where to look for it instead of using `object/name` as we currently do. This would be different from `prompt`, but consistent with other prompt-style functions in package **methods**, and also allow prompting for data set documentation without explicitly having to load the data set.

See Also

`prompt`

Examples

```
promptData(sunspots)
unlink("sunspots.Rd")
```

---

promptPackage *Generate a Shell for Documentation of a Package*

Description

Generates a shell of documentation for an installed or source package.

Usage

```
promptPackage(package, lib.loc = NULL, filename = NULL,
              name = NULL, final = FALSE)
```

Arguments

`package`	the name of an *installed* or *source* package to be documented.
`lib.loc`	a character vector describing the location of R library trees to search through, or `NULL`. The default value of `NULL` corresponds to all libraries currently known. For a source package this should specify the parent directory of the package's sources.
`filename`	usually, a connection or a character string giving the name of the file to which the documentation shell should be written. The default corresponds to a file whose name is `name` followed by `".Rd"`. Can also be `NA` (see below).
`name`	a character string specifying the name of the help topic, typically of the form `<pkg>-package`.
`final`	a logical value indicating whether to attempt to create a usable version of the help topic, rather than just a shell.

Details

Unless `filename` is `NA`, a documentation shell for `package` is written to the file specified by `filename`, and a message about this is given.

If `filename` is `NA`, a list-style representation of the documentation shell is created and returned. Writing the shell to a file amounts to `cat(unlist(x), file = filename, sep = "\n")`, where `x` is the list-style representation.

If `final` is `TRUE`, the generated documentation will not include the place-holder slots for manual editing, it will be usable as-is. In most cases a manually edited file is preferable (but `final = TRUE` is certainly less work).

Value

If `filename` is `NA`, a list-style representation of the documentation shell. Otherwise, the name of the file written to is returned invisibly.

See Also

`prompt`

Examples

```
filename <- tempfile()
promptPackage("utils", file=filename)
file.show(filename)
unlink(filename)
```

---

read.DIF *Data Input from Spreadsheet*

Description

Reads a file in Data Interchange Format (DIF) and creates a data frame from it. DIF is a format for data matrices such as single spreadsheets.

Usage

```
read.DIF(file, header = FALSE,
           dec = ".", row.names, col.names,
           as.is = !stringsAsFactors,
           na.strings = "NA", colClasses = NA, nrows = -1,
           skip = 0, check.names = TRUE,
           blank.lines.skip = TRUE,
           stringsAsFactors = default.stringsAsFactors())
```

Arguments

`file`	the name of the file which the data are to be read from, or a connection, or a complete URL.

`header` a logical value indicating whether the spreadsheet contains the names of the variables as its first line. If missing, the value is determined from the file format: `header` is set to `TRUE` if and only if the first row contains only character values and the top left cell is empty.

`dec` the character used in the file for decimal points.

`row.names` a vector of row names. This can be a vector giving the actual row names, or a single number giving the column of the table which contains the row names, or character string giving the name of the table column containing the row names.

If there is a header and the first row contains one fewer field than the number of columns, the first column in the input is used for the row names. Otherwise if `row.names` is missing, the rows are numbered.

Using `row.names = NULL` forces row numbering.

`col.names` a vector of optional names for the variables. The default is to use `"V"` followed by the column number.

`as.is` the default behavior of `read.DIF` is to convert character variables (which are not converted to logical, numeric or complex) to factors. The variable `as.is` controls the conversion of columns not otherwise specified by `colClasses`. Its value is either a vector of logicals (values are recycled if necessary), or a vector of numeric or character indices which specify which columns should not be converted to factors.

Note: to suppress all conversions including those of numeric columns, set `colClasses = "character"`.

Note that `as.is` is specified per column (not per variable) and so includes the column of row names (if any) and any columns to be skipped.

`na.strings` a character vector of strings which are to be interpreted as `NA` values. Blank fields are also considered to be missing values in logical, integer, numeric and complex fields.

`colClasses` character. A vector of classes to be assumed for the columns. Recycled as necessary, or if the character vector is named, unspecified values are taken to be `NA`.

Possible values are `NA` (when `type.convert` is used), `"NULL"` (when the column is skipped), one of the atomic vector classes (logical, integer, numeric, complex, character, raw), or `"factor"`, `"Date"` or `"POSIXct"`. Otherwise there needs to be an `as` method (from package **methods**) for conversion from `"character"` to the specified formal class.

Note that `colClasses` is specified per column (not per variable) and so includes the column of row names (if any).

`nrows` the maximum number of rows to read in. Negative values are ignored.

`skip` the number of lines of the data file to skip before beginning to read data.

`check.names` logical. If `TRUE` then the names of the variables in the data frame are checked to ensure that they are syntactically valid variable names. If necessary they are adjusted (by `make.names`) so that they are, and also to ensure that there are no duplicates.

`blank.lines.skip`
logical: if `TRUE` blank lines in the input are ignored.

`stringsAsFactors`
logical: should character vectors be converted to factors?

Value

A data frame (data.frame) containing a representation of the data in the file. Empty input is an error unless col.names is specified, when a 0-row data frame is returned: similarly giving just a header line if header = TRUE results in a 0-row data frame.

Note

The columns referred to in as.is and colClasses include the column of row names (if any).

Less memory will be used if colClasses is specified as one of the six atomic vector classes.

References

The DIF format specification can be found by searching on http://www.wotsit.org/; the optional header fields are ignored. See also http://en.wikipedia.org/wiki/Data_Interchange_Format.

The term is likely to lead to confusion: Windows will have a 'Windows Data Interchange Format (DIF) data format' as part of its WinFX system, which may or may not be compatible.

See Also

The *R Data Import/Export* manual.

scan, type.convert, read.fwf for reading *f*ixed *w*idth *f*ormatted input; read.table; data.frame.

---

read.fortran *Read fixed-format data*

Description

Read fixed-format data files using Fortran-style format specifications.

Usage

```
read.fortran(file, format, ..., as.is = TRUE, colClasses = NA)
```

Arguments

file	File or connection to read from
format	Character vector or list of vectors. See Details below.
...	Other arguments for read.table
as.is	Keep characters as characters?
colClasses	Variable classes to override defaults. See read.table for details.

Details

The format for a field is of one of the following forms: `rFl.d`, `rDl.d`, `rXl`, `rAl`, `rIl`, where `l` is the number of columns, `d` is the number of decimal places, and `r` is the number of repeats. `F` and `D` are numeric formats, `A` is character, `I` is integer, and `X` indicates columns to be skipped. The repeat code `r` and decimal place code `d` are always optional. The length code `l` is required except for `X` formats when `r` is present.

For a single-line record, `format` should be a character vector. For a multiline record it should be a list with a character vector for each line.

Skipped (`X`) columns are not passed to `read.table`, so `colClasses`, `col.names`, and similar arguments passed to `read.table` should not reference these columns.

Value

A data frame

See Also

`read.fwf`, `read.csv`

Examples

```
ff <- tempfile()
cat(file=ff, "123456", "987654", sep="\n")
read.fortran(ff, c("F2.1","F2.0","I2"))
read.fortran(ff, c("2F1.0","2X","2A1"))
unlink(ff)
cat(file=ff, "123456AB", "987654CD", sep="\n")
read.fortran(ff, list(c("2F3.1","A2"), c("3I2","2X")))
unlink(ff)
```

`read.fwf` *Read Fixed Width Format Files*

Description

Read a "table" of **f**ixed **w**idth **f**ormatted data into a `data.frame`.

Usage

```
read.fwf(file, widths, header = FALSE, sep = "\t", as.is = FALSE,
         skip = 0, row.names, col.names, n = -1,
         buffersize = 2000, ...)
```

Arguments

`file`	the name of the file which the data are to be read from. Alternatively, `file` can be a `connection`, which will be opened if necessary, and if so closed at the end of the function call.
`widths`	integer vector, giving the widths of the fixed-width fields (of one line), or list of integer vectors giving widths for multiline records.

header	a logical value indicating whether the file contains the names of the variables as its first line. If present, the names must be delimited by sep.
sep	character; the separator used internally; should be a character that does not occur in the file (except in the header).
as.is	see read.table.
skip	number of initial lines to skip; see read.table.
row.names	see read.table.
col.names	see read.table.
n	the maximum number of records (lines) to be read, defaulting to no limit.
buffersize	Maximum number of lines to read at one time
...	further arguments to be passed to read.table. Useful further arguments include na.strings, colClasses and strip.white.

Details

Multiline records are concatenated to a single line before processing. Fields that are of zero-width or are wholly beyond the end of the line in file are replaced by NA.

Negative-width fields are used to indicate columns to be skipped, eg -5 to skip 5 columns. These fields are not seen by read.table and so should not be included in a col.names or colClasses argument (nor in the header line, if present).

Reducing the buffersize argument may reduce memory use when reading large files with long lines. Increasing buffersize may result in faster processing when enough memory is available.

Value

A data.frame as produced by read.table which is called internally.

Author(s)

Brian Ripley for R version: original Perl by Kurt Hornik.

See Also

scan and read.table.

Examples

```
ff <- tempfile()
cat(file=ff, "123456", "987654", sep="\n")
read.fwf(ff, width=c(1,2,3))     #> 1 23 456 \ 9 87 654
read.fwf(ff, width=c(1,-2,3))    #> 1 456 \ 9 654
unlink(ff)
cat(file=ff, "123", "987654", sep="\n")
read.fwf(ff, width=c(1,0, 2,3))     #> 1 NA 23 NA \ 9 NA 87 654
unlink(ff)
cat(file=ff, "123456", "987654", sep="\n")
read.fwf(ff, width=list(c(1,0, 2,3), c(2,2,2)))     #> 1 NA 23 456 98 76 54
unlink(ff)
```

`read.socket` *Read from or Write to a Socket*

Description

`read.socket` reads a string from the specified socket, `write.socket` writes to the specified socket. There is very little error checking done by either.

Usage

```
read.socket(socket, maxlen = 256, loop = FALSE)
write.socket(socket, string)
```

Arguments

`socket`	a socket object
`maxlen`	maximum length of string to read
`loop`	wait for ever if there is nothing to read?
`string`	string to write to socket

Value

`read.socket` returns the string read.

Author(s)

Thomas Lumley

See Also

`close.socket`, `make.socket`

Examples

```
finger <- function(user, host = "localhost", port = 79, print = TRUE)
{
    if (!is.character(user))
        stop("user name must be a string")
    user <- paste(user,"\r\n")
    socket <- make.socket(host, port)
    on.exit(close.socket(socket))
    write.socket(socket, user)
    output <- character(0)
    repeat{
        ss <- read.socket(socket)
        if (ss == "") break
        output <- paste(output, ss)
    }
    close.socket(socket)
    if (print) cat(output)
    invisible(output)
}
```

```
## Not run:
finger("root")  ## only works if your site provides a finger daemon
## End(Not run)
```

`readNEWS` *Read R's NEWS file or a Similar One*

Description

Read R's NEWS file or a similarly formatted one. This is an experimental feature, new in R 2.4.0 and may change in several ways.

Usage

```
readNEWS(file = file.path(R.home(), "NEWS"), trace = FALSE,
         chop = c("first", "1", "par1", "keepAll"))
```

Arguments

`file`	the name of the file which the data are to be read from. Alternatively, `file` can be a `connection`, which will be opened if necessary, and can also be a complete URL. For more details, see the `file` argument of `read.table`.
`trace`	logical indicating if the recursive reading should be traced, i.e., print what it is doing.
`chop`	a character string specifying how the news entries should be *"chopped"*; `chop = "keepAll"` saves the full entries.

Value

An (S3) object of class `"newsTree"`; effectively a `list` of lists which is a tree of NEWS entries.

Note that this is still experimental and may change in the future.

Examples

```
NEWStr  <- readNEWS(trace = TRUE)# chop = "first" ( = "first non-empty")
## keep the full NEWS entry text i.e. "no chopping":
NEWStrA <- readNEWS(chop = "keepAll")
object.size(NEWStr)
object.size(NEWStrA) ## (no chopping) ==> about double the size

str(NEWStr, max = 3)

str(NEWStr[[c("2.3", "2.3.1")]], max=2, vec.len=1)

NEWStr [[c("2.3", "2.3.1", "NEW FEATURES")]]
NEWStrA[[c("2.4", "2.4.0", "NEW FEATURES")]]
```

`recover` *Browsing after an Error*

Description

This function allows the user to browse directly on any of the currently active function calls, and is suitable as an error option. The expression `options(error=recover)` will make this the error option.

Usage

```
recover()
```

Details

When called, `recover` prints the list of current calls, and prompts the user to select one of them. The standard R `browser` is then invoked from the corresponding environment; the user can type ordinary S language expressions to be evaluated in that environment.

When finished browsing in this call, type `c` to return to `recover` from the browser. Type another frame number to browse some more, or type `0` to exit `recover`.

The use of `recover` largely supersedes `dump.frames` as an error option, unless you really want to wait to look at the error. If `recover` is called in non-interactive mode, it behaves like `dump.frames`. For computations involving large amounts of data, `recover` has the advantage that it does not need to copy out all the environments in order to browse in them. If you do decide to quit interactive debugging, call `dump.frames` directly while browsing in any frame (see the examples).

WARNING: The special `Q` command to go directly from the browser to the prompt level of the evaluator currently interacts with `recover` to effectively turn off the error option for the next error (on subsequent errors, `recover` will be called normally).

Value

Nothing useful is returned. However, you *can* invoke `recover` directly from a function, rather than through the error option shown in the examples. In this case, execution continues after you type `0` to exit `recover`.

Compatibility Note

The R `recover` function can be used in the same way as the S-Plus function of the same name; therefore, the error option shown is a compatible way to specify the error action. However, the actual functions are essentially unrelated and interact quite differently with the user. The navigating commands `up` and `down` do not exist in the R version; instead, exit the browser and select another frame.

References

John M. Chambers (1998). *Programming with Data*; Springer.
See the compatibility note above, however.

See Also

browser for details about the interactive computations; options for setting the error option; dump.frames to save the current environments for later debugging.

Examples

```
## Not run:

options(error = recover) # setting the error option

### Example of interaction

> myFit <- lm(y ~ x, data = xy, weights = w)
Error in lm.wfit(x, y, w, offset = offset, ...) :
        missing or negative weights not allowed

Enter a frame number, or 0 to exit
1:lm(y ~ x, data = xy, weights = w)
2:lm.wfit(x, y, w, offset = offset, ...)
Selection: 2
Called from: eval(expr, envir, enclos)
Browse[1]> objects() # all the objects in this frame
[1] "method" "n"      "ny"     "offset" "tol"    "w"
[7] "x"      "y"
Browse[1]> w
[1] -0.5013844  1.3112515  0.2939348 -0.8983705 -0.1538642
[6] -0.9772989  0.7888790 -0.1919154 -0.3026882
Browse[1]> dump.frames() # save for offline debugging
Browse[1]> c # exit the browser

Enter a frame number, or 0 to exit
1:lm(y ~ x, data = xy, weights = w)
2:lm.wfit(x, y, w, offset = offset, ...)
Selection: 0 # exit recover
>

## End(Not run)
```

REMOVE *Remove Add-on Packages*

Description

Utility for removing add-on packages.

Usage

```
R CMD REMOVE [options] [-l lib] pkgs
```

Arguments

pkgs	a list with the names of the packages to be removed.
lib	the path name of the R library tree to remove from. May be absolute or relative.
options	further options.

Details

If used as `R CMD REMOVE pkgs` without explicitly specifying `lib`, packages are removed from the library tree rooted at the first directory given in the environment variable `R_LIBS` if this is set and non-null, and to the default library tree (which is rooted at '$R_HOME/library') otherwise.

To remove from the library tree `lib`, use `R CMD REMOVE -l lib pkgs`.

Use `R CMD REMOVE --help` for more usage information.

Note

Some binary distributions of R have `INSTALL` in a separate bundle, e.g. an `R-devel` RPM.

See Also

`INSTALL`, `remove.packages`

`remove.packages` *Remove Installed Packages*

Description

Removes installed packages/bundles and updates index information as necessary.

Usage

```
remove.packages(pkgs, lib, version)
```

Arguments

`pkgs`	a character vector with the names of the package(s) or bundle(s) to be removed.
`lib`	a character vector giving the library directories to remove the packages from. If missing, defaults to the first element in `.libPaths()`.
`version`	A character vector specifying version(s) with versioned installs of the package(s) to remove. If none is provided, the system will remove an unversioned install of the package if one is found, otherwise the latest versioned install.

Details

If an element of `pkgs` matches a bundle name, all the packages in the bundle will be removed. This takes precedence over matching a package name.

`pkgs` and `version` will be recycled if necessary to the length of the longer one.

See Also

`REMOVE` for a command line version; `install.packages` for installing packages.

RHOME *R Home Directory*

Description

Returns the location of the R home directory, which is the root of the installed R tree.

Usage

```
R RHOME
```

Rprof *Enable Profiling of R's Execution*

Description

Enable or disable profiling of the execution of R expressions.

Usage

```
Rprof(filename = "Rprof.out", append = FALSE, interval = 0.02,
       memory.profiling=FALSE)
```

Arguments

`filename`	The file to be used for recording the profiling results. Set to `NULL` or `""` to disable profiling.
`append`	logical: should the file be over-written or appended to?
`interval`	real: time interval between samples.
`memory.profiling`	logical: write memory use information to the file?

Details

Enabling profiling automatically disables any existing profiling to another or the same file.

Profiling works by writing out the call stack every `interval` seconds, to the file specified. Either the `summaryRprof` function or the Perl script `R CMD Rprof` can be used to process the output file to produce a summary of the usage; use `R CMD Rprof --help` for usage information.

Note that the timing interval cannot be too small: once the timer goes off, the information is not recorded until the next clock tick (probably every 10msecs). Thus the interval is rounded to the nearest integer number of clock ticks, and is made to be at least one clock tick (at which resolution the total time spent is liable to be underestimated).

Note

Profiling is not available on all platforms. By default, it is attempted to compile support for profiling. Configure R with '`--disable-R-profiling`' to change this.

As R profiling uses the same mechanisms as C profiling, the two cannot be used together, so do not use `Rprof` in an executable built for profiling.

See Also

The chapter on "Tidying and profiling R code" in "Writing R Extensions" (see the 'doc/manual' subdirectory of the R source tree).

`summaryRprof`

`tracemem`, `Rprofmem` for other ways to track memory use.

Examples

```
## Not run:
Rprof()
## some code to be profiled
Rprof(NULL)
## some code NOT to be profiled
Rprof(append=TRUE)
## some code to be profiled
Rprof(NULL)
...
## Now post-process the output as described in Details
## End(Not run)
```

`Rprofmem` *Enable Profiling of R's Memory Use*

Description

Enable or disable reporting of memory allocation in R.

Usage

```
Rprofmem(filename = "Rprofmem.out", append = FALSE, threshold = 0)
```

Arguments

`filename`	The file to be used for recording the memory allocations. Set to `NULL` or `""` to disable reporting.
`append`	logical: should the file be over-written or appended to?
`threshold`	numeric: allocations on R's "large vector" heap larger than this number of bytes will be reported.

Details

Enabling profiling automatically disables any existing profiling to another or the same file.

Profiling writes the call stack to the specified file every time `malloc` is called to allocate a large vector object or to allocate a page of memory for small objects. The size of a page of memory and the size above which `malloc` is used for vectors are compile-time constants, by default 2000 and 128 bytes respectively.

The profiler tracks allocations, some of which will be to previously used memory and will not increase the total memory use of R.

Value

None

Note

The memory profiler slows down R even when not in use, and so is a compile-time option. The memory profiler can be used at the same time as other R and C profilers.

See Also

The R sampling profiler, `Rprof` also collects memory information.

`tracemem` traces duplications of specific objects.

The "Writing R Extensions" manual section on "Tidying and profiling R code"

Examples

```
## Not run:
## not supported unless R is compiled to support it.
Rprofmem("Rprofmem.out", threshold=1000)
example(glm)
Rprofmem(NULL)
noquote(readLines("Rprofmem.out", n=5))
## End(Not run)
```

RSiteSearch *Search for Key Words or Phrases in the R-help Mailing List Archives or Documentation*

Description

Search for key words or phrases in the R-help mailing list archives, or R manuals and help pages, using the search engine at `http://search.r-project.org` and view them in a web browser.

Usage

```
RSiteSearch(string,
            restrict = c("Rhelp02a", "functions", "docs"),
            format = "normal", sortby = "score",
            matchesPerPage = 20)
```

Arguments

`string`	word(s) or phrase to search. If the words are to be searched as one entity, enclose all words in braces (see example).
`restrict`	character: What areas to search in: `Rhelp02a` for R-help mailing list archive since 2002, `Rhelp01` for mailing list archive before 2002, `docs` for R manuals, `functions` for help pages. Use `c()` to specify more than one.
`format`	`normal` or `short` (no excerpts).

`sortby` How to sort the search results (`score`, `date:late` for sorting by date with latest results first, `date:early` for earliest first, `subject` for subject in alphabetical order, `subject:descending` for reverse alphabetical order, `from` or `from:descending` for sender (when applicable), `size` or `size:descending` for size.)

`matchesPerPage`
How many items to show per page.

Details

This function is designed to work with the search site at `http://search.r-project.org`, and depends on that site continuing to be made available (thanks to Jonathan Baron and the School of Arts and Sciences of the University of Pennslyvania).

Unique partial matches will work for all arguments. Each new browser window will stay open unless you close it.

Value

(Invisibly) the complete URL passed to the browser, including the query string.

Author(s)

Andy Liaw and Jonathan Baron

See Also

`help.search`, `help.start` for local searches.

`browseURL` for how the help file is displayed.

Examples

```
## Not run:
 # need Internet connection
RSiteSearch("{logistic regression}") # matches exact phrase
RSiteSearch("Baron Liaw", res = "Rhelp02")
## End(Not run)
```

`Rtangle` *R Driver for Stangle*

Description

A driver for `Stangle` that extracts R code chunks.

Usage

```
Rtangle()
RtangleSetup(file, syntax, output = NULL, annotate = TRUE,
             split = FALSE, prefix = TRUE, quiet = FALSE)
```

Arguments

file	Name of Sweave source file.
syntax	An object of class SweaveSyntax.
output	Name of output file, default is to remove extension '.nw', '.Rnw' or '.Snw' and to add extension '.R'. Any directory names in file are also removed such that the output is created in the current working directory.
annotate	By default, code chunks are seperated by comment lines specifying the names and numbers of the code chunks. If FALSE, only the code chunks without any decorating comments are extracted.
split	Split output in single files per code chunk?
prefix	If split = TRUE, prefix the chunk labels by the basename of the input file to get output file names?
quiet	If TRUE all progress messages are suppressed.

Author(s)

Friedrich Leisch

References

Friedrich Leisch: Sweave User Manual, 2002
http://www.ci.tuwien.ac.at/~leisch/Sweave

See Also

Sweave, RweaveLatex

RweaveLatex *R/LaTeX Driver for Sweave*

Description

A driver for Sweave that translates R code chunks in LaTeX files.

Usage

```
RweaveLatex()

RweaveLatexSetup(file, syntax, output = NULL, quiet = FALSE,
                 debug = FALSE, echo = TRUE, eval  =  TRUE,
                 split = FALSE, stylepath = TRUE,
                 pdf = TRUE, eps = TRUE)
```

Arguments

`file`	Name of Sweave source file.
`syntax`	An object of class `SweaveSyntax`.
`output`	Name of output file, default is to remove extension '.nw', '.Rnw' or '.Snw' and to add extension '.tex'. Any directory names in `file` are also removed such that the output is created in the current working directory.
`quiet`	If `TRUE` all progress messages are suppressed.
`debug`	If `TRUE`, input and output of all code chunks is copied to the console.
`stylepath`	If `TRUE`, a hard path to the file 'Sweave.sty' installed with this package is set, if `FALSE`, only `\usepackage{Sweave}` is written. The hard path makes the TeX file less portable, but avoids the problem of installing the current version of 'Sweave.sty' to some place in your TeX input path. The argument is ignored if a `\usepackage{Sweave}` is already present in the Sweave source file.
`echo`	set default for option `echo`, see details below.
`eval`	set default for option `eval`, see details below.
`split`	set default for option `split`, see details below.
`pdf`	set default for option `pdf`, see details below.
`eps`	set default for option `eps`, see details below.

Supported Options

RweaveLatex supports the following options for code chunks (the values in parentheses show the default values):

echo: logical (`TRUE`). Include S code in the output file?

eval: logical (`TRUE`). If `FALSE`, the code chunk is not evaluated, and hence no text or graphical output produced.

results: character string (`verbatim`). If `verbatim`, the output of S commands is included in the verbatim-like Soutput environment. If `tex`, the output is taken to be already proper latex markup and included as is. If `hide` then all output is completely suppressed (but the code executed during the weave).

print: logical (`FALSE`) If `TRUE`, each expression in the code chunk is wrapped into a `print()` statement before evaluation, such that the values of all expressions become visible.

term: logical (`TRUE`). If `TRUE`, visibility of values emulates an interactive R session: values of assignments are not printed, values of single objects are printed. If `FALSE`, output comes only from explicit `print` or `cat` statements.

split: logical (`FALSE`). If `TRUE`, text output is written to separate files for each code chunk.

strip.white: character string (`false`). If `true`, blank lines at the beginning and end of output are removed. If `all`, then all blank lines are removed from the output.

prefix: logical (`TRUE`). If `TRUE` generated filenames of figures and output have a common prefix.

prefix.string: a character string, default is the name of the '.Snw' source file.

include: logical (`TRUE`), indicating whether input statements for text output and includegraphics statements for figures should be auto-generated. Use `include = FALSE` if the output should appear in a different place than the code chunk (by placing the input line manually).

fig: logical (`FALSE`), indicating whether the code chunk produces graphical output. Note that only one figure per code chunk can be processed this way.

eps: logical (TRUE), indicating whether EPS figures shall be generated. Ignored if fig = FALSE.

pdf: logical (TRUE), indicating whether PDF figures shall be generated. Ignored if fig = FALSE.

width: numeric (6), width of figures in inch.

height: numeric (6), height of figures in inch.

Author(s)

Friedrich Leisch

References

Friedrich Leisch: Sweave User Manual, 2002
http://www.ci.tuwien.ac.at/~leisch/Sweave

See Also

Sweave, Rtangle

---

savehistory *Load or Save or Display the Commands History*

Description

Load or save or display the commands history.

Usage

```
loadhistory(file = ".Rhistory")
savehistory(file = ".Rhistory")

history(max.show = 25, reverse = FALSE, pattern, ...)

timestamp(stamp = date(),
          prefix = "##------ ", suffix = " ------##",
          quiet = FALSE)
```

Arguments

file	The name of the file in which to save the history, or from which to load it. The path is relative to the current working directory.
max.show	The maximum number of lines to show. Inf will give all of the currently available history.
reverse	logical. If true, the lines are shown in reverse order. Note: this is not useful when there are continuation lines.
pattern	A character string to be matched against the lines of the history
...	Arguments to be passed to grep when doing the matching.
stamp	A value or vector of values to be written into the history.

prefix	A prefix to apply to each line.
suffix	A suffix to apply to each line.
quiet	If TRUE, suppress printing timestamp to the console.

Details

There are several history mechanisms available for the different R consoles, which work in similar but not identical ways. Other uses of R, in particular embedded uses, may have no history. This works under the readline and GNOME and MacOS X consoles, but not otherwise (for example, in batch use or in an embedded application).

The readline history mechanism is controlled by two environment variables: R_HISTSIZE controls the number of lines that are saved (default 512), and R_HISTFILE sets the filename used for the loading/saving of history if requested at the beginning/end of a session (but not the default for these functions). There is no limit on the number of lines of history retained during a session, so setting R_HISTSIZE to a large value has no penalty unless a large file is actually generated.

These variables are read at the time of saving, so can be altered within a session by the use of Sys.putenv.

history shows only unique matching lines if pattern is supplied.

The timestamp function writes a timestamp (or other message) into the history and echos it to the console. On platforms that do not support a history mechanism (where the mechanism does not yet support timestamps) only the console message is printed.

Note

If you want to save the history (almost) every session, you can put a call to savehistory() in .Last. See the examples.

Examples

```
## Not run:
.Last <- function()
    if(interactive()) try(savehistory("~/.Rhistory"))
## End(Not run)
```

---

select.list *Select Items from a List*

Description

Select item(s) from a character vector.

Usage

```
select.list(list, preselect = NULL, multiple = FALSE, title = NULL)
```

Arguments

list	character. A list of items.
preselect	a character vector, or NULL. If non-null and if the string(s) appear in the list, the item(s) are selected initially.
multiple	logical: can more than one item be selected?
title	optional character string for window title.

Details

Under the AQUA interface for MacOS X this brings up a modal dialog box with a (scrollable) list of items, which can be selected by the mouse.

Otherwise it displays a text list from which the user can choose by number(s). The `multiple = FALSE` case uses `menu`. Preselection is only supported for `multiple = TRUE`, where it is indicated by a `"+"` preceding the item.

Value

A character vector of selected items. If `multiple` is false and no item was selected (or `Cancel` was used), `""` is returned. If `multiple` is true and no item was selected (or `Cancel` was used) then a character vector of length 0 is returned.

See Also

`menu`, `tk_select.list` for a graphical version using Tcl/Tk.

Examples

```
## Not run:
select.list(sort(.packages(all.available = TRUE)))
## End(Not run)
```

---

sessionInfo *Collect Information About the Current R Session*

Description

Print version information about R and attached packages.

Usage

```
sessionInfo(package=NULL)
## S3 method for class 'sessionInfo':
print(x, ...)
## S3 method for class 'sessionInfo':
toLatex(object, ...)
```

Arguments

`package`	a character vector naming installed packages. By default all attached packages are used.
`x`	an object of class `"sessionInfo"`.
`object`	an object of class `"sessionInfo"`.
`...`	currently not used.

See Also

`R.version`

Examples

```
sessionInfo()
toLatex(sessionInfo())
```

`setRepositories` *Select Package Repositories*

Description

Interact with the user to choose the package repositories to be used.

Usage

```
setRepositories(graphics = getOption("menu.graphics"))
```

Arguments

`graphics`	Logical. If true and **tcltk** and an X server are available, use a Tk widget, or if under the AQUA interface use a MacOS X widget, otherwise use a text list in the console.

Details

The default list of known repositories is stored in the file '`R_HOME/etc/repositories`'. That file can be edited for a site, or a user can have a personal copy in '`HOME/.R/repositories`' which will take precedence.

The items that are preselected are those that are currently in `options("repos")` plus those marked as default in the list of known repositories.

Value

This function is invoked mainly for its side effect of updating `options("repos")`. It returns (invisibly) the previous `repos` options setting (as a `list` with component `repos`) or `NULL` if no changes were applied.

See Also

`chooseCRANmirror`, `install.packages`.

SHLIB *Build Shared Library for Dynamic Loading*

Description

Compile the given source files and then link all specified object files into a shared library which can be loaded into R using `dyn.load` or `library.dynam`.

Usage

```
R CMD SHLIB [options] [-o libname] files
```

Arguments

`files`	a list specifying the object files to be included in the shared library. You can also include the name of source files (for which the object files are automagically made from their sources) and library linking commands.
`libname`	the full name of the shared library to be built, including the extension (typically '.so' on Unix systems). If not given, the name of the library is taken from the first file.
`options`	Further options to control the processing. Use `R CMD SHLIB --help` for a current list.

Details

`R CMD SHLIB` is the mechanism used by `INSTALL` to compile source code in packages. Please consult section 'Creating shared objects' in the manual 'Writing R Extensions' for how to customize it (for example to add `cpp` flags and to add libraries to the link step) and for details of some of its quirks.

Items in `files` with extensions `.c`, `.cpp`, `.cc`, `.C`, `.f`, `.f90` and `.f95` are regarded as source files, and those with extension `.o` as object files. All other items are passed to the linker.

Note

Some binary distributions of R have `SHLIB` in a separate bundle, e.g. an `R-devel` RPM.

See Also

`COMPILE`, `dyn.load`, `library.dynam`.

The section on "Customizing compilation under Unix" in "R Administration and Installation" (see the 'doc/manual' subdirectory of the R source tree).

The 'R Installation and Administration' and 'Writing R Extensions' manuals.

Examples

```
## Not run:
R CMD SHLIB -o mylib.so a.f b.f -L/opt/acml3.5.0/gnu64/lib -lacml
## End(Not run)
```

`str` *Compactly Display the Structure of an Arbitrary R Object*

Description

Compactly display the internal **str**ucture of an R object, a "diagnostic" function and an alternative to `summary` (and to some extent, `dput`). Ideally, only one line for each "basic" structure is displayed. It is especially well suited to compactly display the (abbreviated) contents of (possibly nested) lists. The idea is to give reasonable output for **any** R object. It calls `args` for (non-primitive) function objects.

`strOptions()` is a convenience function for setting `options(str = .)`, see the examples.

Usage

```
str(object, ...)

## S3 method for class 'data.frame':
str(object, ...)

## Default S3 method:
str(object, max.level = NA,
    vec.len  = strO$vec.len, digits.d = strO$digits.d,
    nchar.max = 128, give.attr = TRUE, give.length = TRUE,
    width = getOption("width"), nest.lev = 0,
    indent.str = paste(rep.int(" ", max(0, nest.lev + 1)),
                       collapse = ".."),
    comp.str="$ ", no.list = FALSE, envir = baseenv(),
    strict.width = strO$strict.width, ...)

strOptions(strict.width = "no", digits.d = 3, vec.len = 4)
```

Arguments

`object`	any R object about which you want to have some information.
`max.level`	maximal level of nesting which is applied for displaying nested structures, e.g., a list containing sub lists. Default NA: Display all nesting levels.
`vec.len`	numeric (>= 0) indicating how many "first few" elements are displayed of each vector. The number is multiplied by different factors (from .5 to 3) depending on the kind of vector. Defaults to the `vec.len` component of option `"str"` (see `options`) which defaults to 4.
`digits.d`	number of digits for numerical components (as for `print`). Defaults to the `digits.d` component of option `"str"` which defaults to 3.
`nchar.max`	maximal number of characters to show for `character` strings. Longer strings are truncated, see `longch` example below.
`give.attr`	logical; if `TRUE` (default), show attributes as sub structures.
`give.length`	logical; if `TRUE` (default), indicate length (as `[1:...]`).
`width`	the page width to be used. The default is the currently active `options("width")`; note that this has only a weak effect, unless `strict.width` is not `"no"`.

`nest.lev`	current nesting level in the recursive calls to `str`.
`indent.str`	the indentation string to use.
`comp.str`	string to be used for separating list components.
`no.list`	logical; if true, no "list of .." is nor the class is printed.
`envir`	the environment to be used for *promise* (see `delayedAssign`) objects only.
`strict.width`	string indicating if the `width` argument's specification should be followed strictly, one of the values `c("no", "cut", "wrap")`. Defaults to the `strict.width` component of option `"str"` (see `options`) which defaults to `"no"` for back compatibility reasons; `"wrap"` uses `strwrap(*, width=width)` whereas `"cut"` cuts directly to `width`. Note that a small `vec.length` may be better than setting `strict.width = "wrap"`.
`...`	potential further arguments (required for Method/Generic reasons).

Value

`str` does not return anything, for efficiency reasons. The obvious side effect is output to the terminal.

Author(s)

Martin Maechler ⟨maechler@stat.math.ethz.ch⟩ since 1990.

See Also

`ls.str` for *listing* objects with their structure; `summary`, `args`.

Examples

```
require(stats)
## The following examples show some of 'str' capabilities
str(1:12)
str(ls)
str(args) #- more useful than  args(args) !
str(freeny)
str(str)
str(.Machine, digits = 20)
str( lsfit(1:9,1:9))
str( lsfit(1:9,1:9), max = 1)
str( lsfit(1:9,1:9), width = 60, strict.width = "cut")
str( lsfit(1:9,1:9), width = 60, strict.width = "wrap")
op <- options(); str(op) # save first;
                         # otherwise internal options() is used.
need.dev <- !exists(".Device") || is.null(.Device) || .Device == "null device"
{ if(need.dev) postscript()
  str(par())
  if(need.dev) graphics.off()
}
ch <- letters[1:12]; is.na(ch) <- 3:5
str(ch) # character NA's

nchar(longch <- paste(rep(letters,100), collapse=""))
str(longch)
str(longch, nchar.max = 52)
```

```
str(longch, strict.width = "wrap")

## Settings for narrow transcript :
op <- options(width = 60,
              str = strOptions(strict.width = "wrap"))
str(lsfit(1:9,1:9))
str(options())
## reset to previous:
options(op)


str(quote( { A+B; list(C,D) } ))


## S4 classes :
require(stats4)
x <- 0:10; y <- c(26, 17, 13, 12, 20, 5, 9, 8, 5, 4, 8)
ll <- function(ymax=15, xh=6)
      -sum(dpois(y, lambda=ymax/(1+x/xh), log=TRUE))
fit <- mle(ll)
str(fit)
```

`summaryRprof` *Summarise Output of R Sampling Profiler*

Description

Summarise the output of the `Rprof` function to show the amount of time used by different R functions.

Usage

```
summaryRprof(filename = "Rprof.out", chunksize = 5000,
               memory=c("none","both","tseries","stats"),
               index=2, diff=TRUE, exclude=NULL)
```

Arguments

`filename`	Name of a file produced by `Rprof()`
`chunksize`	Number of lines to read at a time
`memory`	Summaries for memory information. See Details below
`index`	How to summarize the stack trace for memory information. See Details below.
`diff`	If `TRUE` memory summaries use change in memory rather than current memory
`exclude`	Functions to exclude when summarizing the stack trace for memory summaries

Details

This function is an alternative to `R CMD Rprof`. It provides the convenience of an all-R implementation but will be slower for large files.

As the profiling output file could be larger than available memory, it is read in blocks of `chunksize` lines. Increasing `chunksize` will make the function run faster if sufficient memory is available.

When called with `memory.profiling=TRUE`, the profiler writes information on three aspects of memory use: vector memory in small blocks on the R heap, vector memory in large blocks (from `malloc`), memory in nodes on the R heap. It also records the number of calls to the internal function `duplicate` in the time interval. `duplicate` is called by C code when arguments need to be copied. Note that the profiler does not track which function actually allocated the memory.

With `memory="both"` the change in total memory (truncated at zero) is reported in addition to timing data.

With `memory="tseries"` or `memory="stats"` the `index` argument specifies how to summarize the stack trace. A positive number specifies that many calls from the bottom of the stack; a negative number specifies the number of calls from the top of the stack. With `memory="tseries"` the index is used to construct labels and may be a vector to give multiple sets of labels. With `memory="stats"` the index must be a single number and specifies how to aggregate the data to the maximum and average of the memory statistics. With both `memory="tseries"` and `memory="stats"` the argument `diff=TRUE` asks for summaries of the increase in memory use over the sampling interval and `diff=FALSE` asks for the memory use at the end of the interval.

Value

If `memory="none"`,a list with components

`by.self` Timings sorted by 'self' time

`by.total` Timings sorted by 'total' time

`sampling.time` Total length of profiling run

If `memory="both"` the same list but with memory consumption in Mb in addition to the timings.

If `memory="tseries"` a data frame giving memory statistics over time

If `memory="stats"` a `by` object giving memory statistics by function.

See Also

The chapter on "Tidying and profiling R code" in "Writing R Extensions" (see the 'doc/manual' subdirectory of the R source tree).

`Rprof`

`tracemem` traces copying of an object via the C function `duplicate`.

`Rprofmem` is a non-sampling memory use profiler.

`http://developer.r-project.org/memory-profiling.html`

Examples

```
## Not run:
## Rprof() is not available on all platforms
Rprof(tmp <- tempfile())
example(glm)
Rprof()
summaryRprof(tmp)
unlink(tmp)
## End(Not run)
```

Sweave *Automatic Generation of Reports*

Description

`Sweave` provides a flexible framework for mixing text and S code for automatic report generation. The basic idea is to replace the S code with its output, such that the final document only contains the text and the output of the statistical anlysis.

Usage

```
Sweave(file, driver = RweaveLatex(),
       syntax = getOption("SweaveSyntax"), ...)

Stangle(file, driver = Rtangle(),
        syntax = getOption("SweaveSyntax"), ...)
```

Arguments

`file`	Name of Sweave source file.
`driver`	The actual workhorse, see details below.
`syntax`	An object of class `SweaveSyntax` or a character string with its name. The default installation provides `SweaveSyntaxNoweb` and `SweaveSyntaxLatex`.
`...`	Further arguments passed to the driver's setup function.

Details

Automatic generation of reports by mixing word processing markup (like latex) and S code. The S code gets replaced by its output (text or graphs) in the final markup file. This allows to re-generate a report if the input data change and documents the code to reproduce the analysis in the same file that also produces the report.

`Sweave` combines the documentation and code chunks together (or their output) into a single document. `Stangle` extracts only the code from the Sweave file creating a valid S source file (that can be run using `source`). Code inside `\Sexpr{}` statements is ignored by `Stangle`.

`Stangle` is just a frontend to `Sweave` using a simple driver by default, which discards the documentation and concatenates all code chunks the current S engine understands.

Hook Functions

Before each code chunk is evaluated, a number of hook functions can be executed. If `getOption("SweaveHooks")` is set, it is taken to be a collection of hook functions. For each logical option of a code chunk (`echo`, `print`, ...) a hook can be specified, which is executed if and only if the respective option is `TRUE`. Hooks must be named elements of the list returned by `getOption("SweaveHooks")` and be functions taking no arguments. E.g., if option `"SweaveHooks"` is defined as `list(fig = foo)`, and `foo` is a function, then it would be executed before the code in each figure chunk. This is especially useful to set defaults for the graphical parameters in a series of figure chunks.

Note that the user is free to define new Sweave options and associate arbitrary hooks with them. E.g., one could define a hook function for option `clean` that removes all objects in the global environment. Then all code chunks with `clean = TRUE` would start operating on an empty workspace.

Syntax Definition

Sweave allows a very flexible syntax framework for marking documentation and text chunks. The default is a noweb-style syntax, as alternative a latex-style syntax can be used. See the user manual for details.

Author(s)

Friedrich Leisch

References

Friedrich Leisch: Sweave User Manual, 2002
http://www.ci.tuwien.ac.at/~leisch/Sweave

Friedrich Leisch: Dynamic generation of statistical reports using literate data analysis. In W. Härdle and B. Rönz, editors, Compstat 2002 - Proceedings in Computational Statistics, pages 575–580. Physika Verlag, Heidelberg, Germany, 2002. ISBN 3-7908-1517-9.

See Also

`RweaveLatex`, `Rtangle`

Examples

```
testfile <- system.file("Sweave", "Sweave-test-1.Rnw", package = "utils")

## enforce par(ask=FALSE)
options(par.ask.default=FALSE)

## create a LaTeX file
Sweave(testfile)

## create an S source file from the code chunks
Stangle(testfile)
## which can be simply sourced
source("Sweave-test-1.R")
```

SweaveSyntConv *Convert Sweave Syntax*

Description

This function converts the syntax of files in Sweave format to another Sweave syntax definition.

Usage

```
SweaveSyntConv(file, syntax, output = NULL)
```

Arguments

file	Name of Sweave source file.
syntax	An object of class SweaveSyntax or a character string with its name giving the target syntax to which the file is converted.
output	Name of output file, default is to remove the extension from the input file and to add the default extension of the target syntax. Any directory names in file are also removed such that the output is created in the current working directory.

Author(s)

Friedrich Leisch

References

Friedrich Leisch: Sweave User Manual, 2002
http://www.ci.tuwien.ac.at/~leisch/Sweave

See Also

RweaveLatex, Rtangle

Examples

```
testfile <- system.file("Sweave", "Sweave-test-1.Rnw", package = "utils")

## convert the file to latex syntax
SweaveSyntConv(testfile, SweaveSyntaxLatex)

## and run it through Sweave
Sweave("Sweave-test-1.Stex")
```

toLatex *Converting R Objects to BibTeX or LaTeX*

Description

These methods convert R objects to character vectors with BibTeX or LaTeX markup.

Usage

```
toBibtex(object, ...)
toLatex(object, ...)
## S3 method for class 'Bibtex':
print(x, prefix="", ...)
## S3 method for class 'Latex':
print(x, prefix="", ...)
```

Arguments

object	object of a class for which a toBibtex or toLatex method exists.
x	object of class "Bibtex" or "Latex".
prefix	a character string which is printed at the beginning of each line, mostly used to insert whitespace for indentation.
...	currently not used in the print methods.

Details

Objects of class "Bibtex" or "Latex" are simply character vectors where each element holds one line of the corresponding BibTeX or LaTeX file.

See Also

citEntry and sessionInfo for examples

tracemem *Trace copying of objects*

Description

This function marks an object so that a message is printed whenever the internal function duplicate is called. This happens when two objects share the same memory and one of them is modified. It is a major cause of hard-to-predict memory use in R.

Usage

```
tracemem(x)
untracemem(x)
retracemem(x, previous=NULL)
```

Arguments

x	An R object, not a function or environment or NULL
previous	An address specification as returned by tracemem or retracemem

Details

This functionality is optional, determined at compilation, because it makes R run a little more slowly even when no objects are being traced. tracemem and untracemem give errors when R is not compiled with memory profiling; retracemem does not (so it can be left in code during development).

When an object is traced any copying of the object by the C function duplicate or by arithmetic or mathmetical operations produces a message to standard output. The message consists of the string memtrace, the identifying strings for the object being copied and the new object being created, and a stack trace showing where the duplication occurred. retracemem() is used to indicate that a variable should be considered a copy of a previous variable (eg after subscripting).

The messages can be turned off with tracingState.

It is not possible to trace functions, as this would conflict with trace and it is not useful to trace NULL, environments, promises, weak references, or external pointer objects, as these are not duplicated.

Value

A string for identifying the object in the trace output.

See Also

trace

Rprofmem

http://developer.r-project.org/memory-profiling.html

Examples

```
## Not run:
a<-1:10
tracemem(a)
## b and a share memory
b<-a
b[1]<-1

## copying in lm
d<-rnorm(10)
tracemem(d)
lm(d~a+log(b))

## f is not a copy and is not traced
f<-d[-1]
f+1
## indicate that f should be traced as a copy of d
retracemem(f, retracemem(d))
f+1

## End(Not run)
```

`update.packages` *Download Packages from CRAN-like repositories*

Description

These functions can be used to automatically compare the version numbers of installed packages with the newest available version on the repositories and update outdated packages on the fly.

Usage

```
update.packages(lib.loc = NULL, repos = getOption("repos"),
                contriburl = contrib.url(repos, type),
                method, instlib = NULL, ask = TRUE, available = NULL,
                ..., checkBuilt = FALSE,
                type = getOption("pkgType"))

available.packages(contriburl = contrib.url(getOption("repos")),
                   method, fields = NULL)

old.packages(lib.loc = NULL, repos = getOption("repos"),
             contriburl = contrib.url(repos),
             method, available = NULL, checkBuilt = FALSE)

new.packages(lib.loc = NULL, repos = getOption("repos"),
             contriburl = contrib.url(repos, type),
             method, available = NULL, ask = FALSE, ...,
             type = getOption("pkgType"))

download.packages(pkgs, destdir, available = NULL,
                  repos = getOption("repos"),
                  contriburl = contrib.url(repos, type),
                  method, type = getOption("pkgType"))

install.packages(pkgs, lib, repos = getOption("repos"),
                 contriburl = contrib.url(repos, type),
                 method, available = NULL, destdir = NULL,
                 installWithVers = FALSE, dependencies = FALSE,
                 type = getOption("pkgType"),
                 configure.args = character(0),
                 clean = FALSE)

contrib.url(repos, type = getOption("pkgType"))
```

Arguments

`lib.loc`	character vector describing the location of R library trees to search through (and update packages therein).
`repos`	character vector, the base URL(s) of the repositories to use, i.e., the URL of the CRAN master such as `"http://cran.r-project.org"` or its Statlib mirror, `"http://lib.stat.cmu.edu/R/CRAN"`. Can be `NULL` to install from local '.tar.gz' files.

`contriburl` URL(s) of the contrib sections of the repositories. Use this argument only if your repository mirror is incomplete, e.g., because you burned only the 'contrib' section on a CD. Overrides argument `repos`. Can also be `NULL` to install from local '.tar.gz' files.

`method` Download method, see `download.file`.

`pkgs` character vector of the short names of packages/bundles whose current versions should be downloaded from the repositories. If `repos = NULL`, a character vector of file paths of '.tar.gz' files. These can be source archives or binary package/bundle archive files (as created by `R CMD build --binary`). Tilde-expansion will be done on the file paths. If this is a zero-length character vector, a listbox of available packages (including those contained in bundles) is presented where possible.

`destdir` directory where downloaded packages are stored.

`available` an object listing packages available at the repositories as returned by `available.packages`.

`lib` character vector giving the library directories where to install the packages. Recycled as needed. If missing, defaults to `.libPaths()[1]`.

`ask` logical indicating whether to ask user before packages are actually downloaded and installed, or the character string `"graphics"`, which brings up a widget to allow the user to (de-)select from the list of packages which could be updated. The latter only works on systems with a GUI version of `select.list`, and is otherwise equivalent to `ask = TRUE`.

`installWithVers`
If `TRUE`, will invoke the install of the package such that it can be referenced by package version.

`checkBuilt` If `TRUE`, a package built under an earlier minor version of R is considered to be 'old'.

`instlib` character string giving the library directory where to install the packages.

`dependencies` logical indicating to also install uninstalled packages on which these packages depend/suggest/import (and so on recursively). Not used if `repos = NULL`. Can also be a character vector, a subset of `c("Depends", "Imports", "Suggests")`.
Only supported if `lib` is of length one (or missing), so it is unambiguous where to install the dependent packages.

`configure.args`
a character vector or a named list. If a character vector with no names is supplied, the elements are concatenated into a single string (separated by a space) and used as the value for the `--configure-args` flag in the call to `R CMD INSTALL`. If the character vector has names, these are assumed to identify values for `--configure-args` for individual packages. This allows one to specify settings for an entire collection of packages which will be used if any of those packages are installed. These settings can therefore be reused and act as default settings. A named list can be used also to the same effect, and that allows multi-element character strings for each package which are concatenated to a single string to be used as the value for `--configure-args`.

`...` (for `update.packages`). Arguments such as `destdir`, `installWithVers` and `dependencies` to be passed to `install.packages`.
(for `new.packages`). Arguments such as `destdir` and `dependencies` to be passed to `install.packages`.

type — character, indicating the type of package to download and install. Possible values are "source" (the default except under the CRAN Mac OS X build), "mac.binary" and "win.binary" (which can be downloaded but not installed).

clean — a logical value indicating whether to specify to add the --clean flag to the call to R CMD INSTALL. This is sometimes used to perform additional operations at the end of the package installation in addition to removing intermediate files.

fields — a character vector giving the fields to extract from the PACKAGES file(s) in addition to the default ones, or NULL (default). Unavailable fields result in NA values.

Details

All of these functions work with the names of a package or bundle (and not the component packages of a bundle, except for install.packages if the repository provides the necessary information).

available.packages returns a matrix of details corresponding to packages/bundles currently available at one or more repositories. The current list of packages is downloaded over the internet (or copied from a local mirror). It returns only packages whose version requirements are met by the running version of R.

old.packages compares the information from available.packages with that from installed.packages and reports installed packages/bundles that have newer versions on the repositories or, if checkBuilt = TRUE, that were built under an earlier minor version of R (for example built under 2.0.x when running R 2.1.1).

new.packages does the same comparison but reports uninstalled packages/bundles that are available at the repositories. It will also give warnings about incompletely installed bundles (provided the information is available) and bundles whose contents has changed. If ask != FALSE it asks which packages should be installed in the first element of lib.loc. NB: versioned installs are not installs of a named package.

download.packages takes a list of package/bundle names and a destination directory, downloads the newest versions and saves them in destdir. If the list of available packages is not given as argument, it is obtained from repositories. If a repository is local, i.e., the URL starts with "file:", then the packages are not downloaded but used directly. (Both "file:" and "file:///" are allowed as prefixes to a file path, the latter for an absolute file path.)

The main function of the set is update.packages. First a list of all packages/bundles found in lib.loc is created and compared with those available at the repostories. If ask = TRUE (the default) packages/bundles with a newer version are reported and for each one the user can specify if it should be updated. If so, the package sources are downloaded from the repositories and installed in the respective library path (or instlib if specified) using the R INSTALL mechanism.

install.packages can be used to install new packages/bundles. It takes a vector of names and a destination library, downloads the packages from the repositories and installs them. (If the library is omitted it defaults to the first directory in .libPaths(), with a warning if there is more than one.) An attempt is made to install the packages in an order that respects their dependencies. This does assume that all the entries in lib are on the default library path for installs (set by R_LIBS).

contrib.url adds the appropriate type-specific path within a repository to each URL in repos.

For install.packages, destdir is the directory to which packages will be downloaded. If it is NULL (the default) a directory downloaded_packages of the session temporary directory will be used (and the files will be deleted at the end of the session).

If repos or contriburl is a vector of length greater than one, the newest version of the package compatible with this version of R is fetched from the first repository on the list within which it is found.

Value

For available.packages, a matrix with one row per package/bundle, row names the package names and column names "Package", "Version", "Priority", "Bundle", "Depends", "Imports", "Suggests" "Contains" and "Repository". Additional columns can be specified using the fields argument.

For old.packages, NULL or a matrix with one row per package/bundle, row names the package names and column names "Package", "LibPath", "Installed" (the version), "Built" (the version built under), "ReposVer" and "Repository".

For new.packages a character vector of package/bundle names, *after* any have been installed.

For download.packages, a two-column matrix of names and destination file names, for those packages/bundles successfully downloaded. If packages are not available or there is a problem with the download, suitable warnings are given.

install.packages and update.packages have no return value.

Warning

Take care when using dependencies with update.packages, for it is unclear where new dependencies should be installed. The current implementation will only allow it if all the packages to be updated are in a single library, when that library will be used.

Note

Some binary distributions of R have INSTALL in a separate bundle, e.g. an R-devel RPM. install.packages will give an error if called on such a system.

See Also

installed.packages, remove.packages

See download.file for how to handle proxies and other options to monitor file transfers.

INSTALL, REMOVE, library, .packages, read.dcf

Examples

```
## Not run:
  install.packages(c("XML_0.99-5.tar.gz",
                     "../../Interfaces/Perl/RSPerl_0.8-0.tar.gz"),
                   repos = NULL,
                   configure.args = c(XML = '--with-xml-config=xml-config',
                                      RSPerl = "--with-modules='IO Fcntl'"))
## End(Not run)
```

url.show *Display a text URL*

Description

Extension of file.show to display text files from a remote server.

Usage

```
url.show(url, title = url, file = tempfile(),
         delete.file = TRUE, method, ...)
```

Arguments

url	The URL to read from.
title	Title for the browser.
file	File to copy to.
delete.file	Delete the file afterwards?
method	File transfer method: see download.file
...	Arguments to pass to file.show.

See Also

url, file.show, download.file

Examples

```
## Not run: url.show("http://lib.stat.cmu.edu/datasets/csb/ch3a.txt")
```

URLencode *Encode or Decode a (partial) URL*

Description

Functions to encode or decode characters in URLs.

Usage

```
URLencode(URL, reserved = FALSE)
URLdecode(URL)
```

Arguments

URL	A character string.
reserved	should reserved characters be encoded? See Details.

Details

Characters in a URL other than the English alphanumeric characters and `$ - _ . + ! * ' ( ) ,` should be encoded as `%` plus a two-digit hexadecimal representation, and any single-byte character can be so encoded. (Multi-byte characters are encoded as byte-by-byte.)

In addition, `; / ? : @ = &` are reserved characters, and should be encoded unless used in their reserved sense, which is scheme specific. The default in `URLencode` is to leave them alone, which is appropriate for `file://` URLs, but probably not for `http://` ones.

Value

A character string.

References

RFC1738, http://www.rfc-editor.org/rfc/rfc1738.txt

Examples

```
(y <- URLencode("a url with spaces and / and @"))
URLdecode(y)
(y <- URLencode("a url with spaces and / and @", reserved=TRUE))
URLdecode(y)
URLdecode("ab%20cd")
```

---

utils-deprecated *Deprecated Functions in Package utils*

Description

These functions are provided for compatibility with older versions of R only, and may be defunct as soon as of the next release.

Usage

```
CRAN.packages(CRAN = getOption("repos"), method,
              contriburl = contrib.url(CRAN))
```

Arguments

`CRAN`	character, an earlier way to specify a repository.
`method`	Download method, see `download.file`.
`contriburl`	URL(s) of the contrib section of the repositories. Use this argument only if your CRAN mirror is incomplete, e.g., because you burned only the 'contrib' section on a CD. Overrides argument `repos`.

See Also

`Deprecated`, `Defunct`

vignette *View or List Vignettes*

Description

View a specified vignette, or list the available ones.

Usage

```
vignette(topic, package = NULL, lib.loc = NULL)

## S3 method for class 'vignette':
print(x, ...)
## S3 method for class 'vignette':
edit(name, ...)
```

Arguments

topic	a character string giving the (base) name of the vignette to view. If omitted, all vignettes from all installed packages are listed.
package	a character vector with the names of packages to search through, or NULL in which case *all* available packages in the library trees specified by lib.loc are searched.
lib.loc	a character vector of directory names of R libraries, or NULL. The default value of NULL corresponds to all libraries currently known.
x, name	Object of class vignette.
...	Ignored by the print method, passed on to file.edit by the edit method.

Details

Function vignette returns an object of the same class, the print method opens a viewer for it. Currently, only PDF versions of vignettes can be viewed. The program specified by the pdfviewer option is used for this. If several vignettes have PDF versions with base name identical to topic, the first one found is used.

If no topics are given, all available vignettes are listed. The corresponding information is returned in an object of class "packageIQR".

The edit method extracts the R code from the vignette to a temporary file and opens the file in an editor (see edit). This makes it very easy to execute the commands line by line, modify them in any way you want to help you test variants, etc.. An alternative way of extracting the R code from the vignette is to run Stangle on the source code of the vignette, see the examples below.

Examples

```
## List vignettes from all *installed* packages (can take long!):
vignette()

## Not run:
## Open the grid intro vignette
vignette("grid")
```

```
## The same
v1 <- vignette("grid")
print(v1)

## Now let us have a closer look at the code
edit(v1)

## An alternative way of extracting the code,
## R file is written to current working directory
Stangle(v1$file)

## A package can have more than one vignette (package grid has several):
vignette(package="grid")
vignette("rotated")
## The same, but without searching for it:
vignette("rotated", package="grid")
## End(Not run)
```

Part II

Chapter 10

The KernSmooth package

`bkde` *Compute a Binned Kernel Density Estimate*

Description

Returns x and y coordinates of the binned kernel density estimate of the probability density of the data.

Usage

```
bkde(x, kernel = "normal", canonical = FALSE, bandwidth,
     gridsize = 401, range.x, truncate = TRUE)
```

Arguments

`x`	vector of observations from the distribution whose density is to be estimated. Missing values are not allowed.
`bandwidth`	the kernel bandwidth smoothing parameter. Larger values of `bandwidth` make smoother estimates, smaller values of `bandwidth` make less smooth estimates.
`kernel`	character string which determines the smoothing kernel. `kernel` can be: `"normal"` - the Gaussian density function (the default). `"box"` - a rectangular box. `"epanech"` - the centred beta(2,2) density. `"biweight"` - the centred beta(3,3) density. `"triweight"` - the centred beta(4,4) density.
`canonical`	logical flag: if `TRUE`, canonically scaled kernels are used.
`gridsize`	the number of equally spaced points at which to estimate the density.
`range.x`	vector containing the minimum and maximum values of `x` at which to compute the estimate. The default is the minimum and maximum data values, extended by the support of the kernel.
`truncate`	logical flag: if `TRUE`, data with `x` values outside the range specified by `range.x` are ignored.

Details

This is the binned approximation to the ordinary kernel density estimate. Linear binning is used to obtain the bin counts. For each `x` value in the sample, the kernel is centered on that `x` and the heights of the kernel at each datapoint are summed. This sum, after a normalization, is the corresponding `y` value in the output.

Value

a list containing the following components:

`x`	vector of sorted `x` values at which the estimate was computed.
`y`	vector of density estimates at the corresponding `x`.

Background

Density estimation is a smoothing operation. Inevitably there is a trade-off between bias in the estimate and the estimate's variability: large bandwidths will produce smooth estimates that may hide local features of the density; small bandwidths may introduce spurious bumps into the estimate.

References

Wand, M. P. and Jones, M. C. (1995). *Kernel Smoothing.* Chapman and Hall, London.

See Also

`density`, `dpik`, `hist`, `ksmooth`.

Examples

```
data(geyser, package="MASS")
x <- geyser$duration
est <- bkde(x, bandwidth=0.25)
plot(est, type="l")
```

---

bkde2D *Compute a 2D Binned Kernel Density Estimate*

Description

Returns the set of grid points in each coordinate direction, and the matrix of density estimates over the mesh induced by the grid points. The kernel is the standard bivariate normal density.

Usage

```
bkde2D(x, bandwidth, gridsize=c(51, 51), range.x, truncate=TRUE)
```

Arguments

x	a two-column matrix containing the observations from the distribution whose density is to be estimated. Missing values are not allowed.
bandwidth	vector containing the bandwidth to be used in each coordinate direction.
gridsize	vector containing the number of equally spaced points in each direction over which the density is to be estimated.
range.x	a list containing two vectors, where each vector contains the minimum and maximum values of x at which to compute the estimate for each direction. The default minimum in each direction is minimum data value minus 1.5 times the bandwidth for that direction. The default maximum is the maximum data value plus 1.5 times the bandwidth for that direction
truncate	logical flag: if TRUE, data with x values outside the range specified by range.x are ignored.

Value

a list containing the following components:

x1	vector of values of the grid points in the first coordinate direction at which the estimate was computed.
x2	vector of values of the grid points in the second coordinate direction at which the estimate was computed.
fhat	matrix of density estimates over the mesh induced by x1 and x2.

Details

This is the binned approximation to the 2D kernel density estimate. Linear binning is used to obtain the bin counts and the Fast Fourier Transform is used to perform the discrete convolutions. For each x1,x2 pair the bivariate Gaussian kernel is centered on that location and the heights of the kernel, scaled by the bandwidths, at each datapoint are summed. This sum, after a normalization, is the corresponding fhat value in the output.

References

Wand, M. P. (1994). Fast Computation of Multivariate Kernel Estimators. *Journal of Computational and Graphical Statistics,* **3**, 433-445.

Wand, M. P. and Jones, M. C. (1995). *Kernel Smoothing.* Chapman and Hall, London.

See Also

bkde, density, hist.

Examples

```
data(geyser, package="MASS")
x <- cbind(geyser$duration, geyser$waiting)
est <- bkde2D(x, bandwidth=c(0.7,7))
contour(est$x1, est$x2, est$fhat)
persp(est$fhat)
```

`bkfe` *Compute a Binned Kernel Functional Estimate*

Description

Returns an estimate of a binned approximation to the kernel estimate of the specified density functional. The kernel is the standard normal density.

Usage

```
bkfe(x, drv, bandwidth, gridsize = 401, range.x, binned = FALSE,
     truncate = TRUE)
```

Arguments

`x`	vector of observations from the distribution whose density is to be estimated. Missing values are not allowed.
`drv`	order of derivative in the density functional. Must be a non-negative even integer.
`bandwidth`	the kernel bandwidth smoothing parameter.
`gridsize`	the number of equally-spaced points over which binning is performed.
`range.x`	vector containing the minimum and maximum values of `x` at which to compute the estimate. The default is the minimum and maximum data values, extended by the support of the kernel.
`binned`	logical flag: if `TRUE`, then `x` and `y` are taken to be grid counts rather than raw data.
`truncate`	logical flag: if `TRUE`, data with `x` values outside the range specified by `range.x` are ignored.

Details

The density functional of order `drv` is the integral of the product of the density and its `drv`th derivative. The kernel estimates of such quantities are computed using a binned implementation, and the kernel is the standard normal density.

Value

the estimated functional.

Background

Estimates of this type were proposed by Sheather and Jones (1991).

References

Sheather, S. J. and Jones, M. C. (1991). A reliable data-based bandwidth selection method for kernel density estimation. *Journal of the Royal Statistical Society, Series B*, **53**, 683–690.

Wand, M. P. and Jones, M. C. (1995). *Kernel Smoothing.* Chapman and Hall, London.

Examples

```
data(geyser, package="MASS")
x <- geyser$duration
est <- bkfe(x, drv=4, bandwidth=0.3)
```

---

dpih *Select a Histogram Bin Width*

Description

Uses direct plug-in methodology to select the bin width of a histogram.

Usage

```
dpih(x, scalest="minim", level=2, gridsize=401,
     range.x=range(x), truncate=TRUE)
```

Arguments

x	vector containing the sample on which the histogram is to be constructed.
scalest	estimate of scale. "stdev" - standard deviation is used. "iqr" - inter-quartile range divided by 1.349 is used. "minim" - minimum of "stdev" and "iqr" is used.
level	number of levels of functional estimation used in the plug-in rule.
gridsize	number of grid points used in the binned approximations to functional estimates.
range.x	range over which functional estimates are obtained. The default is the minimum and maximum data values.
truncate	if truncate is TRUE then observations outside of the interval specified by range.x are omitted. Otherwise, they are used to weight the extreme grid points.

Details

The direct plug-in approach, where unknown functionals that appear in expressions for the asymptotically optimal bin width and bandwidths are replaced by kernel estimates, is used. The normal distribution is used to provide an initial estimate.

Value

the selected bin width.

Background

This method for selecting the bin width of a histogram is described in Wand (1995). It is an extension of the normal scale rule of Scott (1979) and uses plug-in ideas from bandwidth selection for kernel density estimation (e.g. Sheather and Jones, 1991).

References

Scott, D. W. (1979). On optimal and data-based histograms. *Biometrika*, **66**, 605–610.

Sheather, S. J. and Jones, M. C. (1991). A reliable data-based bandwidth selection method for kernel density estimation. *Journal of the Royal Statistical Society, Series B*, **53**, 683–690.

Wand, M. P. (1995). Data-based choice of histogram binwidth. *University of New South Wales*, Australian Graduate School of Management Working Paper Series No. 95–011.

See Also

`hist`

Examples

```
data(geyser, package="MASS")
x <- geyser$duration
h <- dpih(x)
bins <- seq(min(x)-0.1, max(x)+0.1+h, by=h)
hist(x, breaks=bins)
```

`dpik` *Select a Bandwidth for Kernel Density Estimation*

Description

Use direct plug-in methodology to select the bandwidth of a kernel density estimate.

Usage

```
dpik(x, scalest="minim", level=2, kernel="normal",
     canonical=FALSE, gridsize=401, range.x=range(x),
     truncate=TRUE)
```

Arguments

`x`	vector containing the sample on which the kernel density estimate is to be constructed.
`scalest`	estimate of scale. `"stdev"` - standard deviation is used. `"iqr"` - inter-quartile range divided by 1.349 is used. `"minim"` - minimum of `"stdev"` and `"iqr"` is used.
`level`	number of levels of functional estimation used in the plug-in rule.
`kernel`	character string which determines the smoothing kernel. `kernel` can be: `"normal"` - the Gaussian density function (the default). `"box"` - a rectangular box. `"epanech"` - the centred beta(2,2) density. `"biweight"` - the centred beta(3,3) density. `"triweight"` - the centred beta(4,4) density.
`canonical`	logical flag: if `TRUE`, canonically scaled kernels are used
`gridsize`	the number of equally-spaced points over which binning is performed to obtain kernel functional approximation.

`range.x`	vector containing the minimum and maximum values of `x` at which to compute the estimate. The default is the minimum and maximum data values.
`truncate`	logical flag: if `TRUE`, data with `x` values outside the range specified by `range.x` are ignored.

Details

The direct plug-in approach, where unknown functionals that appear in expressions for the asymptotically optimal bandwidths are replaced by kernel estimates, is used. The normal distribution is used to provide an initial estimate.

Value

the selected bandwidth.

Background

This method for selecting the bandwidth of a kernel density estimate was proposed by Sheather and Jones (1991) and is described in Section 3.6 of Wand and Jones (1995).

References

Sheather, S. J. and Jones, M. C. (1991). A reliable data-based bandwidth selection method for kernel density estimation. *Journal of the Royal Statistical Society, Series B*, **53**, 683–690.

Wand, M. P. and Jones, M. C. (1995). *Kernel Smoothing.* Chapman and Hall, London.

See Also

`bkde`, `density`, `ksmooth`

Examples

```
data(geyser, package="MASS")
x <- geyser$duration
h <- dpik(x)
est <- bkde(x,bandwidth=h)
plot(est,type="l")
```

---

dpill *Select a Bandwidth for Local Linear Regression*

Description

Use direct plug-in methodology to select the bandwidth of a local linear Gaussian kernel regression estimate, as described by Ruppert, Sheather and Wand (1995).

Usage

```
dpill(x, y, blockmax = 5, divisor = 20, trim = 0.01, proptrun = 0.05,
      gridsize = 401, range.x, truncate = TRUE)
```

Arguments

x	vector of x data. Missing values are not accepted.
y	vector of y data. This must be same length as x, and missing values are not accepted.
blockmax	the maximum number of blocks of the data for construction of an initial parametric estimate.
divisor	the value that the sample size is divided by to determine a lower limit on the number of blocks of the data for construction of an initial parametric estimate.
trim	the proportion of the sample trimmed from each end in the x direction before application of the plug-in methodology.
proptrun	the proportion of the range of x at each end truncated in the functional estimates.
gridsize	number of equally-spaced grid points over which the function is to be estimated.
range.x	vector containing the minimum and maximum values of x at which to compute the estimate. For density estimation the default is the minimum and maximum data values with 5% of the range added to each end. For regression estimation the default is the minimum and maximum data values.
truncate	logical flag: if TRUE, data with x values outside the range specified by range.x are ignored.

Details

The direct plug-in approach, where unknown functionals that appear in expressions for the asymptotically optimal bandwidths are replaced by kernel estimates, is used. The kernel is the standard normal density. Least squares quartic fits over blocks of data are used to obtain an initial estimate. Mallow's C_p is used to select the number of blocks.

Value

the selected bandwidth.

Warning

If there are severe irregularities (i.e. outliers, sparse regions) in the x values then the local polynomial smooths required for the bandwidth selection algorithm may become degenerate and the function will crash. Outliers in the y direction may lead to deterioration of the quality of the selected bandwidth.

References

Ruppert, D., Sheather, S. J. and Wand, M. P. (1995). An effective bandwidth selector for local least squares regression. *Journal of the American Statistical Association*, **90**, 1257–1270.

Wand, M. P. and Jones, M. C. (1995). *Kernel Smoothing.* Chapman and Hall, London.

See Also

ksmooth, locpoly.

Examples

```
data(geyser, package = "MASS")
x <- geyser$duration
y <- geyser$waiting
plot(x, y)
h <- dpill(x, y)
fit <- locpoly(x, y, bandwidth = h)
lines(fit)
```

---

locpoly *Estimate Functions Using Local Polynomials*

Description

Estimates a probability density function, regression function or their derivatives using local polynomials. A fast binned implementation over an equally-spaced grid is used.

Usage

```
locpoly(x, y, drv = 0, degree =, kernel = "normal",
        bandwidth, gridsize = 401, bwdisc = 25,
        range.x, binned = FALSE, truncate = TRUE)
```

Arguments

`x`	vector of x data. Missing values are not accepted.
`bandwidth`	the kernel bandwidth smoothing parameter. It may be a single number or an array having length `gridsize`, representing a bandwidth that varies according to the location of estimation.
`y`	vector of y data. This must be same length as `x`, and missing values are not accepted.
`drv`	order of derivative to be estimated.
`degree`	degree of local polynomial used. Its value must be greater than or equal to the value of `drv`. The default value is of `degree` is `drv` + 1.
`kernel`	`"normal"` - the Gaussian density function.
`gridsize`	number of equally-spaced grid points over which the function is to be estimated.
`bwdisc`	number of logarithmically-equally-spaced bandwidths on which `bandwidth` is discretised, to speed up computation.
`range.x`	vector containing the minimum and maximum values of `x` at which to compute the estimate.
`binned`	logical flag: if `TRUE`, then `x` and `y` are taken to be grid counts rather than raw data.
`truncate`	logical flag: if `TRUE`, data with `x` values outside the range specified by `range.x` are ignored.

Value

if `y` is specified, a local polynomial regression estimate of E[Y|X] (or its derivative) is computed. If `y` is missing, a local polynomial estimate of the density of `x` (or its derivative) is computed.

a list containing the following components:

`x`	vector of sorted x values at which the estimate was computed.
`y`	vector of smoothed estimates for either the density or the regression at the corresponding `x`.

Details

Local polynomial fitting with a kernel weight is used to estimate either a density, regression function or their derivatives. In the case of density estimation, the data are binned and the local fitting procedure is applied to the bin counts. In either case, binned approximations over an equally-spaced grid is used for fast computation. The bandwidth may be either scalar or a vector of length `gridsize`.

References

Wand, M. P. and Jones, M. C. (1995). *Kernel Smoothing.* Chapman and Hall, London.

See Also

`bkde`, `density`, `dpill`, `ksmooth`, `loess`, `smooth`, `supsmu`.

Examples

```
data(geyser, package = "MASS")
x <- geyser$duration
est <- locpoly(x, bandwidth = 0.25)
plot(est, type = "l")
# local linear density estimate
y <- geyser$waiting
plot(x, y)
fit <- locpoly(x, y, bandwidth = 0.25)
lines(fit)
# local linear regression estimate
```

Chapter 11

The MASS package

`abbey` *Determinations of Nickel Content*

Description

A numeric vector of 31 determinations of nickel content (ppm) in a Canadian syenite rock.

Usage

```
abbey
```

Source

S. Abbey (1988) *Geostandards Newsletter* **12**, 241.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

`accdeaths` *Accidental Deaths in the US 1973-1978*

Description

A regular time series giving the monthly totals of accidental deaths in the USA. The values for first six months of 1979 (p. 326) are `7798 7406 8363 8460 9217 9316`

Usage

```
accdeaths
```

Source

P. J. Brockwell and R. A. Davis (1991) *Time Series: Theory and Methods.* Springer, New York.

References

Venables, W. N. and Ripley, B. D. (1999) *Modern Applied Statistics with S-PLUS.* Third Edition. Springer.

`addterm` *Try All One-Term Additions to a Model*

Description

Try fitting all models that differ from the current model by adding a single term from those supplied, maintaining marginality.

This function is generic; there exist methods for classes `lm` and `glm` and the default method will work for many other classes.

Usage

```
addterm(object, ...)

## Default S3 method:
addterm(object, scope, scale = 0, test = c("none", "Chisq"),
        k = 2, sorted = FALSE, trace = FALSE, ...)
## S3 method for class 'lm':
addterm(object, scope, scale = 0, test = c("none", "Chisq", "F"),
        k = 2, sorted = FALSE, ...)
## S3 method for class 'glm':
addterm(object, scope, scale = 0, test = c("none", "Chisq", "F"),
        k = 2, sorted = FALSE, trace = FALSE, ...)
```

Arguments

`object`	An object fitted by some model-fitting function.
`scope`	a formula specifying a maximal model which should include the current one. All additional terms in the maximal model with all marginal terms in the original model are tried.
`scale`	used in the definition of the AIC statistic for selecting the models, currently only for `lm`, `aov` and `glm` models. Specifying `scale` asserts that the residual standard error or dispersion is known.
`test`	should the results include a test statistic relative to the original model? The F test is only appropriate for `lm` and `aov` models, and perhaps for some over-dispersed `glm` models. The Chisq test can be an exact test (`lm` models with known scale) or a likelihood-ratio test depending on the method.
`k`	the multiple of the number of degrees of freedom used for the penalty. Only `k=2` gives the genuine AIC: `k = log(n)` is sometimes referred to as BIC or SBC.
`sorted`	should the results be sorted on the value of AIC?
`trace`	if `TRUE` additional information may be given on the fits as they are tried.
`...`	arguments passed to or from other methods.

Details

The definition of AIC is only up to an additive constant: when appropriate (`lm` models with specified scale) the constant is taken to be that used in Mallows' Cp statistic and the results are labelled accordingly.

Value

A table of class `"anova"` containing at least columns for the change in degrees of freedom and AIC (or Cp) for the models. Some methods will give further information, for example sums of squares, deviances, log-likelihoods and test statistics.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`dropterm`, `stepAIC`

Examples

```
quine.hi <- aov(log(Days + 2.5) ~ .^4, quine)
quine.lo <- aov(log(Days+2.5) ~ 1, quine)
addterm(quine.lo, quine.hi, test="F")

house.glm0 <- glm(Freq ~ Infl*Type*Cont + Sat, family=poisson,
                  data=housing)
addterm(house.glm0, ~. + Sat:(Infl+Type+Cont), test="Chisq")
house.glm1 <- update(house.glm0, . ~ . + Sat*(Infl+Type+Cont))
addterm(house.glm1, ~. + Sat:(Infl+Type+Cont)^2, test = "Chisq")
```

`Aids2` *Australian AIDS Survival Data*

Description

Data on patients diagnosed with AIDS in Australia before 1 July 1991.

Usage

```
Aids2
```

Format

This data frame contains 2843 rows and the following columns:

`state` Grouped state of origin: NSW includes ACT and `"other"` is WA, SA, NT and TAS.

`sex` Sex of patient

`diag` (Julian) date of diagnosis

`death` (Julian) date of death or end of observation

`status` `"A"` (alive) or `"D"` (dead) at end of observation

`T.categ` Reported transmission category

`age` Age (years) at diagnosis

Note

This data set has been slightly jittered as a condition of its release, to ensure patient confidentiality.

Source

Dr P. J. Solomon and the Australian National Centre in HIV Epidemiology and Clinical Research.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

`Animals` *Brain and Body Weights for 28 Species*

Description

Average brain and body weights for 28 species of land animals.

Usage

```
Animals
```

Format

`body` body weight in kg

`brain` brain weight in g

Note

The name `Animals` avoids conflicts with a system dataset `animals` in S-PLUS 4.5 and later.

Source

P. J. Rousseeuw and A. M. Leroy (1987) *Robust Regression and Outlier Detection.* Wiley, p. 57.

References

Venables, W. N. and Ripley, B. D. (1999) *Modern Applied Statistics with S-PLUS.* Third Edition. Springer.

anorexia *Anorexia Data on Weight Change*

Description

The anorexia data frame has 72 rows and 3 columns. Weight change data for young female anorexia patients.

Usage

```
anorexia
```

Format

This data frame contains the following columns:

Treat Factor of three levels: Cont (Control), CBT (Cognitive Behavioural Treatment) and FT (Family treatment).

Prewt Weight of patient before study period, in lbs.

Postwt Weight of patient after study period, in lbs.

Source

Hand, D. J., Daly, F., McConway, K., Lunn, D. and Ostrowski, E. eds (1993) *A Handbook of Small Data Sets.* Chapman & Hall, Data set 285 (p. 229)

(Note that the original source mistakenly says that weights are in kg.)

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

anova.negbin *Likelihood Ratio Tests for Negative Binomial GLMs*

Description

Method function to perform sequential likelihood ratio tests for Negative Binomial generalized linear models.

Usage

```
## S3 method for class 'negbin':
anova(object, ..., test = "Chisq")
```

Arguments

object	Fitted model object of class "negbin", inheriting from classes "glm" and "lm", specifying a Negative Binomial fitted GLM. Typically the output of glm.nb().
...	Zero or more additional fitted model objects of class "negbin". They should form a nested sequence of models, but need not be specified in any particular order.
test	Argument to match the test argument of anova.glm. Ignored (with a warning if changed) if a sequence of two or more Negative Binomial fitted model objects is specified, but possibly used if only one object is specified.

Details

This function is a method for the generic function anova() for class "negbin". It can be invoked by calling anova(x) for an object x of the appropriate class, or directly by calling anova.negbin(x) regardless of the class of the object.

Note

If only one fitted model object is specified, a sequential analysis of deviance table is given for the fitted model. The theta parameter is kept fixed. If more than one fitted model object is specified they must all be of class "negbin" and likelihood ratio tests are done of each model within the next. In this case theta is assumed to have been re-estimated for each model.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

glm.nb, negative.binomial, summary.negbin

Examples

```
m1 <- glm.nb(Days ~ Eth*Age*Lrn*Sex, quine, link = log)
m2 <- update(m1, . ~ . - Eth:Age:Lrn:Sex)
anova(m2, m1)
anova(m2)
```

---

area *Adaptive Numerical Integration*

Description

Integrate a function of one variable over a finite range using a recursive adaptive method. This function is mainly for demonstration purposes.

Usage

```
area(f, a, b, ..., fa = f(a, ...), fb = f(b, ...),
     limit = 10, eps = 1e-05)
```

Arguments

f	The integrand as an S function object. The variable of integration must be the first argument.
a	Lower limit of integration.
b	Upper limit of integration.
...	Additional arguments needed by the integrand.
fa	Function value at the lower limit.
fb	Function value at the upper limit.
limit	Limit on the depth to which recursion is allowed to go.
eps	Error tolerance to control the process.

Details

The method divides the interval in two and compares the values given by Simpson's rule and the trapezium rule. If these are within eps of each other the Simpson's rule result is given, otherwise the process is applied separately to each half of the interval and the results added together.

Value

The integral from a to b of f(x).

References

Venables, W. N. and Ripley, B. D. (1994) *Modern Applied Statistics with S-Plus.* Springer. pp. 105–110.

Examples

```
area(sin, 0, pi)  # integrate the sin function from 0 to pi.
```

---

bacteria *Presence of Bacteria after Drug Treatments*

Description

Tests of the presence of the bacteria *H. influenzae* in children with otitis media in the Northern Territory of Australia.

Usage

```
bacteria
```

Format

This data frame has 220 rows and the following columns:

y presence or absence: a factor with levels `n` and `y`.

ap active/placebo: a factor with levels `a` and `p`.

hilo hi/low compliance: a factor with levels `hi` amd `lo`.

week numeric: week of test.

ID subject ID: a factor.

trt a factor with levels `placebo`, `drug` and `drug+`, a re-coding of `ap` and `hilo`.

Details

Dr A. Leach tested the effects of a drug on 50 children with a history of otitis media in the Northern Territory of Australia. The children were randomized to the drug or the a placebo, and also to receive active encouragement to comply with taking the drug.

The presence of *H. influenzae* was checked at weeks 0, 2, 4, 6 and 11: 30 of the checks were missing and are not included in this data frame.

Source

Menzies School of Health Research 1999–2000 Annual Report pp. 18–21 (http://www.menzies.edu.au/publications/anreps/MSHR00.pdf).

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*. Fourth edition. Springer.

Examples

```
contrasts(bacteria$trt) <- structure(contr.sdif(3),
     dimnames = list(NULL, c("drug", "encourage")))
## fixed effects analyses
summary(glm(y ~ trt * week, binomial, data = bacteria))
summary(glm(y ~ trt + week, binomial, data = bacteria))
summary(glm(y ~ trt + I(week > 2), binomial, data = bacteria))

# conditional random-effects analysis
library(survival)
bacteria$Time <- rep(1, nrow(bacteria))
coxph(Surv(Time, unclass(y)) ~ week + strata(ID),
      data = bacteria, method = "exact")
coxph(Surv(Time, unclass(y)) ~ factor(week) + strata(ID),
      data = bacteria, method = "exact")
coxph(Surv(Time, unclass(y)) ~ I(week > 2) + strata(ID),
      data = bacteria, method = "exact")

# PQL glmm analysis
library(nlme)
summary(glmmPQL(y ~ trt + I(week > 2), random = ~ 1 | ID,
                family = binomial, data = bacteria))
```

bandwidth.nrd *Bandwidth for density() via Normal Reference Distribution*

Description

A well-supported rule-of-thumb for choosing the bandwidth of a Gaussian kernel density estimator.

Usage

```
bandwidth.nrd(x)
```

Arguments

x	A data vector.

Value

A bandwidth on a scale suitable for the `width` argument of `density`.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Springer, equation (5.5) on page 130.

Examples

```
# The function is currently defined as
function(x)
{
        r <- quantile(x, c(0.25, 0.75))
        h <- (r[2] - r[1])/1.34
        4 * 1.06 * min(sqrt(var(x)), h) * length(x)^(-1/5)
}
```

bcv *Biased Cross-Validation for Bandwidth Selection*

Description

Uses biased cross-validation to select the bandwidth of a Gaussian kernel density estimator.

Usage

```
bcv(x, nb = 1000, lower, upper)
```

Arguments

x	a numeric vector
nb	number of bins to use.
lower, upper	Range over which to minimize. The default is almost always satisfactory.

Value

a bandwidth

References

Scott, D. W. (1992) *Multivariate Density Estimation: Theory, Practice, and Visualization.* Wiley.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

ucv, width.SJ, density

Examples

```
bcv(geyser$duration)
```

---

beav1 *Body Temperature Series of Beaver 1*

Description

Reynolds (1994) describes a small part of a study of the long-term temperature dynamics of beaver *Castor canadensis* in north-central Wisconsin. Body temperature was measured by telemetry every 10 minutes for four females, but data from a one period of less than a day for each of two animals is used there.

Usage

```
beav1
```

Format

The beav1 data frame has 114 rows and 4 columns. This data frame contains the following columns:

day Day of observation (in days since the beginning of 1990), December 12-13.

time Time of observation, in the form 0330 for 3.30am

temp Measured body temperature in degrees Celcius

activ Indicator of activity outside the retreat

Note

The observation at 22:20 is missing.

Source

P. S. Reynolds (1994) Time-series analyses of beaver body temperatures. Chapter 11 of Lange, N., Ryan, L., Billard, L., Brillinger, D., Conquest, L. and Greenhouse, J. eds (1994) *Case Studies in Biometry.* New York: John Wiley and Sons.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

beav2

Examples

```
attach(beav1)
beav1$hours <- 24*(day-346) + trunc(time/100) + (time%%100)/60
plot(beav1$hours, beav1$temp, type="l", xlab="time",
   ylab="temperature", main="Beaver 1")
usr <- par("usr"); usr[3:4] <- c(-0.2, 8); par(usr=usr)
lines(beav1$hours, beav1$activ, type="s", lty=2)
temp <- ts(c(beav1$temp[1:82], NA, beav1$temp[83:114]),
           start = 9.5, frequency = 6)
activ <- ts(c(beav1$activ[1:82], NA, beav1$activ[83:114]),
            start = 9.5, frequency = 6)

acf(temp[1:53])
acf(temp[1:53], type = "partial")
ar(temp[1:53])
act <- c(rep(0, 10), activ)
X <- cbind(1, act = act[11:125], act1 = act[10:124],
          act2 = act[9:123], act3 = act[8:122])
alpha <- 0.80
stemp <- as.vector(temp - alpha*lag(temp, -1))
sX <- X[-1, ] - alpha * X[-115,]
beav1.ls <- lm(stemp ~ -1 + sX, na.action = na.omit)
summary(beav1.ls, cor = FALSE)
detach("beav1"); rm(temp, activ)
```

---

beav2 *Body Temperature Series of Beaver 2*

Description

Reynolds (1994) describes a small part of a study of the long-term temperature dynamics of beaver *Castor canadensis* in north-central Wisconsin. Body temperature was measured by telemetry every 10 minutes for four females, but data from a one period of less than a day for each of two animals is used there.

Usage

```
beav2
```

Format

The beav2 data frame has 100 rows and 4 columns. This data frame contains the following columns:

day Day of observation (in days since the beginning of 1990), November 3-4.

time Time of observation, in the form 0330 for 3.30am

temp Measured body temperature in degrees Celcius

activ Indicator of activity outside the retreat

Source

P. S. Reynolds (1994) Time-series analyses of beaver body temperatures. Chapter 11 of Lange, N., Ryan, L., Billard, L., Brillinger, D., Conquest, L. and Greenhouse, J. eds (1994) *Case Studies in Biometry.* New York: John Wiley and Sons.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

beav1

Examples

```
attach(beav2)
beav2$hours <- 24*(day-307) + trunc(time/100) + (time%%100)/60
plot(beav2$hours, beav2$temp, type = "l", xlab = "time",
   ylab = "temperature", main = "Beaver 2")
usr <- par("usr"); usr[3:4] <- c(-0.2, 8); par(usr = usr)
lines(beav2$hours, beav2$activ, type = "s", lty = 2)

temp <- ts(temp, start = 8+2/3, frequency = 6)
activ <- ts(activ, start = 8+2/3, frequency = 6)
acf(temp[activ == 0]); acf(temp[activ == 1]) # also look at PACFs
ar(temp[activ == 0]); ar(temp[activ == 1])

arima(temp, order = c(1,0,0), xreg = activ)
dreg <- cbind(sin = sin(2*pi*beav2$hours/24), cos = cos(2*pi*beav2$hours/24))
arima(temp, order = c(1,0,0), xreg = cbind(active=activ, dreg))

library(nlme)
beav2.gls <- gls(temp ~ activ, data = beav2, corr = corAR1(0.8),
                 method = "ML")
summary(beav2.gls)
summary(update(beav2.gls, subset = 6:100))
detach("beav2"); rm(temp, activ)
```

Belgian-phones *Belgium Phone Calls 1950-1973*

Description

A list object with the annual numbers of telephone calls, in Belgium. The components are:

year The last two digits of the year.

calls The number of telephone calls made (in millions of calls).

Usage

```
data(phones)
```

Source

P. J. Rousseeuw and A. M. Leroy (1987) *Robust Regression & Outlier Detection.* Wiley.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

---

biopsy *Biopsy Data on Breast Cancer Patients*

Description

This breast cancer database was obtained from the University of Wisconsin Hospitals, Madison from Dr. William H. Wolberg. He assessed biopsies of breast tumours for 699 patients up to 15 July 1992; each of nine attributes has been scored on a scale of 1 to 10, and the outcome is also known. There are 699 rows and 11 columns.

Usage

```
biopsy
```

Format

This data frame contains the following columns:

`ID` Sample code number (not unique)

`V1` Clump thickness

`V2` Uniformity of cell size

`V3` Uniformity of cell shape

`V4` Marginal adhesion

`V5` Single epithelial cell size

`V6` Bare nuclei (16 values are missing)

`V7` Bland chromatin

`V8` Normal nucleoli

`V9` Mitoses

`class` `"benign"` or `"malignant"`

Source

P. M. Murphy and D. W. Aha (1992). UCI Repository of machine learning databases. [Machine-readable data repository]. Irvine, CA: University of California, Department of Information and Computer Science.

O. L. Mangasarian and W. H. Wolberg (1990) Cancer diagnosis via linear programming. *SIAM News* **23**, pp 1 & 18.

William H. Wolberg and O.L. Mangasarian (1990) Multisurface method of pattern separation for medical diagnosis applied to breast cytology. *Proceedings of the National Academy of Sciences, U.S.A.* **87**, pp. 9193–9196.

O. L. Mangasarian, R. Setiono and W.H. Wolberg (1990) Pattern recognition via linear programming: Theory and application to medical diagnosis. In *Large-scale Numerical Optimization* eds Thomas F. Coleman and Yuying Li, SIAM Publications, Philadelphia, pp 22–30.

K. P. Bennett and O. L. Mangasarian (1992) Robust linear programming discrimination of two linearly inseparable sets. *Optimization Methods and Software* **1**, pp. 23–34 (Gordon & Breach Science Publishers).

References

Venables, W. N. and Ripley, B. D. (1999) *Modern Applied Statistics with S-PLUS.* Third Edition. Springer.

---

birthwt *Risk Factors Associated with Low Infant Birth Weight*

Description

The `birthwt` data frame has 189 rows and 10 columns. The data were collected at Baystate Medical Center, Springfield, Mass during 1986.

Usage

```
birthwt
```

Format

This data frame contains the following columns:

`low` indicator of birth weight less than 2.5kg

`age` mother's age in years

`lwt` mother's weight in pounds at last menstrual period

`race` mother's race (`1` = white, `2` = black, `3` = other)

`smoke` smoking status during pregnancy

`ptl` number of previous premature labours

`ht` history of hypertension

`ui` presence of uterine irritability

`ftv` number of physician visits during the first trimester

`bwt` birth weight in grams

Source

Hosmer, D.W. and Lemeshow, S. (1989) *Applied Logistic Regression.* New York: Wiley

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

Examples

```
attach(birthwt)
race <- factor(race, labels = c("white", "black", "other"))
ptd <- factor(ptl > 0)
ftv <- factor(ftv)
levels(ftv)[-(1:2)] <- "2+"
bwt <- data.frame(low = factor(low), age, lwt, race,
    smoke = (smoke > 0), ptd, ht = (ht > 0), ui = (ui > 0), ftv)
detach("birthwt")
options(contrasts = c("contr.treatment", "contr.poly"))
glm(low ~ ., binomial, bwt)
```

Boston — *Housing Values in Suburbs of Boston*

Description

The `Boston` data frame has 506 rows and 14 columns.

Usage

```
Boston
```

Format

This data frame contains the following columns:

`crim` per capita crime rate by town

`zn` proportion of residential land zoned for lots over 25,000 sq.ft.

`indus` proportion of non-retail business acres per town

`chas` Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)

`nox` nitrogen oxides concentration (parts per 10 million)

`rm` average number of rooms per dwelling

`age` proportion of owner-occupied units built prior to 1940

`dis` weighted mean of distances to five Boston employment centres

`rad` index of accessibility to radial highways

`tax` full-value property-tax rate per $10,000

`ptratio` pupil-teacher ratio by town

`black` $1000(Bk - 0.63)^2$ where Bk is the proportion of blacks by town

`lstat` lower status of the population (percent)

`medv` median value of owner-occupied homes in $1000

Source

Harrison, D. and Rubinfeld, D.L. (1978) Hedonic prices and the demand for clean air. *J. Environ. Economics and Management* **5**, 81–102.

Belsley D.A., Kuh, E. and Welsch, R.E. (1980) *Regression Diagnostics. Identifying Influential Data and Sources of Collinearity.* New York: Wiley.

---

boxcox *Box-Cox Transformations for Linear Models*

Description

Computes and optionally plots profile log-likelihoods for the parameter of the Box-Cox power transformation.

Usage

```
boxcox(object, ...)

## Default S3 method:
boxcox(object, lambda = seq(-2, 2, 1/10), plotit = TRUE,
       interp, eps = 1/50, xlab = expression(lambda),
       ylab = "log-Likelihood", ...)

## S3 method for class 'formula':
boxcox(object, lambda = seq(-2, 2, 1/10), plotit = TRUE,
       interp, eps = 1/50, xlab = expression(lambda),
       ylab = "log-Likelihood", ...)

## S3 method for class 'lm':
boxcox(object, lambda = seq(-2, 2, 1/10), plotit = TRUE,
       interp, eps = 1/50, xlab = expression(lambda),
       ylab = "log-Likelihood", ...)
```

Arguments

`object`	a formula or fitted model object. Currently only `lm` and `aov` objects are handled.
`lambda`	vector of values of `lambda` – default $(-2, 2)$ in steps of 0.1.
`plotit`	logical which controls whether the result should be plotted.
`interp`	logical which controls whether spline interpolation is used. Default to `TRUE` if plotting with `lambda` of length less than 100.
`eps`	Tolerance for `lambda = 0`; defaults to 0.02.
`xlab`	defaults to `"lambda"`.
`ylab`	defaults to `"log-Likelihood"`.
`...`	additional parameters to be used in the model fitting.

Value

A list of the `lambda` vector and the computed profile log-likelihood vector, invisibly if the result is plotted.

Side Effects

If `plotit = TRUE` plots loglik *vs* `lambda` and indicates a 95% confidence interval about the maximum observed value of `lambda`. If `interp = TRUE`, spline interpolation is used to give a smoother plot.

References

Box, G. E. P. and Cox, D. R. (1964) An analysis of transformations (with discussion). *Journal of the Royal Statistical Society B*, **26**, 211–252.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

Examples

```
data(trees)
boxcox(Volume ~ log(Height) + log(Girth), data = trees,
       lambda = seq(-0.25, 0.25, length = 10))

boxcox(Days+1 ~ Eth*Sex*Age*Lrn, data = quine,
       lambda = seq(-0.05, 0.45, len = 20))
```

`cabbages` *Data from a cabbage field trial*

Description

The `cabbages` data set has 60 observations and 4 variables

Usage

```
cabbages
```

Format

This data frame contains the following columns:

`Cult` Factor giving the cultivar of the cabbage, two levels: `c39` and `c52`.

`Date` Factor specifying one of three planting dates: `d16`, `d20` or `d21`.

`HeadWt` Weight of the cabbage head, presumably in kg.

`VitC` Ascorbic acid content, in undefined units.

Source

Rawlings, J. O. (1988) *Applied Regression Analysis: A Research Tool.* Wadsworth and Brooks/Cole. Example 8.4, page 219. (Rawlings cites the original source as the files of the late Dr Gertrude M Cox.)

References

Venables, W. N. and Ripley, B. D. (1999) *Modern Applied Statistics with S-PLUS.* Third Edition. Springer.

caith — *Colours of Eyes and Hair of People in Caithness*

Description

Data on the cross-classification of people in Caithness, Scotland, by eye and hair colour. The region of the UK is particularly interesting as there is a mixture of people of Nordic, Celtic and Anglo-Saxon origin.

Usage

```
caith
```

Format

A 4 by 5 table with rows the eye colours (blue, light, medium, dark) and columns the hair colours (fair, red, medium, dark, black).

Source

Fisher, R.A. (1940) The precision of discriminant functions. *Annals of Eugenics (London)* **10**, 422–429.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

Examples

```
corresp(caith)
dimnames(caith)[[2]] <- c("F", "R", "M", "D", "B")
par(mfcol=c(1,3))
plot(corresp(caith, nf=2)); title("symmetric")
plot(corresp(caith, nf=2), type="rows"); title("rows")
plot(corresp(caith, nf=2), type="col"); title("columns")
par(mfrow=c(1,1))
```

Cars93 — *Data from 93 Cars on Sale in the USA in 1993*

Description

The `Cars93` data frame has 93 rows and 27 columns.

Usage

```
Cars93
```

Format

This data frame contains the following columns:

`Manufacturer` Manufacturer
`Model` Model
`Type` Type: Small, Sporty, Compact, Midsize, Large, Van
`Min.Price` Minimum Price (in $1,000) - price for a basic version
`Price` Midrange Price (in $1,000) - average of `Min.Price` and `Max.Price`
`Max.Price` Maximum Price (in $1,000) - price for "a premium version"
`MPG.city` City MPG (miles per US gallon by EPA rating)
`MPG.highway` Highway MPG
`AirBags` Air Bags standard. Factor: none, driver only, or driver & passenger
`DriveTrain` Drive train type: rear wheel, front wheel or 4WD; (factor).
`Cylinders` Number of cylinders (missing for Mazda RX-7, which has a rotary engine).
`EngineSize` Engine size (litres)
`Horsepower` Horsepower (maximum)
`RPM` RPM (revs per minute at maximum horsepower)
`Rev.per.mile` Engine revolutions per mile (in highest gear)
`Man.trans.avail` Is a manual transmission version available? (yes or no, Factor).
`Fuel.tank.capacity` Fuel tank capacity (US gallons)
`Passengers` Passenger capacity (persons)
`Length` Length (inches)
`Wheelbase` Wheelbase (inches)
`Width` Width (inches)
`Turn.circle` U-turn space (feet)
`Rear.seat.room` Rear seat room (inches) (missing for 2-seater vehicles)
`Luggage.room` Luggage capacity (cubic feet) (missing for vans)
`Weight` Weight (pounds)
`Origin` Of non-USA or USA company origins? (factor)
`Make` Combination of Manufacturer and Model (character)

Details

Cars were selected at random from among 1993 passenger car models that were listed in both the *Consumer Reports* issue and the *PACE Buying Guide*. Pickup trucks and Sport/Utility vehicles were eliminated due to incomplete information in the *Consumer Reports* source. Duplicate models (e.g., Dodge Shadow and Plymouth Sundance) were listed at most once.

Further description can be found in Lock (1993). Use the URL `http://www.amstat.org/publications/jse/v1n1/datasets.lock.html`

Source

Lock, R. H. (1993) 1993 New Car Data. *Journal of Statistics Education* **1**(1)

References

Venables, W. N. and Ripley, B. D. (1999) *Modern Applied Statistics with S-PLUS*. Third Edition. Springer.

cats *Anatomical Data from Domestic Cats*

Description

The heart and body weights of samples of male and female cats used for digitalis experiments. The cats were all adult, over 2 kg body weight.

Usage

```
cats
```

Format

This data frame contains the following columns:

Sex Sex factor. Levels "F" and "M".

Bwt Body weight in kg.

Hwt Heart weight in g.

Source

R. A. Fisher (1947) The analysis of covariance method for the relation between a part and the whole, *Biometrics* **3**, 65–68.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

cement *Heat Evolved by Setting Cements*

Description

Experiment on the heat evolved in the setting of each of 13 cements.

Usage

```
cement
```

Format

x1, x2, x3, x4 Proportions (%) of active ingredients

y heat evolved in cals/gm

Details

13 samples of Portland cement were set. For each sample, the percentages of the four main chemical ingredients was accurately measured. While the cement was setting the amount of heat evolved was also measured.

Source

Woods, H., Steinour, H.H. and Starke, H.R. (1932) Effect of composition of Portland cement on heat evolved during hardening. *Industrial Engineering and Chemistry*, **24**, 1207–1214.

References

Hald, A. (1957) *Statistical Theory with Engineering Applications.* Wiley, New York.

Examples

```
lm(y ~ x1 + x2 + x3 + x4, cement)
```

---

chem *Copper in Wholemeal Flour*

Description

A numeric vector of 24 determinations of copper in wholemeal flour, in parts per million.

Usage

```
chem
```

Source

Analytical Methods Committee (1989) Robust statistics – how not to reject outliers. *The Analyst* **114**, 1693–1702, 1989

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

---

con2tr *Convert Lists to Data Frames for use by Trellis*

Description

Convert lists to data frames for use by Trellis.

Usage

```
con2tr(obj)
```

Arguments

`obj`	A list of components `x`, `y` and `z` as passed to `contour`

Details

`con2tr` repeats the `x` and `y` components suitably to match the vector `z`.

Value

A data frame suitable for passing to Trellis functions.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

---

confint-MASS *Confidence Intervals for Model Parameters*

Description

Computes confidence intervals for one or more parameters in a fitted model. Package **MASS** adds methods for `glm` and `nls` fits.

Usage

```
## S3 method for class 'glm':
confint(object, parm, level = 0.95, trace = FALSE, ...)

## S3 method for class 'nls':
confint(object, parm, level = 0.95, ...)
```

Arguments

`object`	a fitted model object. Methods currently exist for the classes `"glm"`, `"nls"` and for profile objects from these classes.
`parm`	a specification of which parameters are to be given confidence intervals, either a vector of numbers or a vector of names. If missing, all parameters are considered.
`level`	the confidence level required.
`trace`	logical. Should profiling be traced?
`...`	additional argument(s) for methods.

Details

`confint` is a generic function in package `base`.

These `confint` methods calls the appropriate profile method, then finds the confidence intervals by interpolation in the profile traces. If the profile object is already available it should be used as the main argument rather than the fitted model object itself.

Value

A matrix (or vector) with columns giving lower and upper confidence limits for each parameter. These will be labelled as (1-level)/2 and 1 - (1-level)/2 in % (by default 2.5% and 97.5%).

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

profile

Examples

```
expn1 <- deriv(y ~ b0 + b1 * 2^(-x/th), c("b0", "b1", "th"),
               function(b0, b1, th, x) {})

wtloss.gr <- nls(Weight ~ expn1(b0, b1, th, Days),
   data = wtloss, start = c(b0=90, b1=95, th=120))

expn2 <- deriv(~b0 + b1*((w0 - b0)/b1)^(x/d0),
         c("b0","b1","d0"), function(b0, b1, d0, x, w0) {})

wtloss.init <- function(obj, w0) {
  p <- coef(obj)
  d0 <-  - log((w0 - p["b0"])/p["b1"])/log(2) * p["th"]
  c(p[c("b0", "b1")], d0 = as.vector(d0))
}

out <- NULL
w0s <- c(110, 100, 90)
for(w0 in w0s) {
    fm <- nls(Weight ~ expn2(b0, b1, d0, Days, w0),
              wtloss, start = wtloss.init(wtloss.gr, w0))
    out <- rbind(out, c(coef(fm)["d0"], confint(fm, "d0")))
  }
dimnames(out) <- list(paste(w0s, "kg:"),  c("d0", "low", "high"))
out

ldose <- rep(0:5, 2)
numdead <- c(1, 4, 9, 13, 18, 20, 0, 2, 6, 10, 12, 16)
sex <- factor(rep(c("M", "F"), c(6, 6)))
SF <- cbind(numdead, numalive = 20 - numdead)
budworm.lg0 <- glm(SF ~ sex + ldose - 1, family = binomial)
confint(budworm.lg0)
confint(budworm.lg0, "ldose")
```

---

contr.sdif *Successive Differences contrast coding*

Description

A coding for unordered factors based on successive differences.

Usage

```
contr.sdif(n, contrasts = TRUE)
```

Arguments

n	The number of levels required.
contrasts	Should there be n - 1 columns orthogonal to the mean (the default) or n columns spanning the space.

Details

The contrast coefficients are chosen so that the coded coefficients in a one-way layout are the differences between the means of the second and first levels, the third and second levels, and so on.

Value

If `contrasts is TRUE`, a matrix with `n` rows and `n - 1` columns, and the `n` by `n` identity matrix if `contrasts` is `FALSE`.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth Edition, Springer.

See Also

`contr.treatment`, `contr.sum`, `contr.helmert`

Examples

```
contr.sdif(6)
```

---

coop *Co-operative Trial in Analytical Chemistry*

Description

Seven specimens were sent to 6 laboratories in 3 separate batches and each analysed for Analyte. Each analysis was duplicated.

Usage

```
coop
```

Format

This data frame contains the following columns:

`Lab` Laboratory, `L1`, `L2`, ..., `L6`.

`Spc` Specimen, `S1`, `S2`, ..., `S7`.

`Bat` Batch, `B1`, `B2`, `B3` (nested within `Spc/Lab`),

`Conc` Concentration of Analyte in g/kg.

Source

Analytical Methods Committee (1987) Recommendations for the conduct and interpretation of co-operative trials, *The Analyst* **112**, 679–686.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`chem`, `abbey`.

corresp *Simple Correspondence Analysis*

Description

Find the principal canonical correlation and corresponding row- and column-scores from a correspondence analysis of a two-way contingency table.

Usage

```
corresp(x, ...)

## S3 method for class 'matrix':
corresp(x, nf = 1, ...)

## S3 method for class 'factor':
corresp(x, y, ...)

## S3 method for class 'data.frame':
corresp(x, ...)

## S3 method for class 'xtabs':
corresp(x, ...)

## S3 method for class 'formula':
corresp(formula, data, ...)
```

Arguments

`x, formula`	The function is generic, accepting various forms of the principal argument for specifying a two-way frequency table. Currently accepted forms are matrices, data frames (coerced to frequency tables), objects of class "`xtabs`" and formulae of the form `~ F1 + F2`, where `F1` and `F2` are factors.
`nf`	The number of factors to be computed. Note that although 1 is the most usual, one school of thought takes the first two singular vectors for a sort of biplot.
`y`	a second factor for a cross-classification
`data`	a data frame against which to preferentially resolve variables in the formula.
`...`	If the principal argument is a formula, a data frame may be specified as well from which variables in the formula are preferentially satisfied.

Details

See Venables & Ripley (2002). The `plot` method produces a graphical representation of the table if `nf=1`, with the *areas* of circles representing the numbers of points. If `nf` is two or more the `biplot` method is called, which plots the second and third columns of the matrices `A = Dr^(-1/2) U L` and `B = Dc^(-1/2) V L` where the singular value decomposition is `U L V`. Thus the x-axis is the canonical correlation times the row and column scores. Although this is called a biplot, it does *not* have any useful inner product relationship between the row and column scores. Think of this as an equally-scaled plot with two unrelated sets of labels. The origin is marked on the plot with a cross. (For other versions of this plot see the book.)

Value

An list object of class "correspondence" for which print, plot and biplot methods are supplied. The main components are the canonical correlation(s) and the row and column scores.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

Gower, J. C. and Hand, D. J. (1996) *Biplots.* Chapman & Hall.

See Also

svd, princomp

Examples

```
(ct <- corresp(~ Age + Eth, data = quine))
## Not run: plot(ct)

corresp(caith)
biplot(corresp(caith, nf = 2))
```

---

cov.rob *Resistant Estimation of Multivariate Location and Scatter*

Description

Compute a multivariate location and scale estimate with a high breakdown point – this can be thought of as estimating the mean and covariance of the good part of the data. cov.mve and cov.mcd are compatibility wrappers.

Usage

```
cov.rob(x, cor = FALSE, quantile.used = floor((n + p + 1)/2),
        method = c("mve", "mcd", "classical"),
        nsamp = "best", seed)

cov.mve(...)
cov.mcd(...)
```

Arguments

x	a matrix or data frame.
cor	should the returned result include a correlation matrix?
quantile.used	the minimum number of the data points regarded as good points.
method	the method to be used – minimum volume ellipsoid, minimum covariance determinant or classical product-moment. Using cov.mve or cov.mcd forces mve or mcd respectively.

nsamp	the number of samples or "best" or "exact" or "sample". If "sample" the number chosen is min(5*p, 3000), taken from Rousseeuw and Hubert (1997). If "best" exhaustive enumeration is done up to 5000 samples: if "exact" exhaustive enumeration will be attempted however many samples are needed.
seed	the seed to be used for random sampling: see RNGkind. The current value of .Random.seed will be preserved if it is set.
...	arguments to cov.rob other than method.

Details

For method "mve", an approximate search is made of a subset of size quantile.used with an enclosing ellipsoid of smallest volume; in method "mcd" it is the volume of the Gaussian confidence ellipsoid, equivalently the determinant of the classical covariance matrix, that is minimized. The mean of the subset provides a first estimate of the location, and the rescaled covariance matrix a first estimate of scatter. The Mahalanobis distances of all the points from the location estimate for this covariance matrix are calculated, and those points within the 97.5% point under Gaussian assumptions are declared to be good. The final estimates are the mean and rescaled covariance of the good points.

The rescaling is by the appropriate percentile under Gaussian data; in addition the first covariance matrix has an *ad hoc* finite-sample correction given by Marazzi.

For method "mve" the search is made over ellipsoids determined by the covariance matrix of p of the data points. For method "mcd" an additional improvement step suggested by Rousseeuw and van Driessen (1999) is used, in which once a subset of size quantile.used is selected, an ellipsoid based on its covariance is tested (as this will have no larger a determinant, and may be smaller).

Value

A list with components

center	the final estimate of location.
cov	the final estimate of scatter.
cor	(only is cor = TRUE) the estimate of the correlation matrix.
sing	message giving number of singular samples out of total
crit	the value of the criterion on log scale. For MCD this is the determinant, and for MVE it is proportional to the volume.
best	the subset used. For MVE the best sample, for MCD the best set of size quantile.used.
n.obs	total number of observations.

Author(s)

B.D. Ripley

References

P. J. Rousseeuw and A. M. Leroy (1987) *Robust Regression and Outlier Detection.* Wiley.

A. Marazzi (1993) *Algorithms, Routines and S Functions for Robust Statistics.* Wadsworth and Brooks/Cole.

P. J. Rousseeuw and B. C. van Zomeren (1990) Unmasking multivariate outliers and leverage points, *Journal of the American Statistical Association*, **85**, 633–639.

P. J. Rousseeuw and K. van Driessen (1999) A fast algorithm for the minimum covariance determinant estimator. *Technometrics* **41**, 212–223.

P. Rousseeuw and M. Hubert (1997) Recent developments in PROGRESS. In *L1-Statistical Procedures and Related Topics* ed Y. Dodge, IMS Lecture Notes volume **31**, pp. 201–214.

See Also

`lqs`

Examples

```
data(stackloss)
set.seed(123)
cov.rob(stackloss)
cov.rob(stack.x, method = "mcd", nsamp = "exact")
```

---

cov.trob *Covariance Estimation for Multivariate t Distribution*

Description

Estimates a covariance or correlation matrix assuming the data came from a multivariate t distribution: this provides some degree of robustness to outlier without giving a high breakdown point.

Usage

```
cov.trob(x, wt = rep(1, n), cor = FALSE, center = TRUE, nu = 5,
         maxit = 25, tol = 0.01)
```

Arguments

`x`	data matrix. Missing values (NAs) are not allowed.
`wt`	A vector of weights for each case: these are treated as if the case `i` actually occurred `wt[i]` times.
`cor`	Flag to choose between returning the correlation (`cor = TRUE`) or covariance (`cor = FALSE`) matrix.
`center`	a logical value or a numeric vector providing the location about which the covariance is to be taken. If `center = FALSE`, no centering is done; if `center = TRUE` the MLE of the location vector is used.
`nu`	"degrees of freedom" for the multivariate t distribution. Must exceed 2 (so that the covariance matrix is finite).
`maxit`	Maximum number of iterations in fitting.
`tol`	Convergence tolerance for fitting.

Value

A list with the following components

`cov`	the fitted covariance matrix.
`center`	the estimated or specified location vector.
`wt`	the specified weights: only returned if the `wt` argument was given.
`n.obs`	the number of cases used in the fitting.
`cor`	the fitted correlation matrix: only returned if `cor = TRUE`.
`call`	The matched call.
`iter`	The number of iterations used.

References

J. T. Kent, D. E. Tyler and Y. Vardi (1994) A curious likelihood identity for the multivariate t-distribution. *Communications in Statistics—Simulation and Computation* **23**, 441–453.

Venables, W. N. and Ripley, B. D. (1999) *Modern Applied Statistics with S-PLUS.* Third Edition. Springer.

See Also

`cov`, `cov.wt`, `cov.mve`

Examples

```
data(stackloss)
cov.trob(stackloss)
```

---

cpus *Performance of Computer CPUs*

Description

A relative performance measure and characteristics of 209 CPUs.

Usage

```
cpus
```

Format

The components are:

`name` Manufacturer and model
`syct` cycle time in nanoseconds
`mmin` minimum main memory in kilobytes
`mmax` maximum main memory in kilobytes
`cach` cache size in kilobytes
`chmin` minimum number of channels
`chmax` maximum number of channels
`perf` published performance on a benchmark mix relative to an IBM 370/158-3
`estperf` estimated performance (by Ein-Dor & Feldmesser)

Source

P. Ein-Dor & J. Feldmesser (1987) Attributes of the performance of central processing units: a relative performance prediction model. *Comm. ACM.* **30**, 308–317.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

---

crabs *Morphological Measurements on Leptograpsus Crabs*

Description

The `crabs` data frame has 200 rows and 8 columns, describing 5 morphological measurements on 50 crabs each of two colour forms and both sexes, of the species *Leptograpsus variegatus* collected at Fremantle, W. Australia.

Usage

```
crabs
```

Format

This data frame contains the following columns:

`sp` `species` - `"B"` or `"O"` for blue or orange

`sex` as it says

`index` index 1:50 within each of the four groups

`FL` frontal lobe size (mm)

`RW` rear width (mm)

`CL` carapace length (mm)

`CW` carapace width (mm)

`BD` body depth (mm)

Source

Campbell, N.A. and Mahon, R.J. (1974) A multivariate study of variation in two species of rock crab of genus *Leptograpsus. Australian Journal of Zoology* **22**, 417–425.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

`Cushings` *Diagnostic Tests on Patients with Cushing's Syndrome*

Description

Cushing's syndrome is a hypertensive disorder associated with over-secretion of cortisol by the adrenal gland. The observations are urinary excretion rates of two steroid metabolites.

Usage

```
Cushings
```

Format

The `Cushings` data frame has 27 rows and 3 columns:

`Tetrahydrocortisone` urinary excretion rate (mg/24hr) of Tetrahydrocortisone.

`Pregnanetriol` urinary excretion rate (mg/24hr) of Pregnanetriol.

`Type` underlying type of syndrome, coded `a` (adenoma) , `b` (bilateral hyperplasia), `c` (carcinoma) or `u` for unknown.

Source

J. Aitchison and I. R. Dunsmore (1975) *Statistical Prediction Analysis.* Cambridge University Press, Tables 11.1–3.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

`DDT` *DDT in Kale*

Description

A numeric vector of 15 measurements by different laboratories of the pesticide DDT in kale, in ppm (parts per million) using the multiple pesticide residue measurement.

Usage

```
DDT
```

Source

C. E. Finsterwalder (1976) Collaborative study of an extension of the Mills *et al* method for the determination of pesticide residues in food. *J. Off. Anal. Chem.* **59**, 169–171

R. G. Staudte and S. J. Sheather (1990) *Robust Estimation and Testing.* Wiley

deaths — *Monthly Deaths from Lung Diseases in the UK*

Description

A time series giving the monthly deaths from bronchitis, emphysema and asthma in the UK, 1974-1979, both sexes (deaths),

Usage

```
deaths
```

Source

P. J. Diggle (1990) *Time Series: A Biostatistical Introduction.* Oxford, table A.3

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

This the same as dataset ldeaths.

denumerate — *Transform an Allowable Formula for 'loglm' into one for 'terms'*

Description

loglm allows dimension numbers to be used in place of names in the formula. denumerate modifies such a formula into one that terms can process.

Usage

```
denumerate(x)
```

Arguments

x	A formula conforming to the conventions of loglm, that is, it may allow dimension numbers to stand in for names when specifying a log-linear model.

Details

The model fitting function loglm fits log-linear models to frequency data using iterative proportional scaling. To specify the model the user must nominate the margins in the data that remain fixed under the log-linear model. It is convenient to allow the user to use dimension numbers, 1, 2, 3, ... for the first, second, third, ..., margins in a similar way to variable names. As the model formula has to be parsed by terms, which treats 1 in a special way and requires parsable variable names, these formulae have to be modified by giving genuine names for these margin, or dimension numbers. denumerate replaces these numbers with names of a special form, namely n is replaced by .vn. This allows terms to parse the formula in the usual way.

Value

A linear model formula like that presented, except that where dimension numbers, say `n`, have been used to specify fixed margins these are replaced by names of the form `.vn` which may be processed by `terms`.

See Also

`renumerate`

Examples

```
denumerate(~(1+2+3)^3 + a/b)
## Not run: ~ (.v1 + .v2 + .v3)^3 + a/b
```

---

dose.p *Predict Doses for Binomial Assay model*

Description

Calibrate binomial assays, generalizing the calculation of LD50.

Usage

```
dose.p(obj, cf = 1:2, p = 0.5)
```

Arguments

`obj`	A fitted model object of class inheriting from `"glm"`.
`cf`	The terms in the coefficient vector giving the intercept and coefficient of (log-)dose
`p`	Probabilities at which to predict the dose needed.

Value

An object of class `"glm.dose"` giving the prediction (attribute `"p"` and standard error (attribute `"SE"`) at each response probability.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Springer.

Examples

```
ldose <- rep(0:5, 2)
numdead <- c(1, 4, 9, 13, 18, 20, 0, 2, 6, 10, 12, 16)
sex <- factor(rep(c("M", "F"), c(6, 6)))
SF <- cbind(numdead, numalive = 20 - numdead)
budworm.lg0 <- glm(SF ~ sex + ldose - 1, family = binomial)

dose.p(budworm.lg0, cf = c(1,3), p = 1:3/4)
dose.p(update(budworm.lg0, family = binomial(link=probit)),
       cf = c(1,3), p = 1:3/4)
```

drivers — *Deaths of Car Drivers in Great Britain 1969-84*

Description

A regular time series giving the monthly totals of car drivers in Great Britain killed or seriously injured Jan 1969 to Dec 1984. Compulsory wearing of seat belts was introduced on 31 Jan 1983.

Usage

```
drivers
```

Source

Harvey, A.C. (1989) *Forecasting, Structural Time Series Models and the Kalman Filter.* Cambridge University Press, pp. 519–523.

References

Venables, W. N. and Ripley, B. D. (1999) *Modern Applied Statistics with S-PLUS.* Third Edition. Springer.

dropterm — *Try All One-Term Deletions from a Model*

Description

Try fitting all models that differ from the current model by dropping a single term, maintaining marginality.

This function is generic; there exist methods for classes `lm` and `glm` and the default method will work for many other classes.

Usage

```
dropterm (object, ...)

## Default S3 method:
dropterm(object, scope, scale = 0, test = c("none", "Chisq"),
         k = 2, sorted = FALSE, trace = FALSE, ...)

## S3 method for class 'lm':
dropterm(object, scope, scale = 0, test = c("none", "Chisq", "F"),
         k = 2, sorted = FALSE, ...)

## S3 method for class 'glm':
dropterm(object, scope, scale = 0, test = c("none", "Chisq", "F"),
         k = 2, sorted = FALSE, trace = FALSE, ...)
```

Arguments

object	A object fitted by some model-fitting function.
scope	a formula giving terms which might be dropped. By default, the model formula. Only terms that can be dropped and maintain marginality are actually tried.
scale	used in the definition of the AIC statistic for selecting the models, currently only for lm, aov and glm models. Specifying scale asserts that the residual standard error or dispersion is known.
test	should the results include a test statistic relative to the original model? The F test is only appropriate for lm and aov models, and perhaps for some over-dispersed glm models. The Chisq test can be an exact test (lm models with known scale) or a likelihood-ratio test depending on the method.
k	the multiple of the number of degrees of freedom used for the penalty. Only k = 2 gives the genuine AIC: k = log(n) is sometimes referred to as BIC or SBC.
sorted	should the results be sorted on the value of AIC?
trace	if TRUE additional information may be given on the fits as they are tried.
...	arguments passed to or from other methods.

Details

The definition of AIC is only up to an additive constant: when appropriate (lm models with specified scale) the constant is taken to be that used in Mallows' Cp statistic and the results are labelled accordingly.

Value

A table of class "anova" containing at least columns for the change in degrees of freedom and AIC (or Cp) for the models. Some methods will give further information, for example sums of squares, deviances, log-likelihoods and test statistics.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

addterm, stepAIC

Examples

```
quine.hi <- aov(log(Days + 2.5) ~ .^4, quine)
quine.nxt <- update(quine.hi, . ~ . - Eth:Sex:Age:Lrn)
dropterm(quine.nxt, test=  "F")
quine.stp <- stepAIC(quine.nxt,
    scope = list(upper = ~Eth*Sex*Age*Lrn, lower = ~1),
    trace = FALSE)
dropterm(quine.stp, test = "F")
quine.3 <- update(quine.stp, . ~ . - Eth:Age:Lrn)
dropterm(quine.3, test = "F")
quine.4 <- update(quine.3, . ~ . - Eth:Age)
dropterm(quine.4, test = "F")
quine.5 <- update(quine.4, . ~ . - Age:Lrn)
```

```
dropterm(quine.5, test = "F")

house.glm0 <- glm(Freq ~ Infl*Type*Cont + Sat, family=poisson,
                  data = housing)
house.glm1 <- update(house.glm0, . ~ . + Sat*(Infl+Type+Cont))
dropterm(house.glm1, test = "Chisq")
```

`eagles` *Foraging Ecology of Bald Eagles*

Description

Knight and Skagen collected during a field study on the foraging behaviour of wintering Bald Eagles in Washington State, USA data concerning 160 attempts by one (pirating) Bald Eagle to steal a chum salmon from another (feeding) Bald Eagle.

Usage

```
eagles
```

Format

The `eagles` data frame has 8 rows and 5 columns.

- **`y`** Number of successful attempts.
- **`n`** Total number of attempts.
- **`P`** Size of pirating eagle (`L` = large, `S` = small).
- **`A`** Age of pirating eagle (`I` = immature, `A` = adult).
- **`V`** Size of victim eagle (`L` = large, `S` = small).

Source

Knight, R. L. and Skagen, S. K. (1988) Agonistic asymmetries and the foraging ecology of Bald Eagles. *Ecology* **69**, 1188–1194.

References

Venables, W. N. and Ripley, B. D. (1999) *Modern Applied Statistics with S-PLUS*. Third Edition. Springer.

Examples

```
eagles.glm <- glm(cbind(y, n - y) ~ P*A + V, data = eagles,
                  family = binomial)
dropterm(eagles.glm)
prof <- profile(eagles.glm)
plot(prof)
pairs(prof)
```

epil *Seizure Counts for Epileptics*

Description

Thall and Vail (1990) give a data set on two-week seizure counts for 59 epileptics. The number of seizures was recorded for a baseline period of 8 weeks, and then patients were randomly assigned to a treatment group or a control group. Counts were then recorded for four successive two-week periods. The subject's age is the only covariate.

Usage

```
epil
```

Format

This data frame has 236 rows and the following 9 columns:

y The count for the 2-week period.

trt The treatment, `"placebo"` or `"progabide"`.

base The counts in the baseline 8-week period.

age The subject's age, in years.

V4 `0/1` indicator variable of period 4.

subject The subject number, 1 to 59.

period The period, 1 to 4.

lbase The log-counts for the baseline period, centred to have zero mean.

lage The log-ages, centred to have zero mean.

Source

Thall, P. F. and Vail, S. C. (1990) Some covariance models for longitudinal count data with overdispersion. *Biometrics* **46**, 657–671.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth Edition. Springer.

Examples

```
summary(glm(y ~ lbase*trt + lage + V4, family = poisson,
            data = epil), cor = FALSE)
epil2 <- epil[epil$period == 1, ]
epil2["period"] <- rep(0, 59); epil2["y"] <- epil2["base"]
epil["time"] <- 1; epil2["time"] <- 4
epil2 <- rbind(epil, epil2)
epil2$pred <- unclass(epil2$trt) * (epil2$period > 0)
epil2$subject <- factor(epil2$subject)
epil3 <- aggregate(epil2, list(epil2$subject, epil2$period > 0),
   function(x) if(is.numeric(x)) sum(x) else x[1])
```

```
epil3$pred <- factor(epil3$pred,
   labels = c("base", "placebo", "drug"))

contrasts(epil3$pred) <- structure(contr.sdif(3),
    dimnames = list(NULL, c("placebo-base", "drug-placebo")))
summary(glm(y ~ pred + factor(subject) + offset(log(time)),
            family = poisson, data = epil3), cor = FALSE)

summary(glmmPQL(y ~ lbase*trt + lage + V4,
                random = ~ 1 | subject,
                family = poisson, data = epil))
summary(glmmPQL(y ~ pred, random = ~1 | subject,
                family = poisson, data = epil3))
```

`eqscplot` *Plots with Geometrically Equal Scales*

Description

Version of a scatterplot with scales chosen to be equal on both axes, that is 1cm represents the same units on each

Usage

```
eqscplot(x, y, ratio = 1, tol = 0.04, uin, ...)
```

Arguments

`x`	vector of x values, or a 2-column matrix, or a list with components `x` and `y`
`y`	vector of y values
`ratio`	desired ratio of units on the axes. Units on the y axis are drawn at `ratio` times the size of units on the x axis. Ignored if `uin` is specified and of length 2.
`tol`	proportion of white space at the margins of plot
`uin`	desired values for the units-per-inch parameter. If of length 1, the desired units per inch on the x axis.
`...`	further arguments for `plot` and graphical parameters. Note that `par(xaxs="i", yaxs="i")` is enforced, and `xlim` and `ylim` will be adjusted accordingly.

Details

Limits for the x and y axes are chosen so that they include the data. One of the sets of limits is then stretched from the midpoint to make the units in the ratio given by `ratio`. Finally both are stretched by `1 + tol` to move points away from the axes, and the points plotted.

Value

invisibly, the values of `uin` used for the plot.

Side Effects

performs the plot.

Note

Arguments `ratio` and `uin` were suggested by Bill Dunlap.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`plot`, `par`

`farms` *Ecological Factors in Farm Management*

Description

The `farms` data frame has 20 rows and 4 columns. The rows are farms on the Dutch island of Terschelling and the columns are factors describing the management of grassland.

Usage

```
farms
```

Format

This data frame contains the following columns:

`Mois` Five levels of soil moisture – level 3 does not occur at these 20 farms.

`Manag` Grassland management type (`SF` = standard, `BF` = biological, `HF` = hobby farming, `NM` = nature conservation).

`Use` Grassland use (`U1` = hay production, `U2` = intermediate, `U3` = grazing)

`Manure` Manure usage – classes `C0` to `C4`.

Source

J.C. Gower and D.J. Hand (1996) *Biplots*. Chapman & Hall, Table 4.6.

Quoted as from:
R.H.G. Jongman, C.J.F. ter Braak and O.F.R. van Tongeren (1987) *Data Analysis in Community and Landscape Ecology.* PUDOC, Wageningen

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

Examples

```
farms.mca <- mca(farms, abbrev = TRUE)  # Use levels as names
eqscplot(farms.mca$cs, type = "n")
text(farms.mca$rs, cex = 0.7)
text(farms.mca$cs, labels = dimnames(farms.mca$cs)[[1]], cex = 0.7)
```

fgl *Measurements of Forensic Glass Fragments*

Description

The `fgl` data frame has 214 rows and 10 columns. It was collected by B. German on fragments of glass collected in forensic work.

Usage

```
fgl
```

Format

This data frame contains the following columns:

`RI` refractive index; more precisely the refractive index is 1.518xxxx.
The remaining 8 measurements are percentages by weight of oxides.

`Na` sodium

`Mg` manganese

`Al` aluminium

`Si` silicon

`K` potassium

`Ca` calcium

`Ba` barium

`Fe` iron

`type` The fragments were originally classed into seven types, one of which was absent in this dataset. The categories which occur are window float glass (`WinF`: 70), window non-float glass (`WinNF`: 76), vehicle window glass (`Veh`: 17), containers (`Con`: 13), tableware (`Tabl`: 9) and vehicle headlamps (`Head`: 29).

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

fitdistr *Maximum-likelihood Fitting of Univariate Distributions*

Description

Maximum-likelihood fitting of univariate distributions, allowing parameters to be held fixed if desired.

Usage

```
fitdistr(x, densfun, start, ...)
```

Arguments

x	A numeric vector.
densfun	Either a character string or a function returning a density evaluated at its first argument. Distributions "beta", "cauchy", "chi-squared", "exponential", "f", "gamma", "geometric", "log-normal", "lognormal", "logistic", "negative binomial", "normal", "Poisson", "t" and "weibull" are recognised, case being ignored.
start	A named list giving the parameters to be optimized with initial values. This can be omitted for some of the named distributions and must be for others (see Details).
...	Additional parameters, either for densfun or for optim. In particular, it can be used to specify bounds via lower or upper or both. If arguments of densfun (or the density function corresponding to a character-string specification) are included they will be held fixed.

Details

For the Normal, log-Normal, exponential and Poisson distributions the closed-form MLEs (and exact standard errors) are used, and start should not be supplied.

For all other distributions, direct optimization of the log-likelihood is performed using optim. The estimated standard errors are taken from the observed information matrix, calculated by a numerical approximation. For one-dimensional problems the Nelder-Mead method is used and for multi-dimensional problems the BFGS method, unless arguments named lower or upper are supplied when L-BFGS-B is used or method is supplied explicitly.

For the "t" named distribution the density is taken to be the location-scale family with location m and scale s.

For the following named distributions, reasonable starting values will be computed if start is omitted or only partially specified: "cauchy", "gamma", "logistic", "negative binomial" (parametrized by mu and size), "t" and "weibull". Note that these starting values may not be good enough if the fit is poor: in particular they are not resistant to outliers unless the fitted distribution is long-tailed.

There are print, coef and logLik methods for class "fitdistr".

Value

An object of class "fitdistr", a list with three components,

estimate	the parameter estimates,
sd	the estimated standard errors, and
loglik	the log-likelihood.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

Examples

```
set.seed(123)
x <- rgamma(100, shape = 5, rate = 0.1)
fitdistr(x, "gamma")
## now do this directly with more control.
fitdistr(x, dgamma, list(shape = 1, rate = 0.1), lower = 0.01)

set.seed(123)
x2 <- rt(250, df = 9)
fitdistr(x2, "t", df = 9)
## allow df to vary: not a very good idea!
fitdistr(x2, "t")
## now do fixed-df fit directly with more control.
mydt <- function(x, m, s, df) dt((x-m)/s, df)/s
fitdistr(x2, mydt, list(m = 0, s = 1), df = 9, lower = c(-Inf, 0))

set.seed(123)
x3 <- rweibull(100, shape = 4, scale = 100)
fitdistr(x3, "weibull")

set.seed(123)
x4 <- rnegbin(500, mu = 5, theta = 4)
fitdistr(x4, "Negative Binomial") # R only
```

---

forbes *Forbes' Data on Boiling Points in the Alps*

Description

A data frame with 17 observations on boiling point (degrees F) and barometric pressure in inches of mercury.

Usage

```
forbes
```

Format

bp boiling point (degrees F)

pres barometric pressure in inches of mercury

Source

A. C. Atkinson (1985) *Plots, Transformations and Regression.* Oxford.

S. Weisberg (1980) *Applied Linear Regression.* Wiley.

`fractions` *Rational Approximation*

Description

Find rational approximations to the components of a real numeric object using a standard continued fraction method.

Usage

```
fractions(x, cycles = 10, max.denominator = 2000, ...)
```

Arguments

`x`	Any object of mode numeric. Missing values are now allowed.
`cycles`	The maximum number of steps to be used in the continued fraction approximation process.
`max.denominator`	An early termination criterion. If any partial denominator exceeds `max.denominator` the continued fraction stops at that point.
`...`	arguments passed to or from other methods.

Details

Each component is first expanded in a continued fraction of the form

```
x = floor(x) + 1/(p1 + 1/(p2 + ...)))
```

where `p1`, `p2`, ...are positive integers, terminating either at `cycles` terms or when a `pj > max.denominator`. The continued fraction is then re-arranged to retrieve the numerator and denominator as integers.

The numerators and denominators are then combined into a character vector that becomes the `"fracs"` attribute and used in printed representations.

Arithmetic operations on `"fractions"` objects have full floating point accuracy, but the character representation printed out may not.

Value

An object of class `"fractions"`. A structure with `.Data` component the same as the input numeric `x`, but with the rational approximations held as a character vector attribute, `"fracs"`. Arithmetic operations on `"fractions"` objects are possible.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth Edition. Springer.

See Also

`rational`

Examples

```
X <- matrix(runif(25), 5, 5)
solve(X, X/5)
##              [,1]        [,2]       [,3]        [,4]        [,5]
##  [1,]  2.0000e-01  3.7199e-17 1.2214e-16  5.7887e-17 -8.7841e-17
##  [2,] -1.1473e-16  2.0000e-01 7.0955e-17  2.0300e-17 -1.0566e-16
##  [3,]  2.7975e-16  1.3653e-17 2.0000e-01 -1.3397e-16  1.5577e-16
##  [4,] -2.9196e-16  2.0412e-17 1.5618e-16  2.0000e-01 -2.1921e-16
##  [5,] -3.6476e-17 -3.6430e-17 3.6432e-17  4.7690e-17  2.0000e-01

fractions(solve(X, X/5))
##      [,1] [,2] [,3] [,4] [,5]
## [1,] 1/5    0    0    0    0
## [2,]   0  1/5    0    0    0
## [3,]   0    0  1/5    0    0
## [4,]   0    0    0  1/5    0
## [5,]   0    0    0    0  1/5

fractions(solve(X, X/5)) + 1
##      [,1] [,2] [,3] [,4] [,5]
## [1,] 6/5    1    1    1    1
## [2,]   1  6/5    1    1    1
## [3,]   1    1  6/5    1    1
## [4,]   1    1    1  6/5    1
## [5,]   1    1    1    1  6/5
```

GAGurine *Level of GAG in Urine of Children*

Description

Data were collected on the concentration of a chemical GAG in the urine of 314 children aged from zero to seventeen years. The aim of the study was to produce a chart to help a paediatrican to assess if a child's GAG concentration is "normal".

Usage

```
GAGurine
```

Format

This data frame contains the following columns:

`Age` age of child in years

`GAG` concentration of GAG (the units have been lost)

Source

Mrs Susan Prosser, Paediatrics Department, University of Oxford, via Department of Statistics Consulting Service.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

galaxies *Velocities for 82 Galaxies*

Description

A numeric vector of velocities in km/sec of 82 galaxies from 6 well-separated conic sections of an `unfilled` survey of the Corona Borealis region. Multimodality in such surveys is evidence for voids and superclusters in the far universe.

Usage

```
galaxies
```

Note

There is an 83rd measurement of 5607 km/sec in the Postman *et al.* paper which is omitted in Roeder (1990) and from the dataset here.

There is also a typo: this dataset has 78th observation 26690 whihc should be 26960.

Source

Roeder, K. (1990) Density estimation with confidence sets exemplified by superclusters and voids in galaxies. *Journal of the American Statistical Association* **85**, 617–624.

Postman, M., Huchra, J. P. and Geller, M. J. (1986) Probes of large-scale structures in the Corona Borealis region. *Astrophysical Journal* **92**, 1238–1247.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*. Fourth edition. Springer.

Examples

```
gal <- galaxies/1000
c(width.SJ(gal, method = "dpi"), width.SJ(gal))
plot(x = c(0, 40), y = c(0, 0.3), type = "n", bty = "l",
     xlab = "velocity of galaxy (1000km/s)", ylab = "density")
rug(gal)
lines(density(gal, width = 3.25, n = 200), lty = 1)
lines(density(gal, width = 2.56, n = 200), lty = 3)
```

gamma.dispersion *Calculate the MLE of the Gamma Dispersion Parameter in a GLM Fit*

Description

A front end to `gamma.shape` for convenience. Finds the reciprocal of the estimate of the shape parameter only.

Usage

```
gamma.dispersion(object, ...)
```

Arguments

object	Fitted model object giving the gamma fit.
...	Additional arguments passed on to gamma.shape.

Value

The MLE of the dispersion parameter of the gamma distribution.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

gamma.shape.glm, including the example on its help page.

---

gamma.shape *Estimate the Shape Parameter of the Gamma Distribution in a GLM Fit*

Description

Find the maximum likelihood estimate of the shape parameter of the gamma distribution after fitting a Gamma generalized linear model.

Usage

```
## S3 method for class 'glm':
gamma.shape(object, it.lim = 10,
            eps.max = .Machine$double.eps^0.25, verbose = FALSE, ...)
```

Arguments

object	Fitted model object from a Gamma family or quasi family with variance = "mu^2".
it.lim	Upper limit on the number of iterations.
eps.max	Maximum discrepancy between approximations for the iteration process to continue.
verbose	If TRUE, causes successive iterations to be printed out. The initial estimate is taken from the deviance.
...	further arguments passed to or from other methods.

Details

A glm fit for a Gamma family correctly calculates the maximum likelihood estimate of the mean parameters but provides only a crude estimate of the dispersion parameter. This function takes the results of the glm fit and solves the maximum likelihood equation for the reciprocal of the dispersion parameter, which is usually called the shape (or exponent) parameter.

Value

List of two components

alpha	the maximum likelihood estimate
SE	the approximate standard error, the square-root of the reciprocal of the observed information.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

gamma.dispersion

Examples

```
clotting <- data.frame(
    u = c(5,10,15,20,30,40,60,80,100),
    lot1 = c(118,58,42,35,27,25,21,19,18),
    lot2 = c(69,35,26,21,18,16,13,12,12))
clot1 <- glm(lot1 ~ log(u), data = clotting, family = Gamma)
gamma.shape(clot1)
## Not run:
Alpha: 538.13
   SE: 253.60
## End(Not run)
gm <- glm(Days + 0.1 ~ Age*Eth*Sex*Lrn,
          quasi(link=log, variance="mu^2"), quine, start = rep(0,32))
gamma.shape(gm, verbose = TRUE)
## Not run:
Initial estimate: 1.0603
Iter.  1  Alpha: 1.23840774338543
Iter.  2  Alpha: 1.27699745778205
Iter.  3  Alpha: 1.27834332265501
Iter.  4  Alpha: 1.27834485787226

Alpha: 1.27834
   SE: 0.13452
## End(Not run)
summary(gm, dispersion = gamma.dispersion(gm))  # better summary
```

---

gehan *Remission Times of Leukaemia Patients*

Description

A data frame from a trial of 42 leukaemia patients. Some were treated with the drug *6-mercaptopurine* and the rest are controls. The trial was designed as matched pairs, both withdrawn from the trial when either came out of remission.

Usage

```
gehan
```

Format

This data frame contains the following columns:

pair label for pair

time remission time in weeks

cens censoring, 0/1

treat treatment, control or 6-MP

Source

Cox, D. R. and Oakes, D. (1984) *Analysis of Survival Data.* Chapman & Hall, p. 7. Taken from

Gehan, E.A. (1965) A generalized Wilcoxon test for comparing arbitrarily single-censored samples. *Biometrika* **52**, 203–233.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

Examples

```
library(survival)
gehan.surv <- survfit(Surv(time, cens) ~ treat, data = gehan,
     conf.type = "log-log")
summary(gehan.surv)
survreg(Surv(time, cens) ~ factor(pair) + treat, gehan, dist = "exp")
summary(survreg(Surv(time, cens) ~ treat, gehan, dist = "exp"))
summary(survreg(Surv(time, cens) ~ treat, gehan))
gehan.cox <- coxph(Surv(time, cens) ~ treat, gehan)
summary(gehan.cox)
```

---

genotype *Rat Genotype Data*

Description

Data from a foster feeding experiment with rat mothers and litters of four different genotypes: `A`, `B`, `I` and `J`. Rice litters were separated from their natural mothers at birth and given to foster mothers to rear.

Usage

```
genotype
```

Format

The data frame has the following components:

Litter The genotype of the litter

Mother The genotype of the foster mother

Wt Litter average weight gain of the litter, in grams at age 28 days. (The source states that the within-litter variability is negligible.)

Source

Scheffe, H. (1959) *The Analysis of Variance* Wiley p. 140.

Bailey, D. W. (1953) *The Inheritance of Maternal Influences on the Growth of the Rat.* Unpublished Ph.D. thesis, University of California. Table B of the Appendix.

References

Venables, W. N. and Ripley, B. D. (1999) *Modern Applied Statistics with S-PLUS.* Third Edition. Springer.

---

geyser *Old Faithful Geyser Data*

Description

A version of the eruptions data from the 'Old Faithful' geyser in Yellowstone National Park, Wyoming. This version comes from Azzalini and Bowman (1990) and is of continuous measurement from August 1 to August 15, 1985.

Some nocturnal duration measurements were coded as 2, 3 or 4 minutes, having originally been described as 'short', 'medium' or 'long'.

Usage

```
geyser
```

Format

A data frame with 299 observations on 2 variables.

`duration`	numeric	Eruption time in mins
`waiting`	numeric	Waiting time to next eruption

References

Azzalini, A. and Bowman, A. W. (1990) A look at some data on the Old Faithful geyser. *Applied Statistics* **39**, 357–365.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`faithful`

---

gilgais *Line Transect of Soil in Gilgai Territory*

Description

This dataset was collected on a line transect survey in gilgai territory in New South Wales, Australia. Gilgais are natural gentle depressions in otherwise flat land, and sometimes seem to be regularly distributed. The data collection was stimulated by the question: are these patterns reflected in soil properties? At each of 365 sampling locations on a linear grid of 4 meters spacing, samples were taken at depths 0-10 cm, 30-40 cm and 80-90 cm below the surface. pH, electrical conductivity and chloride content were measured on a 1:5 soil:water extract from each sample.

Usage

```
gilgais
```

Format

This data frame contains the following columns:

pH00 pH at depth 0-10cm

pH30 pH at depth 30-40cm

pH80 pH at depth 80-90cm

e00 electrical conductivity in mS/cm (0-10 cm)

e30 electrical conductivity in mS/cm (30-40 cm)

e80 electrical conductivity in mS/cm (80-90 cm)

c00 chloride content in ppm (0-10 cm)

c30 chloride content in ppm (30-40 cm)

c80 chloride content in ppm (80-90 cm)

Source

Webster, R. (1977) Spectral analysis of gilgai soil. *Australian Journal of Soil Research* **15**, 191–204.

Laslett, G. M. (1989) Kriging and splines: An empirical comparison of their predictive performance in some applications (with discussion). *Journal of the American Statistical Association* **89**, 319–409

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

---

ginv *Generalized Inverse of a Matrix*

Description

Calculates the Moore-Penrose generalized inverse of a matrix `X`.

Usage

```
ginv(X, tol = sqrt(.Machine$double.eps))
```

Arguments

X	Matrix for which the Moore-Penrose inverse is required.
tol	A relative tolerance to detect zero singular values.

Value

A MP generalized inverse matrix for X.

References

Venables, W. N. and Ripley, B. D. (1999) *Modern Applied Statistics with S-PLUS.* Third Edition. Springer. p.100.

See Also

solve, svd, eigen

Examples

```
## Not run:
# The function is currently defined as
function(X, tol = sqrt(.Machine$double.eps))
{
## Generalized Inverse of a Matrix
  dnx <- dimnames(X)
  if(is.null(dnx)) dnx <- vector("list", 2)
  s <- svd(X)
  nz <- s$d > tol * s$d[1]
  structure(
    if(any(nz)) s$v[, nz] %*% (t(s$u[, nz])/s$d[nz]) else X,
    dimnames = dnx[2:1])
}
## End(Not run)
```

---

glm.convert *Change a Negative Binomial fit to a GLM fit*

Description

This function modifies an output object from glm.nb() to one that looks like the output from glm() with a negative binomial family. This allows it to be updated keeping the theta parameter fixed.

Usage

```
glm.convert(object)
```

Arguments

object	An object of class "negbin", typically the output from glm.nb().

Details

Convenience function needed to effect some low level changes to the structure of the fitted model object.

Value

An object of class "glm" with negative binomial family. The theta parameter is then fixed at its present estimate.

See Also

glm.nb, negative.binomial, glm

Examples

```
quine.nb1 <- glm.nb(Days ~ Sex/(Age + Eth*Lrn), data = quine)
quine.nbA <- glm.convert(quine.nb1)
quine.nbB <- update(quine.nb1, . ~ . + Sex:Age:Lrn)
anova(quine.nbA, quine.nbB)
```

---

glm.nb *Fit a Negative Binomial Generalized Linear Model*

Description

A modification of the system function glm() to include estimation of the additional parameter, theta, for a Negative Binomial generalized linear model.

Usage

```
glm.nb(formula, data, weights, subset, na.action,
       start = NULL, etastart, mustart,
       control = glm.control(...), method = "glm.fit",
       model = TRUE, x = FALSE, y = TRUE, contrasts = NULL, ...,
       init.theta, link = log)
```

Arguments

formula, data, weights, subset, na.action, start, etastart, mustart, control, me
: arguments for the glm() function. Note that these exclude family and offset (but offset() can be used).

init.theta
: Optional initial value for the theta parameter. If omitted a moment estimator after an initial fit using a Poisson GLM is used.

link
: The link function. Currently must be one of log, sqrt or identity.

Details

An alternating iteration process is used. For given `theta` the GLM is fitted using the same process as used by `glm()`. For fixed means the `theta` parameter is estimated using score and information iterations. The two are alternated until convergence of both. (The number of alternations and the number of iterations when estimating `theta` are controlled by the `maxit` parameter of `glm.control`.)

Setting `trace > 0` traces the alternating iteration process. Setting `trace > 1` traces the `glm` fit, and setting `trace > 2` traces the estimation of `theta`.

Value

A fitted model object of class `negbin` inheriting from `glm` and `lm`. The object is like the output of `glm` but contains three additional components, namely `theta` for the ML estimate of theta, `SE.theta` for its approximate standard error (using observed rather than expected information), and `twologlik` for twice the log-likelihood function.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`glm`, `negative.binomial`, `anova.negbin`, `summary.negbin`, `theta.md`

Examples

```
quine.nb1 <- glm.nb(Days ~ Sex/(Age + Eth*Lrn), data = quine)
quine.nb2 <- update(quine.nb1, . ~ . + Sex:Age:Lrn)
quine.nb3 <- update(quine.nb2, Days ~ .^4)
anova(quine.nb1, quine.nb2, quine.nb3)
```

`glmmPQL` *Fit Generalized Linear Mixed Models via PQL*

Description

Fit a GLMM model with multivariate normal random effects, using Penalized Quasi-Likelihood.

Usage

```
glmmPQL(fixed, random, family, data, correlation, weights,
        control, niter = 10, verbose = TRUE, ...)
```

Arguments

`fixed`	a two-sided linear formula giving fixed-effects part of the model.
`random`	A formula or list of formulae describing the random effects.
`family`	a GLM family.
`data`	an optional data frame used as the first place to find variables in the formulae.
`correlation`	an optional correlation structure.

weights	optional case weights as in glm.
control	an optional argument to be passed to lme.
niter	maximum number of iterations.
verbose	logical: print out record of iterations?
...	Further arguments for lme.

Details

glmmPQL works by repeated calls to lme, so package nlme will be loaded at first use if necessary.

Value

A object of class "lme": see lmeObject.

References

Schall, R. (1991) Estimation in generalized linear models with random effects. *Biometrika* **78**, 719–727.

Breslow, N. E. and Clayton, D. G. (1993) Approximate inference in generalized linear mixed models. *Journal of the American Statistical Association* **88**, 9–25.

Wolfinger, R. and O'Connell, M. (1993) Generalized linear mixed models: a pseudo-likelihood approach. *Journal of Statistical Computation and Simulation* **48**, 233–243.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*. Fourth edition. Springer.

See Also

lme

Examples

```
library(nlme) # will be loaded automatically if omitted
summary(glmmPQL(y ~ trt + I(week > 2), random = ~ 1 | ID,
                family = binomial, data = bacteria))
```

---

hills *Record Times in Scottish Hill Races*

Description

The record times in 1984 for 35 Scottish hill races.

Usage

```
hills
```

Format

The components are:

dist distance in miles (on the map)

climb total height gained during the route, in feet.

time record time in minutes.

Source

A.C. Atkinson (1986) Comment: Aspects of diagnostic regression analysis. *Statistical Science* **1**, 397–402.

[A.C. Atkinson (1988) Transformations unmasked. *Technometrics* **30**, 311–318 "corrects" the time for Knock Hill from 78.65 to 18.65. It is unclear if this based on the original records.]

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

`hist.scott` *Plot a Histogram with Automatic Bin Width Selection*

Description

Plot a histogram with automatic bin width selection, using the Scott or Freedman–Diaconis formulae.

Usage

```
hist.scott(x, prob = TRUE, xlab = deparse(substitute(x)), ...)
hist.FD(x, prob = TRUE, xlab = deparse(substitute(x)), ...)
```

Arguments

`x`	A data vector
`prob`	Should the plot have unit area, so be a density estimate?
`xlab, ...`	Further arguments to `hist`.

Value

For the `nclass.*` functions, the suggested number of classes.

Side Effects

Plot a histogram.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Springer.

See Also

`hist`

`housing` *Frequency Table from a Copenhagen Housing Conditions Survey*

Description

The `housing` data frame has 72 rows and 5 variables.

Usage

```
housing
```

Format

`Sat` Satisfaction of householders with their present housing circumstances, (High, Medium or Low, ordered factor).

`Infl` Perceived degree of influence householders have on the management of the property (High, Medium, Low).

`Type` Type of rental accommodation, (Tower, Atrium, Apartment, Terrace).

`Cont` Contact residents are afforded with other residents, (Low, High).

`Freq` Frequencies: the numbers of residents in each class.

Source

Madsen, M. (1976) Statistical analysis of multiple contingency tables. Two examples. *Scand. J. Statist.* **3**, 97–106.

Cox, D. R. and Snell, E. J. (1984) *Applied Statistics, Principles and Examples*. Chapman & Hall.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

Examples

```
options(contrasts = c("contr.treatment", "contr.poly"))

# Surrogate Poisson models
house.glm0 <- glm(Freq ~ Infl*Type*Cont + Sat, family = poisson,
                  data = housing)
summary(house.glm0, cor = FALSE)

addterm(house.glm0, ~. + Sat:(Infl+Type+Cont), test = "Chisq")

house.glm1 <- update(house.glm0, . ~ . + Sat*(Infl+Type+Cont))
summary(house.glm1, cor = FALSE)

1 - pchisq(deviance(house.glm1), house.glm1$df.resid)

dropterm(house.glm1, test = "Chisq")

addterm(house.glm1, ~. + Sat:(Infl+Type+Cont)^2, test  =  "Chisq")

hnames <- lapply(housing[, -5], levels) # omit Freq
```

```
newData <- expand.grid(hnames)
newData$Sat <- ordered(newData$Sat)
house.pm <- predict(house.glm1, newData,
                    type = "response")  # poisson means
house.pm <- matrix(house.pm, ncol = 3, byrow = TRUE,
                   dimnames = list(NULL, hnames[[1]]))
house.pr <- house.pm/drop(house.pm %*% rep(1, 3))
cbind(expand.grid(hnames[-1]), round(house.pr, 2))

# Iterative proportional scaling
loglm(Freq ~ Infl*Type*Cont + Sat*(Infl+Type+Cont), data = housing)

# multinomial model
library(nnet)
(house.mult<- multinom(Sat ~ Infl + Type + Cont, weights = Freq,
                       data = housing))
house.mult2 <- multinom(Sat ~ Infl*Type*Cont, weights = Freq,
                        data = housing)
anova(house.mult, house.mult2)

house.pm <- predict(house.mult, expand.grid(hnames[-1]),
                    type = "probs")
cbind(expand.grid(hnames[-1]), round(house.pm, 2))

# proportional odds model
house.cpr <- apply(house.pr, 1, cumsum)
logit <- function(x) log(x/(1-x))
house.ld <- logit(house.cpr[2, ]) - logit(house.cpr[1, ])
(ratio <- sort(drop(house.ld)))
mean(ratio)

(house.plr <- polr(Sat ~ Infl + Type + Cont,
                   data = housing, weights = Freq))

house.pr1 <- predict(house.plr, expand.grid(hnames[-1]),
                     type = "probs")
cbind(expand.grid(hnames[-1]), round(house.pr1, 2))

Fr <- matrix(housing$Freq, ncol  =  3, byrow = TRUE)
2*sum(Fr*log(house.pr/house.pr1))

house.plr2 <- stepAIC(house.plr, ~.^2)
house.plr2$anova
```

`huber` *Huber M-estimator of Location with MAD Scale*

Description

Finds the Huber M-estimator of location with MAD scale.

Usage

```
huber(y, k = 1.5, tol = 1e-06)
```

Arguments

`y`	vector of data values
`k`	Winsorizes at `k` standard deviations
`tol`	convergence tolerance

Value

list of location and scale parameters

`mu`	location estimate
`s`	MAD scale estimate

References

Huber, P. J. (1981) *Robust Statistics.* Wiley.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`hubers`, `mad`

Examples

```
huber(chem)
```

`hubers` *Huber Proposal 2 Robust Estimator of Location and/or Scale*

Description

Finds the Huber M-estimator for location with scale specified, scale with location specified, or both if neither is specified.

Usage

```
hubers(y, k = 1.5, mu, s, initmu = median(y), tol = 1e-06)
```

Arguments

`y`	vector y of data values
`k`	Winsorizes at `k` standard deviations
`mu`	specified location
`s`	specified scale
`initmu`	initial value of `mu`
`tol`	convergence tolerance

Value

list of location and scale estimates

`mu`	location estimate
`s`	scale estimate

References

Huber, P. J. (1981) *Robust Statistics.* Wiley.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`huber`

Examples

```
hubers(chem)
hubers(chem, mu=3.68)
```

---

immer *Yields from a Barley Field Trial*

Description

The `immer` data frame has 30 rows and 4 columns. Five varieties of barley were grown in six locations in each of 1931 and 1932.

Usage

```
immer
```

Format

This data frame contains the following columns:

`Loc` The location.

`Var` The variety of barley (`"manchuria"`, `"svansota"`, `"velvet"`, `"trebi"` and `"peatland"`).

`Y1` Yield in 1931

`Y2` Yield in 1932

Source

Immer, F.R., Hayes, H.D. and LeRoy Powers (1934) Statistical determination of barley varietal adaptation. *Journal of the American Society for Agronomy* **26**, 403–419.

Fisher, R.A. (1947) *The Design of Experiments.* 4th edition. Edinburgh: Oliver and Boyd.

References

Venables, W. N. and Ripley, B. D. (1999) *Modern Applied Statistics with S-PLUS.* Third Edition. Springer.

Examples

```
immer.aov <- aov(cbind(Y1,Y2) ~ Loc + Var, data = immer)
summary(immer.aov)

immer.aov <- aov((Y1+Y2)/2 ~ Var + Loc, data = immer)
summary(immer.aov)
model.tables(immer.aov, type = "means", se = TRUE, cterms = "Var")
```

Insurance *Numbers of Car Insurance claims*

Description

The data given in data frame `Insurance` consist of the numbers of policyholders of an insurance company who were exposed to risk, and the numbers of car insurance claims made by those policyholders in the third quarter of 1973.

Usage

```
Insurance
```

Format

This data frame contains the following columns:

`District` district of policyholder (1 to 4): 4 is major cities.

`Group` group of car (1 to 4), <1 litre, 1–1.5 litre, 1.5–2 litre, >2 litre.

`Age` of driver in 4 ordered groups, <25, 25–29, 30–35, >35.

`Holders` numbers of policyholders

`Claims` numbers of claims

Source

L. A. Baxter, S. M. Coutts and G. A. F. Ross (1980) Applications of linear models in motor insurance. *Proceedings of the 21st International Congress of Actuaries, Zurich* pp. 11–29

M. Aitkin, D. Anderson, B. Francis and J. Hinde (1989) *Statistical Modelling in GLIM.* Oxford University Press.

References

Venables, W. N. and Ripley, B. D. (1999) *Modern Applied Statistics with S-PLUS.* Third Edition. Springer.

Examples

```
## main-effects fit as Poisson GLM with offset
glm(Claims ~ District + Group + Age + offset(log(Holders)),
    data = Insurance, family = poisson)

# same via loglm
loglm(Claims ~ District + Group + Age + offset(log(Holders)),
      data = Insurance)
```

---

isoMDS *Kruskal's Non-metric Multidimensional Scaling*

Description

One form of non-metric multidimensional scaling

Usage

```
isoMDS(d, y = cmdscale(d, k), k = 2, maxit = 50, trace = TRUE,
       tol = 1e-3, p = 2)

Shepard(d, x, p = 2)
```

Arguments

`d`	distance structure of the form returned by `dist`, or a full, symmetric matrix. Data are assumed to be dissimilarities or relative distances, but must be positive except for self-distance. Both missing and infinite values are allowed.
`y`	An initial configuration. If none is supplied, `cmdscale` is used to provide the classical solution, unless there are missing or infinite dissimilarities.
`k`	The desired dimension for the solution, passed to `cmdscale`.
`maxit`	The maximum number of iterations.
`trace`	Logical for tracing optimization. Default `TRUE`.
`tol`	convergence tolerance.
`p`	Power for Minkowski distance in the configuration space.
`x`	A final configuration.

Details

This chooses a k-dimensional (default k = 2) configuration to minimize the stress, the square root of the ratio of the sum of squared differences between the input distances and those of the configuration to the sum of configuration distances squared. However, the input distances are allowed a monotonic transformation.

An iterative algorithm is used, which will usually converge in around 10 iterations. As this is necessarily an $O(n^2)$ calculation, it is slow for large datasets. Further, since for the default $p = 2$ the configuration is only determined up to rotations and reflections (by convention the centroid is at the origin), the result can vary considerably from machine to machine.

Value

Two components:

points	A k-column vector of the fitted configuration.
stress	The final stress achieved (in percent).

Side Effects

If trace is true, the initial stress and the current stress are printed out every 5 iterations.

References

T. F. Cox and M. A. A. Cox (1994, 2001) *Multidimensional Scaling*. Chapman & Hall.

Ripley, B. D. (1996) *Pattern Recognition and Neural Networks*. Cambridge University Press.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*. Fourth edition. Springer.

See Also

cmdscale, sammon

Examples

```
data(swiss)
swiss.x <- as.matrix(swiss[, -1])
swiss.dist <- dist(swiss.x)
swiss.mds <- isoMDS(swiss.dist)
plot(swiss.mds$points, type = "n")
text(swiss.mds$points, labels = as.character(1:nrow(swiss.x)))
swiss.sh <- Shepard(swiss.dist, swiss.mds$points)
plot(swiss.sh, pch = ".")
lines(swiss.sh$x, swiss.sh$yf, type = "S")
```

---

kde2d — *Two-Dimensional Kernel Density Estimation*

Description

Two-dimensional kernel density estimation with an axis-aligned bivariate normal kernel, evaluated on a square grid.

Usage

```
kde2d(x, y, h, n = 25, lims = c(range(x), range(y)))
```

Arguments

x	x coordinate of data
y	y coordinate of data
h	vector of bandwidths for x and y directions. Defaults to normal reference bandwidth (see bandwidth.nrd).
n	Number of grid points in each direction.
lims	The limits of the rectangle covered by the grid as c(xl, xu, yl, yu).

Value

A list of three components.

`x, y`	The x and y coordinates of the grid points, vectors of length `n`.
`z`	An `n` x `n` matrix of the evaluated density.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

Examples

```
attach(geyser)
plot(duration, waiting, xlim = c(0.5,6), ylim = c(40,100))
f1 <- kde2d(duration, waiting, n = 50, lims = c(0.5, 6, 40, 100))
image(f1, zlim = c(0, 0.05))
f2 <- kde2d(duration, waiting, n = 50, lims = c(0.5, 6, 40, 100),
            h = c(width.SJ(duration), width.SJ(waiting)) )
image(f2, zlim = c(0, 0.05))
persp(f2, phi = 30, theta = 20, d = 5)

plot(duration[-272], duration[-1], xlim = c(0.5, 6),
     ylim = c(1, 6),xlab = "previous duration", ylab = "duration")
f1 <- kde2d(duration[-272], duration[-1],
            h = rep(1.5, 2), n = 50, lims = c(0.5, 6, 0.5, 6))
contour(f1, xlab = "previous duration",
        ylab = "duration", levels  =  c(0.05, 0.1, 0.2, 0.4) )
f1 <- kde2d(duration[-272], duration[-1],
            h = rep(0.6, 2), n = 50, lims = c(0.5, 6, 0.5, 6))
contour(f1, xlab = "previous duration",
        ylab = "duration", levels  =  c(0.05, 0.1, 0.2, 0.4) )
f1 <- kde2d(duration[-272], duration[-1],
            h = rep(0.4, 2), n = 50, lims = c(0.5, 6, 0.5, 6))
contour(f1, xlab = "previous duration",
        ylab = "duration", levels  =  c(0.05, 0.1, 0.2, 0.4) )
detach("geyser")
```

`lda` *Linear Discriminant Analysis*

Description

Linear discriminant analysis.

Usage

```
lda(x, ...)

## S3 method for class 'formula':
lda(formula, data, ..., subset, na.action)

## Default S3 method:
lda(x, grouping, prior = proportions, tol = 1.0e-4,
```

```
    method, CV = FALSE, nu, ...)

## S3 method for class 'data.frame':
lda(x, ...)

## S3 method for class 'matrix':
lda(x, grouping, ..., subset, na.action)
```

Arguments

formula	A formula of the form `groups ~ x1 + x2 + ...` That is, the response is the grouping factor and the right hand side specifies the (non-factor) discriminators.
data	Data frame from which variables specified in `formula` are preferentially to be taken.
x	(required if no formula is given as the principal argument.) a matrix or data frame or Matrix containing the explanatory variables.
grouping	(required if no formula principal argument is given.) a factor specifying the class for each observation.
prior	the prior probabilities of class membership. If unspecified, the class proportions for the training set are used. If present, the probabilities should be specified in the order of the factor levels.
tol	A tolerance to decide if a matrix is singular; it will reject variables and linear combinations of unit-variance variables whose variance is less than `tol^2`.
subset	An index vector specifying the cases to be used in the training sample. (NOTE: If given, this argument must be named.)
na.action	A function to specify the action to be taken if `NA`s are found. The default action is for the procedure to fail. An alternative is `na.omit`, which leads to rejection of cases with missing values on any required variable. (NOTE: If given, this argument must be named.)
method	`"moment"` for standard estimators of the mean and variance, `"mle"` for MLEs, `"mve"` to use `cov.mve`, or `"t"` for robust estimates based on a t distribution.
CV	If true, returns results (classes and posterior probabilities) for leave-one-out cross-validation. Note that if the prior is estimated, the proportions in the whole dataset are used.
nu	degrees of freedom for `method = "t"`.
...	arguments passed to or from other methods.

Details

The function tries hard to detect if the within-class covariance matrix is singular. If any variable has within-group variance less than `tol^2` it will stop and report the variable as constant. This could result from poor scaling of the problem, but is more likely to result from constant variables.

Specifying the `prior` will affect the classification unless over-ridden in `predict.lda`. Unlike in most statistical packages, it will also affect the rotation of the linear discriminants within their space, as a weighted between-groups covariance matrix is used. Thus the first few linear discriminants emphasize the differences between groups with the weights given by the prior, which may differ from their prevalence in the dataset.

If one or more groups is missing in the supplied data, they are dropped with a warning, but the classifications produced are with respect to the original set of levels.

Value

If `CV = TRUE` the return value is a list with components `class`, the MAP classification (a factor), and `posterior`, posterior probabilities for the classes.

Otherwise it is an object of class `"lda"` containing the following components:

`prior`	the prior probabilities used.
`means`	the group means.
`scaling`	a matrix which transforms observations to discriminant functions, normalized so that within groups covariance matrix is spherical.
`svd`	the singular values, which give the ratio of the between- and within-group standard deviations on the linear discriminant variables. Their squares are the canonical F-statistics.
`N`	The number of observations used.
`call`	The (matched) function call.

Note

This function may be called giving either a formula and optional data frame, or a matrix and grouping factor as the first two arguments. All other arguments are optional, but `subset=` and `na.action=`, if required, must be fully named.

If a formula is given as the principal argument the object may be modified using `update()` in the usual way.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*. Fourth edition. Springer.

Ripley, B. D. (1996) *Pattern Recognition and Neural Networks*. Cambridge University Press.

See Also

`predict.lda`, `qda`, `predict.qda`

Examples

```
data(iris3)
Iris <- data.frame(rbind(iris3[,,1], iris3[,,2], iris3[,,3]),
                   Sp = rep(c("s","c","v"), rep(50,3)))
train <- sample(1:150, 75)
table(Iris$Sp[train])
## your answer may differ
##  c  s  v
## 22 23 30
z <- lda(Sp ~ ., Iris, prior = c(1,1,1)/3, subset = train)
predict(z, Iris[-train, ])$class
##  [1] s s s s s s s s s s s s s s s s s s s s s s s s s s s c c c
## [31] c c c c c c c v c c c c v c c c c c c c c c c c v v v v v
## [61] v v v v v v v v v v v v v v v
(z1 <- update(z, . ~ . - Petal.W.))
```

`ldahist` *Histograms or Density Plots of Multiple Groups*

Description

Plot histograms or density plots of data on a single Fisher linear discriminant.

Usage

```
ldahist(data, g, nbins = 25, h, x0 = - h/1000, breaks,
        xlim = range(breaks), ymax = 0, width,
        type = c("histogram", "density", "both"),
        sep = (type != "density"),
        col = 5, xlab = deparse(substitute(data)), bty = "n", ...)
```

Arguments

`data`	vector of data. Missing values (`NA`s) are allowed and omitted.
`g`	factor or vector giving groups, of the same length as `data`.
`nbins`	Suggested number of bins to cover the whole range of the data.
`h`	The bin width (takes precedence over `nbins`).
`x0`	Shift for the bins - the breaks are at `x0 + h * (..., -1, 0, 1, ...)`
`breaks`	The set of breakpoints to be used. (Usually omitted, takes precedence over `h` and `nbins`).
`xlim`	The limits for the x-axis.
`ymax`	The upper limit for the y-axis.
`width`	Bandwidth for density estimates. If missing, the Sheather-Jones selector is used for each group separately.
`type`	Type of plot.
`sep`	Whether there is a separate plot for each group, or one combined plot.
`col`	The colour number for the bar fill.
`xlab`	label for the plot x-axis. By default, this will be the name of `data`.
`bty`	The box type for the plot - defaults to none.
`...`	additional arguments to `polygon`.

Side Effects

Histogram and/or density plots are plotted on the current device.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`plot.lda`.

`leuk` *Survival Times and White Blood Counts for Leukaemia Patients*

Description

A data frame of data from 33 leukaemia patients.

Usage

```
leuk
```

Format

A data frame with columns:

`wbc` white blood count

`ag` a test result, `"present"` or `"absent"`

`time` survival time in weeks

Details

Survival times are given for 33 patients who died from acute myelogenous leukaemia. Also measured was the patient's white blood cell count at the time of diagnosis. The patients were also factored into 2 groups according to the presence or absence of a morphologic characteristic of white blood cells. Patients termed AG positive were identified by the presence of Auer rods and/or significant granulation of the leukaemic cells in the bone marrow at the time of diagnosis.

Source

Cox, D. R. and Oakes, D. (1984) *Analysis of Survival Data*. Chapman & Hall, p. 9.

Taken from

Feigl, P. & Zelen, M. (1965) Estimation of exponential survival probabilities with concomitant information. *Biometrics* **21**, 826–838.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*. Fourth edition. Springer.

Examples

```
library(survival)
plot(survfit(Surv(time) ~ ag, data = leuk), lty = 2:3, col = 2:3)

# now Cox models
leuk.cox <- coxph(Surv(time) ~ ag + log(wbc), leuk)
summary(leuk.cox)
```

`lm.gls` *Fit Linear Models by Generalized Least Squares*

Description

Fit linear models by Generalized Least Squares

Usage

```
lm.gls(formula, data, W, subset, na.action, inverse = FALSE,
       method = "qr", model = FALSE, x = FALSE, y = FALSE,
       contrasts = NULL, ...)
```

Arguments

`formula`	a formula expression as for regression models, of the form `response ~ predictors`. See the documentation of `formula` for other details.
`data`	an optional data frame in which to interpret the variables occurring in `formula`.
`W`	a weight matrix.
`subset`	expression saying which subset of the rows of the data should be used in the fit. All observations are included by default.
`na.action`	a function to filter missing data.
`inverse`	logical: if true `W` specifies the inverse of the weight matrix: this is appropriate if a variance matrix is used.
`method`	method to be used by `lm.fit`.
`model`	should the model frame be returned?
`x`	should the design matrix be returned?
`y`	should the response be returned?
`contrasts`	a list of contrasts to be used for some or all of
`...`	additional arguments to `lm.fit`.

Details

The problem is transformed to uncorrelated form and passed to `lm.fit`.

Value

An object of class `"lm.gls"`, which is similar to an `"lm"` object. There is no `"weights"` component, and only a few `"lm"` methods will work correctly. As from version 7.1-22 the residuals and fitted values refer to the untransformed problem.

See Also

`gls`, `lm`, `lm.ridge`

`lm.ridge` *Ridge Regression*

Description

Fit a linear model by ridge regression.

Usage

```
lm.ridge(formula, data, subset, na.action, lambda = 0, model = FALSE,
         x = FALSE, y = FALSE, contrasts = NULL, ...)
```

Arguments

`formula`	a formula expression as for regression models, of the form `response ~ predictors`. See the documentation of `formula` for other details. `offset` terms are allowed.
`data`	an optional data frame in which to interpret the variables occurring in `formula`.
`subset`	expression saying which subset of the rows of the data should be used in the fit. All observations are included by default.
`na.action`	a function to filter missing data.
`lambda`	A scalar or vector of ridge constants.
`model`	should the model frame be returned?
`x`	should the design matrix be returned?
`y`	should the response be returned?
`contrasts`	a list of contrasts to be used for some or all of factor terms in the formula. See the `contrasts.arg` of `model.matrix.default`.
`...`	additional arguments to `lm.fit`.

Details

If an intercept is present in the model, its coefficient is not penalized. (If you want to penalized an intercept, put in your own constant term and remove the intercept.)

Value

A list with components

`coef`	matrix of coefficients, one row for each value of `lambda`. Note that these are not on the original scale and are for use by the `coef` method.
`scales`	scalings used on the X matrix.
`Inter`	was intercept included?
`lambda`	vector of lambda values
`ym`	mean of `y`
`xm`	column means of `x` matrix
`GCV`	vector of GCV values
`kHKB`	HKB estimate of the ridge constant.
`kLW`	L-W estimate of the ridge constant.

References

Brown, P. J. (1994) *Measurement, Regression and Calibration* Oxford.

See Also

lm

Examples

```
data(longley) # not the same as the S-PLUS dataset
names(longley)[1] <- "y"
lm.ridge(y ~ ., longley)
plot(lm.ridge(y ~ ., longley,
              lambda = seq(0,0.1,0.001)))
select(lm.ridge(y ~ ., longley,
               lambda = seq(0,0.1,0.0001)))
```

---

loglm *Fit Log-Linear Models by Iterative Proportional Scaling*

Description

This function provides a front-end to the standard function, `loglin`, to allow log-linear models to be specified and fitted in a manner similar to that of other fitting functions, such as `glm`.

Usage

```
loglm(formula, data, subset, na.action, ...)
```

Arguments

`formula`	A linear model formula specifying the log-linear model. If the left-hand side is empty, the `data` argument is required and must be a (complete) array of frequencies. In this case the variables on the right-hand side may be the names of the `dimnames` attribute of the frequency array, or may be the positive integers: 1, 2, 3, ... used as alternative names for the 1st, 2nd, 3rd, ... dimension (classifying factor). If the left-hand side is not empty it specifies a vector of frequencies. In this case the data argument, if present, must be a data frame from which the left-hand side vector and the classifying factors on the right-hand side are (preferentially) obtained. The usual abbreviation of a `.` to stand for 'all other variables in the data frame' is allowed. Any non-factors on the right-hand side of the formula are coerced to factor.
`data`	Numeric array or data frame. In the first case it specifies the array of frequencies; in then second it provides the data frame from which the variables occurring in the formula are preferentially obtained in the usual way. This argument may be the result of a call to `xtabs`.
`subset`	Specifies a subset of the rows in the data frame to be used. The default is to take all rows.
`na.action`	Specifies a method for handling missing observations. The default is to fail if missing values are present.
`...`	May supply other arguments to the function `loglm1`.

Details

If the left-hand side of the formula is empty the `data` argument supplies the frequency array and the right-hand side of the formula is used to construct the list of fixed faces as required by `loglin`. Structural zeros may be specified by giving a `start` argument with those entries set to zero, as described in the help information for `loglin`.

If the left-hand side is not empty, all variables on the right-hand side are regarded as classifying factors and an array of frequencies is constructed. If some cells in the complete array are not specified they are treated as structural zeros. The right-hand side of the formula is again used to construct the list of faces on which the observed and fitted totals must agree, as required by `loglin`. Hence terms such as `a:b`, `a*b` and `a/b` are all equivalent.

Value

An object of class `"loglm"` conveying the results of the fitted log-linear model. Methods exist for the generic functions `print`, `summary`, `deviance`, `fitted`, `coef`, `resid`, `anova` and `update`, which perform the expected tasks. Only log-likelihood ratio tests are allowed using `anova`.

The deviance is simply an alternative name for the log-likelihood ratio statistic for testing the current model within a saturated model, in accordance with standard usage in generalized linear models.

Warning

If structural zeros are present, the calculation of degrees of freedom may not be correct. `loglin` itself takes no action to allow for structural zeros. `loglm` deducts one degree of freedom for each structural zero, but cannot make allowance for gains in error degrees of freedom due to loss of dimension in the model space. (This would require checking the rank of the model matrix, but since iterative proportional scaling methods are developed largely to avoid constructing the model matrix explicitly, the computation is at least difficult.)

When structural zeros (or zero fitted values) are present the estimated coefficients will not be available due to infinite estimates. The deviances will normally continue to be correct, though.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`loglm1`, `loglin`

Examples

```
# The data frames  Cars93, minn38 and quine are available
# in the MASS package.

# Case 1: frequencies specified as an array.
sapply(minn38, function(x) length(levels(x)))
## hs phs fol sex f
##  3   4   7   2 0
minn38a <- array(0, c(3,4,7,2), lapply(minn38[, -5], levels))
minn38a[data.matrix(minn38[,-5])] <- minn38$f
fm <- loglm(~1 + 2 + 3 + 4, minn38a)  # numerals as names.
deviance(fm)
##[1] 3711.9
```

```
fm1 <- update(fm, .~.^2)
fm2 <- update(fm, .~.^3, print = TRUE)
## 5 iterations: deviation 0.0750732
anova(fm, fm1, fm2)
## Not run: LR tests for hierarchical log-linear models

Model 1:
  ~  1 + 2 + 3 + 4
Model 2:
 .  ~  1 + 2 + 3 + 4 + 1:2 + 1:3 + 1:4 + 2:3 + 2:4 + 3:4
Model 3:
 .  ~  1 + 2 + 3 + 4 + 1:2 + 1:3 + 1:4 + 2:3 + 2:4 + 3:4 +
       1:2:3 + 1:2:4 + 1:3:4 + 2:3:4

          Deviance  df Delta(Dev) Delta(df) P(> Delta(Dev)
  Model 1 3711.915 155
  Model 2  220.043 108   3491.873        47        0.00000
  Model 3   47.745  36    172.298        72        0.00000
Saturated    0.000   0     47.745        36        0.09114

## End(Not run)
# Case 1. An array generated with xtabs.

loglm(~ Type + Origin, xtabs(~ Type + Origin, Cars93))
## Not run: Call:
loglm(formula = ~Type + Origin, data = xtabs(~Type + Origin,
    Cars93))

Statistics:
                     X^2 df  P(> X^2)
Likelihood Ratio 18.362  5 0.0025255
         Pearson 14.080  5 0.0151101

## End(Not run)
# Case 2.  Frequencies given as a vector in a data frame
names(quine)
## [1] "Eth"  "Sex"  "Age"  "Lrn"  "Days"
fm <- loglm(Days ~ .^2, quine)
gm <- glm(Days ~ .^2, poisson, quine)  # check glm.
c(deviance(fm), deviance(gm))          # deviances agree
## [1] 1368.7 1368.7
c(fm$df, gm$df)                        # resid df do not!
c(fm$df, gm$df.residual)               # resid df do not!
## [1] 127 128
# The loglm residual degrees of freedom is wrong because of
# a non-detectable redundancy in the model matrix.
```

`logtrans` *Estimate log Transformation Parameter*

Description

Find and optionally plot the marginal (profile) likelihood for alpha for a transformation model of the form `log(y + alpha) ~ x1 + x2 + ...`.

Usage

```
logtrans(object, ...)

## Default S3 method:
logtrans(object, ..., alpha = seq(0.5, 6, by = 0.25) - min(y),
         plotit = TRUE, interp =, xlab = "alpha",
         ylab = "log Likelihood")

## S3 method for class 'formula':
logtrans(object, data, ...)

## S3 method for class 'lm':
logtrans(object, ...)
```

Arguments

`object`	Fitted linear model object, or formula defining the untransformed model that is `y ~ x1 + x2 + ...`. The function is generic.
`...`	If `object` is a formula, this argument may specify a data frame as for `lm`.
`alpha`	Set of values for the transformation parameter, alpha.
`plotit`	Should plotting be done?
`interp`	Should the marginal log-likelihood be interpolated with a spline approximation? (Default is `TRUE` if plotting is to be done and the number of real points is less than 100.)
`xlab`	as for `plot`.
`ylab`	as for `plot`.
`data`	optional `data` argument for `lm` fit.

Value

List with components `x` (for alpha) and `y` (for the marginal log-likelihood values).

Side Effects

A plot of the marginal log-likelihood is produced, if requested, together with an approximate mle and 95% confidence interval.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`boxcox`

Examples

```
logtrans(Days ~ Age*Sex*Eth*Lrn, data = quine,
         alpha = seq(0.75, 6.5, len=20))
```

`lqs` *Resistant Regression*

Description

Fit a regression to the *good* points in the dataset, thereby achieving a regression estimator with a high breakdown point. `lmsreg` and `ltsreg` are compatibility wrappers.

Usage

```
lqs(x, ...)

## S3 method for class 'formula':
lqs(formula, data, ...,
    method = c("lts", "lqs", "lms", "S", "model.frame"),
    subset, na.action, model = TRUE,
    x.ret = FALSE, y.ret = FALSE, contrasts = NULL)

## Default S3 method:
lqs(x, y, intercept = TRUE, method = c("lts", "lqs", "lms", "S"),
    quantile, control = lqs.control(...), k0 = 1.548, seed, ...)

lmsreg(...)
ltsreg(...)
```

Arguments

`formula`	a formula of the form `y ~ x1 + x2 + ...`.
`data`	data frame from which variables specified in `formula` are preferentially to be taken.
`subset`	an index vector specifying the cases to be used in fitting. (NOTE: If given, this argument must be named exactly.)
`na.action`	function to specify the action to be taken if `NA`s are found. The default action is for the procedure to fail. Alternatives include `na.omit` and `na.exclude`, which lead to omission of cases with missing values on any required variable. (NOTE: If given, this argument must be named exactly.)
`model, x.ret, y.ret`	logical. If `TRUE` the model frame, the model matrix and the response are returned, respectively.
`contrasts`	an optional list. See the `contrasts.arg` of `model.matrix.default`.
`x`	a matrix or data frame containing the explanatory variables.
`y`	the response: a vector of length the number of rows of `x`.
`intercept`	should the model include an intercept?
`method`	the method to be used. `model.frame` returns the model frame: for the others see the `Details` section. Using `lmsreg` or `ltsreg` forces `"lms"` and `"lts"` respectively.
`quantile`	the quantile to be used: see `Details`. This is over-ridden if `method = "lms"`.

`control`	additional control items: see `Details`.
`k0`	the cutoff / tuning constant used for $\chi()$ and $\psi()$ functions when `method = "S"`, currently corresponding to Tukey's "biweight".
`seed`	the seed to be used for random sampling: see `.Random.seed`. The current value of `.Random.seed` will be preserved if it is set..
`...`	arguments to be passed to `lqs.default` or `lqs.control`, see `control` above and `Details`.

Details

Suppose there are `n` data points and `p` regressors, including any intercept.

The first three methods minimize some function of the sorted squared residuals. For methods `"lqs"` and `"lms"` is the `quantile` squared residual, and for `"lts"` it is the sum of the `quantile` smallest squared residuals. `"lqs"` and `"lms"` differ in the defaults for `quantile`, which are `floor((n+p+1)/2)` and `floor((n+1)/2)` respectively. For `"lts"` the default is `floor(n/2) + floor((p+1)/2)`.

The `"S"` estimation method solves for the scale `s` such that the average of a function chi of the residuals divided by `s` is equal to a given constant.

The `control` argument is a list with components

`psamp`: the size of each sample. Defaults to `p`.

`nsamp`: the number of samples or `"best"` (the default) or `"exact"` or `"sample"`. If `"sample"` the number chosen is `min(5*p, 3000)`, taken from Rousseeuw and Hubert (1997). If `"best"` exhaustive enumeration is done up to 5000 samples; if `"exact"` exhaustive enumeration will be attempted however many samples are needed.

`adjust`: should the intercept be optimized for each sample? Defaults to `TRUE`.

Value

An object of class `"lqs"`. This is a list with components

`crit`	the value of the criterion for the best solution found, in the case of `method == "S"` before IWLS refinement.
`sing`	character. A message about the number of samples which resulted in singular fits.
`coefficients`	of the fitted linear model
`bestone`	the indices of those points fitted by the best sample found (prior to adjustment of the intercept, if requested).
`fitted.values`	the fitted values.
`residuals`	the residuals.
`scale`	estimate(s) of the scale of the error. The first is based on the fit criterion. The second (not present for `method == "S"`) is based on the variance of those residuals whose absolute value is less than 2.5 times the initial estimate.

Note

There seems no reason other than historical to use the `lms` and `lqs` options. LMS estimation is of low efficiency (converging at rate $n^{-1/3}$) whereas LTS has the same asymptotic efficiency as an M estimator with trimming at the quartiles (Marazzi, 1993, p.201). LQS and LTS have the same maximal breakdown value of `(floor((n-p)/2) + 1)/n` attained if `floor((n+p)/2) <= quantile <= floor((n+p+1)/2)`. The only drawback mentioned of LTS is greater computation, as a sort was thought to be required (Marazzi, 1993, p.201) but this is not true as a partial sort can be used (and is used in this implementation).

Adjusting the intercept for each trial fit does need the residuals to be sorted, and may be significant extra computation if `n` is large and `p` small.

Opinions differ over the choice of `psamp`. Rousseeuw and Hubert (1997) only consider p; Marazzi (1993) recommends p+1 and suggests that more samples are better than adjustment for a given computational limit.

The computations are exact for a model with just an intercept and adjustment, and for LQS for a model with an intercept plus one regressor and exhaustive search with adjustment. For all other cases the minimization is only known to be approximate.

Author(s)

B. D. Ripley

References

P. J. Rousseeuw and A. M. Leroy (1987) *Robust Regression and Outlier Detection.* Wiley.

A. Marazzi (1993) *Algorithms, Routines and S Functions for Robust Statistics.* Wadsworth and Brooks/Cole.

P. Rousseeuw and M. Hubert (1997) Recent developments in PROGRESS. In *L1-Statistical Procedures and Related Topics*, ed Y. Dodge, IMS Lecture Notes volume **31**, pp. 201–214.

See Also

`predict.lqs`

Examples

```
data(stackloss)
set.seed(123)
lqs(stack.loss ~ ., data = stackloss)
lqs(stack.loss ~ ., data = stackloss, method = "S", nsamp = "exact")
```

---

mammals — *Brain and Body Weights for 62 Species of Land Mammals*

Description

A data frame with average brain and body weights for 62 species of land mammals.

Usage

```
mammals
```

Format

body body weight in kg

brain brain weight in g

name Common name of species. Rock hyrax-a = *Heterohyrax brucci.* Rock hyrax-b = *Procavia habessinica.*

Source

Weisberg, S. (1985) *Applied Linear Regression.* 2nd edition. Wiley, pp. 144–5.

Selected from: Allison, T. and Cicchetti, D. V. (1976) Sleep in mammals: ecological and constitutional correlates. *Science* **194**, 732–734.

References

Venables, W. N. and Ripley, B. D. (1999) *Modern Applied Statistics with S-PLUS.* Third Edition. Springer.

---

mca *Multiple Correspondence Analysis*

Description

Computes a multiple correspondence analysis of a set of factors.

Usage

```
mca(df, nf = 2, abbrev = FALSE)
```

Arguments

df	A data frame containing only factors
nf	The number of dimensions for the MCA. Rarely 3 might be useful.
abbrev	Should the vertex names be abbreviated? By default these are of the form "factor.level" but if `abbrev = TRUE` they are just "level" which will suffice if the factors have distinct levels.

Value

An object of class `"mca"`, with components

rs	The coordinates of the rows, in `nf` dimensions.
cs	The coordinates of the column vertices, one for each level of each factor.
fs	Weights for each row, used to interpolate additional factors in `predict.mca`.
p	The number of factors
d	The singular values for the `nf` dimensions.
call	The matched call.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`predict.mca`, `plot.mca`, `corresp`

Examples

```
farms.mca <- mca(farms, abbrev=TRUE)
farms.mca
plot(farms.mca)
```

---

mcycle *Data from a Simulated Motorcycle Accident*

Description

A data frame giving a series of measurements of head acceleration in a simulated motorcycle accident, used to test crash helmets.

Usage

```
mcycle
```

Format

`times` in milliseconds after impact

`accel` in g

Source

Silverman, B. W. (1985) Some aspects of the spline smoothing approach to non-parametric curve fitting. *Journal of the Royal Statistical Society series B* **47**, 1–52.

References

Venables, W. N. and Ripley, B. D. (1999) *Modern Applied Statistics with S-PLUS.* Third Edition. Springer.

Melanoma *Survival from Malignant Melanoma*

Description

The Melanoma data frame has data on 205 patients in Denmark with malignant melanoma.

Usage

```
Melanoma
```

Format

This data frame contains the following columns:

time survival time in days, possibly censored

status 1 died from melanoma, 2 alive, 3 dead from other causes

sex 1 = male, 2 = female

age age in years

year of operation

thickness tumour thickness in mm

ulcer 1 = presence, 0 = absence

Source

P. K. Andersen, O. Borgan, R. D. Gill, and N. Keiding (1993) *Statistical Models based on Counting Processes.* Springer.

menarche *Age of Menarche data*

Description

Proportions of female children at various ages during adolescence who have reached menarche.

Usage

```
menarche
```

Format

This data frame contains the following columns:

Age Average age of the group. (The groups are reasonably age homogeneous.)

Total Total number of children in the group.

Menarche Number who have reached menarche.

Source

Milicer, H. and Szczotka, F. (1966) Age at Menarche in Warsaw girls in 1965. *Human Biology* **38**, 199–203.

The data are also given in
Aranda-Ordaz, F.J. (1981) On two families of transformations to additivity for binary response data. *Biometrika* **68**, 357–363.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

Examples

```
mprob <- glm(cbind(Menarche, Total - Menarche) ~ Age,
             binomial(link = probit), data = menarche)
```

---

michelson *Michelson's Speed of Light Data*

Description

Measurements of the speed of light in air, made between 5th June and 2nd July, 1879. The data consists of five experiments, each consisting of 20 consecutive runs. The response is the speed of light in km/s, less 299000. The currently accepted value, on this scale of measurement, is 734.5.

Usage

```
michelson
```

Format

The data frame contains the following components:

`Expt` The experiment number, from 1 to 5

`Run` The run number within each experiment

`Speed` Speed-of-light measurement

Source

A.J. Weekes (1986) *A Genstat Primer.* Edward Arnold.

S. M. Stigler (1977) Do robust estimators work with real data? *Annals of Statistics* **5**, 1055–1098.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

minn38 *Minnesota High School Graduates of 1938*

Description

The Minnesota high school graduates of 1938 were classified according to four factors, described below. The `minn38` data frame has 168 rows and 5 columns.

Usage

```
minn38
```

Format

This data frame contains the following columns:

`hs` High school rank: `"L"`, `"M"` and `"U"` for lower, middle and upper third.

`phs` Post high school status: Enrolled in college, (`"C"`), enrolled in non-collegiate school, (`"N"`), employed full-time, (`"E"`) and other, (`"O"`).

`fol` Father's occupational level, (seven levels, `"F1"`, `"F2"`, ..., `"F7"`).

`sex` Sex factor, `"F"` or `"M"`.

`f` Frequency.

Source

Quoted from R. L. Plackett, (1974) *The Analysis of Categorical Data.* London: Griffin

who quotes the data from

Hoyt, C. J., Krishnaiah, P. R. and Torrance, E. P. (1959) Analysis of complex contingency tables, *J. Exp. Ed.* **27**, 187–194.

motors *Accelerated Life Testing of Motorettes*

Description

The `motors` data frame has 40 rows and 3 columns. It describes an accelerated life test at each of four temperatures of 10 motorettes, and has rather discrete times.

Usage

```
motors
```

Format

This data frame contains the following columns:

`temp` the temperature (degrees C) of the test

`time` the time in hours to failure or censoring at 8064 hours (= 336 days).

`cens` an indicator variable for death

Source

Kalbfleisch, J. D. and Prentice, R. L. (1980) *The Statistical Analysis of Failure Time Data.* New York: Wiley.

taken from

Nelson, W. D. and Hahn, G. J. (1972) Linear regression of a regression relationship from censored data. Part 1 – simple methods and their application. *Technometrics*, **14**, 247–276.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

Examples

```
library(survival)
plot(survfit(Surv(time, cens) ~ factor(temp), motors), conf.int = FALSE)
# fit Weibull model
motor.wei <- survreg(Surv(time, cens) ~ temp, motors)
summary(motor.wei)
# and predict at 130C
unlist(predict(motor.wei, data.frame(temp=130), se.fit = TRUE))

motor.cox <- coxph(Surv(time, cens) ~ temp, motors)
summary(motor.cox)
# predict at temperature 200
plot(survfit(motor.cox, newdata = data.frame(temp=200),
     conf.type = "log-log"))
summary( survfit(motor.cox, newdata = data.frame(temp=130)) )
```

`muscle` *Effect of Calcium Chloride on Muscle Contraction in Rat Hearts*

Description

The purpose of this experiment was to assess the influence of calcium in solution on the contraction of heart muscle in rats. The left auricle of 21 rat hearts was isolated and on several occasions a constant-length strip of tissue was electrically stimulated and dipped into various concentrations of calcium chloride solution, after which the shortening of the strip was accurately measured as the response.

Usage

```
muscle
```

Format

This data frame contains the following columns:

`Strip` which heart muscle strip was used?

`Conc` concentration of calcium chloride solution, in multiples of 2.2 mM.

`Length` the change in length (shortening) of the strip, (allegedly) in mm.

Source

Linder, A., Chakravarti, I. M. and Vuagnat, P. (1964) Fitting asymptotic regression curves with different asymptotes. In *Contributions to Statistics. Presented to Professor P. C. Mahalanobis on the occasion of his 70th birthday,* ed. C. R. Rao, pp. 221–228. Oxford: Pergamon Press.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth Edition. Springer.

Examples

```
A <- model.matrix(~ Strip - 1, data=muscle)
rats.nls1 <- nls(log(Length) ~ cbind(A, rho^Conc),
   data = muscle, start = c(rho=0.1), algorithm="plinear")
B <- coef(rats.nls1)
B

st <- list(alpha = B[2:22], beta = B[23], rho = B[1])
(rats.nls2 <- nls(log(Length) ~ alpha[Strip] + beta*rho^Conc,
                  data = muscle, start = st))

attach(muscle)
Muscle <- expand.grid(Conc = sort(unique(Conc)),
                      Strip = levels(Strip))
Muscle$Yhat <- predict(rats.nls2, Muscle)
Muscle <- cbind(Muscle, logLength = rep(as.numeric(NA), 126))
ind <- match(paste(Strip, Conc),
            paste(Muscle$Strip, Muscle$Conc))
Muscle$logLength[ind] <- log(Length)
detach()

require(lattice)
xyplot(Yhat ~ Conc | Strip, Muscle, as.table = TRUE,
   ylim = range(c(Muscle$Yhat, Muscle$logLength), na.rm = TRUE),
   subscripts = TRUE, xlab = "Calcium Chloride concentration (mM)",
   ylab = "log(Length in mm)", panel =
   function(x, y, subscripts, ...) {
      lines(spline(x, y))
      panel.xyplot(x, Muscle$logLength[subscripts], ...)
   })
```

---

mvrnorm *Simulate from a Multivariate Normal Distribution*

Description

Produces one or more samples from the specified multivariate normal distribution.

Usage

```
mvrnorm(n = 1, mu, Sigma, tol = 1e-6, empirical = FALSE)
```

Arguments

n	the number of samples required.
mu	a vector giving the means of the variables.
Sigma	a positive-definite symmetric matrix specifying the covariance matrix of the variables.
tol	tolerance (relative to largest variance) for numerical lack of positive-definiteness in Sigma.
empirical	logical. If true, mu and Sigma specify the empirical not population mean and covariance matrix.

Details

The matrix decomposition is done via eigen; although a Choleski decomposition might be faster, the eigendecomposition is stabler.

Value

If n = 1 a vector of the same length as mu, otherwise an n by length(mu) matrix with one sample in each row.

Side Effects

Causes creation of the dataset .Random.seed if it does not already exist, otherwise its value is updated.

References

B. D. Ripley (1987) *Stochastic Simulation.* Wiley. Page 98.

See Also

rnorm

Examples

```
Sigma <- matrix(c(10,3,3,2),2,2)
Sigma
var(mvrnorm(n=1000, rep(0, 2), Sigma))
var(mvrnorm(n=1000, rep(0, 2), Sigma, empirical = TRUE))
```

---

negative.binomial *Family function for Negative Binomial GLMs*

Description

Specifies the information required to fit a Negative Binomial generalized linear model, with known theta parameter, using glm().

Usage

```
negative.binomial(theta = stop("'theta' must be specified"), link = "log")
```

Arguments

theta	The known value of the additional parameter, theta.
link	The link function, as a character string, name or one-element character vector specifying one of log, sqrt or identity, or an object of class "link-glm".

Value

An object of class "family", a list of functions and expressions needed by glm() to fit a Negative Binomial generalized linear model.

References

Venables, W. N. and Ripley, B. D. (1999) *Modern Applied Statistics with S-PLUS.* Third Edition. Springer.

See Also

glm.nb, anova.negbin, summary.negbin

Examples

```
# Fitting a Negative Binomial model to the quine data
#   with theta = 2 assumed known.
#
glm(Days ~ .^4, family = negative.binomial(2), data = quine)
```

---

newcomb *Newcomb's Measurements of the Passage Time of Light*

Description

A numeric vector giving the Third Series of measurements of the passage time of light recorded by Newcomb in 1882. The given values divided by 1000 plus 24 give the time in millionths of a second for light to traverse a known distance. The "true" value is now considered to be 33.02.

Usage

```
newcomb
```

Source

S. M. Stigler (1973) Simon Newcombe, Percy Daniell, and the history of robust estimation 1885–1920. *Journal of the American Statistical Association* **68**, 872–879

R. G. Staudte and S. J. Sheather (1990) *Robust Estimation and Testing.* Wiley

`nlschools` *Eighth-Grade Pupils in the Netherlands*

Description

Snijders and Bosker (1999) use as a running example a study of 2287 eighth-grade pupils (aged about 11) in 132 classes in 131 schools in the Netherlands. Only the variables used in our examples are supplied.

Usage

```
nlschools
```

Format

This data frame contains 2287 rows and the following columns:

`lang` language test score

`IQ` Verbal IQ

`class` class ID

`GS` Class size: number of eighth-grade pupils recorded in the class (there may be others: see `COMB`, and some may have been omitted with missing values).

`SES` Social-economic status of pupil's family.

`COMB` were the pupils taught in a multi-grade class (`0/1`)? Classes which contained pupils from grades 7 and 8 are coded `1`, but only eighth-graders were tested.

Source

Snijders, T. A. B. and Bosker, R. J. (1999) *Multilevel Analysis. An Introduction to Basic and Advanced Multilevel Modelling.* London: Sage.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

Examples

```
library(nlme)
nl1 <- nlschools
attach(nl1)
classMeans <- tapply(IQ, class, mean)
nl1$IQave <- classMeans[as.character(class)]
nl1$IQ <- nl1$IQ - nl1$IQave
detach()
cen <- c("IQ", "IQave", "SES")
nl1[cen] <- scale(nl1[cen], center = TRUE, scale = FALSE)

nl.lme <- lme(lang ~ IQ*COMB + IQave + SES,
              random = ~ IQ | class, data = nl1)
summary(nl.lme)
```

npk *Classical N, P, K Factorial Experiment*

Description

A classical N, P, K (nitrogen, phosphate, potassium) factorial experiment on the growth of peas conducted on 6 blocks. Each half of a fractional factorial design confounding the NPK interaction was used on 3 of the plots.

Usage

```
npk
```

Format

The `npk` data frame has 24 rows and 5 columns:

`block` which block (label 1 to 6).

`N` indicator (0/1) for the application of nitrogen.

`P` indicator (0/1) for the application of phosphate.

`K` indicator (0/1) for the application of potassium.

`yield` Yield of peas, in pounds/plot (the plots were (1/70) acre).

Source

Imperial College, London, M.Sc. exercise sheet.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

Examples

```
options(contrasts = c("contr.sum", "contr.poly"))
npk.aov <- aov(yield ~ block + N*P*K, npk)
npk.aov
summary(npk.aov)
alias(npk.aov)
coef(npk.aov)
options(contrasts = c("contr.treatment", "contr.poly"))
npk.aov1 <- aov(yield ~ block + N + K, data = npk)
summary.lm(npk.aov1)
se.contrast(npk.aov1, list(N=="0", N=="1"), data = npk)
model.tables(npk.aov1, type = "means", se = TRUE)
```

`npr1` *US Naval Petroleum Reserve No. 1 data*

Description

Data on the locations, porosity and permeability (a measure of oil flow) on 104 oil wells in the US Naval Petroleum Reserve No. 1 in California.

Usage

```
npr1
```

Format

This data frame contains the following columns:

`x, y` x and y coordinates, in miles from an arbitrary origin

`perm` permeability in milli-Darcies

`por` porosity (%)

Source

Maher, J.C., Carter, R.D. and Lantz, R.J. (1975) Petroleum geology of Naval Petroleum Reserve No. 1, Elk Hills, Kern County, California. *USGS Professional Paper* **912**.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

`Null` *Null Spaces of Matrices*

Description

Given a matrix, `M`, find a matrix `N` giving a basis for the null space. That is `t(N) %*% M` is the zero and `N` has the maximum number of linearly independent columns.

Usage

```
Null(M)
```

Arguments

`M`	Input matrix. A vector is coerced to a 1-column matrix.

Value

The matrix `N` with the basis for the null space, or an empty vector if the matrix `M` is square and of maximal rank.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

qr, qr.Q

Examples

```
# The function is currently defined as
function(M)
{
        tmp <- qr(M)
        set <- if(tmp$rank == 0) 1:ncol(M) else  - (1:tmp$rank)
        qr.Q(tmp, complete = TRUE)[, set, drop = FALSE]
}
```

---

oats *Data from an Oats Field Trial*

Description

The yield of oats from a split-plot field trial using three varieties and four levels of manurial treatment. The experiment was laid out in 6 blocks of 3 main plots, each split into 4 sub-plots. The varieties were applied to the main plots and the manurial treatments to the sub-plots.

Usage

```
oats
```

Format

This data frame contains the following columns:

B Blocks, levels I, II, III, IV, V and VI

V Varieties, 3 levels.

N Nitrogen (manurial) treatment, levels 0.0cwt, 0.2cwt, 0.4cwt and 0.6cwt, showing the application in cwt/acre.

Y Yields in 1/4lbs per sub-plot, each of area 1/80 acre.

Source

Yates, F. (1935) Complex experiments, *Journal of the Royal Statistical Society Suppl.* **2**, 181–247.

Also given in Yates, F. (1970) *Experimental design: Selected papers of Frank Yates, C.B.E, F.R.S.* London: Griffin.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

Examples

```
oats$Nf <- ordered(oats$N, levels = sort(levels(oats$N)))
oats.aov <- aov(Y ~ Nf*V + Error(B/V), data = oats, qr = TRUE)
summary(oats.aov)
summary(oats.aov, split = list(Nf=list(L=1, Dev=2:3)))
par(mfrow = c(1,2), pty = "s")
plot(fitted(oats.aov[[4]]), studres(oats.aov[[4]]))
abline(h = 0, lty = 2)
oats.pr <- proj(oats.aov)
qqnorm(oats.pr[[4]][,"Residuals"], ylab = "Stratum 4 residuals")
qqline(oats.pr[[4]][,"Residuals"])

par(mfrow = c(1,1), pty = "m")
oats.aov2 <- aov(Y ~ N + V + Error(B/V), data = oats, qr = TRUE)
model.tables(oats.aov2, type = "means", se = TRUE)
```

OME *Tests of Auditory Perception in Children with OME*

Description

Experiments were performed on children on their ability to differentiate a signal in broad-band noise. The noise was played from a pair of speakers and a signal was added to just one channel; the subject had to turn his/her head to the channel with the added signal. The signal was either coherent (the amplitude of the noise was increased for a period) or incoherent (independent noise was added for the same period to form the same increase in power).

The threshold used in the original analysis was the stimulus loudness needs to get 75% correct responses. Some of the children had suffered from otitis media with effusion (OME).

Usage

```
OME
```

Format

The `OME` data frame has 1129 rows and 7 columns:

`ID` Subject ID (1 to 99, with some IDs missing). A few subjects were measured at different ages.

`OME` `"low"` or `"high"` or `"N/A"` (at ages other than 30 and 60 months).

`Age` Age of the subject (months).

`Loud` Loudness of stimulus, in decibels.

`Noise` Whether the signal in the stimulus was `"coherent"` or `"incoherent"`.

`Correct` Number of correct responses from `Trials` trials.

`Trials` Number of trials performed.

Background

The experiment was to study otitis media with effusion (OME), a very common childhood condition where the middle ear space, which is normally air-filled, becomes congested by a fluid. There is a concomitant fluctuating, conductive hearing loss which can result in various language, cognitive and social deficits. The term "binaural hearing" is used to describe the listening conditions in which the brain is processing information from both ears at the same time. The brain computes differences in the intensity and/or timing of signals arriving at each ear which contributes to sound localisation and also to our ability to hear in background noise.

Some years ago, it was found that children of 7-8 years with a history of significant OME had significantly worse binaural hearing than children without such a history, despite having equivalent sensitivity. The question remained as to whether it was the timing, the duration, or the degree of severity of the otitis media episodes during critical periods, which affected later binaural hearing. In an attempt to begin to answer this question, 95 children were monitored for the presence of effusion every month since birth. On the basis of OME experience in their first two years, the test population was split into one group of high OME prevalence and one of low prevalence.

Source

Sarah Hogan, Dept of Physiology, University of Oxford, via Dept of Statistics Consulting Service

Examples

```
# Fit logistic curve from p = 0.5 to p = 1.0
fp1 <- deriv(~ 0.5 + 0.5/(1 + exp(-(x-L75)/scal)),
             c("L75", "scal"),
             function(x,L75,scal)NULL)
nls(Correct/Trials ~ fp1(Loud, L75, scal), data = OME,
    start = c(L75=45, scal=3))
nls(Correct/Trials ~ fp1(Loud, L75, scal),
    data = OME[OME$Noise == "coherent",],
    start=c(L75=45, scal=3))
nls(Correct/Trials ~ fp1(Loud, L75, scal),
    data = OME[OME$Noise == "incoherent",],
    start = c(L75=45, scal=3))

# individual fits for each experiment

aa <- factor(OME$Age)
ab <- 10*OME$ID + unclass(aa)
ac <- unclass(factor(ab))
OME$UID <- as.vector(ac)
OME$UIDn <- OME$UID + 0.1*(OME$Noise == "incoherent")
rm(aa, ab, ac)
OMEi <- OME

library(nlme)
fp2 <- deriv(~ 0.5 + 0.5/(1 + exp(-(x-L75)/2)),
            "L75", function(x,L75) NULL)
options(show.error.messages = FALSE)
OMEi.nls <- nlsList(Correct/Trials ~ fp2(Loud, L75) | UIDn,
   data = OMEi, start = list(L75=45), control = list(maxiter=100))
options(show.error.messages = TRUE)
tmp <- sapply(OMEi.nls, function(X)
              {if(is.null(X)) NA else as.vector(coef(X))})
OMEif <- data.frame(UID = round(as.numeric((names(tmp)))),
```

```
         Noise = rep(c("coherent", "incoherent"), 110),
         L75 = as.vector(tmp), stringsAsFactors = TRUE)
OMEif$Age <- OME$Age[match(OMEif$UID, OME$UID)]
OMEif$OME <- OME$OME[match(OMEif$UID, OME$UID)]
OMEif <- OMEif[OMEif$L75 > 30,]
summary(lm(L75 ~ Noise/Age, data = OMEif, na.action = na.omit))
summary(lm(L75 ~ Noise/(Age + OME), data = OMEif,
           subset = (Age >= 30 & Age <= 60),
           na.action = na.omit), cor = FALSE)

# Or fit by weighted least squares
fpl75 <- deriv(~ sqrt(n)*(r/n - 0.5 - 0.5/(1 + exp(-(x-L75)/scal))),
               c("L75", "scal"),
               function(r,n,x,L75,scal) NULL)
nls(0 ~ fpl75(Correct, Trials, Loud, L75, scal),
    data = OME[OME$Noise == "coherent",],
    start = c(L75=45, scal=3))
nls(0 ~ fpl75(Correct, Trials, Loud, L75, scal),
    data = OME[OME$Noise == "incoherent",],
    start = c(L75=45, scal=3))

# Test to see if the curves shift with age
fpl75age <- deriv(~sqrt(n)*(r/n -  0.5 - 0.5/(1 +
                 exp(-(x-L75-slope*age)/scal))),
                 c("L75", "slope", "scal"),
                 function(r,n,x,age,L75,slope,scal) NULL)
OME.nls1 <-
nls(0 ~ fpl75age(Correct, Trials, Loud, Age, L75, slope, scal),
    data = OME[OME$Noise == "coherent",],
    start = c(L75=45, slope=0, scal=2))
sqrt(diag(vcov(OME.nls1)))

OME.nls2 <-
nls(0 ~ fpl75age(Correct, Trials, Loud, Age, L75, slope, scal),
    data = OME[OME$Noise == "incoherent",],
    start = c(L75=45, slope=0, scal=2))
sqrt(diag(vcov(OME.nls2)))

# Now allow random effects by using NLME
OMEf <- OME[rep(1:nrow(OME), OME$Trials),]
attach(OME)
OMEf$Resp <- rep(rep(c(1,0), length(Trials)),
                 t(cbind(Correct, Trials-Correct)))
OMEf <- OMEf[, -match(c("Correct", "Trials"), names(OMEf))]
detach("OME")

## Not run: ## this fails in R on some platforms
fp2 <- deriv(~ 0.5 + 0.5/(1 + exp(-(x-L75)/exp(lsc))),
             c("L75", "lsc"),
             function(x, L75, lsc) NULL)
G1.nlme <- nlme(Resp ~ fp2(Loud, L75, lsc),
     fixed = list(L75 ~ Age, lsc ~ 1),
     random = L75 + lsc ~ 1 | UID,
     data = OMEf[OMEf$Noise == "coherent",], method = "ML",
     start = list(fixed=c(L75=c(48.7, -0.03), lsc=0.24)), verbose = TRUE)
summary(G1.nlme)
```

```
G2.nlme <- nlme(Resp ~ fp2(Loud, L75, lsc),
     fixed = list(L75 ~ Age, lsc ~ 1),
     random = L75 + lsc ~ 1 | UID,
     data = OMEf[OMEf$Noise == "incoherent",], method="ML",
     start = list(fixed=c(L75=c(41.5, -0.1), lsc=0)), verbose = TRUE)
summary(G2.nlme)
## End(Not run)
```

`painters` *The Painter's Data of de Piles*

Description

The subjective assessment, on a 0 to 20 integer scale, of 54 classical painters. The painters were assessed on four characteristics: composition, drawing, colour and expression. The data is due to the Eighteenth century art critic, de Piles.

Usage

```
painters
```

Format

The row names of the data frame are the painters. The components are:

`Composition` Composition score

`Drawing` Drawing score

`Colour` Colour score

`Expression` Expression score

`School` The school to which a painter belongs, as indicated by a factor level code as follows: `"A"`: Renaissance; `"B"`: Mannerist; `"C"`: Seicento; `"D"`: Venetian; `"E"`: Lombard; `"F"`: Sixteenth Century; `"G"`: Seventeenth Century; `"H"`: French.

Source

A. J. Weekes (1986) *A Genstat Primer.* Edward Arnold.

M. Davenport and G. Studdert-Kennedy (1972) The statistical analysis of aesthetic judgement: an exploration. *Applied Statistics* **21**, 324–333.

I. T. Jolliffe (1986) *Principal Component Analysis.* Springer.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

`pairs.lda` *Produce Pairwise Scatterplots from an 'lda' Fit*

Description

Pairwise scatterplot of the data on the linear discriminants.

Usage

```
## S3 method for class 'lda':
pairs(x, labels = colnames(x), panel = panel.lda,
      dimen, abbrev = FALSE, ..., cex=0.7, type = c("std", "trellis"))
```

Arguments

`x`	Object of class `"lda"`.
`labels`	vector of character strings for labelling the variables.
`panel`	panel function to plot the data in each panel.
`dimen`	The number of linear discriminants to be used for the plot; if this exceeds the number determined by `x` the smaller value is used.
`abbrev`	whether the group labels are abbreviated on the plots. If `abbrev > 0` this gives `minlength` in the call to `abbreviate`.
`...`	additional arguments for `pairs.default`.
`cex`	graphics parameter `cex` for labels on plots.
`type`	type of plot. The default is in the style of `pairs.default`; the style `"trellis"` uses the Trellis function `splom`.

Details

This function is a method for the generic function `pairs()` for class `"lda"`. It can be invoked by calling `pairs(x)` for an object `x` of the appropriate class, or directly by calling `pairs.lda(x)` regardless of the class of the object.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`pairs`

`parcoord` *Parallel Coordinates Plot*

Description

Parallel coordinates plot

Usage

```
parcoord(x, col = 1, lty = 1, var.label = FALSE, ...)
```

Arguments

`x`	a matrix or data frame who columns represent variables. Missing values are allowed.
`col`	A vector of colours, recycled as necessary for each observation.
`lty`	A vector of line types, recycled as necessary for each observation.
`var.label`	If `TRUE`, each variable's axis is labelled with maximum and minimum values.
`...`	Further graphics parameters which are passed to `matplot`.

Side Effects

a parallel coordinates plots is drawn.

Author(s)

B. D. Ripley. Enhancements based on ideas and code by Fabian Scheipl.

References

Wegman, E. J. (1990) Hyperdimensional data analysis using parallel coordinates. *Journal of the American Statistical Association* **85**, 664–675.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

Examples

```
data(state)
parcoord(state.x77[, c(7, 4, 6, 2, 5, 3)])

data(iris3)
ir <- rbind(iris3[,,1], iris3[,,2], iris3[,,3])
parcoord(log(ir)[, c(3, 4, 2, 1)], col = 1 + (0:149)%/%50)
```

petrol *N. L. Prater's Petrol Refinery Data*

Description

The yield of a petroleum refining process with four covariates. The crude oil appears to come from only 10 distinct samples.

These data were originally used by Prater (1956) to build an estimation equation for the yield of the refining process of crude oil to gasoline.

Usage

```
petrol
```

Format

The variables are as follows

No Crude oil sample identification label. (factor)

SG Specific gravity, degrees API. (Constant within sample.)

VP Vapour pressure in psi. (Constant within sample.)

V10 Volatility of crude; ASTM 10% point. (Constant within sample.)

EP Desired volatility of gasoline. (The end point. Varies within sample.)

Y Yield as a percentage of crude.

Source

N. H. Prater (1956) Estimate gasoline yields from crudes. *Petroleum Refiner* **35**, 236–238.

This dataset is also given in D. J. Hand, F. Daly, K. McConway, D. Lunn and E. Ostrowski (eds) (1994) *A Handbook of Small Data Sets.* Chapman & Hall.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

Examples

```
library(nlme)
Petrol <- petrol
Petrol[, 2:5] <- scale(as.matrix(Petrol[, 2:5]), scale = FALSE)
pet3.lme <- lme(Y ~ SG + VP + V10 + EP,
                random = ~ 1 | No, data = Petrol)
pet3.lme <- update(pet3.lme, method = "ML")
pet4.lme <- update(pet3.lme, fixed = Y ~ V10 + EP)
anova(pet4.lme, pet3.lme)
```

`Pima.tr` *Diabetes in Pima Indian Women*

Description

A population of women who were at least 21 years old, of Pima Indian heritage and living near Phoenix, Arizona, was tested for diabetes according to World Health Organization criteria. The data were collected by the US National Institute of Diabetes and Digestive and Kidney Diseases. We used the 532 complete records after dropping the (mainly missing) data on serum insulin.

Usage

```
Pima.tr
Pima.tr2
Pima.te
```

Format

These data frames contains the following columns:

`npreg` number of pregnancies

`glu` plasma glucose concentration in an oral glucose tolerance test

`bp` diastolic blood pressure (mm Hg)

`skin` triceps skin fold thickness (mm)

`bmi` body mass index (weight in kg/(height in m)^2)

`ped` diabetes pedigree function

`age` age in years

`type` `Yes` or `No`, for diabetic according to WHO criteria

Details

The training set `Pima.tr` contains a randomly selected set of 200 subjects, and `Pima.te` contains the remaining 322 subjects. `Pima.tr2` contains `Pima.tr` plus 100 subjects with missing values in the explanatory variables.

Source

Smith, J. W., Everhart, J. E., Dickson, W. C., Knowler, W. C. and Johannes, R. S. (1988) Using the ADAP learning algorithm to forecast the onset of *diabetes mellitus*. In *Proceedings of the Symposium on Computer Applications in Medical Care (Washington, 1988),* ed. R. A. Greenes, pp. 261–265. Los Alamitos, CA: IEEE Computer Society Press.

Ripley, B.D. (1996) *Pattern Recognition and Neural Networks.* Cambridge: Cambridge University Press.

`plot.lda` *Plot Method for Class 'lda'*

Description

Plots a set of data on one, two or more linear discriminants.

Usage

```
## S3 method for class 'lda':
plot(x, panel = panel.lda, ..., cex = 0.7, dimen,
     abbrev = FALSE, xlab = "LD1", ylab = "LD2")
```

Arguments

`x`	An object of class `"lda"`.
`panel`	the panel function used to plot the data.
`...`	additional arguments to `pairs`, `ldahist` or `eqscplot`.
`cex`	graphics parameter `cex` for labels on plots.
`dimen`	The number of linear discriminants to be used for the plot; if this exceeds the number determined by `x` the smaller value is used.
`abbrev`	whether the group labels are abbreviated on the plots. If `abbrev > 0` this gives `minlength` in the call to `abbreviate`.
`xlab`	label for the x axis
`ylab`	label for the y axis

Details

This function is a method for the generic function `plot()` for class `"lda"`. It can be invoked by calling `plot(x)` for an object `x` of the appropriate class, or directly by calling `plot.lda(x)` regardless of the class of the object.

The behaviour is determined by the value of `dimen`. For `dimen > 2`, a `pairs` plot is used. For `dimen = 2`, an equiscaled scatter plot is drawn. For `dimen = 1`, a set of histograms or density plots are drawn. Use argument `type` to match `"histogram"` or `"density"` or `"both"`.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`pairs.lda`, `ldahist`, `lda`, `predict.lda`

plot.mca — *Plot Method for Objects of Class 'mca'*

Description

Plot a multiple correspondence analysis.

Usage

```
## S3 method for class 'mca':
plot(x, rows = TRUE, col, cex = par("cex"), ...)
```

Arguments

x	An object of class "mca".
rows	Should the coordinates for the rows be plotted, or just the vertices for the levels?
col, cex	The colours and cex to be used for the row points and level vertices respectively.
...	Additional parameters to plot.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*. Fourth edition. Springer.

See Also

mca, predict.mca

Examples

```
plot(mca(farms, abbrev = TRUE))
```

polr — *Ordered Logistic or Probit Regression*

Description

Fits a logistic or probit regression model to an ordered factor response. The default logistic case is *proportional odds logistic regression*, after which the function is named.

Usage

```
polr(formula, data, weights, start, ..., subset, na.action,
     contrasts = NULL, Hess = FALSE, model = TRUE,
     method = c("logistic", "probit", "cloglog", "cauchit"))
```

Arguments

`formula`	a formula expression as for regression models, of the form `response ~ predictors`. The response should be a factor (preferably an ordered factor), which will be interpreted as an ordinal response, with levels ordered as in the factor. A proportional odds model will be fitted. The model must have an intercept: attempts to remove one will lead to a warning and be ignored. An offset may be used. See the documentation of `formula` for other details.
`data`	an optional data frame in which to interpret the variables occurring in `formula`.
`weights`	optional case weights in fitting. Default to 1.
`start`	initial values for the parameters. This is in the format `c(coefficients, zeta)`: see the Values section.
`...`	additional arguments to be passed to `optim`, most often a `control` argument.
`subset`	expression saying which subset of the rows of the data should be used in the fit. All observations are included by default.
`na.action`	a function to filter missing data.
`contrasts`	a list of contrasts to be used for some or all of the factors appearing as variables in the model formula.
`Hess`	logical for whether the Hessian (the observed information matrix) should be returned.
`model`	logical for whether the model matrix should be returned.
`method`	logistic or probit or complementary log-log or cauchit (corresponding to a Cauchy latent variable and only available in R >= 2.1.0).

Details

This model is what Agresti (2002) calls a *cumulative link* model. The basic interpretation is as a *coarsened* version of a latent variable Y_i which has a logistic or normal or extreme-value or Cauchy distribution with scale parameter one and a linear model for the mean. The ordered factor which is observed is which bin Y_i falls into with breakpoints

$$\zeta_0 = -\infty < \zeta_1 < \cdots < \zeta_K = \infty$$

This leads to the model

$$\text{logit} P(Y \le k|x) = \zeta_k - \eta$$

with *logit* replaced by *probit* for a normal latent variable, and η being the linear predictor, a linear function of the explanatory variables (with no intercept). Note that it is quite common for other software to use the opposite sign for η.

In the logistic case, the left-hand side of the last display is the log odds of category k or less, and since these are log odds which differ only by a constant for different k, the odds are proportional. Hence the term *proportional odds logistic regression*.

In the complementary log-log case, we have a *proportional hazards* model for grouped survival times.

There are methods for the standard model-fitting functions, including `predict`, `summary`, `vcov`, `anova`, `model.frame` and an `extractAIC` method for use with `stepAIC`. There are also `profile` and `confint` methods.

Value

A object of class `"polr"`. This has components

`coefficients`	the coefficients of the linear predictor, which has no intercept.
`zeta`	the intercepts for the class boundaries.
`deviance`	the residual deviance.
`fitted.values`	a matrix, with a column for each level of the response.
`lev`	the names of the response levels.
`terms`	the `terms` structure describing the model.
`df.residual`	the number of residual degrees of freedoms, calculated using the weights.
`edf`	the (effective) number of degrees of freedom used by the model
`n, nobs`	the (effective) number of observations, calculated using the weights. (`nobs` is for use by `stepAIC`.
`call`	the matched call.
`method`	the matched method used.
`convergence`	the convergence code returned by `optim`.
`niter`	the number of function and gradient evaluations used by `optim`.
`Hessian`	(if `Hess` is true).
`model`	(if `model` is true).

References

Agresti, A. (2002) *Categorical Data.* Second edition. Wiley.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`optim`, `glm`, `multinom`.

Examples

```
options(contrasts = c("contr.treatment", "contr.poly"))
house.plr <- polr(Sat ~ Infl + Type + Cont, weights = Freq, data = housing)
house.plr
summary(house.plr)
## slightly worse fit from
summary(update(house.plr, method = "probit"))
## although it is not really appropriate, can fit
summary(update(house.plr, method = "cloglog"))

predict(house.plr, housing, type = "p")
addterm(house.plr, ~.^2, test = "Chisq")
house.plr2 <- stepAIC(house.plr, ~.^2)
house.plr2$anova
anova(house.plr, house.plr2)

house.plr <- update(house.plr, Hess=TRUE)
pr <- profile(house.plr)
confint(pr)
plot(pr)
pairs(pr)
```

`predict.glmmPQL` *Predict Method for glmmPQL Fits*

Description

Obtains predictions from a fitted generalized linear model with random effects.

Usage

```
## S3 method for class 'glmmPQL':
predict(object, newdata = NULL, type = c("link", "response"),
        level, na.action = na.pass, ...)
```

Arguments

`object`	a fitted object of class inheriting from `"glmmPQL"`.
`newdata`	optionally, a data frame in which to look for variables with which to predict.
`type`	the type of prediction required. The default is on the scale of the linear predictors; the alternative `"response"` is on the scale of the response variable. Thus for a default binomial model the default predictions are of log-odds (probabilities on logit scale) and `type = "response"` gives the predicted probabilities.
`level`	an optional integer vector giving the level(s) of grouping to be used in obtaining the predictions. Level values increase from outermost to innermost grouping, with level zero corresponding to the population predictions. Defaults to the highest or innermost level of grouping.
`na.action`	function determining what should be done with missing values in `newdata`. The default is to predict `NA`.
`...`	further arguments passed to or from other methods.

Value

If `level` is a single integer, a vector otherwise a data frame.

See Also

`glmmPQL`, `predict.lme`.

Examples

```
fit <- glmmPQL(y ~ trt + I(week > 2), random = ~1 |  ID,
               family = binomial, data = bacteria)
predict(fit, bacteria, level = 0, type="response")
predict(fit, bacteria, level = 1, type="response")
```

`predict.lda` *Classify Multivariate Observations by Linear Discrimination*

Description

Classify multivariate observations in conjunction with `lda`, and also project data onto the linear discriminants.

Usage

```
## S3 method for class 'lda':
predict(object, newdata, prior = object$prior, dimen,
        method = c("plug-in", "predictive", "debiased"), ...)
```

Arguments

`object`	object of class `"lda"`
`newdata`	data frame of cases to be classified or, if `object` has a formula, a data frame with columns of the same names as the variables used. A vector will be interpreted as a row vector. If newdata is missing, an attempt will be made to retrieve the data used to fit the `lda` object.
`prior`	The prior probabilities of the classes, by default the proportions in the training set or what was set in the call to `lda`.
`dimen`	the dimension of the space to be used. If this is less than `min(p, ng-1)`, only the first `dimen` discriminant components are used (except for `method="predictive"`), and only those dimensions are returned in `x`.
`method`	This determines how the parameter estimation is handled. With `"plug-in"` (the default) the usual unbiased parameter estimates are used and assumed to be correct. With `"debiased"` an unbiased estimator of the log posterior probabilities is used, and with `"predictive"` the parameter estimates are integrated out using a vague prior.
`...`	arguments based from or to other methods

Details

This function is a method for the generic function `predict()` for class `"lda"`. It can be invoked by calling `predict(x)` for an object `x` of the appropriate class, or directly by calling `predict.lda(x)` regardless of the class of the object.

Missing values in `newdata` are handled by returning `NA` if the linear discriminants cannot be evaluated. If `newdata` is omitted and the `na.action` of the fit omitted cases, these will be omitted on the prediction.

This version centres the linear discriminants so that the weighted mean (weighted by `prior`) of the group centroids is at the origin.

Value

a list with components

`class`	The MAP classification (a factor)
`posterior`	posterior probabilities for the classes
`x`	the scores of test cases on up to `dimen` discriminant variables

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*. Fourth edition. Springer.

Ripley, B. D. (1996) *Pattern Recognition and Neural Networks*. Cambridge University Press.

See Also

`lda`, `qda`, `predict.qda`

Examples

```
data(iris3)
tr <- sample(1:50, 25)
train <- rbind(iris3[tr,,1], iris3[tr,,2], iris3[tr,,3])
test <- rbind(iris3[-tr,,1], iris3[-tr,,2], iris3[-tr,,3])
cl <- factor(c(rep("s",25), rep("c",25), rep("v",25)))
z <- lda(train, cl)
predict(z, test)$class
```

---

predict.lqs *Predict from an lqs Fit*

Description

Predict from an resistant regression fitted by `lqs`.

Usage

```
## S3 method for class 'lqs':
predict(object, newdata, na.action = na.pass, ...)
```

Arguments

`object`	object inheriting from class `"lqs"`
`newdata`	matrix or data frame of cases to be predicted or, if `object` has a formula, a data frame with columns of the same names as the variables used. A vector will be interpreted as a row vector. If `newdata` is missing, an attempt will be made to retrieve the data used to fit the `lqs` object.
`na.action`	function determining what should be done with missing values in `newdata`. The default is to predict `NA`.
`...`	arguments to be passed from or to other methods.

Details

This function is a method for the generic function `predict()` for class `lqs`. It can be invoked by calling `predict(x)` for an object `x` of the appropriate class, or directly by calling `predict.lqs(x)` regardless of the class of the object.

Missing values in `newdata` are handled by returning `NA` if the linear fit cannot be evaluated. If `newdata` is omitted and the `na.action` of the fit omitted cases, these will be omitted on the prediction.

Value

A vector of predictions.

Author(s)

B.D. Ripley

See Also

lqs

Examples

```
data(stackloss)
set.seed(123)
fm <- lqs(stack.loss ~ ., data = stackloss, method = "S", nsamp = "exact")
predict(fm, stackloss)
```

---

predict.mca *Predict Method for Class 'mca'*

Description

Used to compute coordinates for additional rows or additional factors in a multiple correspondence analysis.

Usage

```
## S3 method for class 'mca':
predict(object, newdata, type = c("row", "factor"), ...)
```

Arguments

object	An object of class "mca", usually the result of a call to mca.
newdata	A data frame containing *either* additional rows of the factors used to fit object *or* additional factors for the cases used in the original fit.
type	Are predictions required for further rows or for new factors?
...	Additional arguments from predict: unused.

Value

If type = "row", the coordinates for the additional rows.

If type = "factor", the coordinates of the column vertices for the levels of the new factors.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*. Fourth edition. Springer.

See Also

mca, plot.mca

`predict.qda` *Classify from Quadratic Discriminant Analysis*

Description

Classify multivariate observations in conjunction with `qda`

Usage

```
## S3 method for class 'qda':
predict(object, newdata, prior = object$prior,
        method = c("plug-in", "predictive", "debiased", "looCV"), ...)
```

Arguments

`object`	object of class `"qda"`
`newdata`	data frame of cases to be classified or, if `object` has a formula, a data frame with columns of the same names as the variables used. A vector will be interpreted as a row vector. If newdata is missing, an attempt will be made to retrieve the data used to fit the `qda` object.
`prior`	The prior probabilities of the classes, by default the proportions in the training set or what was set in the call to `qda`.
`method`	This determines how the parameter estimation is handled. With `"plug-in"` (the default) the usual unbiased parameter estimates are used and assumed to be correct. With `"debiased"` an unbiased estimator of the log posterior probabilities is used, and with `"predictive"` the parameter estimates are integrated out using a vague prior. With `"looCV"` the leave-one-out cross-validation fits to the original dataset are computed and returned.
`...`	arguments based from or to other methods

Details

This function is a method for the generic function `predict()` for class `"qda"`. It can be invoked by calling `predict(x)` for an object `x` of the appropriate class, or directly by calling `predict.qda(x)` regardless of the class of the object.

Missing values in `newdata` are handled by returning `NA` if the quadratic discriminants cannot be evaluated. If `newdata` is omitted and the `na.action` of the fit omitted cases, these will be omitted on the prediction.

Value

a list with components

`class`	The MAP classification (a factor)
`posterior`	posterior probabilities for the classes

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*. Fourth edition. Springer.

Ripley, B. D. (1996) *Pattern Recognition and Neural Networks*. Cambridge University Press.

See Also

qda, lda, predict.lda

Examples

```
data(iris3)
tr <- sample(1:50, 25)
train <- rbind(iris3[tr,,1], iris3[tr,,2], iris3[tr,,3])
test <- rbind(iris3[-tr,,1], iris3[-tr,,2], iris3[-tr,,3])
cl <- factor(c(rep("s",25), rep("c",25), rep("v",25)))
zq <- qda(train, cl)
predict(zq, test)$class
```

---

qda *Quadratic Discriminant Analysis*

Description

Quadratic discriminant analysis.

Usage

```
qda(x, ...)

## S3 method for class 'formula':
qda(formula, data, ..., subset, na.action)

## Default S3 method:
qda(x, grouping, prior = proportions,
    method, CV = FALSE, nu, ...)

## S3 method for class 'data.frame':
qda(x, ...)

## S3 method for class 'matrix':
qda(x, grouping, ..., subset, na.action)
```

Arguments

formula	A formula of the form `groups ~ x1 + x2 + ...` That is, the response is the grouping factor and the right hand side specifies the (non-factor) discriminators.
data	Data frame from which variables specified in `formula` are preferentially to be taken.
x	(required if no formula is given as the principal argument.) a matrix or data frame or Matrix containing the explanatory variables.
grouping	(required if no formula principal argument is given.) a factor specifying the class for each observation.
prior	the prior probabilities of class membership. If unspecified, the class proportions for the training set are used. If specified, the probabilities should be specified in the order of the factor levels.

subset	An index vector specifying the cases to be used in the training sample. (NOTE: If given, this argument must be named.)
na.action	A function to specify the action to be taken if NAs are found. The default action is for the procedure to fail. An alternative is na.omit, which leads to rejection of cases with missing values on any required variable. (NOTE: If given, this argument must be named.)
method	"moment" for standard estimators of the mean and variance, "mle" for MLEs, "mve" to use cov.mve, or "t" for robust estimates based on a t distribution.
CV	If true, returns results (classes and posterior probabilities) for leave-out-out cross-validation. Note that if the prior is estimated, the proportions in the whole dataset are used.
nu	degrees of freedom for method = "t".
...	arguments passed to or from other methods.

Details

Uses a QR decomposition which will give an error message if the within-group variance is singular for any group.

Value

an object of class "qda" containing the following components:

prior	the prior probabilities used.
means	the group means.
scaling	for each group i, scaling[,,i] is an array which transforms observations so that within-groups covariance matrix is spherical.
ldet	a vector of half log determinants of the dispersion matrix.
lev	the levels of the grouping factor.
terms	(if formula is a formula) an object of mode expression and class term summarizing the formula.
call	the (matched) function call.
class	The MAP classification (a factor)
posterior	posterior probabilities for the classes

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

Ripley, B. D. (1996) *Pattern Recognition and Neural Networks*. Cambridge University Press.

See Also

predict.qda, lda

Examples

```
data(iris3)
tr <- sample(1:50, 25)
train <- rbind(iris3[tr,,1], iris3[tr,,2], iris3[tr,,3])
test <- rbind(iris3[-tr,,1], iris3[-tr,,2], iris3[-tr,,3])
cl <- factor(c(rep("s",25), rep("c",25), rep("v",25)))
z <- qda(train, cl)
predict(z,test)$class
```

---

quine *Absenteeism from School in Rural New South Wales*

Description

The `quine` data frame has 146 rows and 5 columns. Children from Walgett, New South Wales, Australia, were classified by Culture, Age, Sex and Learner status and the number of days absent from school in a particular school year was recorded.

Usage

```
quine
```

Format

This data frame contains the following columns:

`Eth` Ethnic background: Aboriginal or Not, (`"A"` or `"N"`).

`Sex` Sex factor: (`"F"` or `"M"`).

`Age` Age group: Primary (`"F0"`), or forms `"F1,"` `"F2"` or `"F3"`.

`Lrn` Learner status factor: Average or Slow learner, (`"AL"` or `"SL"`).

`Days` Days absent from school in the year.

Source

S. Quine, quoted in Aitkin, M. (1978) The analysis of unbalanced cross classifications (with discussion). *Journal of the Royal Statistical Society series A* **141**, 195–223.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

Rabbit *Blood Pressure in Rabbits*

Description

Five rabbits were studied on two occasions, after treatment with saline (control) and after treatment with the $5-HT_3$ antagonist MDL 72222. After each treatment ascending doses of phenylbiguanide were injected intravenously at 10 minute intervals and the responses of mean blood pressure measured. The goal was to test whether the cardiogenic chemoreflex elicited by phenylbiguanide depends on the activation of $5 - HT_3$ receptors.

Usage

```
Rabbit
```

Format

This data frame contains 60 rows and the following variables:

`BPchange` change in blood pressure relative to the start of the experiment

`Dose` dose of Phenylbiguanide in micrograms

`Run` label of run (`"C1"` to `"C5"`, then `"M1"` to `"M5"`)

`Treatment` placebo or the $5 - HT_3$ antagonist MDL 72222

`Animal` label of animal used (`"R1"` to `"R5"`)

Source

J. Ludbrook (1994) Repeated measurements and multiple comparisons in cardiovascular research. *Cardiovascular Research* **28**, 303–311.
[The numerical data are not in the paper but were supplied by Professor Ludbrook]

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

rational *Rational Approximation*

Description

Find rational approximations to the components of a real numeric object using a standard continued fraction method.

Usage

```
rational(x, cycles = 10, max.denominator = 2000, ...)
```

Arguments

`x`	Any object of mode numeric. Missing values are now allowed.
`cycles`	The maximum number of steps to be used in the continued fraction approximation process.
`max.denominator`	An early termination criterion. If any partial denominator exceeds `max.denominator` the continued fraction stops at that point.
`...`	arguments passed to or from other methods.

Details

Each component is first expanded in a continued fraction of the form

```
x = floor(x) + 1/(p1 + 1/(p2 + ...)))
```

where `p1`, `p2`, ... are positive integers, terminating either at `cycles` terms or when a `pj > max.denominator`. The continued fraction is then re-arranged to retrieve the numerator and denominator as integers and the ratio returned as the value.

Value

A numeric object with the same attributes as `x` but with entries rational approximations to the values. This effectively rounds relative to the size of the object and replaces very small entries by zero.

See Also

`fractions`

Examples

```
X <- matrix(runif(25), 5, 5)
solve(X, X/5)
##              [,1]         [,2]        [,3]         [,4]         [,5]
## [1,]  2.0000e-01  3.7199e-17 1.2214e-16  5.7887e-17 -8.7841e-17
## [2,] -1.1473e-16  2.0000e-01 7.0955e-17  2.0300e-17 -1.0566e-16
## [3,]  2.7975e-16  1.3653e-17 2.0000e-01 -1.3397e-16  1.5577e-16
## [4,] -2.9196e-16  2.0412e-17 1.5618e-16  2.0000e-01 -2.1921e-16
## [5,] -3.6476e-17 -3.6430e-17 3.6432e-17  4.7690e-17  2.0000e-01

## rational(solve(X, X/5))
##      [,1] [,2] [,3] [,4] [,5]
## [1,]  0.2  0.0  0.0  0.0  0.0
## [2,]  0.0  0.2  0.0  0.0  0.0
## [3,]  0.0  0.0  0.2  0.0  0.0
## [4,]  0.0  0.0  0.0  0.2  0.0
## [5,]  0.0  0.0  0.0  0.0  0.2
```

renumerate *Convert a Formula Transformed by 'denumerate'*

Description

denumerate converts a formula written using the conventions of loglm into one that terms is able to process. renumerate converts it back again to a form like the original.

Usage

```
renumerate(x)
```

Arguments

x	A formula, normally as modified by denumerate.

Details

This is an inverse function to denumerate. It is only needed since terms returns an expanded form of the original formula where the non-marginal terms are exposed. This expanded form is mapped back into a form corresponding to the one that the user originally supplied.

Value

A formula where all variables with names of the form .vn, where n is an integer, converted to numbers, n, as allowed by the formula conventions of loglm.

See Also

denumerate

Examples

```
denumerate(~(1+2+3)^3 + a/b)
## ~ (.v1 + .v2 + .v3)^3 + a/b
renumerate(.Last.value)
## ~ (1 + 2 + 3)^3 + a/b
```

rlm *Robust Fitting of Linear Models*

Description

Fit a linear model by robust regression using an M estimator.

Usage

```
rlm(x, ...)

## S3 method for class 'formula':
rlm(formula, data, weights, ..., subset, na.action,
    method = c("M", "MM", "model.frame"),
    wt.method = c("inv.var", "case"),
    model = TRUE, x.ret = TRUE, y.ret = FALSE, contrasts = NULL)

## Default S3 method:
rlm(x, y, weights, ..., w = rep(1, nrow(x)),
    init, psi = psi.huber, scale.est, k2 = 1.345,
    method = c("M", "MM"), wt.method = c("inv.var", "case"),
    maxit = 20, acc = 1e-4, test.vec = "resid", lqs.control = NULL)

psi.huber(u, k = 1.345, deriv = 0)
psi.hampel(u, a = 2, b = 4, c = 8, deriv = 0)
psi.bisquare(u, c = 4.685, deriv = 0)
```

Arguments

formula	a formula of the form `y ~ x1 + x2 + ...`.
data	data frame from which variables specified in `formula` are preferentially to be taken.
weights	a vector of prior weights for each case.
subset	An index vector specifying the cases to be used in fitting.
na.action	A function to specify the action to be taken if NAs are found. The default action is for the procedure to fail. An alternative is `na.omit`, which leads to omission of cases with missing values on any required variable.
x	a matrix or data frame containing the explanatory variables.
y	the response: a vector of length the number of rows of `x`.
method	currently either M-estimation or find the model frame. MM estimation is M-estimation with Tukey's biweight initialized by a specific S-estimator. See the details section.
wt.method	are the weights case weights (giving the relative importance of case, so a weight of 2 means there are two of these) or the inverse of the variances, so a weight of two means this error is half as variable?
model	should the model frame be returned in the object?
x.ret	should the model matrix be returned in the object?
y.ret	should the response be returned in the object?
contrasts	optional contrast specifications: se `lm`.
w	(optional) initial down-weighting for each case.
init	(optional) initial values for the coefficients OR a method to find initial values OR the result of a fit with a `coef` component. Known methods are `"ls"` (the default) for an initial least-squares fit using weights `w*weights`, and `"lts"` for an unweighted least-trimmed squares fit with 200 samples.
psi	the psi function is specified by this argument. It must give (possibly by name) a function `g(x, ..., deriv)` that for `deriv=0` returns psi(x)/x and for `deriv=1` returns psi'(x). Tuning constants will be passed in via `...`.

scale.est	method of scale estimation: re-scaled MAD of the residuals or Huber's proposal 2 (which can be selected by either "Huber" or "proposal 2").
k2	tuning constant used for Huber proposal 2 scale estimation.
maxit	the limit on the number of IWLS iterations.
acc	the accuracy for the stopping criterion.
test.vec	the stopping criterion is based on changes in this vector.
...	additional arguments to be passed to rlm.default or to the psi function.
lqs.control	An optional list of control values for lqs.
u	numeric vector of evalation points.
k,a,b,c	tuning constants
deriv	0 or 1: compute values of the psi function or of its first derivative.

Details

Fitting is done by iterated re-weighted least squares (IWLS).

Psi functions are supplied for the Huber, Hampel and Tukey bisquare proposals as psi.huber, psi.hampel and psi.bisquare. Huber's corresponds to a convex optimization problem and gives a unique solution (up to collinearity). The other two will have multiple local minima, and a good starting point is desirable.

Selecting method = "MM" selects a specific set of options which ensures that the estimator has a high breakdown point. The initial set of coefficients and the final scale are selected by an S-estimator with k0 = 1.548; this gives (for $n \gg p$) breakdown point 0.5. The final estimator is an M-estimator with Tukey's biweight and fixed scale that will inherit this breakdown point provided c > k0; this is true for the default value of c that corresponds to 95% relative efficiency at the normal. Case weights are not supported for method = "MM".

Value

An object of class "rlm" inheriting from "lm". The additional components not in an lm object are

s	the robust scale estimate used
w	the weights used in the IWLS process
psi	the psi function with parameters substituted
conv	the convergence criteria at each iteration
converged	did the IWLS converge?
wresid	a working residual, weighted for "inv.var" weights only.

References

P. J. Huber (1981) *Robust Statistics*. Wiley.

F. R. Hampel, E. M. Ronchetti, P. J. Rousseeuw and W. A. Stahel (1986) *Robust Statistics: The Approach based on Influence Functions*. Wiley.

A. Marazzi (1993) *Algorithms, Routines and S Functions for Robust Statistics*. Wadsworth & Brooks/Cole.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*. Fourth edition. Springer.

See Also

lm, lqs.

Examples

```
data(stackloss)
summary(rlm(stack.loss ~ ., stackloss))
rlm(stack.loss ~ ., stackloss, psi = psi.hampel, init = "lts")
rlm(stack.loss ~ ., stackloss, psi = psi.bisquare)
```

---

rms.curv *Relative Curvature Measures for Non-Linear Regression*

Description

Calculates the root mean square parameter effects and intrinsic relative curvatures, c^{θ} and c^{ι}, for a fitted nonlinear regression, as defined in Bates & Watts, section 7.3, p. 253 et seq.

Usage

```
rms.curv(obj)
```

Arguments

obj	Fitted model object of class "nls". The model must be fitted using the default algorithm.

Details

The method of section 7.3.1 of Bates & Watts is implemented. The function deriv3 should be used generate a model function with first derivative (gradient) matrix and second derivative (Hessian) array attributes. This function should then be used to fit the nonlinear regression model.

A print method, print.rms.curv, prints the pc and ic components only, suitably annotated.

If either pc or ic exceeds some threshold (0.3 has been suggested) the curvature is unacceptably high for the planar assumption.

Value

A list of class rms.curv with components pc and ic for parameter effects and intrinsic relative curvatures multiplied by sqrt(F), ct and ci for c^{θ} and c^{ι} (unmultiplied), and C the C-array as used in section 7.3.1 of Bates & Watts.

References

Bates, D. M, and Watts, D. G. (1988) *Nonlinear Regression Analysis and its Applications.* Wiley, New York.

See Also

deriv3

Examples

```
# The treated sample from the Puromycin data
data(Puromycin)
mmcurve <- deriv3(~ Vm * conc/(K + conc), c("Vm", "K"),
                  function(Vm, K, conc) NULL)
Treated <- Puromycin[Puromycin$state == "treated", ]
(Purfit1 <- nls(rate ~ mmcurve(Vm, K, conc), data = Treated,
                start = list(Vm=200, K=0.1)))
rms.curv(Purfit1)
##Parameter effects: c^theta x sqrt(F) = 0.2121
##        Intrinsic: c^iota  x sqrt(F) = 0.092
```

---

rnegbin *Simulate Negative Binomial Variates*

Description

Function to generate random outcomes from a Negative Binomial distribution, with mean `mu` and variance `mu + mu^2/theta`.

Usage

```
rnegbin(n, mu = n, theta = stop("'theta' must be specified"))
```

Arguments

`n`	If a scalar, the number of sample values required. If a vector, `length(n)` is the number required and `n` is used as the mean vector if `mu` is not specified.
`mu`	The vector of means. Short vectors are recycled.
`theta`	Vector of values of the `theta` parameter. Short vectors are recycled.

Details

The function uses the representation of the Negative Binomial distribution as a continuous mixture of Poisson distributions with Gamma distributed means. Unlike `rnbinom` the index can be arbitrary.

Value

Vector of random Negative Binomial variate values.

Side Effects

Changes `.Random.seed` in the usual way.

Examples

```
# Negative Binomials with means fitted(fm) and theta = 4.5
fm <- glm.nb(Days ~ ., data = quine)
dummy <- rnegbin(fitted(fm), theta = 4.5)
```

road *Road Accident Deaths in US States*

Description

A data frame with the annual deaths in road accidents for half the US states. Components are:

Usage

```
road
```

Format

state name
deaths number of deaths
drivers number of drivers (in 10,000's)
popden population density in people per square mile
rural length of rural roads, in 1000's of miles
temp average daily maximum temperature in January
fuel fuel consumption in 10,000,000 US gallons per year

Source

Imperial College, London M.Sc. exercise

rotifer *Numbers of Rotifers by Fluid Density*

Description

The data give the numbers of rotifers falling out of suspension for different fluid densities. There are two species, `pm` *Polyartha major* and `kc`, *Keratella cochlearis* and for each species the number falling out and the total number are given.

Usage

```
rotifer
```

Format

density specific density of fluid
pm.y number falling out for *P. major*
pm.total total number of *P. major*
kc.y number falling out for *K. cochlearis*
kc.tot total number of *K. cochlearis*

Source

D. Collett (1991) *Modelling Binary Data.* Chapman & Hall. p.217

Rubber *Accelerated Testing of Tyre Rubber*

Description

Data frame from accelerated testing of tyre rubber.

Usage

```
Rubber
```

Format

loss the abrasion loss in gm/hr.

hard the hardness in Shore units.

tens tensile strength in kg/sq m.

Source

O.L. Davies (1947) *Statistical Methods in Research and Production.* Oliver and Boyd, Table 6.1 p. 119.

O.L. Davies and P.L. Goldsmith (1972) *Statistical Methods in Research and Production.* 4th edition, Longmans, Table 8.1 p. 239.

References

Venables, W. N. and Ripley, B. D. (1999) *Modern Applied Statistics with S-PLUS.* Third Edition. Springer.

sammon *Sammon's Non-Linear Mapping*

Description

One form of non-metric multidimensional scaling.

Usage

```
sammon(d, y = cmdscale(d, k), k = 2, niter = 100, trace = TRUE,
       magic = 0.2, tol = 1e-4)
```

Arguments

d	distance structure of the form returned by dist, or a full, symmetric matrix. Data are assumed to be dissimilarities or relative distances, but must be positive except for self-distance. This can contain missing values.
y	An initial configuration. If none is supplied, cmdscale is used to provide the classical solution. (If there are missing values in d, an initial configuration must be provided.) This must not have duplicates.
k	The dimension of the configuration.
niter	The maximum number of iterations.
trace	Logical for tracing optimization. Default TRUE.
magic	initial value of the step size constant in diagonal Newton method.
tol	Tolerance for stopping, in units of stress.

Details

This chooses a two-dimensional configuration to minimize the stress, the sum of squared differences between the input distances and those of the configuration, weighted by the distances, the whole sum being divided by the sum of input distances to make the stress scale-free.

An iterative algorithm is used, which will usually converge in around 50 iterations. As this is necessarily an $O(n^2)$ calculation, it is slow for large datasets. Further, since the configuration is only determined up to rotations and reflections (by convention the centroid is at the origin), the result can vary considerably from machine to machine. In this release the algorithm has been modified by adding a step-length search (magic) to ensure that it always goes downhill.

Value

Two components:

points	A two-column vector of the fitted configuration.
stress	The final stress achieved.

Side Effects

If trace is true, the initial stress and the current stress are printed out every 10 iterations.

References

Sammon, J. W. (1969) A non-linear mapping for data structure analysis. *IEEE Trans. Comput.*, **C-18** 401–409.

Ripley, B. D. (1996) *Pattern Recognition and Neural Networks*. Cambridge University Press.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*. Fourth edition. Springer.

See Also

cmdscale, isoMDS

Examples

```
data(swiss)
swiss.x <- as.matrix(swiss[, -1])
swiss.sam <- sammon(dist(swiss.x))
plot(swiss.sam$points, type = "n")
text(swiss.sam$points, labels = as.character(1:nrow(swiss.x)))
```

---

ships *Ships Damage Data*

Description

Data frame giving the number of damage incidents and aggregate months of service by ship type, year of construction, and period of operation.

Usage

```
ships
```

Format

type type: "A" to "E"

year year of construction: 1960-64, 65-69, 70-74, 75-79 (coded as "60", "65", "70", "75")

period period of operation : 1960-74, 75-79

service aggregate months of service

incidents number of damage incidents

Source

P. McCullagh and J. A. Nelder, (1983), *Generalized Linear Models.* Chapman & Hall, section 6.3.2, page 137

---

shoes *Shoe wear data of Box, Hunter and Hunter*

Description

A list of two vectors, giving the wear of shoes of materials A and B for one foot each of ten boys.

Usage

```
shoes
```

Source

G. E. P. Box, W. G. Hunter and J. S. Hunter (1978) *Statistics for Experimenters.* Wiley, p. 100

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

shrimp — *Percentage of Shrimp in Shrimp Cocktail*

Description

A numeric vector with 18 determinations by different laboratories of the amount (percentage of the declared total weight) of shrimp in shrimp cocktail.

Usage

```
shrimp
```

Source

F. J. King and J. J. Ryan (1976) Collaborative study of the determination of the amount of shrimp in shrimp cocktail. *J. Off. Anal. Chem.* **59**, 644–649

R. G. Staudte and S. J. Sheather (1990) *Robust Estimation and Testing.* Wiley

shuttle — *Space Shuttle Autolander Problem*

Description

The `shuttle` data frame has 256 rows and 7 columns. The first six columns are categorical variables giving example conditions; the seventh is the decision. The first 253 rows are the training set, the last 3 the test conditions.

Usage

```
shuttle
```

Format

This data frame contains the following columns:

`stability` Stable positioning or not (`stab / xstab`)

`error` Size of error (`MM / SS / LX / XL`)

`sign` Sign of error, positive or negative (`pp / nn`)

`wind` Wind sign (`head / tail`)

`magn` Wind strength (`Light / Medium / Strong / Out of Range`)

`vis` Visibility (`yes / no`)

`use` Use the autolander or not

Source

D. Michie (1989) Problems of computer-aided concept formation. In *Applications of Expert Systems 2*, ed. J. R. Quinlan, Turing Institute Press / Addison-Wesley, pp. 310–333.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

Sitka — *Growth Curves for Sitka Spruce Trees in 1988*

Description

The `Sitka` data frame has 395 rows and 4 columns. It gives repeated measurements on the log-size of 79 Sitka spruce trees, 54 of which were grown in ozone-enriched chambers and 25 were controls. The size was measured five times in 1988, at roughly monthly intervals.

Usage

```
Sitka
```

Format

This data frame contains the following columns:

`size` measured size (height times diameter squared) of tree, on log scale

`Time` time of measurement in days since 1 January 1988

`tree` number of tree

`treat` either `"ozone"` for an ozone-enriched chamber or `"control"`

Source

P. J. Diggle, K.-Y. Liang and S. L. Zeger (1994) *Analysis of Longitudinal Data.* Clarendon Press, Oxford

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`Sitka89`

Sitka89 — *Growth Curves for Sitka Spruce Trees in 1989*

Description

The `Sitka89` data frame has 632 rows and 4 columns. It gives repeated measurements on the log-size of 79 Sitka spruce trees, 54 of which were grown in ozone-enriched chambers and 25 were controls. The size was measured eight times in 1989, at roughly monthly intervals.

Usage

```
Sitka89
```

Format

This data frame contains the following columns:

`size` measured size (height times diameter squared) of tree, on log scale

`Time` time of measurement in days since 1 January 1988

`tree` number of tree

`treat` either `"ozone"` for an ozone-enriched chamber or `"control"`

Source

P. J. Diggle, K.-Y. Liang and S. L. Zeger (1994) *Analysis of Longitudinal Data.* Clarendon Press, Oxford

See Also

`Sitka`

`Skye` *AFM Compositions of Aphyric Skye Lavas*

Description

The `Skye` data frame has 23 rows and 3 columns.

Usage

```
Skye
```

Format

This data frame contains the following columns:

`A` Percentage of sodium and potassium oxides

`F` Percentage of iron oxide

`M` Percentage of magnesium oxide

Source

R. N. Thompson, J. Esson and A. C. Duncan (1972) Major element chemical variation in the Eocene lavas of the Isle of Skye. *J. Petrology*, **13**, 219–253.

References

J. Aitchison (1986) *The Statistical Analysis of Compositional Data.* Chapman and Hall, p.360.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

Examples

```
# ternary() is from the on-line answers.
ternary <- function(X, pch = par("pch"), lcex = 1,
                    add = FALSE, ord = 1:3, ...)
{
  X <- as.matrix(X)
  if(any(X) < 0) stop("X must be non-negative")
  s <- drop(X %*% rep(1, ncol(X)))
  if(any(s<=0)) stop("each row of X must have a positive sum")
  if(max(abs(s-1)) > 1e-6) {
    warning("row(s) of X will be rescaled")
    X <- X / s
  }
  X <- X[, ord]
  s3 <- sqrt(1/3)
  if(!add)
  {
    oldpty <- par("pty")
    on.exit(par(pty=oldpty))
    par(pty="s")
    plot(c(-s3, s3), c(0.5-s3, 0.5+s3), type="n", axes=FALSE,
         xlab="", ylab="")
    polygon(c(0, -s3, s3), c(1, 0, 0), density=0)
    lab <- NULL
    if(!is.null(dn <- dimnames(X))) lab <- dn[[2]]
    if(length(lab) < 3) lab <- as.character(1:3)
    eps <- 0.05 * lcex
    text(c(0, s3+eps*0.7, -s3-eps*0.7),
         c(1+eps, -0.1*eps, -0.1*eps), lab, cex=lcex)
  }
  points((X[,2] - X[,3])*s3, X[,1], ...)
}

ternary(Skye/100, ord=c(1,3,2))
```

`snails` *Snail Mortality Data*

Description

Groups of 20 snails were held for periods of 1, 2, 3 or 4 weeks in carefully controlled conditions of temperature and relative humidity. There were two species of snail, A and B, and the experiment was designed as a 4 by 3 by 4 by 2 completely randomized design. At the end of the exposure time the snails were tested to see if they had survived; the process itself is fatal for the animals. The object of the exercise was to model the probability of survival in terms of the stimulus variables, and in particular to test for differences between species.

The data are unusual in that in most cases fatalities during the experiment were fairly small.

Usage

```
snails
```

Format

The data frame contains the following components:

`Species` Snail species A (1) or B (2)

`Exposure` Exposure in weeks

`Rel.Hum` Relative humidity (4 levels)

`Temp` Temperature, in degrees Celsius (3 levels)

`Deaths` Number of deaths

`N` Number of snails exposed

Source

Zoology Department, The University of Adelaide.

References

Venables, W. N. and Ripley, B. D. (1999) *Modern Applied Statistics with S-PLUS.* Third Edition. Springer.

SP500 *Returns of the Standard and Poors 500*

Description

Returns of the Standard and Poors 500 Index in the 1990's

Usage

```
SP500
```

Format

A vector of returns of the Standard and Poors 500 index for all the trading days in 1990, 1991, . . . , 1999.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

stdres *Extract Standardized Residuals from a Linear Model*

Description

The standardized residuals. These are normalized to unit variance, fitted including the current data point.

Usage

```
stdres(object)
```

Arguments

`object`	any object representing a linear model.

Value

The vector of appropriately transformed residuals.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`residuals`, `studres`

steam *The Saturated Steam Pressure Data*

Description

Temperature and pressure in a saturated steam driven experimental device.

Usage

```
steam
```

Format

The data frame contains the following components:

`Temp` Temperature, in degrees Celsius

`Press` Pressure, in Pascals

Source

N.R. Draper and H. Smith (1981) *Applied Regression Analysis.* Wiley, pp. 518–9.

References

Venables, W. N. and Ripley, B. D. (1999) *Modern Applied Statistics with S-PLUS*. Third Edition. Springer.

`stepAIC` *Choose a model by AIC in a Stepwise Algorithm*

Description

Performs stepwise model selection by exact AIC.

Usage

```
stepAIC(object, scope, scale = 0,
        direction = c("both", "backward", "forward"),
        trace = 1, keep = NULL, steps = 1000, use.start = FALSE,
        k = 2, ...)
```

Arguments

`object`	an object representing a model of an appropriate class. This is used as the initial model in the stepwise search.
`scope`	defines the range of models examined in the stepwise search. This should be either a single formula, or a list containing components `upper` and `lower`, both formulae. See the details for how to specify the formulae and how they are used.
`scale`	used in the definition of the AIC statistic for selecting the models, currently only for `lm` and `aov` models (see `extractAIC` for details).
`direction`	the mode of stepwise search, can be one of `"both"`, `"backward"`, or `"forward"`, with a default of `"both"`. If the `scope` argument is missing the default for `direction` is `"backward"`.
`trace`	if positive, information is printed during the running of `stepAIC`. Larger values may give more information on the fitting process.
`keep`	a filter function whose input is a fitted model object and the associated `AIC` statistic, and whose output is arbitrary. Typically `keep` will select a subset of the components of the object and return them. The default is not to keep anything.
`steps`	the maximum number of steps to be considered. The default is 1000 (essentially as many as required). It is typically used to stop the process early.
`use.start`	if true the updated fits are done starting at the linear predictor for the currently selected model. This may speed up the iterative calculations for `glm` (and other fits), but it can also slow them down. **Not used** in R.
`k`	the multiple of the number of degrees of freedom used for the penalty. Only `k = 2` gives the genuine AIC: `k = log(n)` is sometimes referred to as BIC or SBC.
`...`	any additional arguments to `extractAIC`. (None are currently used.)

Details

The set of models searched is determined by the `scope` argument. The right-hand-side of its `lower` component is always included in the model, and right-hand-side of the model is included in the `upper` component. If `scope` is a single formula, it specifes the `upper` component, and the `lower` model is empty. If `scope` is missing, the initial model is used as the `upper` model.

Models specified by `scope` can be templates to update `object` as used by `update.formula`.

There is a potential problem in using `glm` fits with a variable `scale`, as in that case the deviance is not simply related to the maximized log-likelihood. The `glm` method for `extractAIC` makes the appropriate adjustment for a `gaussian` family, but may need to be amended for other cases. (The `binomial` and `poisson` families have fixed `scale` by default and do not correspond to a particular maximum-likelihood problem for variable `scale`.)

Where a conventional deviance exists (e.g. for `lm`, `aov` and `glm` fits) this is quoted in the analysis of variance table: it is the *unscaled* deviance.

Value

the stepwise-selected model is returned, with up to two additional components. There is an `"anova"` component corresponding to the steps taken in the search, as well as a `"keep"` component if the `keep=` argument was supplied in the call. The `"Resid. Dev"` column of the analysis of deviance table refers to a constant minus twice the maximized log likelihood: it will be a deviance only in cases where a saturated model is well-defined (thus excluding `lm`, `aov` and `survreg` fits, for example).

Note

The model fitting must apply the models to the same dataset. This may be a problem if there are missing values and an `na.action` other than `na.fail` is used (as is the default in R). We suggest you remove the missing values first.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`addterm`, `dropterm`, `step`

Examples

```
quine.hi <- aov(log(Days + 2.5) ~ .^4, quine)
quine.nxt <- update(quine.hi, . ~ . - Eth:Sex:Age:Lrn)
quine.stp <- stepAIC(quine.nxt,
    scope = list(upper = ~Eth*Sex*Age*Lrn, lower = ~1),
    trace = FALSE)
quine.stp$anova

cpus1 <- cpus
attach(cpus)
for(v in names(cpus)[2:7])
  cpus1[[v]] <- cut(cpus[[v]], unique(quantile(cpus[[v]])),
                    include.lowest = TRUE)
detach()
cpus0 <- cpus1[, 2:8]  # excludes names, authors' predictions
```

```
cpus.samp <- sample(1:209, 100)
cpus.lm <- lm(log10(perf) ~ ., data = cpus1[cpus.samp,2:8])
cpus.lm2 <- stepAIC(cpus.lm, trace = FALSE)
cpus.lm2$anova

example(birthwt)
birthwt.glm <- glm(low ~ ., family = binomial, data = bwt)
birthwt.step <- stepAIC(birthwt.glm, trace = FALSE)
birthwt.step$anova
birthwt.step2 <- stepAIC(birthwt.glm, ~ .^2 + I(scale(age)^2)
    + I(scale(lwt)^2), trace = FALSE)
birthwt.step2$anova

quine.nb <- glm.nb(Days ~ .^4, data = quine)
quine.nb2 <- stepAIC(quine.nb)
quine.nb2$anova
```

`stormer` *The Stormer Viscometer Data*

Description

The stormer viscometer measures the viscosity of a fluid by measuring the time taken for an inner cylinder in the mechanism to perform a fixed number of revolutions in response to an actuating weight. The viscometer is calibrated by measuring the time taken with varying weights while the mechanism is suspended in fluids of accurately known viscosity. The data comes from such a calibration, and theoretical considerations suggest a nonlinear relationship between time, weight and viscosity, of the form `Time = (B1*Viscosity)/(Weight - B2) + E` where `B1` and `B2` are unknown parameters to be estimated, and `E` is error.

Usage

```
stormer
```

Format

The data frame contains the following components:

`Viscosity` Viscosity of fluid

`Wt` Actuating weight

`Time` Time taken

Source

E. J. Williams (1959) *Regression Analysis.* Wiley.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

studres *Extract Studentized Residuals from a Linear Model*

Description

The Studentized residuals. Like standardized residuals, these are normalized to unit variance, but the Studentized version is fitted ignoring the current data point. (They are sometimes called jackknifed residuals).

Usage

```
studres(object)
```

Arguments

object	any object representing a linear model.

Value

The vector of appropriately transformed residuals.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

residuals, stdres

summary.loglm *Summary Method Function for Objects of Class 'loglm'*

Description

Returns a summary list for log-linear models fitted by iterative proportional scaling using loglm.

Usage

```
## S3 method for class 'loglm':
summary(object, fitted = FALSE, ...)
```

Arguments

object	a fitted loglm model object.
fitted	if TRUE return observed and expected frequencies in the result. Using fitted = TRUE may necessitate re-fitting the object.
...	arguments to be passed to or from other methods.

Details

This function is a method for the generic function `summary()` for class `"loglm"`. It can be invoked by calling `summary(x)` for an object `x` of the appropriate class, or directly by calling `summary.loglm(x)` regardless of the class of the object.

Value

a list is returned for use by `print.summary.loglm`. This has components

`formula`	the formula used to produce `object`
`tests`	the table of test statistics (likelihood ratio, Pearson) for the fit.
`oe`	if `fitted = TRUE`, an array of the observed and expected frequencies, otherwise `NULL`.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`loglm`, `summary`

`summary.negbin` *Summary Method Function for Objects of Class 'negbin'*

Description

Identical to `summary.glm`, but with three lines of additional output: the ML estimate of theta, its standard error, and twice the log-likelihood function.

Usage

```
## S3 method for class 'negbin':
summary(object, dispersion = 1, correlation = TRUE, ...)
```

Arguments

`object`	Fitted model object of class `negbin` inheriting from `glm` and `lm`. Typically the output of `glm.nb`.
`dispersion`	as for `summary.glm`, with a default of 1.
`correlation`	as for `summary.glm`.
`...`	arguments passed to or from other methods.

Details

`summary.glm` is used to produce the majority of the output and supply the result. This function is a method for the generic function `summary()` for class `"negbin"`. It can be invoked by calling `summary(x)` for an object `x` of the appropriate class, or directly by calling `summary.negbin(x)` regardless of the class of the object.

Value

As for `summary.glm`; the additional lines of output are not included in the resultant object.

Side Effects

A summary table is produced as for `summary.glm`, with the additional information described above.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*. Fourth edition. Springer.

See Also

`summary`, `glm.nb`, `negative.binomial`, `anova.negbin`

Examples

```
summary(glm.nb(Days ~ Eth*Age*Lrn*Sex, quine, link = log))
```

`summary.rlm` *Summary Method for Robust Linear Models*

Description

`summary` method for objects of class `"rlm"`

Usage

```
## S3 method for class 'rlm':
summary(object, method = c("XtX", "XtWX"), correlation = TRUE, ...)
```

Arguments

`object`	the fitted model. This is assumed to be the result of some fit that produces an object inheriting from the class `rlm`, in the sense that the components returned by the `rlm` function will be available.
`method`	Should the weighted (by the IWLS weights) or unweighted cross-products matrix be used?
`correlation`	logical. Should correlations be computed (and printed)?
`...`	arguments passed to or from other methods.

Details

This function is a method for the generic function `summary()` for class `"rlm"`. It can be invoked by calling `summary(x)` for an object `x` of the appropriate class, or directly by calling `summary.rlm(x)` regardless of the class of the object.

Value

If printing takes place, only a null value is returned. Otherwise, a list is returned with the following components. Printing always takes place if this function is invoked automatically as a method for the `summary` function.

`correlation`	The computed correlation coefficient matrix for the coefficients in the model.
`cov.unscaled`	The unscaled covariance matrix; i.e, a matrix such that multiplying it by an estimate of the error variance produces an estimated covariance matrix for the coefficients.
`sigma`	The scale estimate.
`stddev`	A scale estimate used for the standard errors.
`df`	The number of degrees of freedom for the model and for residuals.
`coefficients`	A matrix with three columns, containing the coefficients, their standard errors and the corresponding t statistic.
`terms`	The terms object used in fitting this model.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`summary`

Examples

```
summary(rlm(calls ~ year, data = phones, maxit = 50))
## Not run:
Call:
rlm(formula = calls ~ year, data = phones, maxit = 50)

Residuals:
   Min     1Q Median     3Q    Max
-18.31  -5.95  -1.68  26.46 173.77

Coefficients:
            Value    Std. Error t value
(Intercept) -102.622   26.553   -3.86
year           2.041    0.429    4.76

Residual standard error: 9.03 on 22 degrees of freedom

Correlation of Coefficients:
[1] -0.994

## End(Not run)
```

`survey` *Student Survey Data*

Description

This data frame contains the responses of 237 Statistics I students at the University of Adelaide to a number of questions.

Usage

```
survey
```

Format

The components of the data frame are:

`Sex` The sex of the student. (Factor with levels `"Male"` and `"Female"`.)

`Wr.Hnd` The span (distance from tip of thumb to tip of little finger of spread hand) of writing hand, in centimetres.

`NW.Hnd` Span of non-writing hand

`W.Hnd` Writing hand of student. (Factor, with levels `"Left"` and `"Right"`.)

`Fold` "Fold your arms! Which is on top" (Factor, with levels `"R on L"`, `"L on R"`, `"Neither"`.)

`Pulse` Pulse rate of student (beats per minute.)

`Clap` 'Clap your hands! Which hand is on top?' (Factor, with levels `"Right"`, `"Left"`, `"Neither"`.)

`Exer` How often the student exercises. (Factor, with levels `"Freq"` (frequently), `"Some"`, `"None"`)

`Smoke` How much the student smokes. (Factor, levels `"Heavy"`, `"Regul"` (regularly), `"Occas"` (occasionally), `"Never"`.)

`Height` The height of the student, in centimetres.

`M.I` Indicates whether the student expressed height in imperial (feet/inches) or metric (centimetres/metres) units. (Factor, levels `"Metric"`, `"Imperial"`.)

`Age` Age of the student, in years.

References

Venables, W. N. and Ripley, B. D. (1999) *Modern Applied Statistics with S-PLUS.* Third Edition. Springer.

synth.tr *Synthetic Classification Problem*

Description

The `synth.tr` data frame has 250 rows and 3 columns. The `synth.te` data frame has 100 rows and 3 columns. It is intended that `synth.tr` be used from training and `synth.te` for testing.

Usage

```
synth.tr
synth.te
```

Format

These data frames contains the following columns:

`xs` x-coordinate

`ys` y-coordinate

`yc` class, coded as 0 or 1.

Source

Ripley, B.D. (1994) Neural networks and related methods for classification (with discussion). *Journal of the Royal Statistical Society series B* **56**, 409–456.

Ripley, B.D. (1996) *Pattern Recognition and Neural Networks.* Cambridge: Cambridge University Press.

theta.md *Estimate theta of the Negative Binomial*

Description

Given the estimated mean vector, estimate `theta` of the Negative Binomial Distribution.

Usage

```
theta.md(y, mu, dfr, weights, limit = 20, eps = .Machine$double.eps^0.25)

theta.ml(y, mu, n, weights, limit = 10, eps = .Machine$double.eps^0.25,
         trace = FALSE)

theta.mm(y, mu, dfr, weights, limit = 10, eps = .Machine$double.eps^0.25)
```

Arguments

y	Vector of observed values from the Negative Binomial.
mu	Estimated mean vector.
n	Number of data points (defaults to the sum of weights)
dfr	Residual degrees of freedom (assuming theta known). For a weighted fit this is the sum of the weights minus the number of fitted parameters.
weights	Case weights. If missing, taken as 1.
limit	Limit on the number of iterations.
eps	Tolerance to determine convergence.
trace	logical: should iteration progress be printed?

Details

theta.md estimates by equating the deviance to the residual degrees of freedom, an analogue of a moment estimator.

theta.ml uses maximum likelihood.

theta.mm calculates the moment estimator of theta by equating the Pearson chi-square $sum((y - \mu)^2/(\mu + \mu^2/\theta))$ to the residual degrees of freedom.

Value

The required estimate of theta, as a scalar. For theta.ml, the standard error is given as attribute "SE".

See Also

glm.nb

Examples

```
quine.nb <- glm.nb(Days ~ .^2, data = quine)
theta.md(quine$Days, fitted(quine.nb), dfr = df.residual(quine.nb))
theta.ml(quine$Days, fitted(quine.nb))
theta.mm(quine$Days, fitted(quine.nb), dfr = df.residual(quine.nb))

## weighted example
yeast <- data.frame(cbind(numbers = 0:5, fr = c(213, 128, 37, 18, 3, 1)))
fit <- glm.nb(numbers ~ 1, weights = fr, data = yeast)
summary(fit)
attach(yeast)
mu <- fitted(fit)
theta.md(numbers, mu, dfr = 399, weights = fr)
theta.ml(numbers, mu, weights = fr)
theta.mm(numbers, mu, dfr = 399, weights = fr)
detach()
```

topo *Spatial Topographic Data*

Description

The `topo` data frame has 52 rows and 3 columns, of topographic heights within a 310 feet square.

Usage

```
topo
```

Format

This data frame contains the following columns:

`x` x coordinates (units of 50 feet)

`y` y coordinates (units of 50 feet)

`z` heights (feet)

Source

Davis, J.C. (1973) *Statistics and Data Analysis in Geology.* Wiley.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

Traffic *Effect of Swedish Speed Limits on Accidents*

Description

An experiment was performed in Sweden in 1961-2 to assess the effect of a speed limit on the motorway accident rate. The experiment was conducted on 92 days in each year, matched so that day `j` in 1962 was comparable to day `j` in 1961. On some days the speed limit was in effect and enforced, while on other days there was no speed limit and cars tended to be driven faster. The speed limit days tended to be in contiguous blocks.

Usage

```
Traffic
```

Format

This data frame contains the following columns:

`year` 1961 or 1962

`day` of year

`limit` was there a speed limit?

`y` traffic accident count for that day

Source

Svensson, A. (1981) On the goodness-of-fit test for the multiplicative Poisson model. *Annals of Statistics,* **9**, 697–704.

References

Venables, W. N. and Ripley, B. D. (1999) *Modern Applied Statistics with S-PLUS.* Third Edition. Springer.

`truehist` *Plot a Histogram*

Description

Creates a histogram on the current graphics device.

Usage

```
truehist(data, nbins = "Scott", h, x0 = -h/1000,
         breaks, prob = TRUE, xlim = range(breaks),
         ymax = max(est), col,
         xlab = deparse(substitute(data)), bty = "n", ...)
```

Arguments

`data`	numeric vector of data for histogram. Missing values (`NA`s) are allowed and omitted.
`nbins`	The suggested number of bins. Either a number, or a character string naming a rule: `"Scott"` or `"Freedman-Diaconis"` or `"FD"`. (Case is ignored.)
`h`	The bin width (takes precedence over `nbins`).
`x0`	Shift for the bins - the breaks are at `x0 + h * (..., -1, 0, 1, ...)`
`breaks`	The set of breakpoints to be used. (Usually omitted, takes precedence over `h` and `nbins`).
`prob`	If true (the default) plot a true histogram. The vertical axis has a *relative frequency density* scale, so the product of the dimensions of any panel gives the relative frequency. Hence the total area under the histogram is 1 and it is directly comparable with most other estimates of the probability density function. If false plot the counts in the bins.
`xlim`	The limits for the x-axis.
`ymax`	The upper limit for the y-axis.
`col`	The colour number for the bar fill.
`xlab`	label for the plot x-axis. By default, this will be the name of `data`.
`bty`	The box type for the plot - defaults to none.
`...`	additional arguments to `rect` or `plot`.

Details

This plots a true histogram, a density estimate of total area 1. If `breaks` is specified, those breakpoints are used. Otherwise if `h` is specified, a regular grid of bins is used with width `h`. If neither `breaks` nor `h` is specified, `nbins` is used to select a suitable `h`.

Side Effects

A histogram is plotted on the current device.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`hist`

`ucv` *Unbiased Cross-Validation for Bandwidth Selection*

Description

Uses unbiased cross-validation to select the bandwidth of a Gaussian kernel density estimator.

Usage

```
ucv(x, nb = 1000, lower, upper)
```

Arguments

`x`	a numeric vector
`nb`	number of bins to use.
`lower, upper`	Range over which to minimize. The default is almost always satisfactory.

Value

a bandwidth.

References

Scott, D. W. (1992) *Multivariate Density Estimation: Theory, Practice, and Visualization.* Wiley.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`bcv`, `width.SJ`, `density`

Examples

```
ucv(geyser$duration)
```

`UScereal` *Nutritional and Marketing Information on US Cereals*

Description

The `UScereal` data frame has 65 rows and 11 columns. The data come from the 1993 ASA Statistical Graphics Exposition, and are taken from the mandatory F&DA food label. The data have been normalized here to a portion of one American cup.

Usage

```
UScereal
```

Format

This data frame contains the following columns:

`mfr` Manufacturer, represented by its first initial: G=General Mills, K=Kelloggs, N=Nabisco, P=Post, Q=Quaker Oats, R=Ralston Purina.

`calories` number of calories in one portion

`protein` grams of protein in one portion

`fat` grams of fat in one portion

`sodium` milligrams of sodium in one portion

`fibre` grams of dietary fibre in one portion

`carbo` grams of complex carbohydrates in one portion

`sugars` grams of sugars in one portion

`shelf` display shelf (1, 2, or 3, counting from the floor)

`potassium` grams of potassium

`vitamins` vitamins and minerals (none, enriched, or 100%)

Source

The original data are available at http://lib.stat.cmu.edu/datasets/1993.expo/.

References

Venables, W. N. and Ripley, B. D. (1999) *Modern Applied Statistics with S-PLUS.* Third Edition. Springer.

UScrime *The Effect of Punishment Regimes on Crime Rates*

Description

Criminologists are interested in the effect of punishment regimes on crime rates. This has been studied using aggregate data on 47 states of the USA for 1960 given in this data frame. The variables seem to have been re-scaled to convenient numbers.

Usage

```
UScrime
```

Format

This data frame contains the following columns:

M percentage of males aged 14-24

So indicator variable for a southern state

Ed mean years of schooling

Po1 police expenditure in 1960

Po2 police expenditure in 1959

LF labour force participation rate

M.F number of males per 1000 females

Pop state population

NW number of nonwhites per 1000 people

U1 unemployment rate of urban males 14-24

U2 unemployment rate of urban males 35-39

GDP gross domestic product per head

Ineq income inequality

Prob probability of imprisonment

Time average time served in state prisons

y rate of crimes in a particular category per head of population

Source

Ehrlich, I. (1973) Participation in illegitimate activities: a theoretical and empirical investigation. *Journal of Political Economy*, **81**, 521–565.

Vandaele, W. (1978) Participation in illegitimate activities: Ehrlich revisited. In *Deterrence and Incapacitation*, eds A. Blumstein, J. Cohen and D. Nagin, pp. 270–335. US National Academy of Sciences.

References

Venables, W. N. and Ripley, B. D. (1999) *Modern Applied Statistics with S-PLUS*. Third Edition. Springer.

VA *Veteran's Administration Lung Cancer Trial*

Description

Veteran's Administration lung cancer trial from Kalbfleisch & Prentice.

Usage

```
VA
```

Format

A data frame with columns:

stime survival or follow-up time in days.

status dead or censored.

treat standard or test

age patient's age in years

Karn Karnofsky score of patient's performance on a scale of 0 to 100.

diag.time times since diagnosis in months at entry to trial.

cell one of four cell types.

prior prior therapy?

Source

Kalbfleisch, J.D. and Prentice R.L. (1980) *The Statistical Analysis of Failure Time Data.*

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

waders *Counts of Waders at 15 Sites in South Africa*

Description

The `waders` data frame has 15 rows and 19 columns. The entries are counts of waders in summer.

Usage

```
waders
```

Format

This data frame contains the following columns (species)

S1 Oystercatcher

S2 White-fronted Plover

S3 Kitt Lutz's Plover

S4 Three-banded Plover

S5 Grey Plover

S6 Ringed Plover

S7 Bar-tailed Godwit

S8 Whimbrel

S9 Marsh Sandpiper

S10 Greenshank

S11 Common Sandpiper

S12 Turnstone

S13 Knot

S14 Sanderling

S15 Little Stint

S16 Curlew Sandpiper

S17 Ruff

S18 Avocet

S19 Black-winged Stilt

The rows are the sites: A = Namibia North coast B = Namibia North wetland C = Namibia South coast D = Namibia South wetland E = Cape North coast F = Cape North wetland G = Cape West coast H = Cape West wetland I = Cape South coast J= Cape South wetland K = Cape East coast L = Cape East wetland M = Transkei coast N = Natal coast O = Natal wetland

Source

J.C. Gower and D.J. Hand (1996) *Biplots* Chapman & Hall Table 9.1. Quoted as from:

R.W. Summers, L.G. Underhill, D.J. Pearson and D.A. Scott (1987) Wader migration systems in south and eastern Africa and western Asia. *Wader Study Group Bulletin* **49** Supplement, 15–34.

Examples

```
plot(corresp(waders, nf=2))
```

`whiteside` *House Insulation: Whiteside's Data*

Description

Mr Derek Whiteside of the UK Building Research Station recorded the weekly gas consumption and average external temperature at his own house in south-east England for two heating seasons, one of 26 weeks before, and one of 30 weeks after cavity-wall insulation was installed. The object of the exercise was to assess the effect of the insulation on gas consumption.

Usage

```
whiteside
```

Format

The `whiteside` data frame has 56 rows and 3 columns.:

`Insul` A factor, before or after insulation.

`Temp` Purportedly the average outside temperature in degrees Celsius. (These values is far too low for any 56-week period in the 1960s in South-East England. It might be the weekly average of daily minima.)

`Gas` The weekly gas consumption in 1000s of cubic feet.

Source

A data set collected in the 1960s by Mr Derek Whiteside of the UK Building Research Station. Reported by

Hand, D. J., Daly, F., McConway, K., Lunn, D. and Ostrowski, E. eds (1993) *A Handbook of Small Data Sets.* Chapman & Hall, p. 69.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

Examples

```
require(lattice)
xyplot(Gas ~ Temp | Insul, whiteside, panel =
  function(x, y, ...) {
    panel.xyplot(x, y, ...)
    panel.lmline(x, y, ...)
  }, xlab = "Average external temperature (deg. C)",
  ylab = "Gas consumption  (1000 cubic feet)", aspect = "xy",
  strip = function(...) strip.default(..., style = 1))

gasB <- lm(Gas ~ Temp, whiteside, subset = Insul=="Before")
gasA <- update(gasB, subset = Insul=="After")
summary(gasB)
summary(gasA)
gasBA <- lm(Gas ~ Insul/Temp - 1, whiteside)
summary(gasBA)
```

```
gasQ <- lm(Gas ~ Insul/(Temp + I(Temp^2)) - 1, whiteside)
summary(gasQ)$coef

gasPR <- lm(Gas ~ Insul + Temp, whiteside)
anova(gasPR, gasBA)
options(contrasts = c("contr.treatment", "contr.poly"))
gasBA1 <- lm(Gas ~ Insul*Temp, whiteside)
summary(gasBA1)$coef
```

`width.SJ` *Bandwidth Selection by Pilot Estimation of Derivatives*

Description

Uses the method of Sheather & Jones (1991) to select the bandwidth of a Gaussian kernel density estimator.

Usage

```
width.SJ(x, nb = 1000, lower, upper, method = c("ste", "dpi"))
```

Arguments

`x`	a numeric vector
`nb`	number of bins to use.
`upper, lower`	range over which to search for solution if `method = "ste"`.
`method`	Either `"ste"` ("solve-the-equation") or `"dpi"` ("direct plug-in").

Value

a bandwidth.

References

Sheather, S. J. and Jones, M. C. (1991) A reliable data-based bandwidth selection method for kernel density estimation. *Journal of the Royal Statistical Society series B* **53**, 683–690.

Scott, D. W. (1992) *Multivariate Density Estimation: Theory, Practice, and Visualization.* Wiley.

Wand, M. P. and Jones, M. C. (1995) *Kernel Smoothing.* Chapman & Hall.

See Also

`ucv`, `bcv`, `density`

Examples

```
attach(geyser)
width.SJ(duration, method = "dpi")
width.SJ(duration)
detach()

width.SJ(galaxies, method = "dpi")
width.SJ(galaxies)
```

write.matrix *Write a Matrix or Data Frame*

Description

Writes a matrix or data frame to a file or the console, using column labels and a layout respecting columns.

Usage

```
write.matrix(x, file = "", sep = " ", blocksize)
```

Arguments

x	matrix or data frame.
file	name of output file. The default ("") is the console.
sep	The separator between columns.
blocksize	If supplied and positive, the output is written in blocks of blocksize rows. Choose as large as possible consistent with the amount of memory available.

Details

If x is a matrix, supplying blocksize is more memory-efficient and enables larger matrices to be written, but each block of rows might be formatted slightly differently.

If x is a data frame, the conversion to a matrix may negate the memory saving.

Side Effects

A formatted file is produced, with column headings (if x has them) and columns of data.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

write.table

wtloss *Weight Loss Data from an Obese Patient*

Description

The data frame gives the weight, in kilograms, of an obese patient at 52 time points over an 8 month period of a weight rehabilitation programme.

Usage

```
wtloss
```

Format

This data frame contains the following columns:

`Days` Time in days since the start of the programme.

`Weight` Weight in kilograms of the patient.

Source

Dr T. Davies, Adelaide.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

Examples

```
wtloss.fm <- nls(Weight ~ b0 + b1*2^(-Days/th),
    data = wtloss, start = list(b0=90, b1=95, th=120),
    trace = TRUE)
```

Chapter 12

The boot package

`abc.ci` *Nonparametric ABC Confidence Intervals*

Description

Calculate equi-tailed two-sided nonparametric approximate bootstrap confidence intervals for a parameter, given a set of data and an estimator of the parameter, using numerical differentiation.

Usage

```
abc.ci(data, statistic, index=1, strata=rep(1, n), conf=0.95,
       eps=0.001/n, ...)
```

Arguments

`data`	A data set expressed as a vector, matrix or data frame.
`statistic`	A function which returns the statistic of interest. The function must take at least 2 arguments; the first argument should be the data and the second a vector of weights. The weights passed to `statistic` will be normalized to sum to 1 within each stratum. Any other arguments should be passed to `abc.ci` as part of the `...{}` argument.
`index`	If `statistic` returns a vector of length greater than 1, then this indicates the position of the variable of interest within that vector.
`strata`	A factor or numerical vector indicating to which sample each observation belongs in multiple sample problems. The default is the one-sample case.
`conf`	A scalar or vector containing the confidence level(s) of the required interval(s).
`eps`	The value of epsilon to be used for the numerical differentiation.
`...`	Any other arguments for `statistic`. These will be passed unchanged to `statistic` each time it is called within `abc.ci`.

Details

This function is based on the function `abcnon` written by R. Tibshirani. A listing of the original function is available in DiCiccio and Efron (1996). The function uses numerical differentiation for the first and second derivatives of the statistic and then uses these values to approximate the bootstrap BCa intervals. The total number of evaluations of the statistic is `2*n+2+2*length(conf)` where `n` is the number of data points (plus calculation of the original value of the statistic). The function works for the multiple sample case without the need to rewrite the statistic in an artificial form since the stratified normalization is done internally by the function.

Value

A `length(conf)` by 3 matrix where each row contains the confidence level followed by the lower and upper end-points of the ABC interval at that level.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*, Chapter 5. Cambridge University Press.

DiCiccio, T. J. and Efron B. (1992) More accurate confidence intervals in exponential families. *Biometrika*, **79**, 231–245.

DiCiccio, T. J. and Efron B. (1996) Bootstrap confidence intervals (with Discussion). *Statistical Science*, **11**, 189–228.

See Also

`boot.ci`

Examples

```
# 90% and 95% confidence intervals for the correlation
# coefficient between the columns of the bigcity data

abc.ci(bigcity, corr, conf=c(0.90,0.95))

# A 95% confidence interval for the difference between the means of
# the last two samples in gravity
mean.diff <- function(y, w)
{    gp1 <- 1:table(as.numeric(y$series))[1]
     sum(y[gp1,1] * w[gp1]) - sum(y[-gp1,1] * w[-gp1])
}
grav1 <- gravity[as.numeric(gravity[,2])>=7,]
abc.ci(grav1, mean.diff, strata=grav1$series)
```

`acme` *Monthly Excess Returns*

Description

The `acme` data frame has 60 rows and 3 columns.

The excess return for the Acme Cleveland Corporation are recorded along with those for all stocks listed on the New York and American Stock Exchanges were recorded over a five year period. These excess returns are relative to the return on a risk-less investment such a U.S. Treasury bills.

Usage

```
acme
```

Format

This data frame contains the following columns:

month A character string representing the month of the observation.

market The excess return of the market as a whole.

acme The excess return for the Acme Cleveland Corporation.

Source

The data were obtained from

Simonoff, J.S. and Tsai, C.-L. (1994) Use of modified profile likelihood for improved tests of constancy of variance in regression. *Applied Statistics*, **43**, 353–370.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

---

aids *Delay in AIDS Reporting in England and Wales*

Description

The `aids` data frame has 570 rows and 6 columns.

Although all cases of AIDS in England and Wales must be reported to the Communicable Disease Surveillance Centre, there is often a considerable delay between the time of diagnosis and the time that it is reported. In estimating the prevalence of AIDS, account must be taken of the unknown number of cases which have been diagnosed but not reported. The data set here records the reported cases of AIDS diagnosed from July 1983 and until the end of 1992. The data are cross-classified by the date of diagnosis and the time delay in the reporting of the cases.

Usage

```
aids
```

Format

This data frame contains the following columns:

year The year of the diagnosis.

quarter The quarter of the year in which diagnosis was made.

delay The time delay (in months) between diagnosis and reporting. 0 means that the case was reported within one month. Longer delays are grouped in 3 month intervals and the value of `delay` is the midpoint of the interval (therefore a value of `2` indicates that reporting was delayed for between 1 and 3 months).

dud An indicator of censoring. These are categories for which full information is not yet available and the number recorded is a lower bound only.

time The time interval of the diagnosis. That is the number of quarters from July 1983 until the end of the quarter in which these cases were diagnosed.

y The number of AIDS cases reported.

Source

The data were obtained from

De Angelis, D. and Gilks, W.R. (1994) Estimating acquired immune deficiency syndrome accounting for reporting delay. *Journal of the Royal Statistical Society, A*, **157**, 31–40.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

---

aircondit *Failures of Air-conditioning Equipment*

Description

Proschan (1963) reported on the times between failures of the air-conditioning equipment in 10 Boeing 720 aircraft. The `aircondit` data frame contains the intervals for the ninth aircraft while `aircondit7` contains those for the seventh aircraft.

Both data frames have just one column. Note that the data have been sorted into increasing order.

Usage

```
aircondit
```

Format

The data frames contain the following column:

hours The time interval in hours between successive failures of the air-conditioning equipment

Source

The data were taken from

Cox, D.R. and Snell, E.J. (1981) *Applied Statistics: Principles and Examples*. Chapman and Hall.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

Proschan, F. (1963) Theoretical explanation of observed decreasing failure rate. *Technometrics*, **5**, 375-383.

amis *Car Speeding and Warning Signs*

Description

The amis data frame has 8437 rows and 4 columns.

In a study into the effect that warning signs have on speeding patterns, Cambridgeshire County Council considered 14 pairs of locations. The locations were paired to account for factors such as traffic volume and type of road. One site in each pair had a sign erected warning of the dangers of speeding and asking drivers to slow down. No action was taken at the second site. Three sets of measurements were taken at each site. Each set of measurements was nominally of the speeds of 100 cars but not all sites have exactly 100 measurements. These speed measurements were taken before the erection of the sign, shortly after the erection of the sign, and again after the sign had been in place for some time.

Usage

```
amis
```

Format

This data frame contains the following columns:

speed Speeds of cars (in miles per hour).

period A numeric column indicating the time that the reading was taken. A value of 1 indicates a reading taken before the sign was erected, a 2 indicates a reading taken shortly after erection of the sign and a 3 indicates a reading taken after the sign had been in place for some time.

warning A numeric column indicating whether the location of the reading was chosen to have a warning sign erected. A value of 1 indicates presence of a sign and a value of 2 indicates that no sign was erected.

pair A numeric column giving the pair number at which the reading was taken. Pairs were numbered from 1 to 14.

Source

The data were kindly made available by Mr. Graham Amis, Cambridgeshire County Council, U.K.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

aml *Remission Times for Acute Myelogenous Leukaemia*

Description

The aml data frame has 23 rows and 3 columns.

A clinical trial to evaluate the efficacy of maintenance chemotherapy for acute myelogenous leukaemia was conducted by Embury et al. (1977) at Stanford University. After reaching a stage of remission through treatment by chemotherapy, patients were randomized into two groups. The first group received maintenance chemotherapy and the second group did not. The aim of the study was to see if maintenance chemotherapy increased the length of the remission. The data here formed a preliminary analysis which was conducted in October 1974.

Usage

```
aml
```

Format

This data frame contains the following columns:

time The length of the complete remission (in weeks).

cens An indicator of right censoring. 1 indicates that the patient had a relapse and so time is the length of the remission. 0 indicates that the patient had left the study or was still in remission in October 1974, that is the length of remission is right-censored.

group The group into which the patient was randomized. Group 1 received maintenance chemotherapy, group 2 did not.

Note

Package **survival** also has a dataset aml. It is the same data with different names and with group replaced by a factor x.

Source

The data were obtained from

Miller, R.G. (1981) *Survival Analysis*. John Wiley.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

Embury, S.H, Elias, L., Heller, P.H., Hood, C.E., Greenberg, P.L. and Schrier, S.L. (1977) Remission maintenance therapy in acute myelogenous leukaemia. *Western Journal of Medicine*, **126**, 267-272.

`beaver` *Beaver Body Temperature Data*

Description

The `beaver` data frame has 100 rows and 4 columns. It is a multivariate time series of class `"ts"` and also inherits from class `"data.frame"`.

This data set is part of a long study into body temperature regulation in beavers. Four adult female beavers were live-trapped and had a temperature-sensitive radiotransmitter surgically implanted. Readings were taken every 10 minutes. The location of the beaver was also recorded and her activity level was dichotomized by whether she was in the retreat or outside of it since high-intensity activities only occur outside of the retreat.

The data in this data frame are those readings for one of the beavers on a day in autumn.

Usage

```
beaver
```

Format

This data frame contains the following columns:

`day` The day number. The data includes only data from day 307 and early 308.

`time` The time of day formatted as hour-minute.

`temp` The body temperature in degrees Celsius.

`activ` The dichotomized activity indicator. `1` indicates that the beaver is outside of the retreat and therefore engaged in high-intensity activity.

Source

The data were obtained from

Reynolds, P.S. (1994) Time-series analyses of beaver body temperatures. In *Case Studies in Biometry*. N. Lange, L. Ryan, L. Billard, D. Brillinger, L. Conquest and J. Greenhouse (editors), 211–228. John Wiley.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

bigcity *Population of U.S. Cities*

Description

The `bigcity` data frame has 49 rows and 2 columns.

The `city` data frame has 10 rows and 2 columns.

The measurements are the population (in 1000's) of 49 U.S. cities in 1920 and 1930. The 49 cities are a random sample taken from the 196 largest cities in 1920. The `city` data frame consists of the first 10 observations in `bigcity`.

Usage

```
bigcity
```

Format

This data frame contains the following columns:

u The 1920 population.

x The 1930 population.

Source

The data were obtained from

Cochran, W.G. (1977) *Sampling Techniques*. Third edition. John Wiley

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

boot *Bootstrap Resampling*

Description

Generate `R` bootstrap replicates of a statistic applied to data. Both parametric and nonparametric resampling are possible. For the nonparametric bootstrap, possible resampling methods are the ordinary bootstrap, the balanced bootstrap, antithetic resampling, and permutation. For nonparametric multi-sample problems stratified resampling is used. This is specified by including a vector of strata in the call to boot. Importance resampling weights may be specified.

Usage

```
boot(data, statistic, R, sim="ordinary", stype="i",
     strata=rep(1,n), L=NULL, m=0, weights=NULL,
     ran.gen=function(d, p) d, mle=NULL, ...)
```

Arguments

`data`	The data as a vector, matrix or data frame. If it is a matrix or data frame then each row is considered as one multivariate observation.
`statistic`	A function which when applied to data returns a vector containing the statistic(s) of interest. When `sim="parametric"`, the first argument to `statistic` must be the data. For each replicate a simulated dataset returned by `ran.gen` will be passed. In all other cases `statistic` must take at least two arguments. The first argument passed will always be the original data. The second will be a vector of indices, frequencies or weights which define the bootstrap sample. Further, if predictions are required, then a third argument is required which would be a vector of the random indices used to generate the bootstrap predictions. Any further arguments can be passed to `statistic` through the `...{}` argument.
`R`	The number of bootstrap replicates. Usually this will be a single positive integer. For importance resampling, some resamples may use one set of weights and others use a different set of weights. In this case `R` would be a vector of integers where each component gives the number of resamples from each of the rows of weights.
`sim`	A character string indicating the type of simulation required. Possible values are `"ordinary"` (the default), `"parametric"`, `"balanced"`, `"permutation"`, or `"antithetic"`. Importance resampling is specified by including importance weights; the type of importance resampling must still be specified but may only be `"ordinary"` or `"balanced"` in this case.
`stype`	A character string indicating what the second argument of statistic represents. Possible values of stype are `"i"` (indices - the default), `"f"` (frequencies), or `"w"` (weights).
`strata`	An integer vector or factor specifying the strata for multi-sample problems. This may be specified for any simulation, but is ignored when `sim` is `"parametric"`. When `strata` is supplied for a nonparametric bootstrap, the simulations are done within the specified strata.
`L`	Vector of influence values evaluated at the observations. This is used only when `sim` is `"antithetic"`. If not supplied, they are calculated through a call to `empinf`. This will use the infinitesimal jackknife provided that `stype` is `"w"`, otherwise the usual jackknife is used.
`m`	The number of predictions which are to be made at each bootstrap replicate. This is most useful for (generalized) linear models. This can only be used when `sim` is `"ordinary"`. `m` will usually be a single integer but, if there are strata, it may be a vector with length equal to the number of strata, specifying how many of the errors for prediction should come from each strata. The actual predictions should be returned as the final part of the output of `statistic`, which should also take a vector of indices of the errors to be used for the predictions.
`weights`	Vector or matrix of importance weights. If a vector then it should have as many elements as there are observations in `data`. When simulation from more than one set of weights is required, `weights` should be a matrix where each row of the matrix is one set of importance weights. If `weights` is a matrix then `R` must be a vector of length `nrow(weights)`. This parameter is ignored if `sim` is not `"ordinary"` or `"balanced"`.
`ran.gen`	This function is used only when `sim` is `"parametric"` when it describes how random values are to be generated. It should be a function of two arguments.

The first argument should be the observed data and the second argument consists of any other information needed (e.g. parameter estimates). The second argument may be a list, allowing any number of items to be passed to `ran.gen`. The returned value should be a simulated data set of the same form as the observed data which will be passed to statistic to get a bootstrap replicate. It is important that the returned value be of the same shape and type as the original dataset. If `ran.gen` is not specified, the default is a function which returns the original `data` in which case all simulation should be included as part of `statistic`. Use of `sim="parametric"` with a suitable `ran.gen` allows the user to implement any types of nonparametric resampling which are not supported directly.

`mle` The second argument to be passed to `ran.gen`. Typically these will be maximum likelihood estimates of the parameters. For efficiency `mle` is often a list containing all of the objects needed by `ran.gen` which can be calculated using the original data set only.

`...` Any other arguments for `statistic` which are passed unchanged each time it is called. Any such arguments to `statistic` must follow the arguments which `statistic` is required to have for the simulation.

Details

The statistic to be bootstrapped can be as simple or complicated as desired as long as its arguments correspond to the dataset and (for a nonparametric bootstrap) a vector of indices, frequencies or weights. `statistic` is treated as a black box by the `boot` function and is not checked to ensure that these conditions are met.

The first order balanced bootstrap is described in Davison, Hinkley and Schechtman (1986). The antithetic bootstrap is described by Hall (1989) and is experimental, particularly when used with strata. The other non-parametric simulation types are the ordinary bootstrap (possibly with unequal probabilities), and permutation which returns random permutations of cases. All of these methods work independently within strata if that argument is supplied.

For the parametric bootstrap it is necessary for the user to specify how the resampling is to be conducted. The best way of accomplishing this is to specify the function `ran.gen` which will return a simulated data set from the observed data set and a set of parameter estimates specified in `mle`.

Value

The returned value is an object of class `"boot"`, containing the following components :

`t0` The observed value of `statistic` applied to `data`.

`t` A matrix with `R` rows each of which is a bootstrap replicate of `statistic`.

`R` The value of `R` as passed to `boot`.

`data` The `data` as passed to `boot`.

`seed` The value of `.Random.seed` when `boot` was called.

`statistic` The function `statistic` as passed to `boot`.

`sim` Simulation type used.

`stype` Statistic type as passed to `boot`.

`call` The original call to `boot`.

`strata` The strata used. This is the vector passed to `boot`, if it was supplied or a vector of ones if there were no strata. It is not returned if `sim` is `"parametric"`.

weights	The importance sampling weights as passed to boot or the empirical distribution function weights if no importance sampling weights were specified. It is omitted if sim is not one of "ordinary" or "balanced".
pred.i	If predictions are required (m>0) this is the matrix of indices at which predictions were calculated as they were passed to statistic. Omitted if m is 0 or sim is not "ordinary".
L	The influence values used when sim is "antithetic". If no such values were specified and stype is not "w" then L is returned as consecutive integers corresponding to the assumption that data is ordered by influence values. This component is omitted when sim is not "antithetic".
ran.gen	The random generator function used if sim is "parametric". This component is omitted for any other value of sim.
mle	The parameter estimates passed to boot when sim is "parametric". It is omitted for all other values of sim.

References

There are many references explaining the bootstrap and its variations. Among them are :

Booth, J.G., Hall, P. and Wood, A.T.A. (1993) Balanced importance resampling for the bootstrap. *Annals of Statistics*, **21**, 286–298.

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

Davison, A.C., Hinkley, D.V. and Schechtman, E. (1986) Efficient bootstrap simulation. *Biometrika*, **73**, 555–566.

Efron, B. and Tibshirani, R. (1993) *An Introduction to the Bootstrap*. Chapman & Hall.

Gleason, J.R. (1988) Algorithms for balanced bootstrap simulations. *American Statistician*, **42**, 263–266.

Hall, P. (1989) Antithetic resampling for the bootstrap. *Biometrika*, **73**, 713–724.

Hinkley, D.V. (1988) Bootstrap methods (with Discussion). *Journal of the Royal Statistical Society, B*, **50**, 312–337, 355–370.

Hinkley, D.V. and Shi, S. (1989) Importance sampling and the nested bootstrap. *Biometrika*, **76**, 435–446.

Johns M.V. (1988) Importance sampling for bootstrap confidence intervals. *Journal of the American Statistical Association*, **83**, 709–714.

Noreen, E.W. (1989) *Computer Intensive Methods for Testing Hypotheses*. John Wiley & Sons.

See Also

boot.array, boot.ci, censboot, empinf, jack.after.boot, tilt.boot, tsboot

Examples

```
# usual bootstrap of the ratio of means using the city data
ratio <- function(d, w)
     sum(d$x * w)/sum(d$u * w)
boot(city, ratio, R=999, stype="w")

# Stratified resampling for the difference of means.  In this
# example we will look at the difference of means between the final
```

```
# two series in the gravity data.
diff.means <- function(d, f)
{    n <- nrow(d)
     gp1 <- 1:table(as.numeric(d$series))[1]
     m1 <- sum(d[gp1,1] * f[gp1])/sum(f[gp1])
     m2 <- sum(d[-gp1,1] * f[-gp1])/sum(f[-gp1])
     ss1 <- sum(d[gp1,1]^2 * f[gp1]) -
            (m1 *  m1 * sum(f[gp1]))
     ss2 <- sum(d[-gp1,1]^2 * f[-gp1]) -
            (m2 *  m2 * sum(f[-gp1]))
     c(m1-m2, (ss1+ss2)/(sum(f)-2))
}
grav1 <- gravity[as.numeric(gravity[,2])>=7,]
boot(grav1, diff.means, R=999, stype="f", strata=grav1[,2])

#  In this example we show the use of boot in a prediction from
#  regression based on the nuclear data.  This example is taken
#  from Example 6.8 of Davison and Hinkley (1997).  Notice also
#  that two extra arguments to statistic are passed through boot.
nuke <- nuclear[,c(1,2,5,7,8,10,11)]
nuke.lm <- glm(log(cost)~date+log(cap)+ne+ ct+log(cum.n)+pt, data=nuke)
nuke.diag <- glm.diag(nuke.lm)
nuke.res <- nuke.diag$res*nuke.diag$sd
nuke.res <- nuke.res-mean(nuke.res)

#  We set up a new data frame with the data, the standardized
#  residuals and the fitted values for use in the bootstrap.
nuke.data <- data.frame(nuke,resid=nuke.res,fit=fitted(nuke.lm))

#  Now we want a prediction of plant number 32 but at date 73.00
new.data <- data.frame(cost=1, date=73.00, cap=886, ne=0,
                       ct=0, cum.n=11, pt=1)
new.fit <- predict(nuke.lm, new.data)

nuke.fun <- function(dat, inds, i.pred, fit.pred, x.pred)
{
     assign(".inds", inds, envir=.GlobalEnv)
     lm.b <- glm(fit+resid[.inds] ~date+log(cap)+ne+ct+
          log(cum.n)+pt, data=dat)
     pred.b <- predict(lm.b,x.pred)
     remove(".inds", envir=.GlobalEnv)
     c(coef(lm.b), pred.b-(fit.pred+dat$resid[i.pred]))
}

nuke.boot <- boot(nuke.data, nuke.fun, R=999, m=1,
     fit.pred=new.fit, x.pred=new.data)
#  The bootstrap prediction error would then be found by
mean(nuke.boot$t[,8]^2)
#  Basic bootstrap prediction limits would be
new.fit-sort(nuke.boot$t[,8])[c(975,25)]



#  Finally a parametric bootstrap.  For this example we shall look
#  at the air-conditioning data.  In this example our aim is to test
#  the hypothesis that the true value of the index is 1 (i.e. that
#  the data come from an exponential distribution) against the
```

```
#  alternative that the data come from a gamma distribution with
#  index not equal to 1.
air.fun <- function(data)
{    ybar <- mean(data$hours)
     para <- c(log(ybar),mean(log(data$hours)))
     ll <- function(k) {
          if (k <= 0) out <- 1e200 # not NA
          else out <- lgamma(k)-k*(log(k)-1-para[1]+para[2])
         out
     }
     khat <- nlm(ll,ybar^2/var(data$hours))$estimate
     c(ybar, khat)
}

air.rg <- function(data, mle)
#  Function to generate random exponential variates.  mle will contain
#  the mean of the original data
{    out <- data
     out$hours <- rexp(nrow(out), 1/mle)
     out
}

air.boot <- boot(aircondit, air.fun, R=999, sim="parametric",
     ran.gen=air.rg, mle=mean(aircondit$hours))

# The bootstrap p-value can then be approximated by
sum(abs(air.boot$t[,2]-1) > abs(air.boot$t0[2]-1))/(1+air.boot$R)
```

`boot.array` *Bootstrap Resampling Arrays*

Description

This function takes a bootstrap object calculated by one of the functions `boot`, `censboot`, or `tilt.boot` and returns the frequency (or index) array for the the bootstrap resamples.

Usage

```
boot.array(boot.out, indices=)
```

Arguments

`boot.out`	An object of class `"boot"` returned by one of the generation functions for such an object.
`indices`	A logical argument which specifies whether to return the frequency array or the raw index array. The default is `indices=FALSE` unless `boot.out` was created by `tsboot` in which case the default is `indices=TRUE`.

Details

The process by which the original index array was generated is repeated with the same value of `.Random.seed`. If the frequency array is required then `freq.array` is called to convert the index array to a frequency array.

A resampling array can only be returned when such a concept makes sense. In particular it cannot be found for any parametric or model-based resampling schemes. Hence for objects generated by `censboot` the only resampling scheme for which such an array can be found is ordinary case resampling. Similarly if `boot.out$sim` is `"parametric"` in the case of `boot` or `"model"` in the case of `tsboot` the array cannot be found. Note also that for post-blackened bootstraps from `tsboot` the indices found will relate to those prior to any post-blackening and so will not be useful.

Frequency arrays are used in many post-bootstrap calculations such as the jackknife-after-bootstrap and finding importance sampling weights. They are also used to find empirical influence values through the regression method.

Value

A matrix with `boot.out$R` rows and `n` columns where `n` is the number of observations in `boot.out$data`. If `indices` is `FALSE` then this will give the frequency of each of the original observations in each bootstrap resample. If `indices` is `TRUE` it will give the indices of the bootstrap resamples in the order in which they would have been passed to the statistic.

Side Effects

This function temporarily resets `.Random.seed` to the value in `boot.out$seed` and then returns it to its original value at the end of the function.

See Also

`boot`, `censboot`, `freq.array`, `tilt.boot`, `tsboot`

Examples

```
#  A frequency array for a nonparametric bootstrap
city.boot <- boot(city, corr, R=40, stype="w")
boot.array(city.boot)

perm.cor <- function(d,i)
     cor(d$x,d$u[i])
city.perm <- boot(city, perm.cor, R=40, sim="permutation")
boot.array(city.perm, indices=TRUE)
```

---

boot.ci *Nonparametric Bootstrap Confidence Intervals*

Description

This function generates 5 different types of equi-tailed two-sided nonparametric confidence intervals. These are the first order normal approximation, the basic bootstrap interval, the studentized bootstrap interval, the bootstrap percentile interval, and the adjusted bootstrap percentile (BCa) interval. All or a subset of these intervals can be generated.

Usage

```
boot.ci(boot.out, conf = 0.95, type = "all",
        index = 1:min(2,length(boot.out$t0)), var.t0 = NULL,
        var.t = NULL, t0 = NULL, t = NULL, L = NULL, h = function(t) t,
        hdot = function(t) rep(1,length(t)), hinv = function(t) t, ...)
```

Arguments

boot.out	An object of class "boot" containing the output of a bootstrap calculation.
conf	A scalar or vector containing the confidence level(s) of the required interval(s).
type	A vector of character strings representing the type of intervals required. The value should be any subset of the values c("norm","basic", "stud", "perc", "bca") or simply "all" which will compute all five types of intervals.
index	This should be a vector of length 1 or 2. The first element of index indicates the position of the variable of interest in boot.out$t0 and the relevant column in boot.out$t. The second element indicates the position of the variance of the variable of interest. If both var.t0 and var.t are supplied then the second element of index (if present) is ignored. The default is that the variable of interest is in position 1 and its variance is in position 2 (as long as there are 2 positions in boot.out$t0).
var.t0	If supplied, a value to be used as an estimate of the variance of the statistic for the normal approximation and studentized intervals. If it is not supplied and length(index) is 2 then var.t0 defaults to boot.out$t0[index[2]] otherwise var.t0 is undefined. For studentized intervals var.t0 must be defined. For the normal approximation, if var.t0 is undefined it defaults to var(t). If a transformation is supplied through the argument h then var.t0 should be the variance of the untransformed statistic.
var.t	This is a vector (of length boot.out$R) of variances of the bootstrap replicates of the variable of interest. It is used only for studentized intervals. If it is not supplied and length(index) is 2 then var.t defaults to boot.out$t[,index[2]], otherwise its value is undefined which will cause an error for studentized intervals. If a transformation is supplied through the argument h then var.t should be the variance of the untransformed bootstrap statistics.
t0	The observed value of the statistic of interest. The default value is boot.out$t0[index[1]]. Specification of t0 and t allows the user to get intervals for a transformed statistic which may not be in the bootstrap output object. See the second example below. An alternative way of achieving this would be to supply the functions h, hdot, and hinv below.
t	The bootstrap replicates of the statistic of interest. It must be a vector of length boot.out$R. It is an error to supply one of t0 or t but not the other. Also if studentized intervals are required and t0 and t are supplied then so should be var.t0 and var.t. The default value is boot.out$t[,index].
L	The empirical influence values of the statistic of interest for the observed data. These are used only for BCa intervals. If a transformation is supplied through the parameter h then L should be the influence values for t; the values for h(t) are derived from these and hdot within the function. If L is not supplied then the values are calculated using empinf if they are needed.
h	A function defining a transformation. The intervals are calculated on the scale of h(t) and the inverse function hinv applied to the resulting intervals. It must be a function of one variable only and for a vector argument, it must return a vector of the same length, i.e. h(c(t1,t2,t3)) should return c(h(t1),h(t2),h(t3)). The default is the identity function.

hdot	A function of one argument returning the derivative of h. It is a required argument if h is supplied and normal, studentized or BCa intervals are required. The function is used for approximating the variances of h(t0) and h(t) using the delta method, and also for finding the empirical influence values for BCa intervals. Like h it should be able to take a vector argument and return a vector of the same length. The default is the constant function 1.
hinv	A function, like h, which returns the inverse of h. It is used to transform the intervals calculated on the scale of h(t) back to the original scale. The default is the identity function. If h is supplied but hinv is not, then the intervals returned will be on the transformed scale.
...	Any extra arguments that boot.out$statistic is expecting. These arguments are needed only if BCa intervals are required and L is not supplied since in that case L is calculated through a call to empinf which calls boot.out$statistic.

Details

The formulae on which the calculations are based can be found in Chapter 5 of Davison and Hinkley (1997). Function boot must be run prior to running this function to create the object to be passed as boot.out.

Variance estimates are required for studentized intervals. The variance of the observed statistic is optional for normal theory intervals. If it is not supplied then the bootstrap estimate of variance is used. The normal intervals also use the bootstrap bias correction.

Interpolation on the normal quantile scale is used when a non-integer order statistic is required. If the order statistic used is the smallest or largest of the R values in boot.out a warning is generated and such intervals should not be considered reliable.

Value

An object of type "bootci" which contains the intervals. It has components

R	The number of bootstrap replicates on which the intervals were based.
t0	The observed value of the statistic on the same scale as the intervals.
call	The call to boot.ci which generated the object.
	It will also contain one or more of the following components depending on the value of type used in the call to bootci.
normal	A matrix of intervals calculated using the normal approximation. It will have 3 columns, the first being the level and the other two being the upper and lower endpoints of the intervals.
basic	The intervals calculated using the basic bootstrap method.
student	The intervals calculated using the studentized bootstrap method.
percent	The intervals calculated using the bootstrap percentile method.
bca	The intervals calculated using the adjusted bootstrap percentile (BCa) method.
	These latter four components will be matrices with 5 columns, the first column containing the level, the next two containing the indices of the order statistics used in the calculations and the final two the calculated endpoints themselves.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*, Chapter 5. Cambridge University Press.

DiCiccio, T.J. and Efron B. (1996) Bootstrap confidence intervals (with Discussion). *Statistical Science*, **11**, 189–228.

Efron, B. (1987) Better bootstrap confidence intervals (with Discussion). *Journal of the American Statistical Association*, **82**, 171–200.

See Also

abc.ci, boot, empinf, norm.ci

Examples

```
# confidence intervals for the city data
ratio <- function(d, w) sum(d$x * w)/sum(d$u * w)
city.boot <- boot(city, ratio, R = 999, stype = "w",sim = "ordinary")
boot.ci(city.boot, conf = c(0.90,0.95),
        type = c("norm","basic","perc","bca"))

# studentized confidence interval for the two sample
# difference of means problem using the final two series
# of the gravity data.
diff.means <- function(d, f)
{    n <- nrow(d)
     gp1 <- 1:table(as.numeric(d$series))[1]
     m1 <- sum(d[gp1,1] * f[gp1])/sum(f[gp1])
     m2 <- sum(d[-gp1,1] * f[-gp1])/sum(f[-gp1])
     ss1 <- sum(d[gp1,1]^2 * f[gp1]) - (m1 *  m1 * sum(f[gp1]))
     ss2 <- sum(d[-gp1,1]^2 * f[-gp1]) - (m2 *  m2 * sum(f[-gp1]))
     c(m1-m2, (ss1+ss2)/(sum(f)-2))
}
grav1 <- gravity[as.numeric(gravity[,2])>=7,]
grav1.boot <- boot(grav1, diff.means, R=999, stype="f", strata=grav1[,2])
boot.ci(grav1.boot, type=c("stud","norm"))

# Nonparametric confidence intervals for mean failure time
# of the air-conditioning data as in Example 5.4 of Davison
# and Hinkley (1997)
mean.fun <- function(d, i)
{    m <- mean(d$hours[i])
     n <- length(i)
     v <- (n-1)*var(d$hours[i])/n^2
     c(m, v)
}
air.boot <- boot(aircondit, mean.fun, R=999)
boot.ci(air.boot, type = c("norm", "basic", "perc", "stud"))

# Now using the log transformation
# There are two ways of doing this and they both give the
# same intervals.

# Method 1
boot.ci(air.boot, type = c("norm", "basic", "perc", "stud"),
        h = log, hdot = function(x) 1/x)
```

```
# Method 2
vt0 <- air.boot$t0[2]/air.boot$t0[1]^2
vt <- air.boot$t[,2]/air.boot$t[,1]^2
boot.ci(air.boot, type = c("norm", "basic", "perc", "stud"),
        t0 = log(air.boot$t0[1]), t = log(air.boot$t[,1]),
        var.t0 = vt0, var.t = vt)
```

`brambles` *Spatial Location of Bramble Canes*

Description

The `brambles` data frame has 823 rows and 3 columns.

The location of living bramble canes in a 9m square plot was recorded. We take 9m to be the unit of distance so that the plot can be thought of as a unit square. The bramble canes were also classified by their age.

Usage

```
brambles
```

Format

This data frame contains the following columns:

`x` The x coordinate of the position of the cane in the plot.

`y` The y coordinate of the position of the cane in the plot.

`age` The age classification of the canes; `0` indicates a newly emerged cane, `1` indicates a one year old cane and `2` indicates a two year old cane.

Source

The data were obtained from

Diggle, P.J. (1983) *Statistical Analysis of Spatial Point Patterns*. Academic Press.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

`breslow` *Smoking Deaths Among Doctors*

Description

The `breslow` data frame has 10 rows and 5 columns.

In 1961 Doll and Hill sent out a questionnaire to all men on the British Medical Register enquiring about their smoking habits. Almost 70% of such men replied. Death certificates were obtained for medical practitioners and causes of death were assigned on the basis of these certificates. The `breslow` data set contains the person-years of observations and deaths from coronary artery disease accumulated during the first ten years of the study.

Usage

```
breslow
```

Format

This data frame contains the following columns:

`age` The mid-point of the 10 year age-group for the doctors.

`smoke` An indicator of whether the doctors smoked (1) or not (0).

`n` The number of person-years in the category.

`y` The number of deaths attributed to coronary artery disease.

`ns` The number of smoker years in the category (`smoke*n`).

Source

The data were obtained from

Breslow, N.E. (1985) Cohort Analysis in Epidemiology. In *A Celebration of Statistics* A.C. Atkinson and S.E. Fienberg (editors), 109–143. Springer-Verlag.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

Doll, R. and Hill, A.B. (1966) Mortality of British doctors in relation to smoking: Observations on coronary thrombosis. *National Cancer Institute Monograph*, **19**, 205-268.

calcium *Calcium Uptake Data*

Description

The `calcium` data frame has 27 rows and 2 columns.

Howard Grimes from the Botany Department, North Carolina State University, conducted an experiment for biochemical analysis of intracellular storage and transport of calcium across plasma membrane. Cells were suspended in a solution of radioactive calcium for a certain length of time and then the amount of radioactive calcium that was absorbed by the cells was measured. The experiment was repeated independently with 9 different times of suspension each replicated 3 times.

Usage

```
calcium
```

Format

This data frame contains the following columns:

`time` The time (in minutes) that the cells were suspended in the solution.

`cal` The amount of calcium uptake (nmoles/mg).

Source

The data were obtained from

Rawlings, J.O. (1988) *Applied Regression Analysis*. Wadsworth and Brooks/Cole Statistics/Probability Series.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

cane *Sugar-cane Disease Data*

Description

The `cane` data frame has 180 rows and 5 columns. The data frame represents a randomized block design with 45 varieties of sugar-cane and 4 blocks.

Usage

```
cane
```

Format

This data frame contains the following columns:

n The total number of shoots in each plot.

r The number of diseased shoots.

x The number of pieces of the stems, out of 50, planted in each plot.

var A factor indicating the variety of sugar-cane in each plot.

block A factor for the blocks.

Details

The aim of the experiment was to classify the varieties into resistant, intermediate and susceptible to a disease called "coal of sugar-cane" (carvao da cana-de-acucar). This is a disease that is common in sugar-cane plantations in certain areas of Brazil.

For each plot, fifty pieces of sugar-cane stem were put in a solution containing the disease agent and then some were planted in the plot. After a fixed period of time, the total number of shoots and the number of diseased shoots were recorded.

Source

The data were kindly supplied by Dr. C.G.B. Demetrio of Escola Superior de Agricultura, Universidade de Sao Paolo, Brazil.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

---

capability *Simulated Manufacturing Process Data*

Description

The `capability` data frame has 75 rows and 1 columns.

The data are simulated successive observations from a process in equilibrium. The process is assumed to have specification limits (5.49, 5.79).

Usage

```
capability
```

Format

This data frame contains the following column:

y The simulated measurements.

Source

The data were obtained from

Bissell, A.F. (1990) How reliable is your capability index? *Applied Statistics*, **39**, 331–340.

References

Canty, A.J. and Davison, A.C. (1996) Implementation of saddlepoint approximations to resampling distributions. To appear in *Computing Science and Statistics; Proceedings of the 28th Symposium on the Interface*.

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

`catsM` *Weight Data for Domestic Cats*

Description

The `catsM` data frame has 97 rows and 3 columns.

144 adult (over 2kg in weight) cats used for experiments with the drug digitalis had their heart and body weight recorded. 47 of the cats were female and 97 were male. The `catsM` data frame consists of the data for the male cats. The full data are in dataset `cats` in package `MASS`.

Usage

```
cats
```

Format

This data frames contain the following columns:

`Sex` A factor for the sex of the cat (levels are `F` and `M`).

`Bwt` Body weight in kg.

`Hwt` Heart weight in g.

Source

The data were obtained from

Fisher, R.A. (1947) The analysis of covariance method for the relation between a part and the whole. *Biometrics*, **3**, 65–68.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

Venables, W.N. and Ripley, B.D. (1994) *Modern Applied Statistics with S-Plus*. Springer-Verlag.

See Also

`cats`

cav *Position of Muscle Caveolae*

Description

The `cav` data frame has 138 rows and 2 columns.

The data gives the positions of the individual caveolae in a square region with sides of length 500 units. This grid was originally on a 2.65mum square of muscle fibre. The data are those points falling in the lower left hand quarter of the region used for the dataset `caveolae.dat` in the `spatial` library by B.D. Ripley (1994) which is available from Statlib

Usage

```
cav
```

Format

This data frame contains the following columns:

x The x coordinate of the caveola's position in the region.

y The y coordinate of the caveola's position in the region.

References

Appleyard, S.T., Witkowski, J.A., Ripley, B.D., Shotton, D.M. and Dubowicz, V. (1985) A novel procedure for pattern analysis of features present on freeze fractured plasma membranes. *Journal of Cell Science*, **74**, 105–117.

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

cd4 *CD4 Counts for HIV-Positive Patients*

Description

The `cd4` data frame has 20 rows and 2 columns.

CD4 cells are carried in the blood as part of the human immune system. One of the effects of the HIV virus is that these cells die. The count of CD4 cells is used in determining the onset of full-blown AIDS in a patient. In this study of the effectiveness of a new anti-viral drug on HIV, 20 HIV-positive patients had their CD4 counts recorded and then were put on a course of treatment with this drug. After using the drug for one year, their CD4 counts were again recorded. The aim of the experiment was to show that patients taking the drug had increased CD4 counts which is not generally seen in HIV-positive patients.

Usage

```
cd4
```

Format

This data frame contains the following columns:

baseline The CD4 counts (in 100's) on admission to the trial.

oneyear The CD4 counts (in 100's) after one year of treatment with the new drug.

Source

The data were obtained from

DiCiccio, T.J. and Efron B. (1996) Bootstrap confidence intervals (with Discussion). *Statistical Science*, **11**, 189–228.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

---

cd4.nested *Nested Bootstrap of cd4 data*

Description

This is an example of a nested bootstrap for the correlation coefficient of the `cd4` data frame. It is used in a practical in Chapter 5 of Davison and Hinkley (1997).

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

See Also

`cd4`

---

censboot *Bootstrap for Censored Data*

Description

This function applies types of bootstrap resampling which have been suggested to deal with right-censored data. It can also do model-based resampling using a Cox regression model.

Usage

```
censboot(data, statistic, R, F.surv, G.surv, strata=matrix(1,n,2),
         sim="ordinary", cox=NULL, index=c(1, 2), ...)
```

Arguments

`data`	The data frame or matrix containing the data. It must have at least two columns, one of which contains the times and the other the censoring indicators. It is allowed to have as many other columns as desired (although efficiency is reduced for large numbers of columns) except for `sim="weird"` when it should only have two columns - the times and censoring indicators. The columns of `data` referenced by the components of `index` are taken to be the times and censoring indicators.
`statistic`	A function which operates on the data frame and returns the required statistic. Its first argument must be the data. Any other arguments that it requires can be passed using the `...{}` argument. In the case of `sim="weird"`, the data passed to `statistic` only contains the times and censoring indicator regardless of the actual number of columns in `data`. In all other cases the data passed to statistic will be of the same form as the original data. When `sim="weird"`, the actual number of observations in the resampled data sets may not be the same as the number in `data`. For this reason, if `sim="weird"` and `strata` is supplied, `statistic` should also take a numeric vector indicating the strata. This allows the statistic to depend on the strata if required.
`R`	The number of bootstrap replicates.
`F.surv`	An object returned from a call to `survfit` giving the survivor function for the data. This is a required argument unless `sim="ordinary"` or `sim="model"` and `cox` is missing.
`G.surv`	Another object returned from a call to `survfit` but with the censoring indicators reversed to give the product-limit estimate of the censoring distribution. Note that for consistency the uncensored times should be reduced by a small amount in the call to `survfit`. This is a required argument whenever `sim="cond"` or when `sim="model"` and `cox` is supplied.
`strata`	The strata used in the calls to `survfit`. It can be a vector or a matrix with 2 columns. If it is a vector then it is assumed to be the strata for the survival distribution, and the censoring distribution is assumed to be the same for all observations. If it is a matrix then the first column is the strata for the survival distribution and the second is the strata for the censoring distribution. When `sim="weird"` only the strata for the survival distribution are used since the censoring times are considered fixed. When `sim="ordinary"`, only one set of strata is used to stratify the observations, this is taken to be the first column of `strata` when it is a matrix.
`sim`	The simulation type. Possible types are `"ordinary"` (case resampling), `"model"` (equivalent to `"ordinary"` if `cox` is missing, otherwise it is model based resampling), `"weird"` (the weird bootstrap - this cannot be used if `cox` is supplied), and `"cond"` (the conditional bootstrap, in which censoring times are resampled from the conditional censoring distribution).
`cox`	An object returned from `coxph`. If it is supplied, then `F.surv` should have been generated by a call of the form `survfit(cox)`.
`index`	A vector of length two giving the positions of the columns in `data` which correspond to the times and censoring indicators respectively.
`...`	Any other arguments which are passed to `statistic`.

Details

The various types of resampling are described in Davison and Hinkley (1997) in sections 3.5 and 7.3. The simplest is case resampling which simply resamples with replacement from the observa-

tions.

The conditional bootstrap simulates failure times from the estimate of the survival distribution. Then, for each observation its simulated censoring time is equal to the observed censoring time if the observation was censored and generated from the estimated censoring distribution conditional on being greater than the observed failure time if the observation was uncensored. If the largest value is censored then it is given a nominal failure time of `Inf` and conversely if it is uncensored it is given a nominal censoring time of `Inf`. This is necessary to allow the largest observation to be in the resamples.

If a Cox regression model is fitted to the data and supplied, then the failure times are generated from the survival distribution using that model. In this case the censoring times can either be simulated from the estimated censoring distribution (`sim="model"`) or from the conditional censoring distribution as in the previous paragraph (`sim="cond"`).

The weird bootstrap holds the censored observations as fixed and also the observed failure times. It then generates the number of events at each failure time using a binomial distribution with mean 1 and denominator the number of failures that could have occurred at that time in the original data set. In our implementation we insist that there is a least one simulated event in each stratum for every bootstrap dataset.

When there are strata involved and `sim` is either `"model"` or `"cond"` the situation becomes more difficult. Since the strata for the survival and censoring distributions are not the same it is possible that for some observations both the simulated failure time and the simulated censoring time are infinite. To see this consider an observation in stratum 1F for the survival distribution and stratum 1G for the censoring distribution. Now if the largest value in stratum 1F is censored it is given a nominal failure time of `Inf`, also if the largest value in stratum 1G is uncensored it is given a nominal censoring time of `Inf` and so both the simulated failure and censoring times could be infinite. When this happens the simulated value is considered to be a failure at the time of the largest observed failure time in the stratum for the survival distribution.

Value

An object of class `"boot"` containing the following components:

`t0`	The value of `statistic` when applied to the original data.
`t`	A matrix of bootstrap replicates of the values of `statistic`.
`R`	The number of bootstrap replicates performed.
`sim`	The simulation type used. This will usually be the input value of `sim` unless that was `"model"` but `cox` was not supplied, in which case it will be `"ordinary"`.
`data`	The data used for the bootstrap. This will generally be the input value of `data` unless `sim="weird"`, in which case it will just be the columns containing the times and the censoring indicators.
`seed`	The value of `.Random.seed` when `censboot` was called.
`statistic`	The input value of `statistic`.
`strata`	The strata used in the resampling. When `sim="ordinary"` this will be a vector which stratifies the observations, when `sim="weird"` it is the strata for the survival distribution and in all other cases it is a matrix containing the strata for the survival distribution and the censoring distribution.
`call`	The original call to `censboot`.

Author(s)

Angelo J. Canty

References

Andersen, P.K., Borgan, O., Gill, R.D. and Keiding, N. (1993) *Statistical Models Based on Counting Processes*. Springer-Verlag.

Burr, D. (1994) A comparison of certain bootstrap confidence intervals in the Cox model. *Journal of the American Statistical Association*, **89**, 1290–1302.

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

Efron, B. (1981) Censored data and the bootstrap. *Journal of the American Statistical Association*, **76**, 312–319.

Hjort, N.L. (1985) Bootstrapping Cox's regression model. Technical report NSF-241, Dept. of Statistics, Stanford University.

See Also

boot, coxph, survfit

Examples

```
data(aml, package="boot")
library(survival)
# Example 3.9 of Davison and Hinkley (1997) does a bootstrap on some
# remission times for patients with a type of leukaemia.  The patients
# were divided into those who received maintenance chemotherapy and
# those who did not.  Here we are interested in the median remission
# time for the two groups.
aml.fun <- function(data) {
     surv <- survfit(Surv(time, cens)~group, data=data)
     out <- NULL
     st <- 1
     for (s in 1:length(surv$strata)) {
          inds <- st:(st+surv$strata[s]-1)
          md <- min(surv$time[inds[1-surv$surv[inds]>=0.5]])
          st <- st+surv$strata[s]
          out <- c(out,md)
     }
     out
}
aml.case <- censboot(aml,aml.fun,R=499,strata=aml$group)

# Now we will look at the same statistic using the conditional
# bootstrap and the weird bootstrap.  For the conditional bootstrap
# the survival distribution is stratified but the censoring
# distribution is not.

aml.s1 <- survfit(Surv(time,cens)~group, data=aml)
aml.s2 <- survfit(Surv(time-0.001*cens,1-cens)~1, data=aml)
aml.cond <- censboot(aml,aml.fun,R=499,strata=aml$group,
     F.surv=aml.s1,G.surv=aml.s2,sim="cond")

# For the weird bootstrap we must redefine our function slightly since
```

```
# the data will not contain the group number.
aml.fun1 <- function(data,str) {
     surv <- survfit(Surv(data[,1],data[,2])~str)
     out <- NULL
     st <- 1
     for (s in 1:length(surv$strata)) {
          inds <- st:(st+surv$strata[s]-1)
          md <- min(surv$time[inds[1-surv$surv[inds]>=0.5]])
          st <- st+surv$strata[s]
          out <- c(out,md)
     }
}
aml.wei <- censboot(cbind(aml$time,aml$cens),aml.fun1,R=499,
     strata=aml$group, F.surv=aml.s1,sim="weird")

# Now for an example where a cox regression model has been fitted
# the data we will look at the melanoma data of Example 7.6 from
# Davison and Hinkley (1997).  The fitted model assumes that there
# is a different survival distribution for the ulcerated and
# non-ulcerated groups but that the thickness of the tumour has a
# common effect.  We will also assume that the censoring distribution
# is different in different age groups.  The statistic of interest
# is the linear predictor.  This is returned as the values at a
# number of equally spaced points in the range of interest.
data(melanoma, package="boot")
library(splines)# for ns
mel.cox <- coxph(Surv(time,status==1)~ns(thickness,df=4)+strata(ulcer),
     data=melanoma)
mel.surv <- survfit(mel.cox)
agec <- cut(melanoma$age,c(0,39,49,59,69,100))
mel.cens <- survfit(Surv(time-0.001*(status==1),status!=1)~
     strata(agec),data=melanoma)
mel.fun <- function(d) {
     t1 <- ns(d$thickness,df=4)
     cox <- coxph(Surv(d$time,d$status==1) ~ t1+strata(d$ulcer))
     ind <- !duplicated(d$thickness)
     u <- d$thickness[!ind]
     eta <- cox$linear.predictors[!ind]
     sp <- smooth.spline(u,eta,df=20)
     th <- seq(from=0.25,to=10,by=0.25)
     predict(sp,th)$y
}
mel.str<-cbind(melanoma$ulcer,agec)
# this is slow!
system.time(mel.mod <- censboot(melanoma,mel.fun,R=999,F.surv=mel.surv,
     G.surv=mel.cens,cox=mel.cox,strata=mel.str,sim="model"))
# To plot the original predictor and a 95% pointwise envelope for it
mel.env <- envelope(mel.mod)$point
plot(seq(0.25,10,by=0.25),mel.env[1,], ylim=c(-2,2),
     xlab="thickness (mm)", ylab="linear predictor",type="n")
lines(seq(0.25,10,by=0.25),mel.env[1,],lty=2)
lines(seq(0.25,10,by=0.25),mel.env[2,],lty=2)
lines(seq(0.25,10,by=0.25),mel.mod$t0,lty=1)
```

`channing` *Channing House Data*

Description

The `channing` data frame has 462 rows and 5 columns.

Channing House is a retirement centre in Palo Alto, California. These data were collected between the opening of the house in 1964 until July 1, 1975. In that time 97 men and 365 women passed through the centre. For each of these, their age on entry and also on leaving or death was recorded. A large number of the observations were censored mainly due to the resident being alive on July 1, 1975 when the data was collected. Over the time of the study 130 women and 46 men died at Channing House. Differences between the survival of the sexes, taking age into account, was one of the primary concerns of this study.

Usage

```
channing
```

Format

This data frame contains the following columns:

`sex` A factor for the sex of each resident (`"Male"` or `"Female"`).

`entry` The residents age (in months) on entry to the centre

`exit` The age (in months) of the resident on death, leaving the centre or July 1, 1975 whichever event occurred first.

`time` The length of time (in months) that the resident spent at Channing House. (`time=exit-entry`)

`cens` The indicator of right censoring. 1 indicates that the resident died at Channing House, 0 indicates that they left the house prior to July 1, 1975 or that they were still alive and living in the centre at that date.

Source

The data were obtained from

Hyde, J. (1980) Testing survival with incomplete observations. *Biostatistics Casebook*. R.G. Miller, B. Efron, B.W. Brown and L.E. Moses (editors), 31–46. John Wiley.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

`claridge` *Genetic Links to Left-handedness*

Description

The `claridge` data frame has 37 rows and 2 columns.

The data are from an experiment which was designed to look for a relationship between a certain genetic characteristic and handedness. The 37 subjects were women who had a son with mental retardation due to inheriting a defective X-chromosome. For each such mother a genetic measurement of their DNA was made. Larger values of this measurement are known to be linked to the defective gene and it was hypothesized that larger values might also be linked to a progressive shift away from right-handednesss. Each woman also filled in a questionnaire regarding which hand they used for various tasks. From these questionnaires a measure of hand preference was found for each mother. The scale of this measure goes from 1, indicating someone who always favours their right hand, to 8, indicating someone who always favours their left hand. Between these two extremes are people who favour one hand for some tasks and the other for other tasks.

Usage

```
claridge
```

Format

This data frame contains the following columns:

dnan The genetic measurement on each woman's DNA.

hand The measure of left-handedness on an integer scale from 1 to 8.

Source

The data were kindly made available by Dr. Gordon S. Claridge from the Department of Experimental Psychology, University of Oxford.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

---

cloth *Number of Flaws in Cloth*

Description

The `cloth` data frame has 32 rows and 2 columns.

Usage

```
cloth
```

Format

This data frame contains the following columns:

x The length of the roll of cloth.

y The number of flaws found in the roll.

Source

The data were obtained from

Bissell, A.F. (1972) A negative binomial model with varying element size. *Biometrika*, **59**, 435–441.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

---

co.transfer *Carbon Monoxide Transfer*

Description

The `co.transfer` data frame has 7 rows and 2 columns. Seven smokers with chickenpox had their levels of carbon monoxide transfer measured on entry to hospital and then again after 1 week. The main question being whether one week of hospitalization has changed the carbon monoxide transfer factor.

Usage

```
co.transfer
```

Format

This data frame contains the following columns:

`entry` Carbon monoxide transfer factor on entry to hospital.

`week` Carbon monoxide transfer one week after admittance to hospital.

Source

The data were obtained from

Hand, D.J., Daly, F., Lunn, A.D., McConway, K.J. and Ostrowski, E (1994) *A Handbook of Small Data Sets*. Chapman and Hall.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

Ellis, M.E., Neal, K.R. and Webb, A.K. (1987) Is smoking a risk factor for pneumonia in patients with chickenpox? *British Medical Journal*, **294**, 1002.

coal *Dates of Coal Mining Disasters*

Description

The `coal` data frame has 191 rows and 1 columns.

This data frame gives the dates of 191 explosions in coal mines which resulted in 10 or more fatalities. The time span of the data is from March 15, 1851 until March 22 1962.

Usage

```
coal
```

Format

This data frame contains the following column:

`date` The date of the disaster. The integer part of `date` gives the year. The day is represented as the fraction of the year that had elapsed on that day.

Source

The data were obtained from

Hand, D.J., Daly, F., Lunn, A.D., McConway, K.J. and Ostrowski, E. (1994) *A Handbook of Small Data Sets*, Chapman and Hall.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

Jarrett, R.G. (1979) A note on the intervals between coal-mining disasters. *Biometrika*, **66**, 191-193.

control *Control Variate Calculations*

Description

This function will find control variate estimates from a bootstrap output object. It can either find the adjusted bias estimate using post-simulation balancing or it can estimate the bias, variance, third cumulant and quantiles, using the linear approximation as a control variate.

Usage

```
control(boot.out, L = NULL, distn = NULL, index = 1, t0 = NULL,
        t = NULL, bias.adj = FALSE, alpha = NULL, ...)
```

Arguments

`boot.out`	A bootstrap output object returned from `boot`. The bootstrap replicates must have been generated using the usual nonparametric bootstrap.
`L`	The empirical influence values for the statistic of interest. If `L` is not supplied then `empinf` is called to calculate them from `boot.out`.
`distn`	If present this must be the output from `smooth.spline` giving the distribution function of the linear approximation. This is used only if `bias.adj` is `FALSE`. Normally this would be found using a saddlepoint approximation. If it is not supplied in that case then it is calculated by `saddle.distn`.
`index`	The index of the variable of interest in the output of `boot.out$statistic`.
`t0`	The observed value of the statistic of interest on the original data set `boot.out$data`. This argument is used only if `bias.adj` is `FALSE`. The input value is ignored if `t` is not also supplied. The default value is is `boot.out$t0[index]`.
`t`	The bootstrap replicate values of the statistic of interest. This argument is used only if `bias.adj` is `FALSE`. The input is ignored if `t0` is not supplied also. The default value is `boot.out$t[,index]`.
`bias.adj`	A logical variable which if `TRUE` specifies that the adjusted bias estimate using post-simulation balance is all that is required. If `bias.adj` is `FALSE` (default) then the linear approximation to the statistic is calculated and used as a control variate in estimates of the bias, variance and third cumulant as well as quantiles.
`alpha`	The alpha levels for the required quantiles if `bias.adj` is `FALSE`.
`...`	Any additional arguments that `boot.out$statistic` requires. These are passed unchanged every time `boot.out$statistic` is called. `boot.out$statistic` is called once if `bias.adj` is `TRUE`, otherwise it may be called by `empinf` for empirical influence calculations if `L` is not supplied.

Details

If `bias.adj` is `FALSE` then the linear approximation to the statistic is found and evaluated at each bootstrap replicate. Then using the equation $T^* = Tl^* + (T^* - Tl^*)$, moment estimates can be found. For quantile estimation the distribution of the linear approximation to `t` is approximated very accurately by saddlepoint methods, this is then combined with the bootstrap replicates to approximate the bootstrap distribution of `t` and hence to estimate the bootstrap quantiles of `t`.

Value

If `bias.adj` is `TRUE` then the returned value is the adjusted bias estimate.

If `bias.adj` is `FALSE` then the returned value is a list with the following components

`L`	The empirical influence values used. These are the input values if supplied, and otherwise they are the values calculated by `empinf`.
`tL`	The linear approximations to the bootstrap replicates `t` of the statistic of interest.
`bias`	The control estimate of bias using the linear approximation to `t` as a control variate.
`var`	The control estimate of variance using the linear approximation to `t` as a control variate.

`k3` The control estimate of the third cumulant using the linear approximation to `t` as a control variate.

`quantiles` A matrix with two columns; the first column are the alpha levels used for the quantiles and the second column gives the corresponding control estimates of the quantiles using the linear approximation to `t` as a control variate.

`distn` An output object from `smooth.spline` describing the saddlepoint approximation to the bootstrap distribution of the linear approximation to `t`. If `distn` was supplied on input then this is the same as the input otherwise it is calculated by a call to `saddle.distn`.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

Davison, A.C., Hinkley, D.V. and Schechtman, E. (1986) Efficient bootstrap simulation. *Biometrika*, **73**, 555–566.

Efron, B. (1990) More efficient bootstrap computations. *Journal of the American Statistical Association*, **55**, 79–89.

See Also

`boot`, `empinf`, `k3.linear`, `linear.approx`, `saddle.distn`, `smooth.spline`, `var.linear`

Examples

```
# Use of control variates for the variance of the air-conditioning data
mean.fun <- function(d, i)
{    m <- mean(d$hours[i])
     n <- nrow(d)
     v <- (n-1)*var(d$hours[i])/n^2
     c(m, v)
}
air.boot <- boot(aircondit, mean.fun, R = 999)
control(air.boot, index = 2, bias.adj = TRUE)
air.cont <- control(air.boot, index = 2)
# Now let us try the variance on the log scale.
air.cont1 <- control(air.boot, t0=log(air.boot$t0[2]),
                     t=log(air.boot$t[,2]))
```

---

corr *Correlation Coefficient*

Description

Calculates the weighted correlation given a data set and a set of weights.

Usage

```
corr(d, w=rep(1,nrow(d))/nrow(d))
```

Arguments

`d`	A matrix with two columns corresponding to the two variables whose correlation we wish to calculate.
`w`	A vector of weights to be applied to each pair of observations. The default is equal weights for each pair. Normalization takes place within the function so `sum(w)` need not equal 1.

Details

This function finds the correlation coefficient in weighted form. This is often useful in bootstrap methods since it allows for numerical differentiation to get the empirical influence values. It is also necessary to have the statistic in this form to find ABC intervals.

Value

The correlation coefficient between `d[,1]` and `d[,2]`.

See Also

`cor`

---

cum3 *Calculate Third Order Cumulants*

Description

Calculates an estimate of the third cumulant, or skewness, of a vector. Also, if more than one vector is specified, a product-moment of order 3 is estimated.

Usage

```
cum3(a, b=a, c=a, unbiased=TRUE)
```

Arguments

`a`	A vector of observations.
`b`	Another vector of observations, if not supplied it is set to the value of `a`. If supplied then it must be the same length as `a`.
`c`	Another vector of observations, if not supplied it is set to the value of `a`. If supplied then it must be the same length as `a`.
`unbiased`	A logical value indicating whether the unbiased estimator should be used.

Details

The unbiased estimator uses a multiplier of `n/((n-1)*(n-2))` where `n` is the sample size, if `unbiased` is `FALSE` then a multiplier of `1/n` is used. This is multiplied by `sum((a-mean(a))*(b-mean(b))*(c-mean(c)))` to give the required estimate.

Value

The required estimate.

`cv.glm` *Cross-validation for Generalized Linear Models*

Description

This function calculates the estimated K-fold cross-validation prediction error for generalized linear models.

Usage

```
cv.glm(data, glmfit, cost, K)
```

Arguments

`data`	A matrix or data frame containing the data. The rows should be cases and the columns correspond to variables, one of which is the response.
`glmfit`	An object of class `"glm"` containing the results of a generalized linear model fitted to `data`.
`cost`	A function of two vector arguments specifying the cost function for the cross-validation. The first argument to `cost` should correspond to the observed responses and the second argument should correspond to the predicted or fitted responses from the generalized linear model. `cost` must return a non-negative scalar value. The defualt is the average squared error function.
`K`	The number of groups into which the data should be split to estimate the cross-validation prediction error. The value of `K` must be such that all groups are of approximately equal size. If the supplied value of `K` does not satisfy this criterion then it will be set to the closest integer which does and a warning is generated specifying the value of `K` used. The default is to set `K` equal to the number of observations in `data` which gives the usual leave-one-out cross-validation.

Details

The data is divided randomly into `K` groups. For each group the generalized linear model is fit to `data` omitting that group, then the function `cost` is applied to the observed responses in the group that was omitted from the fit and the prediction made by the fitted models for those observations.

When `K` is the number of observations leave-one-out cross-validation is used and all the possible splits of the data are used. When `K` is less than the number of observations the `K` splits to be used are found by randomly partitioning the data into `K` groups of approximately equal size. In this latter case a certain amount of bias is introduced. This can be reduced by using a simple adjustment (see equation 6.48 in Davison and Hinkley, 1997). The second value returned in `delta` is the estimate adjusted by this method.

Value

The returned value is a list with the following components.

`call`	The original call to `cv.glm`.
`K`	The value of `K` used for the K-fold cross validation.

`delta` A vector of length two. The first component is the raw cross-validation estimate of prediction error. The second component is the adjusted cross-validation estimate. The adjustment is designed to compensate for the bias introduced by not using leave-one-out cross-validation.

`seed` The value of `.Random.seed` when `cv.glm` was called.

Side Effects

The value of `.Random.seed` is updated.

References

Brieman, L., Friedman, J.H., Olshen, R.A. and Stone, C.J. (1984) *Classification and Regression Trees*. Wadsworth.

Burman, P. (1989) A comparitive study of ordinary cross-validation, *v*-fold cross-validation and repeated learning-testing methods. *Biometrika*, **76**, 503–514

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

Efron, B. (1986) How biased is the apparent error rate of a prediction rule? *Journal of the American Statistical Association*, **81**, 461–470.

Stone, M. (1974) Cross-validation choice and assessment of statistical predictions (with Discussion). *Journal of the Royal Statistical Society, B*, **36**, 111–147.

See Also

`glm`, `glm.diag`, `predict`

Examples

```
# leave-one-out and 6-fold cross-validation prediction error for
# the mammals data set.
data(mammals, package="MASS")
mammals.glm <- glm(log(brain)~log(body),data=mammals)
cv.err <- cv.glm(mammals,mammals.glm)
cv.err.6 <- cv.glm(mammals, mammals.glm, K=6)

# As this is a linear model we could calculate the leave-one-out
# cross-validation estimate without any extra model-fitting.
muhat <- mammals.glm$fitted
mammals.diag <- glm.diag(mammals.glm)
cv.err <- mean((mammals.glm$y-muhat)^2/(1-mammals.diag$h)^2)

# leave-one-out and 11-fold cross-validation prediction error for
# the nodal data set.  Since the response is a binary variable an
# appropriate cost function is
cost <- function(r, pi=0) mean(abs(r-pi)>0.5)

nodal.glm <- glm(r~stage+xray+acid,binomial,data=nodal)
cv.err <- cv.glm(nodal, nodal.glm, cost, K=nrow(nodal))$delta
cv.11.err <- cv.glm(nodal, nodal.glm, cost, K=11)$delta
```

darwin *Darwin's Plant Height Differences*

Description

The `darwin` data frame has 15 rows and 1 columns.

Charles Darwin conducted an experiment to examine the superiority of cross-fertilized plants over self-fertilized plants. 15 pairs of plants were used. Each pair consisted of one cross-fertilized plant and one self-fertilized plant which germinated at the same time and grew in the same pot. The plants were measured at a fixed time after planting and the difference in heights between the cross- and self-fertilized plants are recorded in eighths of an inch.

Usage

```
darwin
```

Format

This data frame contains the following column:

`y` The difference in heights for the pairs of plants (in units of 0.125 inches).

Source

The data were obtained from

Fisher, R.A. (1935) *Design of Experiments*. Oliver and Boyd.

References

Darwin, C. (1876) *The Effects of Cross- and Self-fertilisation in the Vegetable Kingdom*. John Murray.

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

dogs *Cardiac Data for Domestic Dogs*

Description

The `dogs` data frame has 7 rows and 2 columns.

Data on the cardiac oxygen consumption and left ventricular pressure were gathered on 7 domestic dogs.

Usage

```
dogs
```

Format

This data frame contains the following columns:

mvo Cardiac Oxygen Consumption

lvp Left Ventricular Pressure

References

Davison, A. C. and Hinkley, D. V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

`downs.bc` *Incidence of Down's Syndrome in British Columbia*

Description

The `downs.bc` data frame has 30 rows and 3 columns.

Down's syndrome is a genetic disorder caused by an extra chromosome 21 or a part of chromosome 21 being translocated to another chromosome. The incidence of Down's syndrome is highly dependent on the mother's age and rises sharply after age 30. In the 1960's a large scale study of the effect of maternal age on the incidence of Down's syndrome was conducted at the British Columbia Health Surveillance Registry. These are the data which was collected in that study.

Mothers were classified by age. Most groups correspond to the age in years but the first group comprises all mothers with ages in the range 15-17 and the last is those with ages 46-49. No data for mothers over 50 or below 15 were collected.

Usage

```
downs.bc
```

Format

This data frame contains the following columns:

`age` The average age of all mothers in the age category.

`m` The total number of live births to mothers in the age category.

`r` The number of cases of Down's syndrome.

Source

The data were obtained from

Geyer, C.J. (1991) Constrained maximum likelihood exemplified by isotonic convex logistic regression. *Journal of the American Statistical Association*, **86**, 717–724.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

ducks *Behavioral and Plumage Characteristics of Hybrid Ducks*

Description

The `ducks` data frame has 11 rows and 2 columns.

Each row of the data frame represents a male duck who is a second generation cross of mallard and pintail ducks. For 11 such ducks a behavioural and plumage index were calculated. These were measured on scales devised for this experiment which was to examine whether there was any link between which species the ducks resembled physically and which they resembled in behaviour. The scale for the physical appearance ranged from 0 (identical in appearance to a mallard) to 20 (identical to a pintail). The behavioural traits of the ducks were on a scale from 0 to 15 with lower numbers indicating closer to mallard-like in behaviour.

Usage

```
ducks
```

Format

This data frame contains the following columns:

`plumage` The index of physical appearance based on the plumage of individual ducks.

`behaviour` The index of behavioural characteristics of the ducks.

Source

The data were obtained from

Larsen, R.J. and Marx, M.L. (1986) *An Introduction to Mathematical Statistics and its Applications* (Second Edition). Prentice-Hall.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

Sharpe, R.S., and Johnsgard, P.A. (1966) Inheritance of behavioral characters in F_2 mallard x pintail (*Anas Platyrhynchos L. x Anas Acuta L.*) hybrids. *Behaviour*, **27**, 259-272.

EEF.profile *Empirical Likelihoods*

Description

Construct the empirical log likelihood or empirical exponential family log likelihood for a mean.

Usage

```
EEF.profile(y, tmin=min(y) + 0.1, tmax=max(y) - 0.1, n.t=25,
            u=function(y, t) { y-t})
EL.profile(y, tmin = min(y) + 0.1, tmax = max(y) - 0.1, n.t = 25,
           u = function(y, t) y - t)
```

Arguments

`y`	A vector or matrix of data
`tmin`	The minimum value of the range over which the likelihood should be computed. This must be larger than `min(y)`.
`tmax`	The maximum value of the range over which the likelihood should be computed. This must be smaller than `max(y)`.
`n.t`	The number of points between `tmin` and `tmax` at which the value of the log-liklihood should be computed.
`u`	A function of the data and the parameter.

Details

These functions calculate the log likelihood for a mean using either an empirical likelihood or an empirical exponential family likelihood. They are supplied as part of the package `boot` for demonstration purposes with the practicals in chapter 10 of Davison and Hinkley (1997). The functions are not intended for general use and are not supported as part of the `boot`package. For more general and more robust code to calculate empirical likelihoods see Professor A. B. Owen's empirical likelihood home page at the URL `http://www-stat.stanford.edu/~owen/empirical/`.

Value

A matrix with `n.t` rows. The first column contains the values of the parameter used. The second colmn of the output of `EL.profile` contains the values of the empirical log likelihood. The second and third columns of the output of `EEF.profile` contain two versions of the empirical exponential family log-likelihood. The final column of the output matrix contains the values of the Lagrange multiplier used in the optimization procedure.

Author(s)

Angelo J. Canty

References

Davison, A. C. and Hinkley, D. V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

empinf *Empirical Influence Values*

Description

This function calculates the empirical influence values for a statistic applied to a data set. It allows four types of calculation, namely the infinitesimal jackknife (using numerical differentiation), the usual jackknife estimates, the "positive" jackknife estimates and a method which estimates the empirical influence values using regression of bootstrap replicates of the statistic. All methods can be used with one or more samples.

Usage

```
empinf(boot.out = NULL, data = NULL, statistic = NULL,
       type = NULL, stype = NULL ,index = 1, t = NULL,
       strata = rep(1, n), eps = 0.001, ...)
```

Arguments

`boot.out`	A bootstrap object created by the function `boot`. If `type` is `"reg"` then this argument is required. For any of the other types it is an optional argument. If it is included when optional then the values of `data`, `statistic`, `stype`, and `strata` are taken from the components of `boot.out` and any values passed to `empinf` directly are ignored.
`data`	A vector, matrix or data frame containing the data for which empirical influence values are required. It is a required argument if `boot.out` is not supplied. If `boot.out` is supplied then `data` is set to `boot.out$data` and any value supplied is ignored.
`statistic`	The statistic for which empirical influence values are required. It must be a function of at least two arguments, the data set and a vector of weights, frequencies or indices. The nature of the second argument is given by the value of `stype`. Any other arguments that it takes must be supplied to `empinf` and will be passed to `statistic` unchanged. This is a required argument if `boot.out` is not supplied, otherwise its value is taken from `boot.out` and any value supplied here will be ignored.
`type`	The calculation type to be used for the empirical influence values. Possible values of `type` are `"inf"` (infinitesimal jackknife), `"jack"` (usual jackknife), `"pos"` (positive jackknife), and `"reg"` (regression estimation). The default value depends on the other arguments. If `t` is supplied then the default value of `type` is `"reg"` and `boot.out` should be present so that its frequency array can be found. It `t` is not supplied then if `stype` is `"w"`, the default value of `type` is `"inf"`; otherwise, if `boot.out` is present the default is `"reg"`. If none of these conditions apply then the default is `"jack"`. Note that it is an error for `type` to be `"reg"` if `boot.out` is missing or to be `"inf"` if `stype` is not `"w"`.
`stype`	A character variable giving the nature of the second argument to `statistic`. It can take on three values: `"w"` (weights), `"f"` (frequencies), or `"i"` (indices). If `boot.out` is supplied the value of `stype` is set to `boot.out$stype` and any value supplied here is ignored. Otherwise it is an optional argument which defaults to `"w"`. If `type` is `"inf"` then `stype` MUST be `"w"`.

index	An integer giving the position of the variable of interest in the output of statistic.
t	A vector of length boot.out$R which gives the bootstrap replicates of the statistic of interest. t is used only when type is reg and it defaults to boot.out$t[,index].
strata	An integer vector or a factor specifying the strata for multi-sample problems. If boot.out is supplied the value of strata is set to boot.out$strata. Otherwise it is an optional argument which has default corresponding to the single sample situation.
eps	This argument is used only if type is "inf". In that case the value of epsilon to be used for numerical differentiation will be eps divided by the number of observations in data.
...	Any other arguments that statistic takes. They will be passed unchanged to statistic every time that it is called.

Details

If type is "inf" then numerical differentiation is used to approximate the empirical influence values. This makes sense only for statistics which are written in weighted form (i.e. stype is "w"). If type is "jack" then the usual leave-one-out jackknife estimates of the empirical influence are returned. If type is "pos" then the positive (include-one-twice) jackknife values are used. If type is "reg" then a bootstrap object must be supplied. The regression method then works by regressing the bootstrap replicates of statistic on the frequency array from which they were derived. The bootstrap frequency array is obtained through a call to boot.array. Further details of the methods are given in Section 2.7 of Davison and Hinkley (1997).

Empirical influence values are often used frequently in nonparametric bootstrap applications. For this reason many other functions call empinf when they are required. Some examples of their use are for nonparametric delta estimates of variance, BCa intervals and finding linear approximations to statistics for use as control variates. They are also used for antithetic bootstrap resampling.

Value

A vector of the empirical influence values of statistic applied to data. The values will be in the same order as the observations in data.

Warning

All arguments to empinf must be passed using the name = value convention. If this is not followed then unpredictable errors can occur.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

Efron, B. (1982) *The Jackknife, the Bootstrap and Other Resampling Plans*. CBMS-NSF Regional Conference Series in Applied Mathematics, **38**, SIAM.

Fernholtz, L.T. (1983) *von Mises Calculus for Statistical Functionals*. Lecture Notes in Statistics, **19**, Springer-Verlag.

See Also

boot, boot.array, boot.ci, control, jack.after.boot, linear.approx, var.linear

Examples

```
# The empirical influence values for the ratio of means in
# the city data.
ratio <- function(d, w) sum(d$x *w)/sum(d$u*w)
empinf(data=city,statistic=ratio)
city.boot <- boot(city,ratio,499,stype="w")
empinf(boot.out=city.boot,type="reg")

# A statistic that may be of interest in the difference of means
# problem is the t-statistic for testing equality of means.  In
# the bootstrap we get replicates of the difference of means and
# the variance of that statistic and then want to use this output
# to get the empirical influence values of the t-statistic.
grav1 <- gravity[as.numeric(gravity[,2])>=7,]
grav.fun <- function(dat, w)
{    strata <- tapply(dat[, 2], as.numeric(dat[, 2]))
     d <- dat[, 1]
     ns <- tabulate(strata)
     w <- w/tapply(w, strata, sum)[strata]
     mns <- tapply(d * w, strata, sum)
     mn2 <- tapply(d * d * w, strata, sum)
     s2hat <- sum((mn2 - mns^2)/ns)
     c(mns[2]-mns[1],s2hat)
}

grav.boot <- boot(grav1, grav.fun, R=499, stype="w", strata=grav1[,2])

# Since the statistic of interest is a function of the bootstrap
# statistics, we must calculate the bootstrap replicates and pass
# them to empinf using the t argument.
grav.z <- (grav.boot$t[,1]-grav.boot$t0[1])/sqrt(grav.boot$t[,2])
empinf(boot.out=grav.boot,t=grav.z)
```

---

envelope *Confidence Envelopes for Curves*

Description

This function calculates overall and pointwise confidence envelopes for a curve based on bootstrap replicates of the curve evaluated at a number of fixed points.

Usage

```
envelope(boot.out=NULL, mat=NULL, level=0.95, index=1:ncol(mat))
```

Arguments

boot.out	An object of class "boot" for which boot.out$t contains the replicates of the curve at a number of fixed points.
mat	A matrix of bootstrap replicates of the values of the curve at a number of fixed points. This is a required argument if boot.out is not supplied and is set to boot.out$t otherwise.
level	The confidence level of the envelopes required. The default is to find 95% confidence envelopes. It can be a scalar or a vector of length 2. If it is scalar then both the pointwise and the overall envelopes are found at that level. If is a vector then the first element gives the level for the pointwise envelope and the second gives the level for the overall envelope.
index	The numbers of the columns of mat which contain the bootstrap replicates. This can be used to ensure that other statistics which may have been calculated in the bootstrap are not considered as values of the function.

Details

The pointwise envelope is found by simply looking at the quantiles of the replicates at each point. The overall error for that envelope is then calculated using equation (4.17) of Davison and Hinkley (1997). A sequence of pointwise envelopes is then found until one of them has overall error approximately equal to the level required. If no such envelope can be found then the envelope returned will just contain the extreme values of each column of mat.

Value

A list with the following components :

point	A matrix with two rows corresponding to the values of the upper and lower pointwise confidence envelope at the same points as the bootstrap replicates were calculated.
overall	A matrix similar to point but containing the envelope which controls the overall error.
k.pt	The quantiles used for the pointwise envelope.
err.pt	A vector with two components, the first gives the pointwise error rate for the pointwise envelope, and the second the overall error rate for that envelope.
k.ov	The quantiles used for the overall envelope.
err.ov	A vector with two components, the first gives the pointwise error rate for the overall envelope, and the second the overall error rate for that envelope.
err.nom	A vector of length 2 giving the nominal error rates for the pointwise and the overall envelopes.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

See Also

boot, boot.ci

Examples

```
# Testing whether the final series of measurements of the gravity data
# may come from a normal distribution.  This is done in Examples 4.7
# and 4.8 of Davison and Hinkley (1997).
grav1 <- gravity$g[gravity$series==8]
grav.z <- (grav1-mean(grav1))/sqrt(var(grav1))
grav.gen <- function(dat,mle)
     rnorm(length(dat))
grav.qqboot <- boot(grav.z,sort,R=999,sim="parametric",ran.gen=grav.gen)
grav.qq <- qqnorm(grav.z,plot=FALSE)
grav.qq <- lapply(grav.qq,sort)
plot(grav.qq,ylim=c(-3.5,3.5),ylab="Studentized Order Statistics",
     xlab="Normal Quantiles")
grav.env <- envelope(grav.qqboot,level=0.9)
lines(grav.qq$x,grav.env$point[1,],lty=4)
lines(grav.qq$x,grav.env$point[2,],lty=4)
lines(grav.qq$x,grav.env$overall[1,],lty=1)
lines(grav.qq$x,grav.env$overall[2,],lty=1)
```

---

exp.tilt *Exponential Tilting*

Description

This function calculates exponentially tilted multinomial distributions such that the resampling distributions of the linear approximation to a statistic have the required means.

Usage

```
exp.tilt(L, theta=NULL, t0=0, lambda=NULL,
         strata=rep(1, length(L)))
```

Arguments

`L`	The empirical influence values for the statistic of interest based on the observed data. The length of `L` should be the same as the size of the original data set. Typically `L` will be calculated by a call to `empinf`.
`theta`	The value at which the tilted distribution is to be centred. This is not required if `lambda` is supplied but is needed otherwise.
`t0`	The current value of the statistic. The default is that the statistic equals 0.
`lambda`	The Lagrange multiplier(s). For each value of `lambda` a multinomial distribution is found with probabilities proportional to `exp(lambda * L)`. In general `lambda` is not known and so `theta` would be supplied, and the corresponding value of `lambda` found. If both `lambda` and `theta` are supplied then `lambda` is ignored and the multipliers for tilting to `theta` are found.
`strata`	A vector or factor of the same length as `L` giving the strata for the observed data and the empirical influence values `L`.

Details

Exponential tilting involves finding a set of weights for a data set to ensure that the bootstrap distribution of the linear approximation to a statistic of interest has mean `theta`. The weights chosen to achieve this are given by `p[j]` proportional to `exp(lambda*L[j]/n)`, where `n` is the number of data points. `lambda` is then chosen to make the mean of the bootstrap distribution, of the linear approximation to the statistic of interest, equal to the the required value `theta`. Thus `lambda` is defined as the solution of a nonlinear equation. The equation is solved by minimizing the Euclidean distance between the left and right hand sides of the equation using the function `nlmin`. If this minimum is not equal to zero then the method fails.

Typically exponential tilting is used to find suitable weights for importance resampling. If a small tail probability or quantile of the distribution of the statistic of interest is required then a more efficient simulation is to centre the resampling distribution close to the point of interest and then use the functions `imp.prob` or `imp.quantile` to estimate the required quantity.

Another method of achieving a similar shifting of the distribution is through the use of `smooth.f`. The function `tilt.boot` uses `exp.tilt` or `smooth.f` to find the weights for a tilted bootstrap.

Value

A list with the following components :

`p`	The tilted probabilities. There will be `m` distributions where `m` is the length of `theta` (or `lambda` if supplied). If `m` is 1 then `p` is a vector of `length(L)` probabilities. If `m` is greater than 1 then `p` is a matrix with `m` rows, each of which contain `length(L)` probabilities. In this case the vector `p[i,]` is the distribution tilted to `theta[i]`. `p` is in the form required by the argument `weights` of the function `boot` for importance resampling.
`lambda`	The Lagrange multiplier used in the equation to determine the tilted probabilities. `lambda` is a vector of the same length as `theta`.
`theta`	The values of `theta` to which the distributions have been tilted. In general this will be the input value of `theta` but if `lambda` was supplied then this is the vector of the corresponding `theta` values.

References

Davison, A. C. and Hinkley, D. V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

Efron, B. (1981) Nonparametric standard errors and confidence intervals (with Discussion). *Canadian Journal of Statistics*, **9**, 139–172.

See Also

`empinf`, `imp.prob`, `imp.quantile`, `optim`, `smooth.f`, `tilt.boot`

Examples

```
# Example 9.8 of Davison and Hinkley (1997) requires tilting the resampling
# distribution of the studentized statistic to be centred at the observed
# value of the test statistic 1.84.  This can be achieved as follows.
grav1 <- gravity[as.numeric(gravity[,2])>=7,]
grav.fun <- function(dat, w, orig)
{    strata <- tapply(dat[, 2], as.numeric(dat[, 2]))
     d <- dat[, 1]
```

```
      ns <- tabulate(strata)
      w <- w/tapply(w, strata, sum)[strata]
      mns <- tapply(d * w, strata, sum)
      mn2 <- tapply(d * d * w, strata, sum)
      s2hat <- sum((mn2 - mns^2)/ns)
      as.vector(c(mns[2]-mns[1],s2hat,(mns[2]-mns[1]-orig)/sqrt(s2hat)))
}
grav.z0 <- grav.fun(grav1,rep(1,26),0)
grav.L <- empinf(data=grav1, statistic=grav.fun, stype="w",
                 strata=grav1[,2], index=3, orig=grav.z0[1])
grav.tilt <- exp.tilt(grav.L, grav.z0[3], strata=grav1[,2])
boot(grav1, grav.fun, R=499, stype="w", weights=grav.tilt$p,
      strata=grav1[,2], orig=grav.z0[1])
```

`fir` *Counts of Balsam-fir Seedlings*

Description

The `fir` data frame has 50 rows and 3 columns.

The number of balsam-fir seedlings in each quadrant of a grid of 50 five foot square quadrants were counted. The grid consisted of 5 rows of 10 quadrants in each row.

Usage

```
fir
```

Format

This data frame contains the following columns:

`count` The number of seedlings in the quadrant.

`row` The row number of the quadrant.

`col` The quadrant number within the row.

Source

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

freq.array *Bootstrap Frequency Arrays*

Description

Take a matrix of indices for nonparametric bootstrap resamples and return the frequencies of the original observations in each resample.

Usage

```
freq.array(i.array)
```

Arguments

i.array	This will be an matrix of integers between 1 and n, where n is the number of observations in a data set. The matrix will have n columns and R rows where R is the number of bootstrap resamples. Such matrices are found by boot when doing nonparametric bootstraps. They can also be found after a bootstrap has been run through the function boot.array.

Value

A matrix of the same dimensions as the input matrix. Each row of the matrix corresponds to a single bootstrap resample. Each column of the matrix corresponds to one of the original observations and specifies its frequency in each bootstrap resample. Thus the first column tells us how often the first observation appeared in each bootstrap resample. Such frequency arrays are often useful for diagnostic purposes such as the jackknife-after-bootstrap plot. They are also necessary for the regression estimates of empirical influence values and for finding importance sampling weights.

See Also

boot.array

frets *Head Dimensions in Brothers*

Description

The frets data frame has 25 rows and 4 columns.

The data consist of measurements of the length and breadth of the heads of pairs of adult brothers in 25 randomly sampled families. All measurements are expressed in millimetres.

Usage

```
frets
```

Format

This data frame contains the following columns:

l1 The head length of the eldest son.

b1 The head breadth of the eldest son.

l2 The head length of the second son.

b2 The head breadth of the second son.

Source

The data were obtained from

Frets, G.P. (1921) Heredity of head form in man. *Genetica*, **3**, 193.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

Whittaker, J. (1990) *Graphical Models in Applied Multivariate Statistics*. John Wiley.

---

glm.diag *Generalized Linear Model Diagnostics*

Description

Calculates jackknife deviance residuals, standardized deviance residuals, standardized Pearson residuals, approximate Cook statistic, leverage and estimated dispersion.

Usage

```
glm.diag(glmfit)
```

Arguments

glmfit	glmfit is a glm.object - the result of a call to glm()

Value

Returns a list with the following components

res	The vector of jackknife deviance residuals.
rd	The vector of standardized deviance residuals.
rp	The vector of standardized Pearson residuals.
cook	The vector of approximate Cook statistics.
h	The vector of leverages of the observations.
sd	The value used to standardize the residuals. This is the the estimate of residual standard deviation in the Gaussian family and is the square root of the estimated shape parameter in the Gamma family. In all other cases it is 1.

Note

See the helpfile for `glm.diag.plots` for an example of the use of `glm.diag`.

References

Davison, A.C. and Snell, E.J. (1991) Residuals and diagnostics. In *Statistical Theory and Modelling: In Honour of Sir David Cox*. D.V. Hinkley, N. Reid and E.J. Snell (editors), 83–106. Chapman and Hall.

See Also

`glm`, `glm.diag.plots`, `summary.glm`

`glm.diag.plots` *Diagnostics plots for generalized linear models*

Description

Makes plot of jackknife deviance residuals against linear predictor, normal scores plots of standardized deviance residuals, plot of approximate Cook statistics against leverage/(1-leverage), and case plot of Cook statistic.

Usage

```
glm.diag.plots(glmfit, glmdiag=glm.diag(glmfit),
     subset=NULL, iden=FALSE, labels=NULL, ret=FALSE)
```

Arguments

`glmfit`	`glm.object` : the result of a call to `glm()`
`glmdiag`	Diagnostics of `glmfit` obtained from a call to `glm.diag`. If it is not supplied then it is calculated.
`subset`	Subset of `data` for which `glm` fitting performed: should be the same as the `subset` option used in the call to `glm()` which generated `glmfit`. Needed only if the `subset=` option was used in the call to `glm`.
`iden`	A logical argument. If `TRUE` then, after the plots are drawn, the user will be prompted for an integer between 0 and 4. A positive integer will select a plot and invoke `identify()` on that plot. After exiting `identify()`, the user is again prompted, this loop continuing until the user responds to the prompt with 0. If `iden` is `FALSE` (default) the user cannot interact with the plots.
`labels`	A vector of labels for use with `identify()` if `iden` is `TRUE`. If it is not supplied then the labels are derived from `glmfit`.
`ret`	A logical argument indicating if `glmdiag` should be returned. The default is `FALSE`.

Details

The diagnostics required for the plots are calculated by `glm.diag`. These are then used to produce the four plots on the current graphics device.

The plot on the top left is a plot of the jackknife deviance residuals against the fitted values.

The plot on the top right is a normal QQ plot of the standardized deviance residuals. The dotted line is the expected line if the standardized residuals are normally distributed, i.e. it is the line with intercept 0 and slope 1.

The bottom two panels are plots of the Cook statistics. On the left is a plot of the Cook statistics against the standardized leverages. In general there will be two dotted lines on this plot. The horizontal line is at 8/(n-2p) where n is the number of observations and p is the number of parameters estimated. Points above this line may be points with high influence on the model. The vertical line is at 2p/(n-2p) and points to the right of this line have high leverage compared to the variance of the raw residual at that point. If all points are below the horizontal line or to the left of the vertical line then the line is not shown.

The final plot again shows the Cook statistic this time plotted against case number enabling us to find which observations are influential.

Use of `iden=T` is encouraged for proper exploration of these four plots as a guide to how well the model fits the data and whether certain observations have an unduly large effect on parameter estimates.

Value

If `ret` is `TRUE` then the value of `glmdiag` is returned otherwise there is no returned value.

Side Effects

The current device is cleared and four plots are plotted by use of `split.screen(c(2,2))`. If `iden` is `TRUE`, interactive identification of points is enabled. All screens are closed, but not cleared, on termination of the function.

References

Davison, A. C. and Hinkley, D. V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

Davison, A.C. and Snell, E.J. (1991) Residuals and diagnostics. In *Statistical Theory and Modelling: In Honour of Sir David Cox* D.V. Hinkley, N. Reid, and E.J. Snell (editors), 83–106. Chapman and Hall.

See Also

`glm`, `glm.diag`, `identify`

Examples

```
# In this example we look at the leukaemia data which was looked at in
# Example 7.1 of Davison and Hinkley (1997)
data(leuk, package="MASS")
leuk.mod <- glm(time~ag-1+log10(wbc),family=Gamma(log),data=leuk)
leuk.diag <- glm.diag(leuk.mod)
glm.diag.plots(leuk.mod,leuk.diag)
```

gravity *Acceleration Due to Gravity*

Description

The `gravity` data frame has 81 rows and 2 columns.

The `grav` data set has 26 rows and 2 columns.

Between May 1934 and July 1935, the National Bureau of Standards in Washington D.C. conducted a series of experiments to estimate the acceleration due to gravity, *g*, at Washington. Each experiment produced a number of replicate estimates of *g* using the same methodology. Although the basic method remained the same for all experiments, that of the reversible pendulum, there were changes in configuration.

The `gravity` data frame contains the data from all eight experiments. The `grav` data frame contains the data from the experiments 7 and 8. The data are expressed as deviations from 980.000 in centimetres per second squared.

Usage

```
gravity
```

Format

This data frame contains the following columns:

`g` The deviation of the estimate from 980.000 centimetres per second squared.

`series` A factor describing from which experiment the estimate was derived.

Source

The data were obtained from

Cressie, N. (1982) Playing safe with misweighted means. *Journal of the American Statistical Association*, **77**, 754–759.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

hirose *Failure Time of PET Film*

Description

The `hirose` data frame has 44 rows and 3 columns.

PET film is used in electrical insulation. In this accelerated life test the failure times for 44 samples in gas insulated transformers. 4 different voltage levels were used.

Usage

```
hirose
```

Format

This data frame contains the following columns:

volt The voltage (in kV).

time The failure or censoring time in hours.

cens The censoring indicator; 1 means right-censored data.

Source

The data were obtained from

Hirose, H. (1993) Estimation of threshold stress in accelerated life-testing. *IEEE Transactions on Reliability*, **42**, 650–657.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

Imp.Estimates *Importance Sampling Estimates*

Description

Central moment, tail probability, and quantile estimates for a statistic under importance resampling.

Usage

```
imp.moments(boot.out=NULL, index=1, t=boot.out$t[, index],
          w=NULL, def=TRUE, q=NULL)
imp.prob(boot.out=NULL, index=1, t0=boot.out$t0[index],
          t=boot.out$t[, index], w=NULL, def=TRUE, q=NULL)
imp.quantile(boot.out=NULL, alpha=NULL, index=1,
          t=boot.out$t[, index], w=NULL, def=TRUE, q=NULL)
```

Arguments

boot.out	A object of class "boot" generated by a call to boot or tilt.boot. Use of these functions makes sense only when the bootstrap resampling used unequal weights for the observations. If the importance weights w are not supplied then boot.out is a required argument. It is also required if t is not supplied.
alpha	The alpha levels for the required quantiles. The default is to calculate the 1%, 2.5%, 5%, 10%, 90%, 95%, 97.5% and 99% quantiles.
index	The index of the variable of interest in the output of boot.out$statistic. This is not used if the argument t is supplied.

`t0` The values at which tail probability estimates are required. For each value `t0[i]` the function will estimate the bootstrap cdf evaluated at `t0[i]`. If `imp.prob` is called without the argument `t0` then the bootstrap cdf evaluated at the observed value of the statistic is found.

`t` The bootstrap replicates of a statistic. By default these are taken from the bootstrap output object `boot.out` but they can be supplied separately if required (e.g. when the statistic of interest is a function of the calculated values in `boot.out`). Either `boot.out` or `t` must be supplied.

`w` The importance resampling weights for the bootstrap replicates. If they are not supplied then `boot.out` must be supplied, in which case the importance weights are calculated by a call to `imp.weights`.

`def` A logical value indicating whether a defensive mixture is to be used for weight calculation. This is used only if `w` is missing and it is passed unchanged to `imp.weights` to calculate `w`.

`q` A vector of probabilities specifying the resampling distribution from which any estimates should be found. In general this would correspond to the usual bootstrap resampling distribution which gives equal weight to each of the original observations. The estimates depend on this distribution only through the importance weights `w` so this argument is ignored if `w` is supplied. If `w` is missing then `q` is passed as an argument to `imp.weights` and used to find `w`.

Value

A list with the following components :

`alpha` The `alpha` levels used for the quantiles, if `imp.quantile` is used.

`t0` The values at which the tail probabilities are estimated, if `imp.prob` is used.

`raw` The raw importance resampling estimates. For `imp.moments` this has length 2, the first component being the estimate of the mean and the second being the variance estimate. For `imp.prob`, `raw` is of the same length as `t0`, and for `imp.quantile` it is of the same length as `alpha`.

`rat` The ratio importance resampling estimates. In this method the weights `w` are rescaled to have average value one before they are used. The format of this vector is the same as `raw`.

`reg` The regression importance resampling estimates. In this method the weights which are used are derived from a regression of `t*w` on `w`. This choice of weights can be shown to minimize the variance of the weights and also the Euclidean distance of the weights from the uniform weights. The format of this vector is the same as `raw`.

References

Davison, A. C. and Hinkley, D. V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

Hesterberg, T. (1995) Weighted average importance sampling and defensive mixture distributions. *Technometrics*, **37**, 185–194.

Johns, M.V. (1988) Importance sampling for bootstrap confidence intervals. *Journal of the American Statistical Association*, **83**, 709–714.

See Also

boot, exp.tilt, imp.weights, smooth.f, tilt.boot

Examples

```
# Example 9.8 of Davison and Hinkley (1997) requires tilting the
# resampling distribution of the studentized statistic to be centred
# at the observed value of the test statistic, 1.84.  In this example
# we show how certain estimates can be found using resamples taken from
# the tilted distribution.
grav1 <- gravity[as.numeric(gravity[,2])>=7,]
grav.fun <- function(dat, w, orig)
{    strata <- tapply(dat[, 2], as.numeric(dat[, 2]))
     d <- dat[, 1]
     ns <- tabulate(strata)
     w <- w/tapply(w, strata, sum)[strata]
     mns <- tapply(d * w, strata, sum)
     mn2 <- tapply(d * d * w, strata, sum)
     s2hat <- sum((mn2 - mns^2)/ns)
     as.vector(c(mns[2]-mns[1],s2hat,(mns[2]-mns[1]-orig)/sqrt(s2hat)))
}
grav.z0 <- grav.fun(grav1,rep(1,26),0)
grav.L <- empinf(data=grav1, statistic=grav.fun, stype="w",
                 strata=grav1[,2], index=3, orig=grav.z0[1])
grav.tilt <- exp.tilt(grav.L,grav.z0[3],strata=grav1[,2])
grav.tilt.boot <- boot(grav1, grav.fun, R=199, stype="w",
                       strata=grav1[,2], weights=grav.tilt$p,
                       orig=grav.z0[1])
# Since the weights are needed for all calculations, we shall calculate
# them once only.
grav.w <- imp.weights(grav.tilt.boot)
grav.mom <- imp.moments(grav.tilt.boot, w=grav.w, index=3)
grav.p <- imp.prob(grav.tilt.boot, w=grav.w, index=3, t0=grav.z0[3])
grav.q <- imp.quantile(grav.tilt.boot, w=grav.w, index=3,
                       alpha=c(0.9,0.95,0.975,0.99))
```

`imp.weights` *Importance Sampling Weights*

Description

This function calculates the importance sampling weight required to correct for simulation from a distribution with probabilities `p` when estimates are required assuming that simulation was from an alternative distribution with probabilities `q`.

Usage

```
imp.weights(boot.out, def=TRUE, q=NULL)
```

Arguments

boot.out	A object of class "boot" generated by boot or tilt.boot. Typically the bootstrap simulations would have been done using importance resampling and we wish to do our calculations under the assumption of sampling with equal probabilities.
def	A logical variable indicating whether the defensive mixture distribution weights should be calculated. This makes sense only in the case where the replicates in boot.out were simulated under a number of different distributions. If this is the case then the defensive mixture weights use a mixture of the distributions used in the bootstrap. The alternative is to calculate the weights for each replicate using knowledge of the distribution from which the bootstrap resample was generated.
q	A vector of probabilities specifying the resampling distribution from which we require inferences to be made. In general this would correspond to the usual bootstrap resampling distribution which gives equal weight to each of the original observations and this is the default. q must have length equal to the number of observations in the boot.out$data and all elements of q must be positive.

Details

The importance sampling weight for a bootstrap replicate with frequency vector f is given by prod((q/p)^f). This reweights the replicates so that estimates can be found as if the bootstrap resamples were generated according to the probabilities q even though, in fact, they came from the distribution p.

Value

A vector of importance weights of the same length as boot.out$t. These weights can then be used to reweight boot.out$t so that estimates can be found as if the simulations were from a distribution with probabilities q.

Note

See the example in the help for imp.moments for an example of using imp.weights.

References

Davison, A. C. and Hinkley, D. V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

Hesterberg, T. (1995) Weighted average importance sampling and defensive mixture distributions. *Technometrics*, **37**, 185–194.

Johns, M.V. (1988) Importance sampling for bootstrap confidence intervals. *Journal of the American Statistical Association*, **83**, 709–714.

See Also

boot, exp.tilt, imp.moments, smooth.f, tilt.boot

inv.logit *Inverse Logit Function*

Description

Given a numeric object return the inverse logit of the values.

Usage

```
inv.logit(x)
```

Arguments

x	A numeric object. Missing values (NAs) are allowed.

Details

The inverse logit is defined by `exp(x)/(1+exp(x))`. Values in `x` of `-Inf` or `Inf` return logits of 0 or 1 respectively. Any `NA`s in the input will also be `NA`s in the output.

Value

An object of the same type as `x` containing the inverse logits of the input values.

See Also

`logit`, `plogis` which is the underlying function.

islay *Jura Quartzite Azimuths on Islay*

Description

The `islay` data frame has 18 rows and 1 columns.

Measurements were taken of paleocurrent azimuths from the Jura Quartzite on the Scottish island of Islay.

Usage

```
islay
```

Format

This data frame contains the following column:

`theta` The angle of the azimuth in degrees East of North.

Source

The data were obtained from

Hand, D.J., Daly, F., Lunn, A.D., McConway, K.J. and Ostrowski, E. (1994) *A Handbook of Small Data Sets*, Chapman and Hall.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

Till, R. (1974) *Statistical Methods for the Earth Scientist*. Macmillan.

---

jack.after.boot *Jackknife-after-Bootstrap Plots*

Description

This function calculates the jackknife influence values from a bootstrap output object and plots the corresponding jackknife-after-bootstrap plot.

Usage

```
jack.after.boot(boot.out, index=1, t=NULL, L=NULL, useJ=TRUE,
                stinf=TRUE, alpha=NULL, main="", ylab=NULL, ...)
```

Arguments

boot.out	An object of class "boot" which would normally be created by a call to boot. It should represent a nonparametric bootstrap. For reliable results boot.out$R should be reasonably large.
index	The index of the statistic of interest in the output of boot.out$statistic.
t	A vector of length boot.out$R giving the bootstrap replicates of the statistic of interest. This is useful if the statistic of interest is a function of the calculated bootstrap output. If it is not supplied then the default is boot.out$t[,index].
L	The empirical influence values for the statistic of interest. These are used only if useJ is FALSE. If they are not supplied and are needed, they are calculated by a call to empinf. If L is supplied then it is assumed that they are the infinitesimal jackknife values.
useJ	A logical variable indicating if the jackknife influence values calculated from the bootstrap replicates should be used. If FALSE the empirical influence values are used. The default is TRUE.
stinf	A logical variable indicating whether to standardize the jackknife values before plotting them. If TRUE then the jackknife values used are divided by their standard error.
alpha	The quantiles at which the plots are required. The default is c(0.05, 0.1, 0.16, 0.5, 0.84, 0.9, 0.95).
main	A character string giving the main title for the plot.

`ylab`	The label for the Y axis. If the default values of `alpha` are used and `ylab` is not supplied then a label indicating which percentiles are plotted is used. If `alpha` is supplied then the default label will not say which percentiles were used.
`...`	Any extra arguments required by `boot.out$statistic`. These are required only if `useJ` is `FALSE` and `L` is not supplied, in which case they are passed to `empinf` for use in calculation of the empirical influence values.

Details

The centred jackknife quantiles for each observation are estimated from those bootstrap samples in which the particular observation did not appear. These are then plotted against the influence values. If `useJ` is `TRUE` then the influence values are found in the same way as the difference between the mean of the statistic in the samples excluding the observations and the mean in all samples. If `useJ` is `FALSE` then empirical influence values are calculated by calling `empinf`.

The resulting plots are useful diagnostic tools for looking at the way individual observations affect the bootstrap output.

The plot will consist of a number of horizontal dotted lines which correspond to the quantiles of the centred bootstrap distribution. For each data point the quantiles of the bootstrap distribution calculated by omitting that point are plotted against the (possibly standardized) jackknife values. The observation number is printed below the plots. To make it easier to see the effect of omitting points on quantiles, the plotted quantiles are joined by line segments. These plots provide a useful diagnostic tool in establishing the effect of individual observations on the bootstrap distribution. See the references below for some guidelines on the interpretation of the plots.

Value

There is no returned value but a plot is generated on the current graphics display.

Side Effects

A plot is created on the current graphics device.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

Efron, B. (1992) Jackknife-after-bootstrap standard errors and influence functions (with Discussion). *Journal of the Royal Statistical Society, B*, **54**, 83–127.

See Also

`boot`, `empinf`

Examples

```
#  To draw the jackknife-after-bootstrap plot for the head size data as in
#  Example 3.24 of Davison and Hinkley (1997)
pcorr <- function( x )
{
#  function to find the correlations and partial correlations between
#  the four measurements.
     v <- cor(x)
```

```
      v.d <- diag(var(x))
      iv <- solve(v)
      iv.d <- sqrt(diag(iv))
      iv <- - diag(1/iv.d) %*% iv %*% diag(1/iv.d)
      q <- NULL
      n <- nrow(v)
      for (i in 1:(n-1))
           q <- rbind( q, c(v[i,1:i],iv[i,(i+1):n]) )
      q <- rbind( q, v[n,] )
      diag(q) <- round(diag(q))
      q
}

frets.fun <- function( data, i )
{     d <- data[i,]
      v <- pcorr( d )
      c(v[1,],v[2,],v[3,],v[4,])
}
frets.boot <- boot(log(as.matrix(frets)), frets.fun, R=999)
#  we will concentrate on the partial correlation between head breadth
#  for the first son and head length for the second.  This is the 7th
#  element in the output of frets.fun so we set index=7
jack.after.boot(frets.boot,useJ=FALSE,stinf=FALSE,index=7)
```

`k3.linear` *Linear Skewness Estimate*

Description

Estimates the skewness of a statistic from its empirical influence values.

Usage

```
k3.linear(L, strata=NULL)
```

Arguments

`L`	Vector of the empirical influence values of a statistic. These will usually be calculated by a call to `empinf`.
`strata`	A numeric vector or factor specifying which observations (and hence which components of `L`) come from which strata.

Value

The skewness estimate calculated from `L`.

References

Davison, A. C. and Hinkley, D. V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

See Also

`empinf`, `linear.approx`, `var.linear`

Examples

```
#  To estimate the skewness of the ratio of means for the city data.
ratio <- function(d,w)
     sum(d$x * w)/sum(d$u * w)
k3.linear(empinf(data=city,statistic=ratio))
```

---

linear.approx *Linear Approximation of Bootstrap Replicates*

Description

This function takes a bootstrap object and for each bootstrap replicate it calculates the linear approximation to the statistic of interest for that bootstrap sample.

Usage

```
linear.approx(boot.out, L=NULL, index=1, type=NULL, t0=NULL,
              t=NULL, ...)
```

Arguments

boot.out	An object of class "boot" representing a nonparametric bootstrap. It will usually be created by the function boot.
L	A vector containing the empirical influence values for the statistic of interest. If it is not supplied then L is calculated through a call to empinf.
index	The index of the variable of interest within the output of boot.out$statistic.
type	This gives the type of empirical influence values to be calculated. It is not used if L is supplied. The possible types of empirical influence values are described in the helpfile for empinf.
t0	The observed value of the statistic of interest. The input value is used only if one of t or L is also supplied. The default value is boot.out$t0[index]. If t0 is supplied but neither t nor L are supplied then t0 is set to boot.out$t0[index] and a warning is generated.
t	A vector of bootstrap replicates of the statistic of interest. If t0 is missing then t is not used, otherwise it is used to calculate the empirical influence values (if they are not supplied in L).
...	Any extra arguments required by boot.out$statistic. These are needed if L is not supplied as they are used by empinf to calculate empirical influence values.

Details

The linear approximation to a bootstrap replicate with frequency vector f is given by t0 + sum(L * f)/n in the one sample with an easy extension to the stratified case. The frequencies are found by calling boot.array.

Value

A vector of length `boot.out$R` with the linear approximations to the statistic of interest for each of the bootstrap samples.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

See Also

`boot`, `empinf`, `control`

Examples

```
# Using the city data let us look at the linear approximation to the
# ratio statistic and its logarithm. We compare these with the
# corresponding plots for the bigcity data

ratio <- function(d, w)
     sum(d$x * w)/sum(d$u * w)
city.boot <- boot(city, ratio, R=499, stype="w")
bigcity.boot <- boot(bigcity, ratio, R=499, stype="w")
par(pty="s")
par(mfrow=c(2,2))

# The first plot is for the city data ratio statistic.
city.lin1 <- linear.approx(city.boot)
lim <- range(c(city.boot$t,city.lin1))
plot(city.boot$t, city.lin1, xlim=lim,ylim=lim,
     main="Ratio; n=10", xlab="t*", ylab="tL*")
abline(0,1)

# Now for the log of the ratio statistic for the city data.
city.lin2 <- linear.approx(city.boot,t0=log(city.boot$t0),
                           t=log(city.boot$t))
lim <- range(c(log(city.boot$t),city.lin2))
plot(log(city.boot$t), city.lin2, xlim=lim, ylim=lim,
     main="Log(Ratio); n=10", xlab="t*", ylab="tL*")
abline(0,1)

# The ratio statistic for the bigcity data.
bigcity.lin1 <- linear.approx(bigcity.boot)
lim <- range(c(bigcity.boot$t,bigcity.lin1))
plot(bigcity.lin1,bigcity.boot$t, xlim=lim,ylim=lim,
     main="Ratio; n=49", xlab="t*", ylab="tL*")
abline(0,1)

# Finally the log of the ratio statistic for the bigcity data.
bigcity.lin2 <- linear.approx(bigcity.boot,t0=log(bigcity.boot$t0),
                              t=log(bigcity.boot$t))
lim <- range(c(log(bigcity.boot$t),bigcity.lin2))
plot(bigcity.lin2,log(bigcity.boot$t), xlim=lim,ylim=lim,
     main="Log(Ratio); n=49", xlab="t*", ylab="tL*")
abline(0,1)
```

```
par(mfrow=c(1,1))
```

`lines.saddle.distn` *Add a Saddlepoint Approximation to a Plot*

Description

This function adds a line corresponding to a saddlepoint density or distribution function approximation to the current plot.

Usage

```
## S3 method for class 'saddle.distn':
lines(x, dens = TRUE, h = function(u) u, J = function(u) 1,
     npts = 50, lty = 1, ...)
```

Arguments

`x`	An object of class `"saddle.distn"` (see `saddle.distn.object` representing a saddlepoint approximation to a distribution.
`dens`	A logical variable indicating whether the saddlepoint density (`TRUE`; the default) or the saddlepoint distribution function (`FALSE`) should be plotted.
`h`	Any transformation of the variable that is required. Its first argument must be the value at which the approximation is being performed and the function must be vectorized.
`J`	When `dens=TRUE` this function specifies the Jacobian for any transformation that may be necessary. The first argument of `J` must the value at which the approximation is being performed and the function must be vectorized. If `h` is supplied `J` must also be supplied and both must have the same argument list.
`npts`	The number of points to be used for the plot. These points will be evenly spaced over the range of points used in finding the saddlepoint approximation.
`lty`	The line type to be used.
`...`	Any additional arguments to `h` and `J`.

Details

The function uses `smooth.spline` to produce the saddlepoint curve. When `dens=TRUE` the spline is on the log scale and when `dens=FALSE` it is on the probit scale.

Value

`sad.d` is returned invisibly.

Side Effects

A line is added to the current plot.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

See Also

saddle.distn

Examples

```
# In this example we show how a plot such as that in Figure 9.9 of
# Davison and Hinkley (1997) may be produced.  Note the large number of
# bootstrap replicates required in this example.
expdata <- rexp(12)
vfun <- function(d, i)
{    n <- length(d)
     (n-1)/n*var(d[i])
}
exp.boot <- boot(expdata,vfun, R = 9999)
exp.L <- (expdata-mean(expdata))^2 - exp.boot$t0
exp.tL <- linear.approx(exp.boot, L = exp.L)
hist(exp.tL, nclass = 50, prob = TRUE)
exp.t0 <- c(0,sqrt(var(exp.boot$t)))
exp.sp <- saddle.distn(A = exp.L/12,wdist = "m", t0 = exp.t0)

# The saddlepoint approximation in this case is to the density of
# t-t0 and so t0 must be added for the plot.
lines(exp.sp,h = function(u,t0) u+t0, J = function(u,t0) 1,
      t0 = exp.boot$t0)
```

---

logit *Logit of Proportions*

Description

This function calculates the logit of proportions.

Usage

```
logit(p)
```

Arguments

p	A numeric Splus object, all of whose values are in the range [0,1]. Missing values (NAs) are allowed.

Details

If any elements of p are outside the unit interval then an error message is generated. Values of p equal to 0 or 1 (to within machine precision) will return -Inf or Inf respectively. Any NAs in the input will also be NAs in the output.

Value

A numeric object of the same type as p containing the logits of the input values.

See Also

inv.logit

`manaus` *Average Heights of the Rio Negro river at Manaus*

Description

The `manaus` time series is of class `"ts"` and has 1080 observations on one variable.

The data values are monthly averages of the daily stages (heights) of the Rio Negro at Manaus. Manaus is 18km upstream from the confluence of the Rio Negro with the Amazon but because of the tiny slope of the water surface and the lower courses of its flatland affluents, they may be regarded as a good approximation of the water level in the Amazon at the confluence. The data here cover 90 years from January 1903 until December 1992.

The Manaus gauge is tied in with an arbitrary bench mark of 100m set in the steps of the Municipal Prefecture; gauge readings are usually referred to sea level, on the basis of a mark on the steps leading to the Parish Church (Matriz), which is assumed to lie at an altitude of 35.874 m according to observations made many years ago under the direction of Samuel Pereira, an engineer in charge of the Manaus Sanitation Committee Whereas such an altitude cannot, by any means, be considered to be a precise datum point, observations have been provisionally referred to it. The measurements are in metres.

Source

The data were kindly made available by Professors H. O'Reilly Sternberg and D. R. Brillinger of the University of California at Berkeley.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

Sternberg, H. O'R. (1987) Aggravation of floods in the Amazon river as a consequence of deforestation? *Geografiska Annaler*, **69A**, 201-219.

Sternberg, H. O'R. (1995) Waters and wetlands of Brazilian Amazonia: An uncertain future. In *The Fragile Tropics of Latin America: Sustainable Management of Changing Environments*, Nishizawa, T. and Uitto, J.I. (editors), United Nations University Press, 113-179.

`melanoma` *Survival from Malignant Melanoma*

Description

The `melanoma` data frame has 205 rows and 7 columns.

The data consist of measurements made on patients with malignant melanoma. Each patient had their tumour removed by surgery at the Department of Plastic Surgery, University Hospital of Odense, Denmark during the period 1962 to 1977. The surgery consisted of complete removal of the tumour together with about 2.5cm of the surrounding skin. Among the measurements taken were the thickness of the tumour and whether it was ulcerated or not. These are thought to be important prognostic variables in that patients with a thick and/or ulcerated tumour have an increased chance of death from melanoma. Patients were followed until the end of 1977.

Usage

```
melanoma
```

Format

This data frame contains the following columns:

time Survival time in days since the operation, possibly censored.

status The patients status at the end of the study. 1 indicates that they had died from melanoma, 2 indicates that they were still alive and 3 indicates that they had died from causes unrelated to their melanoma.

sex The patients sex; 1=male, 2=female.

age Age in years at the time of the operation.

year Year of operation.

thickness Tumour thickness in mm.

ulcer Indicator of ulceration; 1=present, 0=absent.

Note

This dataset is not related to the dataset in the **lattice** package with the same name.

Source

The data were obtained from

Andersen, P.K., Borgan, O., Gill, R.D. and Keiding, N. (1993) *Statistical Models Based on Counting Processes*. Springer-Verlag.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

Venables, W.N. and Ripley, B.D. (1994) *Modern Applied Statistics with S-Plus*. Springer-Verlag.

---

motor *Data from a Simulated Motorcycle Accident*

Description

The `motor` data frame has 94 rows and 4 columns. The rows are obtained by removing replicate values of `time` from the dataset `mcycle`. Two extra columns are added to allow for strata with a different residual variance in each stratum.

Usage

```
motor
```

Format

This data frame contains the following columns:

`times` The time in milliseconds since impact.

`accel` The recorded head acceleration (in g).

`strata` A numeric column indicating to which of the three strata (numbered 1, 2 and 3) the observations belong.

`v` An estimate of the residual variance for the observation. `v` is constant within the strata but a different estimate is used for each of the three strata.

Source

The data were obtained from

Silverman, B.W. (1985) Some aspects of the spline smoothing approach to non-parametric curve fitting. *Journal of the Royal Statistical Society, B*, **47**, 1–52.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

Venables, W.N. and Ripley, B.D. (1994) *Modern Applied Statistics with S-Plus*. Springer-Verlag.

See Also

`mcycle`

`neuro` *Neurophysiological Point Process Data*

Description

`neuro` is a matrix containing times of observed firing of a neuron in windows of 250ms either side of the application of a stimulus to a human subject. Each row of the matrix is a replication of the experiment and there were a total of 469 replicates.

Note

There are a lot of missing values in the matrix as different numbers of firings were observed in different replicates. The number of firings observed varied from 2 to 6.

Source

The data were collected and kindly made available by Dr. S.J. Boniface of the Neurophysiology Unit at the Radcliffe Infirmary, Oxford.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

Ventura, V., Davison, A.C. and Boniface, S.J. (1997) A stochastic model for the effect of magnetic brain stimulation on a motoneurone. To appear in *Applied Statistics*.

`nitrofen` *Toxicity of Nitrofen in Aquatic Systems*

Description

The `nitrofen` data frame has 50 rows and 5 columns.

Nitrofen is a herbicide that was used extensively for the control of broad-leaved and grass weeds in cereals and rice. Although it is relatively non-toxic to adult mammals, nitrofen is a significant tetragen and mutagen. It is also acutely toxic and reproductively toxic to cladoceran zooplankton. Nitrofen is no longer in commercial use in the U.S., having been the first pesticide to be withdrawn due to tetrogenic effects.

The data here come from an experiment to measure the reproductive toxicity of nitrofen on a species of zooplankton (*Ceriodaphnia dubia*). 50 animals were randomized into batches of 10 and each batch was put in a solution with a measured concentration of nitrofen. Then the number of live offspring in each of the three broods to each animal was recorded.

Usage

```
nitrofen
```

Format

This data frame contains the following columns:

`conc` The nitrofen concentration in the solution (mug/litre).

`brood1` The number of live offspring in the first brood.

`brood2` The number of live offspring in the second brood.

`brood3` The number of live offspring in the third brood.

`total` The total number of live offspring in the first three broods.

Source

The data were obtained from

Bailer, A.J. and Oris, J.T. (1994) Assessing toxicity of pollutants in aquatic systems. In *Case Studies in Biometry*. N. Lange, L. Ryan, L. Billard, D. Brillinger, L. Conquest and J. Greenhouse (editors), 25–40. John Wiley.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

`nodal` *Nodal Involvement in Prostate Cancer*

Description

The `nodal` data frame has 53 rows and 7 columns.

The treatment strategy for a patient diagnosed with cancer of the prostate depend highly on whether the cancer has spread to the surrounding lymph nodes. It is common to operate on the patient to get samples from the nodes which can then be analysed under a microscope but clearly it would be preferable if an accurate assessment of nodal involvement could be made without surgery.

For a sample of 53 prostate cancer patients, a number of possible predictor variables were measured before surgery. The patients then had surgery to determine nodal involvement. It was required to see if nodal involvement could be accurately predicted from the predictor variables and which ones were most important.

Usage

```
nodal
```

Format

This data frame contains the following columns:

`m` A column of ones.

`r` An indicator of nodal involvement.

`aged` The patients age dichotomized into less than 60 (`0`) and 60 or over `1`.

`stage` A measurement of the size and position of the tumour observed by palpatation with the fingers via the rectum. A value of `1` indicates a more serious case of the cancer.

`grade` Another indicator of the seriousness of the cancer, this one is determined by a pathology reading of a biopsy taken by needle before surgery. A value of `1` indicates a more serious case of the cancer.

`xray` A third measure of the seriousness of the cancer taken from an X-ray reading. A value of `1` indicates a more serious case of the cancer.

`acid` The level of acid phosphatase in the blood serum.

Source

The data were obtained from

Brown, B.W. (1980) Prediction analysis for binary data. In *Biostatistics Casebook*. R.G. Miller, B. Efron, B.W. Brown and L.E. Moses (editors), 3–18. John Wiley.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

norm.ci *Normal Approximation Confidence Intervals*

Description

Using the normal approximation to a statistic, calculate equi-tailed two-sided confidence intervals.

Usage

```
norm.ci(boot.out=NULL, conf=0.95, index=1, var.t0=NULL,
        t0=NULL, t=NULL, L=NULL, h=function(t) t,
        hdot=function(t) 1, hinv=function(t) t)
```

Arguments

boot.out	A bootstrap output object returned from a call to boot. If t0 is missing then boot.out is a required argument. It is also required if both var.t0 and t are missing.
conf	A scalar or vector containing the confidence level(s) of the required interval(s).
index	The index of the statistic of interest within the output of a call to boot.out$statistic. It is not used if boot.out is missing, in which case t0 must be supplied.
var.t0	The variance of the statistic of interest. If it is not supplied then var(t) is used.
t0	The observed value of the statistic of interest. If it is missing then it is taken from boot.out which is required in that case.
t	Bootstrap replicates of the variable of interest. These are used to estimate the variance of the statistic of interest if var.t0 is not supplied. The default value is boot.out$t[,index].
L	The empirical influence values for the statistic of interest. These are used to calculate var.t0 if neither var.t0 nor boot.out are supplied. If a transformation is supplied through h then the influence values must be for the untransformed statistic t0.
h	A function defining a monotonic transformation, the intervals are calculated on the scale of h(t) and the inverse function hinv is applied to the resulting intervals. h must be a function of one variable only and must be vectorized. The default is the identity function.
hdot	A function of one argument returning the derivative of h. It is a required argument if h is supplied and is used for approximating the variance of h(t0). The default is the constant function 1.
hinv	A function, like h, which returns the inverse of h. It is used to transform the intervals calculated on the scale of h(t) back to the original scale. The default is the identity function. If h is supplied but hinv is not, then the intervals returned will be on the transformed scale.

Details

It is assumed that the statistic of interest has an approximately normal distribution with variance `var.t0` and so a confidence interval of length `2*qnorm((1+conf)/2)*sqrt(var.t0)` is found. If `boot.out` or `t` are supplied then the interval is bias-corrected using the bootstrap bias estimate, and so the interval would be centred at `2*t0-mean(t)`. Otherwise the interval is centred at `t0`.

Value

If `length(conf)` is 1 then a vector containing the confidence level and the endpoints of the interval is returned. Otherwise, the returned value is a matrix where each row corresponds to a different confidence level.

Note

This function is primarily designed to be called by `boot.ci` to calculate the normal approximation after a bootstrap but it can also be used without doing any bootstrap calculations as long as `t0` and `var.t0` can be supplied. See the examples below.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

See Also

`boot.ci`

Examples

```
#  In Example 5.1 of Davison and Hinkley (1997), normal approximation
#  confidence intervals are found for the air-conditioning data.
air.mean <- mean(aircondit$hours)
air.n <- nrow(aircondit)
air.v <- air.mean^2/air.n
norm.ci(t0=air.mean, var.t0=air.v)
exp(norm.ci(t0=log(air.mean), var.t0=1/air.n)[2:3])

# Now a more complicated example - the ratio estimate for the city data.
ratio <- function(d, w)
     sum(d$x * w)/sum(d$u *w)
city.v <- var.linear(empinf(data=city, statistic=ratio))
norm.ci(t0=ratio(city,rep(0.1,10)), var.t0=city.v)
```

`nuclear` *Nuclear Power Station Construction Data*

Description

The `nuclear` data frame has 32 rows and 11 columns.

The data relate to the construction of 32 light water reactor (LWR) plants constructed in the U.S.A in the late 1960's and early 1970's. The data was collected with the aim of predicting the cost of construction of further LWR plants. 6 of the power plants had partial turnkey guarantees and it is possible that, for these plants, some manufacturers' subsidies may be hidden in the quoted capital costs.

Usage

```
nuclear
```

Format

This data frame contains the following columns:

`cost` The capital cost of construction in millions of dollars adjusted to 1976 base.

`date` The date on which the construction permit was issued. The data are measured in years since January 1 1990 to the nearest month.

`t1` The time between application for and issue of the construction permit.

`t2` The time between issue of operating license and construction permit.

`cap` The net capacity of the power plant (MWe).

`pr` A binary variable where `1` indicates the prior existence of a LWR plant at the same site.

`ne` A binary variable where `1` indicates that the plant was constructed in the north-east region of the U.S.A.

`ct` A binary variable where `1` indicates the use of a cooling tower in the plant.

`bw` A binary variable where `1` indicates that the nuclear steam supply system was manufactured by Babcock-Wilcox.

`cum.n` The cumulative number of power plants constructed by each architect-engineer.

`pt` A binary variable where `1` indicates those plants with partial turnkey guarantees.

Source

The data were obtained from

Cox, D.R. and Snell, E.J. (1981) *Applied Statistics: Principles and Examples*. Chapman and Hall.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

paulsen *Neurotransmission in Guinea Pig Brains*

Description

The `paulsen` data frame has 346 rows and 1 columns.

Sections were prepared from the brain of adult guinea pigs. Spontaneous currents that flowed into individual brain cells were then recorded and the peak amplitude of each current measured. The aim of the experiment was to see if the current flow was quantal in nature (i.e. that it is not a single burst but instead is built up of many smaller bursts of current). If the current was indeed quantal then it would be expected that the distribution of the current amplitude would be multimodal with modes at regular intervals. The modes would be expected to decrease in magnitude for higher current amplitudes.

Usage

```
paulsen
```

Format

This data frame contains the following column:

`y` The current flowing into individual brain cells. The currents are measured in pico-amperes.

Source

The data were kindly made available by Dr. O. Paulsen from the Department of Pharmacology at the University of Oxford.

Paulsen, O. and Heggelund, P. (1994) The quantal size at retinogeniculate synapses determined from spontaneous and evoked EPSCs in guinea-pig thalamic slices. *Journal of Physiology*, **480**, 505–511.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

plot.boot *Plots of the Output of a Bootstrap Simulation*

Description

This takes a bootstrap object and produces plots for the bootstrap replicates of the variable of interest.

Usage

```
## S3 method for class 'boot':
plot(x, index=1, t0=NULL, t=NULL, jack=FALSE, qdist="norm",
     nclass=NULL, df, ...)
```

Arguments

x	An object of class "boot" returned from one of the bootstrap generation functions.
index	The index of the variable of interest within the output of boot.out. This is ignored if t and t0 are supplied.
t0	The original value of the statistic. This defaults to boot.out$t0[index] unless t is supplied when it defaults to NULL. In that case no vertical line is drawn on the histogram.
t	The bootstrap replicates of the statistic. Usually this will take on its default value of boot.out$t[,index], however it may be useful sometimes to supply a different set of values which are a function of boot.out$t.
jack	A logical value indicating whether a jackknife-after-bootstrap plot is required. The default is not to produce such a plot.
qdist	The distribution against which the Q-Q plot should be drawn. At present "norm" (normal distribution - the default) and "chisq" (chi-squared distribution) are the only possible values.
nclass	An integer giving the number of classes to be used in the bootstrap histogram. The default is the integer between 10 and 100 closest to ceiling(length(t)/25).
df	If qdist is "chisq" then this is the degrees of freedom for the chi-squared distribution to be used. It is a required argument in that case.
...	When jack is TRUE additional parameters to jack.after.boot can be supplied. See the help file for jack.after.boot for details of the possible parameters.

Details

This function will generally produce two side-by-side plots. The left plot will be a histogram of the bootstrap replicates. Usually the breaks of the histogram will be chosen so that t0 is at a breakpoint and all intervals are of equal length. A vertical dotted line indicates the position of t0. This cannot be done if t is supplied but t0 is not and so, in that case, the breakpoints are computed by hist using the nclass argument and no vertical line is drawn.

The second plot is a Q-Q plot of the bootstrap replicates. The order statistics of the replicates can be plotted against normal or chi-squared quantiles. In either case the expected line is also plotted. For the normal, this will have intercept mean(t) and slope sqrt(var(t)) while for the chi-squared it has intercept 0 and slope 1.

If jack is TRUE a third plot is produced beneath these two. That plot is the jackknife-after-bootstrap plot. This plot may only be requested when nonparametric simulation has been used. See jack.after.boot for further details of this plot.

Value

boot.out is returned invisibly.

Side Effects

All screens are closed and cleared and a number of plots are produced on the current graphics device. Screens are closed but not cleared at termination of this function.

See Also

boot, jack.after.boot, print.boot

Examples

```
# We fit an exponential model to the air-conditioning data and use
# that for a parametric bootstrap.  Then we look at plots of the
# resampled means.
air.rg <- function(data, mle)
     rexp(length(data), 1/mle)

air.boot <- boot(aircondit$hours, mean, R=999, sim="parametric",
                 ran.gen=air.rg, mle=mean(aircondit$hours))
plot(air.boot)

# In the difference of means example for the last two series of the
# gravity data
grav1 <- gravity[as.numeric(gravity[,2])>=7,]
grav.fun <- function(dat, w)
{    strata <- tapply(dat[, 2], as.numeric(dat[, 2]))
     d <- dat[, 1]
     ns <- tabulate(strata)
     w <- w/tapply(w, strata, sum)[strata]
     mns <- tapply(d * w, strata, sum)
     mn2 <- tapply(d * d * w, strata, sum)
     s2hat <- sum((mn2 - mns^2)/ns)
     c(mns[2]-mns[1],s2hat)
}

grav.boot <- boot(grav1, grav.fun, R=499, stype="w", strata=grav1[,2])
plot(grav.boot)
# now suppose we want to look at the studentized differences.
grav.z <- (grav.boot$t[,1]-grav.boot$t0[1])/sqrt(grav.boot$t[,2])
plot(grav.boot,t=grav.z,t0=0)

# In this example we look at the one of the partial correlations for the
# head dimensions in the dataset frets.
pcorr <- function( x )
{
#  Function to find the correlations and partial correlations between
#  the four measurements.
     v <- cor(x);
     v.d <- diag(var(x));
     iv <- solve(v);
     iv.d <- sqrt(diag(iv));
     iv <- - diag(1/iv.d) %*% iv %*% diag(1/iv.d);
     q <- NULL;
     n <- nrow(v);
     for (i in 1:(n-1))
          q <- rbind( q, c(v[i,1:i],iv[i,(i+1):n]) );
     q <- rbind( q, v[n,] );
     diag(q) <- round(diag(q));
     q
}

frets.fun <- function( data, i )
```

```
{    d <- data[i,];
     v <- pcorr( d );
     c(v[1,],v[2,],v[3,],v[4,])
}
frets.boot <- boot(log(as.matrix(frets)), frets.fun, R=999)
plot(frets.boot, index=7, jack=TRUE, stinf=FALSE, useJ=FALSE)
```

`poisons` *Animal Survival Times*

Description

The `poisons` data frame has 48 rows and 3 columns.

The data form a 3x4 factorial experiment, the factors being three poisons and four treatments. Each combination of the two factors was used for four animals, the allocation to animals having been completely randomized.

Usage

```
poisons
```

Format

This data frame contains the following columns:

`time` The survival time of the animal in units of 10 hours.

`poison` A factor with levels `1`, `2` and `3` giving the type of poison used.

`treat` A factor with levels `A`, `B`, `C` and `D` giving the treatment.

Source

The data were obtained from

Box, G.E.P. and Cox, D.R. (1964) An analysis of transformations (with Discussion). *Journal of the Royal Statistical Society, B*, **26**, 211–252.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

polar *Pole Positions of New Caledonian Laterites*

Description

The `polar` data frame has 50 rows and 2 columns.

The data are the pole positions from a paleomagnetic study of New Caledonian laterites.

Usage

```
polar
```

Format

This data frame contains the following columns:

`lat` The latitude (in degrees) of the pole position. Note that all latitudes are negative as the axis is taken to be in the lower hemisphere.

`long` The longitude (in degrees) of the pole position.

Source

The data were obtained from

Fisher, N.I., Lewis, T. and Embleton, B.J.J. (1987) *Statistical Analysis of Spherical Data*. Cambridge University Press.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

print.boot *Print a Summary of a Bootstrap Object*

Description

This is a method for the function print for objects of the class `"boot"`.

Usage

```
## S3 method for class 'boot':
print(x, digits = getOption("digits"),
      index = 1:ncol(boot.out$t), ...)
```

Arguments

x	A bootstrap output object of class "boot" generated by one of the bootstrap functions.
digits	The number of digits to be printed in the summary statistics.
index	Indices indicating for which elements of the bootstrap output summary statistics are required.
...	further arguments passed to or from other methods.

Details

For each statistic calculated in the bootstrap the original value and the bootstrap estimates of its bias and standard error are printed. If boot.out$t0 is missing (such as when it was created by a call to tsboot with orig.t=FALSE) the bootstrap mean and standard error are printed. If resampling was done using importance resampling weights, then the bootstrap estimates are reweighted as if uniform resampling had been done. The ratio importance sampling estimates are used and if there were a number of distributions then defensive mixture distributions are used. In this case an extra column with the mean of the observed bootstrap statistics is also printed.

Value

The bootstrap object is returned invisibly.

See Also

boot, censboot, imp.moments, plot.boot, tilt.boot, tsboot

---

print.bootci *Print Bootstrap Confidence Intervals*

Description

This is a method for the function print() to print objects of the class "bootci".

Usage

```
## S3 method for class 'bootci':
print(x, hinv = NULL, ...)
```

Arguments

x	The output from a call to boot.ci.
hinv	A transformation to be made to the interval end-points before they are printed.
...	further arguments passed to or from other methods.

Details

This function prints out the results from `boot.ci` in a "nice" format. It also notes whether the scale of the intervals is the original scale of the input to `boot.ci` or a different scale and whether the calculations were done on a transformed scale. It also looks at the order statistics that were used in calculating the intervals. If the smallest or largest values were used then it prints a message

```
Warning :  Intervals used Extreme Quantiles
```

Such intervals should be considered very unstable and not relied upon for inferences. Even if the extreme values are not used, it is possible that the intervals are unstable if they used quantiles close to the extreme values. The function alerts the user to intervals which use the upper or lower 10 order statistics with the message

```
Some intervals may be unstable
```

Value

The object `ci.out` is returned invisibly.

See Also

`boot.ci`

`print.saddle.distn` *Print Quantiles of Saddlepoint Approximations*

Description

This is a method for the function `print()` to print objects of class `"saddle.distn"`.

Usage

```
## S3 method for class 'saddle.distn':
print(x, ...)
```

Arguments

`x`	An object of class `"saddle.distn"` created by a call to `saddle.distn`.
`...`	further arguments passed to or from other methods.

Details

The quantiles of the saddlepoint approximation to the distribution are printed along with the original call and some other useful information.

Value

The input is returned invisibly.

See Also

`lines.saddle.distn`, `saddle.distn`

print.simplex *Print Solution to Linear Programming Problem*

Description

This is a method for the function `print()` to print objects of class `"simplex"`.

Usage

```
## S3 method for class 'simplex':
print(x, ...)
```

Arguments

`x`	An object of class `"simplex"` created by calling the function `simplex` to solve a linear programming problem.
`...`	further arguments passed to or from other methods.

Details

The coefficients of the objective function are printed. If a solution to the linear programming problem was found then the solution and the optimal value of the objective function are printed. If a feasible solution was found but the maximum number of iterations was exceeded then the last feasible solution and the objective function value at that point are printed. If no feasible solution could be found then a message stating that is printed.

Value

`x` is returned silently.

See Also

`simplex`

remission *Cancer Remission and Cell Activity*

Description

The `remission` data frame has 27 rows and 3 columns.

Usage

```
remission
```

Format

This data frame contains the following columns:

LI A measure of cell activity.

m The number of patients in each group (all values are actually 1 here).

r The number of patients (out of m) who went into remission.

Source

The data were obtained from

Freeman, D.H. (1987) *Applied Categorical Data Analysis*. Marcel Dekker.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

---

saddle *Saddlepoint Approximations for Bootstrap Statistics*

Description

This function calculates a saddlepoint approximation to the distribution of a linear combination of **W** at a particular point u, where **W** is a vector of random variables. The distribution of **W** may be multinomial (default), Poisson or binary. Other distributions are possible also if the adjusted cumulant generating function and its second derivative are given. Conditional saddlepoint approximations to the distribution of one linear combination given the values of other linear combinations of **W** can be calculated for **W** having binary or Poisson distributions.

Usage

```
saddle(A=NULL, u=NULL, wdist="m", type="simp", d=NULL, d1=1,
       init=rep(0.1, d), mu=rep(0.5, n), LR=FALSE, strata=NULL,
       K.adj=NULL, K2=NULL)
```

Arguments

A	A vector or matrix of known coefficients of the linear combinations of **W**. It is a required argument unless K.adj and K2 are supplied, in which case it is ignored.
u	The value at which it is desired to calculate the saddlepoint approximation to the distribution of the linear combination of **W**. It is a required argument unless K.adj and K2 are supplied, in which case it is ignored.
wdist	The distribution of **W**. This can be one of "m" (multinomial), "p" (Poisson), "b" (binary) or "o" (other). If K.adj and K2 are given wdist is set to "o".
type	The type of saddlepoint approximation. Possible types are "simp" for simple saddlepoint and "cond" for the conditional saddlepoint. When wdist is "o" or "m", type is automatically set to "simp", which is the only type of saddlepoint currently implemented for those distributions.

d	This specifies the dimension of the whole statistic. This argument is required only when wdist="o" and defaults to 1 if not supplied in that case. For other distributions it is set to ncol(A).
d1	When type is "cond" this is the dimension of the statistic of interest which must be less than length(u). Then the saddlepoint approximation to the conditional distribution of the first d1 linear combinations given the values of the remaining combinations is found. Conditional distribution function approximations can only be found if the value of d1 is 1.
init	Used if wdist is either "m" or "o", this gives initial values to nlmin which is used to solve the saddlepoint equation.
mu	The values of the parameters of the distribution of **W** when wdist is "m", "p" "b". mu must be of the same length as W (i.e. nrow(A)). The default is that all values of mu are equal and so the elements of **W** are identically distributed.
LR	If TRUE then the Lugananni-Rice approximation to the cdf is used, otherwise the approximation used is based on Barndorff-Nielsen's r*.
strata	The strata for stratified data.
K.adj	The adjusted cumulant generating function used when wdist is "o". This is a function of a single parameter, zeta, which calculates K(zeta)-u%*%zeta, where K(zeta) is the cumulant generating function of **W**.
K2	This is a function of a single parameter zeta which returns the matrix of second derivatives of K(zeta) for use when wdist is "o". If K.adj is given then this must be given also. It is called only once with the calculated solution to the saddlepoint equation being passed as the argument. This argument is ignored if K.adj is not supplied.

Details

If wdist is "o" or "m", the saddlepoint equations are solved using nlmin to minimize K.adj with respect to its parameter zeta. For the Poisson and binary cases, a generalized linear model is fitted such that the parameter estimates solve the saddlepoint equations. The response variable 'y' for the glm must satisfy the equation t(A)%*%y=u (t() being the transpose function). Such a vector can be found as a feasible solution to a linear programming problem. This is done by a call to simplex. The covariate matrix for the glm is given by A.

Value

A list consisting of the following components

spa	The saddlepoint approximations. The first value is the density approximation and the second value is the distribution function approximation.
zeta.hat	The solution to the saddlepoint equation. For the conditional saddlepoint this is the solution to the saddlepoint equation for the numerator.
zeta2.hat	If type is "cond" this is the solution to the saddlepoint equation for the denominator. This component is not returned for any other value of type.

References

Booth, J.G. and Butler, R.W. (1990) Randomization distributions and saddlepoint approximations in generalized linear models. *Biometrika*, **77**, 787–796.

Canty, A.J. and Davison, A.C. (1997) Implementation of saddlepoint approximations to resampling distributions. *Computing Science and Statistics; Proceedings of the 28th Symposium on the Interface*, 248–253.

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and their Application*. Cambridge University Press.

Jensen, J.L. (1995) *Saddlepoint Approximations*. Oxford University Press.

See Also

saddle.distn, simplex

Examples

```
# To evaluate the bootstrap distribution of the mean failure time of
# air-conditioning equipment at 80 hours
saddle(A=aircondit$hours/12, u=80)

# Alternatively this can be done using a conditional poisson
saddle(A=cbind(aircondit$hours/12,1), u=c(80,12), wdist="p", type="cond")

# To use the Lugananni-Rice approximation to this
saddle(A=cbind(aircondit$hours/12,1), u=c(80,12), wdist="p", type="cond",
       LR = TRUE)

# Example 9.16 of Davison and Hinkley (1997) calculates saddlepoint
# approximations to the distribution of the ratio statistic for the
# city data. Since the statistic is not in itself a linear combination
# of random Variables, its distribution cannot be found directly.
# Instead the statistic is expressed as the solution to a linear
# estimating equation and hence its distribution can be found.  We
# get the saddlepoint approximation to the pdf and cdf evaluated at
# t=1.25 as follows.
jacobian <- function(dat,t,zeta)
{
     p <- exp(zeta*(dat$x-t*dat$u))
     abs(sum(dat$u*p)/sum(p))
}
city.sp1 <- saddle(A=city$x-1.25*city$u, u=0)
city.sp1$spa[1] <- jacobian(city, 1.25, city.sp1$zeta.hat) * city.sp1$spa[1]
city.sp1
```

---

saddle.distn *Saddlepoint Distribution Approximations for Bootstrap Statistics*

Description

Approximate an entire distribution using saddlepoint methods. This function can calculate simple and conditional saddlepoint distribution approximations for a univariate quantity of interest. For the simple saddlepoint the quantity of interest is a linear combination of **W** where **W** is a vector of random variables. For the conditional saddlepoint we require the distribution of one linear combination given the values of any number of other linear combinations. The distribution of **W** must be one of multinomial, Poisson or binary. The primary use of this function is to calculate quantiles of bootstrap distributions using saddlepoint approximations. Such quantiles are required by the library function control to approximate the distribution of the linear approximation to a statistic.

Usage

```
saddle.distn(A, u=NULL, alpha=NULL, wdist="m",
             type="simp", npts=20, t=NULL, t0=NULL,
             init=rep(0.1, d), mu=rep(0.5, n), LR=FALSE,
             strata=NULL, ...)
```

Arguments

A	This is a matrix of known coefficients or a function which returns such a matrix. If a function then its first argument must be the point t at which a saddlepoint is required. The most common reason for A being a function would be if the statistic is not itself a linear combination of the **W** but is the solution to a linear estimating equation.
u	If A is a function then u must also be a function returning a vector with length equal to the number of columns of the matrix returned by A. Usually all components other than the first will be constants as the other components are the values of the conditioning variables. If A is a matrix with more than one column (such as when wdist="cond") then u should be a vector with length one less than ncol(A). In this case u specifies the values of the conditioning variables. If A is a matrix with one column or a vector then u is not used.
alpha	The alpha levels for the quantiles of the distribution which should be returned. By default the 0.1, 0.5, 1, 2.5, 5, 10, 20, 50, 80, 90, 95, 97.5, 99, 99.5 and 99.9 percentiles are calculated.
wdist	The distribution of **W**. Possible values are "m" (multinomial), "p" (Poisson), or "b" (binary).
type	The type of saddlepoint to be used. Possible values are "simp" (simple saddlepoint) and "cond" (conditional). If wdist is "m", type is set to "simp".
npts	The number of points at which the saddlepoint approximation should be calculated and then used to fit the spline.
t	A vector of points at which the saddlepoint approximations are calculated. These points should extend beyond the extreme quantiles required but still be in the possible range of the bootstrap distribution. The observed value of the statistic should not be included in t as the distribution function approximation breaks down at that point. The points should, however cover the entire effective range of the distribution including close to the centre. If t is supplied then npts is set to length(t). When t is not supplied, the function attempts to find the effective range of the distribution and then selects points to cover this range.
t0	If t is not supplied then a vector of length 2 should be passed as t0. The first component of t0 should be the centre of the distribution and the second should be an estimate of spread (such as a standard error). These two are then used to find the effective range of the distribution. The range finding mechanism does rely on an accurate estimate of location in t0[1].
init	When wdist is "m", this vector should contain the initial values to be passed to nlmin when it is called to solve the saddlepoint equations.
mu	The vector of parameter values for the distribution. The default is that the components of **W** are identically distributed.
LR	A logical flag. When LR is TRUE the Lugananni-Rice cdf approximations are calculated and used to fit the spline. Otherwise the cdf approximations used are based on Barndorff-Nielsen's r*.

strata	A vector giving the strata when the rows of A relate to stratified data. This is used only when wdist is "m".
...	When A and u are functions any additional arguments are passed unchanged each time one of them is called.

Details

The range at which the saddlepoint is used is such that the cdf approximation at the endpoints is more extreme than required by the extreme values of alpha. The lower endpoint is found by evaluating the saddlepoint at the points t0[1]-2*t0[2], t0[1]-4*t0[2], t0[1]-8*t0[2] etc. until a point is found with a cdf approximation less than min(alpha)/10, then a bisection method is used to find the endpoint which has cdf approximation in the range (min(alpha)/1000, min(alpha)/10). Then a number of, equally spaced, points are chosen between the lower endpoint and t0[1] until a total of npts/2 approximations have been made. The remaining npts/2 points are chosen to the right of t0[1] in a similar manner. Any points which are very close to the centre of the distribution are then omitted as the cdf approximations are not reliable at the centre. A smoothing spline is then fitted to the probit of the saddlepoint distribution function approximations at the remaining points and the required quantiles are predicted from the spline.

Sometimes the function will terminate with the message "Unable to find range". There are two main reasons why this may occur. One is that the distribution is too discrete and/or the required quantiles too extreme, this can cause the function to be unable to find a point within the allowable range which is beyond the extreme quantiles. Another possibility is that the value of t0[2] is too small and so too many steps are required to find the range. The first problem cannot be solved except by asking for less extreme quantiles, although for very discrete distributions the approximations may not be very good. In the second case using a larger value of t0[2] will usually solve the problem.

Value

The returned value is an object of class "saddle.distn". See the help file for saddle.distn.object for a description of such an object.

References

Booth, J.G. and Butler, R.W. (1990) Randomization distributions and saddlepoint approximations in generalized linear models. *Biometrika*, **77**, 787–796.

Canty, A.J. and Davison, A.C. (1997) Implementation of saddlepoint approximations to resampling distributions. *Computing Science and Statistics; Proceedings of the 28th Symposium on the Interface* 248–253.

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and their Application*. Cambridge University Press.

Jensen, J.L. (1995) *Saddlepoint Approximations*. Oxford University Press.

See Also

lines.saddle.distn, saddle, saddle.distn.object, smooth.spline

Examples

```
#  The bootstrap distribution of the mean of the air-conditioning
#  failure data: fails to find value on R (and probably on S too)
```

```
air.t0 <- c(mean(aircondit$hours), sqrt(var(aircondit$hours)/12))
## Not run: saddle.distn(A = aircondit$hours/12, t0 = air.t0)

# alternatively using the conditional poisson
saddle.distn(A = cbind(aircondit$hours/12, 1), u = 12, wdist = "p",
             type = "cond", t0 = air.t0)

# Distribution of the ratio of a sample of size 10 from the bigcity
# data, taken from Example 9.16 of Davison and Hinkley (1997).
ratio <- function(d, w) sum(d$x *w)/sum(d$u * w)
city.v <- var.linear(empinf(data = city, statistic = ratio))
bigcity.t0 <- c(mean(bigcity$x)/mean(bigcity$u), sqrt(city.v))
Afn <- function(t, data) cbind(data$x - t*data$u, 1)
ufn <- function(t, data) c(0,10)
saddle.distn(A = Afn, u = ufn, wdist = "b", type = "cond",
             t0 = bigcity.t0, data = bigcity)

# From Example 9.16 of Davison and Hinkley (1997) again, we find the
# conditional distribution of the ratio given the sum of city$u.
Afn <- function(t, data) cbind(data$x-t*data$u, data$u, 1)
ufn <- function(t, data) c(0, sum(data$u), 10)
city.t0 <- c(mean(city$x)/mean(city$u), sqrt(city.v))
saddle.distn(A = Afn, u = ufn, wdist = "p", type = "cond", t0 = city.t0,
             data = city)
```

---

saddle.distn.object

Saddlepoint Distribution Approximation Objects

Description

Class of objects that result from calculating saddlepoint distribution approximations by a call to `saddle.distn`.

Generation

This class of objects is returned from calls to the function `saddle.distn`.

Methods

The class `"saddle.distn"` has methods for the functions `lines` and `print`.

Structure

Objects of class `"saddle.distn"` are implemented as a list with the following components.

quantiles A matrix with 2 columns. The first column contains the probabilities `alpha` and the second column contains the estimated quantiles of the distribution at those probabilities derived from the spline.

points A matrix of evaluations of the saddlepoint approximation. The first column contains the values of `t` which were used, the second and third contain the density and cdf approximations at those points and the rest of the columns contain the solutions to the saddlepoint equations. When `type` is `"simp"`, there is only one of those. When `type` is `"cond"` there are `2*d-1` where `d` is the

number of columns in `A` or the output of `A(t,...{})`. The first `d` of these correspond to the numerator and the remainder correspond to the denominator.

distn An object of class `smooth.spline`. This corresponds to the spline fitted to the saddlepoint cdf approximations in points in order to approximate the entire distribution. For the structure of the object see `smooth.spline`.

call The original call to `saddle.distn` which generated the object.

LR A logical variable indicating whether the Luganani-Rice approximations were used.

See Also

`lines.saddle.distn`, `saddle.distn`, `print.saddle.distn`

`salinity` *Water Salinity and River Discharge*

Description

The `salinity` data frame has 28 rows and 4 columns.

Biweekly averages of the water salinity and river discharge in Pamlico Sound, North Carolina were recorded between the years 1972 and 1977. The data in this set consists only of those measurements in March, April and May.

Usage

```
salinity
```

Format

This data frame contains the following columns:

`sal` The average salinity of the water over two weeks.

`lag` The average salinity of the water lagged two weeks. Since only spring is used, the value of `lag` is not always equal to the previous value of `sal`.

`trend` A factor indicating in which of the 6 biweekly periods between March and May, the observations were taken. The levels of the factor are from 0 to 5 with 0 being the first two weeks in March.

`dis` The amount of river discharge during the two weeks for which `sal` is the average salinity.

Source

The data were obtained from

Ruppert, D. and Carroll, R.J. (1980) Trimmed least squares estimation in the linear model. *Journal of the American Statistical Association*, **75**, 828–838.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

`simplex` *Simplex Method for Linear Programming Problems*

Description

This function will optimize the linear function `a%*%x` subject to the constraints `A1%*%x <= b1`, `A2%*%x >= b2`, `A3%*%x = b3` and `x >= 0`. Either maximization or minimization is possible but the default is minimization.

Usage

```
simplex(a, A1 = NULL, b1 = NULL, A2 = NULL, b2 = NULL, A3 = NULL,
        b3 = NULL, maxi = FALSE, n.iter = n + 2 * m, eps = 1e-10)
```

Arguments

`a`	A vector of length `n` which gives the coefficients of the objective function.
`A1`	An `m1` by `n` matrix of coefficients for the "<=" type of constraints.
`b1`	A vector of length `m1` giving the right hand side of the "<=" constraints. This argument is required if `A1` is given and ignored otherwise. All values in `b1` must be non-negative.
`A2`	An `m2` by `n` matrix of coefficients for the ">=" type of constraints.
`b2`	A vector of length `m2` giving the right hand side of the ">=" constraints. This argument is required if `A2` is given and ignored otherwise. All values in `b2` must be non-negative. Note that the constraints `x >= 0` are included automatically and so should not be repeated here.
`A3`	An `m3` by `n` matrix of coefficients for the equality constraints.
`b3`	A vector of length `m3` giving the right hand side of equality constraints. This argument is required if `A3` is given and ignored otherwise. All values in `b3` must be non-negative.
`maxi`	A logical flag which specifies minimization if `FALSE` (default) and maximization otherwise. If `maxi` is `TRUE` then the maximization problem is recast as a minimization problem by changing the objective function coefficients to their negatives.
`n.iter`	The maximum number of iterations to be conducted in each phase of the simplex method. The default is `n+2*(m1+m2+m3)`.
`eps`	The floating point tolerance to be used in tests of equality.

Details

The method employed by this function is the two phase tableau simplex method. If there are ">=" or equality constraints an initial feasible solution is not easy to find. To find a feasible solution an artificial variable is introduced into each ">=" or equality constraint and an auxiliary objective function is defined as the sum of these artificial variables. If a feasible solution to the set of constraints exists then the auxiliary objective will be minimized when all of the artificial variables are 0. These are then discarded and the original problem solved starting at the solution to the auxiliary problem. If the only constraints are of the "<=" form, the origin is a feasible solution and so the first stage can be omitted.

Value

An object of class "simplex": see simplex.object.

Note

The method employed here is suitable only for relatively small systems. Also if possible the number of constraints should be reduced to a minimum in order to speed up the execution time which is approximately proportional to the cube of the number of constraints. In particular if there are any constraints of the form x[i] >= b2[i] they should be omitted by setting x[i] = x[i]-b2[i], changing all the constraints and the objective function accordingly and then transforming back after the solution has been found.

References

Gill, P.E., Murray, W. and Wright, M.H. (1991) *Numerical Linear Algebra and Optimization Vol. 1*. Addison-Wesley.

Press, W.H., Teukolsky, S.A., Vetterling, W.T. and Flannery, B.P. (1992) *Numerical Recipes: The Art of Scientific Computing (Second Edition)*. Cambridge University Press.

Examples

```
#  This example is taken from Exercise 7.5 of Gill, Murray,
#  and Wright (1991).
enj <- c(200, 6000, 3000, -200)
fat <- c(800, 6000, 1000, 400)
vitx <- c(50, 3, 150, 100)
vity <- c(10, 10, 75, 100)
vitz <- c(150, 35, 75, 5)
simplex(a = enj, A1 = fat, b1 = 13800, A2 = rbind(vitx, vity, vitz),
        b2 = c(600, 300, 550), maxi = TRUE)
```

---

simplex.object *Linear Programming Solution Objects*

Description

Class of objects that result from solving a linear programming problem using simplex.

Generation

This class of objects is returned from calls to the function simplex.

Methods

The class "saddle.distn" has a method for the function print.

Structure

Objects of class `"simplex"` are implemented as a list with the following components.

soln The values of `x` which optimize the objective function under the specified constraints provided those constraints are jointly feasible.

solved This indicates whether the problem was solved. A value of `-1` indicates that no feasible solution could be found. A value of `0` that the maximum number of iterations was reached without termination of the second stage. This may indicate an unbounded function or simply that more iterations are needed. A value of `1` indicates that an optimal solution has been found.

value The value of the objective function at `soln`.

val.aux This is `NULL` if a feasible solution is found. Otherwise it is a positive value giving the value of the auxiliary objective function when it was minimized.

obj The original coefficients of the objective function.

a The objective function coefficients re-expressed such that the basic variables have coefficient zero.

a.aux This is `NULL` if a feasible solution is found. Otherwise it is the re-expressed auxiliary objective function at the termination of the first phase of the simplex method.

A The final constraint matrix which is expressed in terms of the non-basic variables. If a feasible solution is found then this will have dimensions `m1+m2+m3` by `n+m1+m2`, where the final `m1+m2` columns correspond to slack and surplus variables. If no feasible solution is found there will be an additional `m1+m2+m3` columns for the artificial variables introduced to solve the first phase of the problem.

basic The indices of the basic (non-zero) variables in the solution. Indices between `n+1` and `n+m1` correspond to slack variables, those between `n+m1+1` and `n+m2` correspond to surplus variables and those greater than `n+m2` are artificial variables. Indices greater than `n+m2` should occur only if `solved` is `-1` as the artificial variables are discarded in the second stage of the simplex method.

slack The final values of the `m1` slack variables which arise when the "<=" constraints are re-expressed as the equalities `A1%*%x + slack = b1`.

surplus The final values of the `m2` surplus variables which arise when the "<=" constraints are re-expressed as the equalities `A2%*%x - surplus = b2`.

artificial This is NULL if a feasible solution can be found. If no solution can be found then this contains the values of the `m1+m2+m3` artificial variables which minimize their sum subject to the original constraints. A feasible solution exists only if all of the artificial variables can be made 0 simultaneously.

See Also

`print.simplex`, `simplex`

`smooth.f` *Smooth Distributions on Data Points*

Description

This function uses the method of frequency smoothing to find a distribution on a data set which has a required value, `theta`, of the statistic of interest. The method results in distributions which vary smoothly with `theta`.

Usage

```
smooth.f(theta, boot.out, index=1, t=boot.out$t[, index], width=0.5)
```

Arguments

`theta`	The required value for the statistic of interest. If `theta` is a vector, a separate distribution will be found for each element of `theta`.
`boot.out`	A bootstrap output object returned by a call to `boot`.
`index`	The index of the variable of interest in the output of `boot.out$statistic`. This argument is ignored if `t` is supplied. `index` must be a scalar.
`t`	The bootstrap values of the statistic of interest. This must be a vector of length `boot.out$R` and the values must be in the same order as the bootstrap replicates in `boot.out`.
`width`	The standardized width for the kernel smoothing. The smoothing uses a value of `width*s` for epsilon, where `s` is the bootstrap estimate of the standard error of the statistic of interest. `width` should take a value in the range (0.2, 1) to produce a reasonable smoothed distribution. If `width` is too large then the distribution becomes closer to uniform.

Details

The new distributional weights are found by applying a normal kernel smoother to the observed values of `t` weighted by the observed frequencies in the bootstrap simulation. The resulting distribution may not have parameter value exactly equal to the required value `theta` but it will typically have a value which is close to `theta`. The details of how this method works can be found in Davison, Hinkley and Worton (1995) and Section 3.9.2 of Davison and Hinkley (1997).

Value

If `length(theta)` is 1 then a vector with the same length as the data set `boot.out$data` is returned. The value in position `i` is the probability to be given to the data point in position `i` so that the distribution has parameter value approximately equal to `theta`. If `length(theta)` is bigger than 1 then the returned value is a matrix with `length(theta)` rows each of which corresponds to a distribution with the parameter value approximately equal to the corresponding value of `theta`.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

Davison, A.C., Hinkley, D.V. and Worton, B.J. (1995) Accurate and efficient construction of bootstrap likelihoods. *Statistics and Computing*, **5**, 257–264.

See Also

boot, exp.tilt, tilt.boot

Examples

```
# Example 9.8 of Davison and Hinkley (1997) requires tilting the resampling
# distribution of the studentized statistic to be centred at the observed
# value of the test statistic 1.84.  In the book exponential tilting was used
# but it is also possible to use smooth.f.
grav1 <- gravity[as.numeric(gravity[,2])>=7,]
grav.fun <- function(dat, w, orig)
{    strata <- tapply(dat[, 2], as.numeric(dat[, 2]))
     d <- dat[, 1]
     ns <- tabulate(strata)
     w <- w/tapply(w, strata, sum)[strata]
     mns <- tapply(d * w, strata, sum)
     mn2 <- tapply(d * d * w, strata, sum)
     s2hat <- sum((mn2 - mns^2)/ns)
     c(mns[2]-mns[1], s2hat, (mns[2]-mns[1]-orig)/sqrt(s2hat))
}
grav.z0 <- grav.fun(grav1,rep(1,26),0)
grav.boot <- boot(grav1, grav.fun, R=499, stype="w",
                  strata=grav1[,2], orig=grav.z0[1])
grav.sm <- smooth.f(grav.z0[3], grav.boot, index=3)

# Now we can run another bootstrap using these weights
grav.boot2 <- boot(grav1, grav.fun, R=499, stype="w",
                   strata=grav1[,2], orig=grav.z0[1],
                   weights=grav.sm)

# Estimated p-values can be found from these as follows
mean(grav.boot$t[,3] >= grav.z0[3])
imp.prob(grav.boot2,t0=-grav.z0[3],t=-grav.boot2$t[,3])

# Note that for the importance sampling probability we must
# multiply everything by -1 to ensure that we find the correct
# probability.  Raw resampling is not reliable for probabilities
# greater than 0.5. Thus
1-imp.prob(grav.boot2,index=3,t0=grav.z0[3])$raw
# can give very strange results (negative probabilities).
```

`sunspot` *Annual Mean Sunspot Numbers*

Description

`sunspot` is a time series of class `"rts"` and contains 289 observations.

The Zurich sunspot numbers have been analyzed in almost all books on time series analysis as well as numerous papers. The data set, usually attributed to Rudolf Wolf, consists of means of daily relative numbers of sunspot sightings. The relative number for a day is given by k(f+10g) where g is the number of sunspot groups observed, f is the total number of spots within the groups and k is a scaling factor relating the observer and telescope to a baseline. The relative numbers are then averaged to give an annual figure. See Insenman (1983) for a discussion of the relative numbers. The figures are for the years 1700-1988.

Note

These data are not related to dataset of the same name in the `trellis` library. They are, however, similar to the `S-Plus` dataset `sunspots` except that this set contains annual averages instead of monthly averages and it is for a longer period of time.

Source

The data were obtained from

Tong, H. (1990) *Nonlinear Time Series: A Dynamical System Approach*. Oxford University Press

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

Inzenman, A.J. (1983) J.R. Wolf and H.A. Wolfer: An historical note on the Zurich sunspot relative numbers. *Journal of the Royal Statistical Society, A*, **146**, 311-318.

Waldmeir, M. (1961) *The Sunspot Activity in the Years 1610-1960*. Schulthess and Co.

`survival` *Survival of Rats after Radiation Doses*

Description

The `survival` data frame has 14 rows and 2 columns.

The data measured the survival percentages of batches of rats who were given varying doses of radiation. At each of 6 doses there were two or three replications of the experiment.

Usage

```
survival
```

Format

This data frame contains the following columns:

`dose` The dose of radiation administered (rads).

`surv` The survival rate of the batches expressed as a percentage.

Source

The data were obtained from

Efron, B. (1988) Computer-intensive methods in statistical regression. *SIAM Review*, **30**, 421–449.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

`tau` *Tau Particle Decay Modes*

Description

The `tau` data frame has 60 rows and 2 columns.

The tau particle is a heavy electron-like particle discovered in the 1970's by Martin Perl at the Stanford Linear Accelerator Center. Soon after its production the tau particle decays into various collections of more stable particles. About 86% of the time the decay involves just one charged particle. This rate has been measured independently 13 times.

The one-charged-particle event is made up of four major modes of decay as well as a collection of other events. The four main types of decay are denoted rho, pi, e and mu. These rates have been measured independently 6, 7, 14 and 19 times respectively. Due to physical constraints each experiment can only estimate the composite one-charged-particle decay rate or the rate of one of the major modes of decay.

Each experiment consists of a major research project involving many years work. One of the goals of the experiments was to estimate the rate of decay due to events other than the four main modes of decay. These are uncertain events and so cannot themselves be observed directly.

Usage

```
tau
```

Format

This data frame contains the following columns:

`rate` The decay rate expressed as a percentage.

`decay` The type of decay measured in the experiment. It is a factor with levels `1`, `rho`, `pi`, `e` and `mu`.

Source

The data were obtained from

Efron, B. (1992) Jackknife-after-bootstrap standard errors and influence functions (with Discussion). *Journal of the Royal Statistical Society, B*, **54**, 83–127.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

Hayes, K.G., Perl, M.L. and Efron, B. (1989) Application of the bootstrap statistical method to the tau-decay-mode problem. *Physical Review, D*, **39**, 274-279.

---

tilt.boot *Non-parametric Tilted Bootstrap*

Description

This function will run an initial bootstrap with equal resampling probabilities (if required) and will use the output of the initial run to find resampling probabilities which put the value of the statistic at required values. It then runs an importance resampling bootstrap using the calculated probabilities as the resampling distribution.

Usage

```
tilt.boot(data, statistic, R, sim="ordinary", stype="i",
          strata=rep(1, n), L=NULL, theta=NULL,
          alpha=c(0.025, 0.975), tilt=TRUE, width=0.5,
          index=1, ...)
```

Arguments

`data`	The data as a vector, matrix or data frame. If it is a matrix or data frame then each row is considered as one (multivariate) observation.
`statistic`	A function which when applied to data returns a vector containing the statistic(s) of interest. It must take at least two arguments. The first argument will always be `data` and the second should be a vector of indices, weights or frequencies describing the bootstrap sample. Any other arguments must be supplied to `tilt.boot` and will be passed unchanged to statistic each time it is called.
`R`	The number of bootstrap replicates required. This will generally be a vector, the first value stating how many uniform bootstrap simulations are to be performed at the initial stage. The remaining values of `R` are the number of simulations to be performed resampling from each reweighted distribution. The first value of `R` must always be present, a value of 0 implying that no uniform resampling is to be carried out. Thus `length(R)` should always equal `1+length(theta)`.
`sim`	This is a character string indicating the type of bootstrap simulation required. There are only two possible values that this can take: `"ordinary"` and `"balanced"`. If other simulation types are required for the initial un-weighted bootstrap then it will be necessary to run `boot`, calculate the weights appropriately, and run `boot` again using the calculated weights.
`stype`	A character string indicating the type of second argument expected by `statistic`. The possible values that `stype` can take are `"i"` (indices), `"w"` (weights) and `"f"` (frequencies).
`strata`	An integer vector or factor representing the strata for multi-sample problems.

L	The empirical influence values for the statistic of interest. They are used only for exponential tilting when tilt is TRUE. If tilt is TRUE and they are not supplied then tilt.boot uses empinf to calculate them.
theta	The required parameter value(s) for the tilted distribution(s). There should be one value of theta for each of the non-uniform distributions. If R[1] is 0 theta is a required argument. Otherwise theta values can be estimated from the initial uniform bootstrap and the values in alpha.
alpha	The alpha level to which tilting is required. This parameter is ignored if R[1] is 0 or if theta is supplied, otherwise it is used to find the values of theta as quantiles of the initial uniform bootstrap. In this case R[1] should be large enough that min(c(alpha, 1-alpha))*R[1] > 5, if this is not the case then a warning is generated to the effect that the theta are extreme values and so the tilted output may be unreliable.
tilt	A logical variable which if TRUE (the default) indicates that exponential tilting should be used, otherwise local frequency smoothing (smooth.f) is used. If tilt is FALSE then R[1] must be positive. In fact in this case the value of R[1] should be fairly large (in the region of 500 or more).
width	This argument is used only if tilt is FALSE, in which case it is passed unchanged to smooth.f as the standardized bandwidth for the smoothing operation. The value should generally be in the range (0.2, 1). See smooth.f for for more details.
index	The index of the statistic of interest in the output from statistic. By default the first element of the output of statistic is used.
...	Any additional arguments required by statistic. These are passed unchanged to statistic each time it is called.

Value

An object of class "boot" with the following components

t0	The observed value of the statistic on the original data.
t	The values of the bootstrap replicates of the statistic. There will be sum(R) of these, the first R[1] corresponding to the uniform bootstrap and the remainder to the tilted bootstrap(s).
R	The input vector of the number of bootstrap replicates.
data	The original data as supplied.
statistic	The statistic function as supplied.
sim	The simulation type used in the bootstrap(s), it can either be "ordinary" or "balanced".
stype	The type of statistic supplied, it is the same as the input value stype.
call	A copy of the original call to tilt.boot.
strata	The strata as supplied.
weights	The matrix of weights used. If R[1] is greater than 0 then the first row will be the uniform weights and each subsequent row the tilted weights. If R[1] equals 0 then the uniform weights are omitted and only the tilted weights are output.
theta	The values of theta used for the tilted distributions. These are either the input values or the values derived from the uniform bootstrap and alpha.

References

Booth, J.G., Hall, P. and Wood, A.T.A. (1993) Balanced importance resampling for the bootstrap. *Annals of Statistics*, **21**, 286–298.

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

Hinkley, D.V. and Shi, S. (1989) Importance sampling and the nested bootstrap. *Biometrika*, **76**, 435–446.

See Also

boot, exp.tilt, Imp.Estimates, imp.weights, smooth.f

Examples

```
# Note that these examples can take a while to run.

# Example 9.9 of Davison and Hinkley (1997).
grav1 <- gravity[as.numeric(gravity[,2])>=7,]
grav.fun <- function(dat, w, orig)
{    strata <- tapply(dat[, 2], as.numeric(dat[, 2]))
     d <- dat[, 1]
     ns <- tabulate(strata)
     w <- w/tapply(w, strata, sum)[strata]
     mns <- tapply(d * w, strata, sum)
     mn2 <- tapply(d * d * w, strata, sum)
     s2hat <- sum((mn2 - mns^2)/ns)
     c(mns[2]-mns[1],s2hat,(mns[2]-mns[1]-orig)/sqrt(s2hat))
}
grav.z0 <- grav.fun(grav1,rep(1,26),0)
tilt.boot(grav1, grav.fun, R=c(249,375,375), stype="w",
          strata=grav1[,2], tilt=TRUE, index=3, orig=grav.z0[1])

#  Example 9.10 of Davison and Hinkley (1997) requires a balanced
#  importance resampling bootstrap to be run.  In this example we
#  show how this might be run.
acme.fun <- function(data, i, bhat)
{    d <- data[i,]
     n <- nrow(d)
     d.lm <- glm(d$acme~d$market)
     beta.b <- coef(d.lm)[2]
     d.diag <- glm.diag(d.lm)
     SSx <- (n-1)*var(d$market)
     tmp <- (d$market-mean(d$market))*d.diag$res*d.diag$sd
     sr <- sqrt(sum(tmp^2))/SSx
     c(beta.b, sr, (beta.b-bhat)/sr)
}
acme.b <- acme.fun(acme,1:nrow(acme),0)
acme.boot1 <- tilt.boot(acme, acme.fun, R=c(499, 250, 250),
                        stype="i", sim="balanced", alpha=c(0.05, 0.95),
                        tilt=TRUE, index=3, bhat=acme.b[1])
```

`tsboot` *Bootstrapping of Time Series*

Description

Generate `R` bootstrap replicates of a statistic applied to a time series. The replicate time series can be generated using fixed or random block lengths or can be model based replicates.

Usage

```
tsboot(tseries, statistic, R, l=NULL, sim="model", endcorr=TRUE,
       n.sim=NROW(tseries), orig.t=TRUE, ran.gen,
       ran.args=NULL, norm=TRUE, ...)
```

Arguments

`tseries`	A univariate or multivariate time series.
`statistic`	A function which when applied to `tseries` returns a vector containing the statistic(s) of interest. Each time `statistic` is called it is passed a time series of length `n.sim` which is of the same class as the original `tseries`. Any other arguments which `statistic` takes must remain constant for each bootstrap replicate and should be supplied through the `...` argument to `tsboot`.
`R`	A positive integer giving the number of bootstrap replicates required.
`sim`	The type of simulation required to generate the replicate time series. The possible input values are `"model"` (model based resampling), `"fixed"` (block resampling with fixed block lengths of `l`), `"geom"` (block resampling with block lengths having a geometric distribution with mean `l`) or `"scramble"` (phase scrambling).
`l`	If `sim` is `"fixed"` then `l` is the fixed block length used in generating the replicate time series. If `sim` is `"geom"` then `l` is the mean of the geometric distribution used to generate the block lengths. `l` should be a positive integer less than the length of `tseries`. This argument is not required when `sim` is `"model"` but it is required for all other simulation types.
`endcorr`	A logical variable indicating whether end corrections are to be applied when `sim` is `"fixed"`. When `sim` is `"geom"`, `endcorr` is automatically set to `TRUE`; `endcorr` is not used when `sim` is `"model"` or `"scramble"`.
`n.sim`	The length of the simulated time series. Typically this will be equal to the length of the original time series but there are situations when it will be larger. One obvious situation is if prediction is required. Another situation in which `n.sim` is larger than the original length is if `tseries` is a residual time series from fitting some model to the original time series. In this case, `n.sim` would usually be the length of the original time series.
`orig.t`	A logical variable which indicates whether `statistic` should be applied to `tseries` itself as well as the bootstrap replicate series. If `statistic` is expecting a longer time series than `tseries` or if applying `statistic` to `tseries` will not yield any useful information then `orig.t` should be set to `FALSE`.

ran.gen	This is a function of three arguments. The first argument is a time series. If sim is code"model" then it will always be tseries that is passed. For other simulation types it is the result of selecting n.sim observations from tseries by some scheme and converting the result back into a time series of the same form as tseries (although of length n.sim). The second argument to ran.gen is always the value n.sim, and the third argument is ran.args, which is used to supply any other objects needed by ran.gen. If sim is "model" then the generation of the replicate time series will be done in ran.gen (for example through use of arima.sim). For the other simulation types ran.gen is used for "post-blackening". The default is that the function simply returns the time series passed to it.
ran.args	This will be supplied to ran.gen each time it is called. If ran.gen needs any extra arguments then they should be supplied as components of ran.args. Multiple arguments may be passed by making ran.args a list. If ran.args is NULL then it should not be used within ran.gen but note that ran.gen must still have its third argument.
norm	A logical argument indicating whether normal margins should be used for phase scrambling. If norm is FALSE then margins corresponding to the exact empirical margins are used.
...	Any extra arguments to statistic may be supplied here.

Details

If sim is "fixed" then each replicate time series is found by taking blocks of length l, from the original time series and putting them end-to-end until a new series of length n.sim is created. When sim is "geom" a similar approach is taken except that now the block lengths are generated from a geometric distribution with mean l. Post-blackening can be carried out on these replicate time series by including the function ran.gen in the call to tsboot and having tseries as a time series of residuals.

Model based resampling is very similar to the parametric bootstrap and all simulation must be in one of the user specified functions. This avoids the complicated problem of choosing the block length but relies on an accurate model choice being made.

Phase scrambling is described in Section 8.2.4 of Davison and Hinkley (1997). The types of statistic for which this method produces reasonable results is very limited and the other methods seem to do better in most situations. Other types of resampling in the frequency domain can be accomplished using the function boot with the argument sim="parametric".

Value

An object of class "boot" with the following components.

t0	If orig.t is TRUE then t0 is the result of statistic(tseries,...{}) otherwise it is NULL.
t	The results of applying statistic to the replicate time series.
R	The value of R as supplied to tsboot.
tseries	The original time series.
statistic	The function statistic as supplied.
sim	The simulation type used in generating the replicates.
endcorr	The value of endcorr used. The value is meaningful only when sim is "fixed"; it is ignored for model based simulation or phase scrambling and is always set to TRUE if sim is "geom".

n.sim	The value of n.sim used.
l	The value of l used for block based resampling. This will be NULL if block based resampling was not used.
ran.gen	The ran.gen function used for generating the series or for "post-blackening".
ran.args	The extra arguments passed to ran.gen.
call	The original call to tsboot.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

Kunsch, H.R. (1989) The jackknife and the bootstrap for general stationary observations. *Annals of Statistics*, **17**, 1217–1241.

Politis, D.N. and Romano, J.P. (1994) The stationary bootstrap. *Journal of the American Statistical Association*, **89**, 1303–1313.

See Also

boot, arima.sim

Examples

```
lynx.fun <- function(tsb)
{    ar.fit <- ar(tsb, order.max=25)
     c(ar.fit$order, mean(tsb), tsb)
}

# the stationary bootstrap with mean block length 20
lynx.1 <- tsboot(log(lynx), lynx.fun, R=99, l=20, sim="geom")

# the fixed block bootstrap with length 20
lynx.2 <- tsboot(log(lynx), lynx.fun, R=99, l=20, sim="fixed")

# Now for model based resampling we need the original model
# Note that for all of the bootstraps which use the residuals as their
# data, we set orig.t to FALSE since the function applied to the residual
# time series will be meaningless.
lynx.ar <- ar(log(lynx))
lynx.model <- list(order=c(lynx.ar$order,0,0),ar=lynx.ar$ar)
lynx.res <- lynx.ar$resid[!is.na(lynx.ar$resid)]
lynx.res <- lynx.res - mean(lynx.res)

lynx.sim <- function(res,n.sim, ran.args) {
# random generation of replicate series using arima.sim
     rg1 <- function(n, res)
          sample(res, n, replace=TRUE)
     ts.orig <- ran.args$ts
     ts.mod <- ran.args$model
     mean(ts.orig)+ts(arima.sim(model=ts.mod, n=n.sim,
                      rand.gen=rg1, res=as.vector(res)))
}

lynx.3 <- tsboot(lynx.res, lynx.fun, R=99, sim="model", n.sim=114,
                 orig.t=FALSE, ran.gen=lynx.sim,
```

```
                  ran.args=list(ts=log(lynx), model=lynx.model))

#  For "post-blackening" we need to define another function
lynx.black <- function(res, n.sim, ran.args)
{    ts.orig <- ran.args$ts
     ts.mod <- ran.args$model
     mean(ts.orig) + ts(arima.sim(model=ts.mod,n=n.sim,innov=res))
}

# Now we can run apply the two types of block resampling again but this
# time applying post-blackening.
lynx.1b <- tsboot(lynx.res, lynx.fun, R=99, l=20, sim="fixed",
                  n.sim=114, orig.t=FALSE, ran.gen=lynx.black,
                  ran.args=list(ts=log(lynx), model=lynx.model))

lynx.2b <- tsboot(lynx.res, lynx.fun, R=99, l=20, sim="geom",
                  n.sim=114, orig.t=FALSE, ran.gen=lynx.black,
                  ran.args=list(ts=log(lynx), model=lynx.model))

# To compare the observed order of the bootstrap replicates we
# proceed as follows.
table(lynx.1$t[,1])
table(lynx.1b$t[,1])
table(lynx.2$t[,1])
table(lynx.2b$t[,1])
table(lynx.3$t[,1])
# Notice that the post-blackened and model-based bootstraps preserve
# the true order of the model (11) in many more cases than the others.
```

`tuna` *Tuna Sighting Data*

Description

The `tuna` data frame has 64 rows and 1 columns.

The data come from an aerial line transect survey of Southern Bluefin Tuna in the Great Australian Bight. An aircraft with two spotters on board flies randomly allocated line transects. Each school of tuna sighted is counted and its perpendicular distance from the transect measured. The survey was conducted in summer when tuna tend to stay on the surface.

Usage

```
tuna
```

Format

This data frame contains the following column:

`y` The perpendicular distance, in miles, from the transect for 64 independent sightings of tuna schools.

Source

The data were obtained from

Chen, S.X. (1996) Empirical likelihood confidence intervals for nonparametric density estimation. *Biometrika*, **83**, 329–341.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

---

urine *Urine Analysis Data*

Description

The `urine` data frame has 79 rows and 7 columns.

79 urine specimens were analyzed in an effort to determine if certain physical characteristics of the urine might be related to the formation of calcium oxalate crystals.

Usage

```
urine
```

Format

This data frame contains the following columns:

`r` Indicator of the presence of calcium oxalate crystals.

`gravity` The specific gravity of the urine.

`ph` The pH reading of the urine.

`osmo` The osmolarity of the urine. Osmolarity is proportional to the concentration of molecules in solution.

`cond` The conductivity of the urine. Conductivity is proportional to the concentration of charged ions in solution.

`urea` The urea concentration in millimoles per litre.

`calc` The calcium concentration in millimoles per litre.

Source

The data were obtained from

Andrews, D.F. and Herzberg, A.M. (1985) *Data: A Collection of Problems from Many Fields for the Student and Research Worker*. Springer-Verlag.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

var.linear *Linear Variance Estimate*

Description

Estimates the variance of a statistic from its empirical influence values.

Usage

```
var.linear(L, strata=NULL)
```

Arguments

L	Vector of the empirical influence values of a statistic. These will usually be calculated by a call to empinf.
strata	A numeric vector or factor specifying which observations (and hence empirical influence values) come from which strata.

Value

The variance estimate calculated from L.

References

Davison, A. C. and Hinkley, D. V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

See Also

empinf, linear.approx, k3.linear

Examples

```
#  To estimate the variance of the ratio of means for the city data.
ratio <- function(d,w)
     sum(d$x * w)/sum(d$u * w)
var.linear(empinf(data=city,statistic=ratio))
```

wool *Australian Relative Wool Prices*

Description

wool is a time series of class "ts" and contains 309 observations.

Each week that the market is open the Australian Wool Corporation set a floor price which determines their policy on intervention and is therefore a reflection of the overall price of wool for the week in question. Actual prices paid can vary considerably about the floor price. The series here is the log of the ratio between the price for fine grade wool and the floor price, each market week between July 1976 and Jun 1984.

Source

The data were obtained from

Diggle, P.J. (1990) *Time Series: A Biostatistical Introduction*. Oxford University Press.

References

Davison, A.C. and Hinkley, D.V. (1997) *Bootstrap Methods and Their Application*. Cambridge University Press.

Chapter 13

The `class` package

`batchSOM` *Self-Organizing Maps: Batch Algorithm*

Description

Kohonen's Self-Organizing Maps are a crude form of multidimensional scaling.

Usage

```
batchSOM(data, grid = somgrid(), radii, init)
```

Arguments

`data`	a matrix or data frame of observations, scaled so that Euclidean distance is appropriate.
`grid`	A grid for the representatives: see `somgrid`.
`radii`	the radii of the neighbourhood to be used for each pass: one pass is run for each element of `radii`.
`init`	the initial representatives. If missing, chosen (without replacement) randomly from `data`.

Details

The batch SOM algorithm of Kohonen(1995, section 3.14) is used.

Value

an object of class `"SOM"` with components

`grid`	the grid, an object of class `"somgrid"`.
`codes`	a matrix of representatives.

References

Kohonen, T. (1995) *Self-Organizing Maps.* Springer-Verlag.

Ripley, B. D. (1996) *Pattern Recognition and Neural Networks.* Cambridge.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

somgrid, SOM

Examples

```
data(crabs, package = "MASS")

lcrabs <- log(crabs[, 4:8])
crabs.grp <- factor(c("B", "b", "O", "o")[rep(1:4, rep(50,4))])
gr <- somgrid(topo = "hexagonal")
crabs.som <- batchSOM(lcrabs, gr, c(4, 4, 2, 2, 1, 1, 1, 0, 0))
plot(crabs.som)

bins <- as.numeric(knn1(crabs.som$code, lcrabs, 0:47))
plot(crabs.som$grid, type = "n")
symbols(crabs.som$grid$pts[, 1], crabs.som$grid$pts[, 2],
        circles = rep(0.4, 48), inches = FALSE, add = TRUE)
text(crabs.som$grid$pts[bins, ] + rnorm(400, 0, 0.1),
     as.character(crabs.grp))
```

---

condense *Condense training set for k-NN classifier*

Description

Condense training set for k-NN classifier

Usage

```
condense(train, class, store, trace = TRUE)
```

Arguments

train	matrix for training set
class	vector of classifications for test set
store	initial store set. Default one randomly chosen element of the set.
trace	logical. Trace iterations?

Details

The store set is used to 1-NN classify the rest, and misclassified patterns are added to the store set. The whole set is checked until no additions occur.

Value

index vector of cases to be retained (the final store set).

References

P. A. Devijver and J. Kittler (1982) *Pattern Recognition. A Statistical Approach.* Prentice-Hall, pp. 119–121.

Ripley, B. D. (1996) *Pattern Recognition and Neural Networks.* Cambridge.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

reduce.nn, multiedit

Examples

```
data(iris3)
train <- rbind(iris3[1:25,,1], iris3[1:25,,2], iris3[1:25,,3])
test <- rbind(iris3[26:50,,1], iris3[26:50,,2], iris3[26:50,,3])
cl <- factor(c(rep("s",25), rep("c",25), rep("v",25)))
keep <- condense(train, cl)
knn(train[keep, , drop=FALSE], test, cl[keep])
keep2 <- reduce.nn(train, keep, cl)
knn(train[keep2, , drop=FALSE], test, cl[keep2])
```

---

knn — *k-Nearest Neighbour Classification*

Description

k-nearest neighbour classification for test set from training set. For each row of the test set, the `k` nearest (in Euclidean distance) training set vectors are found, and the classification is decided by majority vote, with ties broken at random. If there are ties for the `k`th nearest vector, all candidates are included in the vote.

Usage

```
knn(train, test, cl, k = 1, l = 0, prob = FALSE, use.all = TRUE)
```

Arguments

`train`	matrix or data frame of training set cases.
`test`	matrix or data frame of test set cases. A vector will be interpreted as a row vector for a single case.
`cl`	factor of true classifications of training set
`k`	number of neighbours considered.
`l`	minimum vote for definite decision, otherwise `doubt`. (More precisely, less than `k-l` dissenting votes are allowed, even if `k` is increased by ties.)
`prob`	If this is true, the proportion of the votes for the winning class are returned as attribute `prob`.
`use.all`	controls handling of ties. If true, all distances equal to the `k`th largest are included. If false, a random selection of distances equal to the `k`th is chosen to use exactly `k` neighbours.

Value

factor of classifications of test set. `doubt` will be returned as `NA`.

References

Ripley, B. D. (1996) *Pattern Recognition and Neural Networks.* Cambridge.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

knn1, knn.cv

Examples

```
data(iris3)
train <- rbind(iris3[1:25,,1], iris3[1:25,,2], iris3[1:25,,3])
test <- rbind(iris3[26:50,,1], iris3[26:50,,2], iris3[26:50,,3])
cl <- factor(c(rep("s",25), rep("c",25), rep("v",25)))
knn(train, test, cl, k = 3, prob=TRUE)
attributes(.Last.value)
```

---

knn.cv *k-Nearest Neighbour Cross-Validatory Classification*

Description

k-nearest neighbour cross-validatory classification from training set.

Usage

```
knn.cv(train, cl, k = 1, l = 0, prob = FALSE, use.all = TRUE)
```

Arguments

train	matrix or data frame of training set cases.
cl	factor of true classifications of training set
k	number of neighbours considered.
l	minimum vote for definite decision, otherwise doubt. (More precisely, less than k-l dissenting votes are allowed, even if k is increased by ties.)
prob	If this is true, the proportion of the votes for the winning class are returned as attribute prob.
use.all	controls handling of ties. If true, all distances equal to the kth largest are included. If false, a random selection of distances equal to the kth is chosen to use exactly k neighbours.

Details

This uses leave-one-out cross validation. For each row of the training set train, the k nearest (in Euclidean distance) other training set vectors are found, and the classification is decided by majority vote, with ties broken at random. If there are ties for the kth nearest vector, all candidates are included in the vote.

Value

factor of classifications of training set. doubt will be returned as NA.

References

Ripley, B. D. (1996) *Pattern Recognition and Neural Networks.* Cambridge.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

knn

Examples

```
data(iris3)
train <- rbind(iris3[,,1], iris3[,,2], iris3[,,3])
cl <- factor(c(rep("s",50), rep("c",50), rep("v",50)))
knn.cv(train, cl, k = 3, prob = TRUE)
attributes(.Last.value)
```

---

knn1 *1-nearest neighbour classification*

Description

Nearest neighbour classification for test set from training set. For each row of the test set, the nearest (by Euclidean distance) training set vector is found, and its classification used. If there is more than one nearest, a majority vote is used with ties broken at random.

Usage

```
knn1(train, test, cl)
```

Arguments

train	matrix or data frame of training set cases.
test	matrix or data frame of test set cases. A vector will be interpreted as a row vector for a single case.
cl	factor of true classification of training set.

Value

factor of classifications of test set.

References

Ripley, B. D. (1996) *Pattern Recognition and Neural Networks.* Cambridge.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

knn

Examples

```
data(iris3)
train <- rbind(iris3[1:25,,1], iris3[1:25,,2], iris3[1:25,,3])
test <- rbind(iris3[26:50,,1], iris3[26:50,,2], iris3[26:50,,3])
cl <- factor(c(rep("s",25), rep("c",25), rep("v",25)))
knn1(train, test, cl)
```

`lvq1` *Learning Vector Quantization 1*

Description

Moves examples in a codebook to better represent the training set.

Usage

```
lvq1(x, cl, codebk, niter = 100 * nrow(codebk$x), alpha = 0.03)
```

Arguments

`x`	a matrix or data frame of examples
`cl`	a vector or factor of classifications for the examples
`codebk`	a codebook
`niter`	number of iterations
`alpha`	constant for training

Details

Selects `niter` examples at random with replacement, and adjusts the nearest example in the codebook for each.

Value

A codebook, represented as a list with components `x` and `cl` giving the examples and classes.

References

Kohonen, T. (1990) The self-organizing map. *Proc. IEEE* **78**, 1464–1480.

Kohonen, T. (1995) *Self-Organizing Maps.* Springer, Berlin.

Ripley, B. D. (1996) *Pattern Recognition and Neural Networks.* Cambridge.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`lvqinit`, `olvq1`, `lvq2`, `lvq3`, `lvqtest`

Examples

```
data(iris3)
train <- rbind(iris3[1:25,,1], iris3[1:25,,2], iris3[1:25,,3])
test <- rbind(iris3[26:50,,1], iris3[26:50,,2], iris3[26:50,,3])
cl <- factor(c(rep("s",25), rep("c",25), rep("v",25)))
cd <- lvqinit(train, cl, 10)
lvqtest(cd, train)
cd0 <- olvq1(train, cl, cd)
lvqtest(cd0, train)
cd1 <- lvq1(train, cl, cd0)
lvqtest(cd1, train)
```

`lvq2` *Learning Vector Quantization 2.1*

Description

Moves examples in a codebook to better represent the training set.

Usage

```
lvq2(x, cl, codebk, niter = 100 * nrow(codebk$x), alpha = 0.03,
     win = 0.3)
```

Arguments

`x`	a matrix or data frame of examples
`cl`	a vector or factor of classifications for the examples
`codebk`	a codebook
`niter`	number of iterations
`alpha`	constant for training
`win`	a tolerance for the closeness of the two nearest vectors.

Details

Selects `niter` examples at random with replacement, and adjusts the nearest two examples in the codebook if one is correct and the other incorrect.

Value

A codebook, represented as a list with components `x` and `cl` giving the examples and classes.

References

Kohonen, T. (1990) The self-organizing map. *Proc. IEEE* **78**, 1464–1480.

Kohonen, T. (1995) *Self-Organizing Maps.* Springer, Berlin.

Ripley, B. D. (1996) *Pattern Recognition and Neural Networks.* Cambridge.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`lvqinit`, `lvq1`, `olvq1`, `lvq3`, `lvqtest`

Examples

```
data(iris3)
train <- rbind(iris3[1:25,,1], iris3[1:25,,2], iris3[1:25,,3])
test <- rbind(iris3[26:50,,1], iris3[26:50,,2], iris3[26:50,,3])
cl <- factor(c(rep("s",25), rep("c",25), rep("v",25)))
cd <- lvqinit(train, cl, 10)
lvqtest(cd, train)
cd0 <- olvq1(train, cl, cd)
```

```
lvqtest(cd0, train)
cd2 <- lvq2(train, cl, cd0)
lvqtest(cd2, train)
```

`lvq3` *Learning Vector Quantization 3*

Description

Moves examples in a codebook to better represent the training set.

Usage

```
lvq3(x, cl, codebk, niter = 100*nrow(codebk$x), alpha = 0.03,
     win = 0.3, epsilon = 0.1)
```

Arguments

`x`	a matrix or data frame of examples
`cl`	a vector or factor of classifications for the examples
`codebk`	a codebook
`niter`	number of iterations
`alpha`	constant for training
`win`	a tolerance for the closeness of the two nearest vectors.
`epsilon`	proportion of move for correct vectors

Details

Selects `niter` examples at random with replacement, and adjusts the nearest two examples in the codebook for each.

Value

A codebook, represented as a list with components `x` and `cl` giving the examples and classes.

References

Kohonen, T. (1990) The self-organizing map. *Proc. IEEE* **78**, 1464–1480.

Kohonen, T. (1995) *Self-Organizing Maps.* Springer, Berlin.

Ripley, B. D. (1996) *Pattern Recognition and Neural Networks.* Cambridge.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`lvqinit`, `lvq1`, `olvq1`, `lvq2`, `lvqtest`

Examples

```
data(iris3)
train <- rbind(iris3[1:25,,1], iris3[1:25,,2], iris3[1:25,,3])
test <- rbind(iris3[26:50,,1], iris3[26:50,,2], iris3[26:50,,3])
cl <- factor(c(rep("s",25), rep("c",25), rep("v",25)))
cd <- lvqinit(train, cl, 10)
lvqtest(cd, train)
cd0 <- olvq1(train, cl, cd)
lvqtest(cd0, train)
cd3 <- lvq3(train, cl, cd0)
lvqtest(cd3, train)
```

---

lvqinit *Initialize a LVQ Codebook*

Description

Construct an initial codebook for LVQ methods.

Usage

```
lvqinit(x, cl, size, prior, k = 5)
```

Arguments

x	a matrix or data frame of training examples, n by p.
cl	the classifications for the training examples. A vector or factor of length n.
size	the size of the codebook. Defaults to min(round(0.4*ng*(ng-1 + p/2),0), n) where ng is the number of classes.
prior	Probabilities to represent classes in the codebook. Default proportions in the training set.
k	k used for k-NN test of correct classification. Default is 5.

Details

Selects size examples from the training set without replacement with proportions proportional to the prior or the original proportions.

Value

A codebook, represented as a list with components x and cl giving the examples and classes.

References

Kohonen, T. (1990) The self-organizing map. *Proc. IEEE* **78**, 1464–1480.

Kohonen, T. (1995) *Self-Organizing Maps.* Springer, Berlin.

Ripley, B. D. (1996) *Pattern Recognition and Neural Networks.* Cambridge.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

lvq1, lvq2, lvq3, olvq1, lvqtest

Examples

```
data(iris3)
train <- rbind(iris3[1:25,,1], iris3[1:25,,2], iris3[1:25,,3])
test <- rbind(iris3[26:50,,1], iris3[26:50,,2], iris3[26:50,,3])
cl <- factor(c(rep("s",25), rep("c",25), rep("v",25)))
cd <- lvqinit(train, cl, 10)
lvqtest(cd, train)
cd1 <- olvq1(train, cl, cd)
lvqtest(cd1, train)
```

---

lvqtest *Classify Test Set from LVQ Codebook*

Description

Classify a test set by 1-NN from a specified LVQ codebook.

Usage

```
lvqtest(codebk, test)
```

Arguments

codebk	codebook object returned by other LVQ software
test	matrix of test examples

Details

uses 1-NN to classify each test example against the codebook.

Value

factor of classification for each row of x

References

Ripley, B. D. (1996) *Pattern Recognition and Neural Networks.* Cambridge.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

lvqinit, olvq1

Examples

```
# The function is currently defined as
function(codebk, test) knn1(codebk$x, test, codebk$cl)
```

`multiedit` *Multiedit for k-NN Classifier*

Description

Multiedit for k-NN classifier

Usage

```
multiedit(x, class, k = 1, V = 3, I = 5, trace = TRUE)
```

Arguments

`x`	matrix of training set.
`class`	vector of classification of training set.
`k`	number of neighbours used in k-NN.
`V`	divide training set into V parts.
`I`	number of null passes before quitting.
`trace`	logical for statistics at each pass.

Value

index vector of cases to be retained.

References

P. A. Devijver and J. Kittler (1982) *Pattern Recognition. A Statistical Approach.* Prentice-Hall, p. 115.

Ripley, B. D. (1996) *Pattern Recognition and Neural Networks.* Cambridge.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`condense`, `reduce.nn`

Examples

```
data(iris3)
tr <- sample(1:50, 25)
train <- rbind(iris3[tr,,1], iris3[tr,,2], iris3[tr,,3])
test <- rbind(iris3[-tr,,1], iris3[-tr,,2], iris3[-tr,,3])
cl <- factor(c(rep(1,25),rep(2,25), rep(3,25)), labels=c("s", "c", "v"))
table(cl, knn(train, test, cl, 3))
ind1 <- multiedit(train, cl, 3)
length(ind1)
table(cl, knn(train[ind1, , drop=FALSE], test, cl[ind1], 1))
ntrain <- train[ind1,]; ncl <- cl[ind1]
ind2 <- condense(ntrain, ncl)
length(ind2)
table(cl, knn(ntrain[ind2, , drop=FALSE], test, ncl[ind2], 1))
```

olvq1 *Optimized Learning Vector Quantization 1*

Description

Moves examples in a codebook to better represent the training set.

Usage

```
olvq1(x, cl, codebk, niter = 40 * nrow(codebk$x), alpha = 0.3)
```

Arguments

x	a matrix or data frame of examples
cl	a vector or factor of classifications for the examples
codebk	a codebook
niter	number of iterations
alpha	constant for training

Details

Selects niter examples at random with replacement, and adjusts the nearest example in the codebook for each.

Value

A codebook, represented as a list with components x and cl giving the examples and classes.

References

Kohonen, T. (1990) The self-organizing map. *Proc. IEEE* **78**, 1464–1480.

Kohonen, T. (1995) *Self-Organizing Maps.* Springer, Berlin.

Ripley, B. D. (1996) *Pattern Recognition and Neural Networks.* Cambridge.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

lvqinit, lvqtest, lvq1, lvq2, lvq3

Examples

```
data(iris3)
train <- rbind(iris3[1:25,,1], iris3[1:25,,2], iris3[1:25,,3])
test <- rbind(iris3[26:50,,1], iris3[26:50,,2], iris3[26:50,,3])
cl <- factor(c(rep("s",25), rep("c",25), rep("v",25)))
cd <- lvqinit(train, cl, 10)
lvqtest(cd, train)
cd1 <- olvq1(train, cl, cd)
lvqtest(cd1, train)
```

`reduce.nn` *Reduce Training Set for a k-NN Classifier*

Description

Reduce training set for a k-NN classifier. Used after `condense`.

Usage

```
reduce.nn(train, ind, class)
```

Arguments

`train`	matrix for training set
`ind`	Initial list of members of the training set (from `condense`).
`class`	vector of classifications for test set

Details

All the members of the training set are tried in random order. Any which when dropped do not cause any members of the training set to be wrongly classified are dropped.

Value

index vector of cases to be retained.

References

Gates, G.W. (1972) The reduced nearest neighbor rule. *IEEE Trans. Information Theory* **IT-18**, 431–432.

Ripley, B. D. (1996) *Pattern Recognition and Neural Networks.* Cambridge.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`condense`, `multiedit`

Examples

```
data(iris3)
train <- rbind(iris3[1:25,,1], iris3[1:25,,2], iris3[1:25,,3])
test <- rbind(iris3[26:50,,1], iris3[26:50,,2], iris3[26:50,,3])
cl <- factor(c(rep("s",25), rep("c",25), rep("v",25)))
keep <- condense(train, cl)
knn(train[keep,], test, cl[keep])
keep2 <- reduce.nn(train, keep, cl)
knn(train[keep2,], test, cl[keep2])
```

SOM *Self-Organizing Maps: Online Algorithm*

Description

Kohonen's Self-Organizing Maps are a crude form of multidimensional scaling.

Usage

```
SOM(data, grid = somgrid(), rlen = 10000, alpha, radii, init)
```

Arguments

data	a matrix or data frame of observations, scaled so that Euclidean distance is appropriate.
grid	A grid for the representatives: see somgrid.
rlen	the number of updates: used only in the defaults for alpha and radii.
alpha	the amount of change: one update is done for each element of alpha. Default is to decline linearly from 0.05 to 0 over rlen updates.
radii	the radii of the neighbourhood to be used for each update: must be the same length as alpha. Default is to decline linearly from 4 to 1 over rlen updates.
init	the initial representatives. If missing, chosen (without replacement) randomly from data.

Details

alpha and radii can also be lists, in which case each component is used in turn, allowing two- or more phase training.

Value

an object of class "SOM" with components

grid	the grid, an object of class "somgrid".
codes	a matrix of representatives.

References

Kohonen, T. (1995) *Self-Organizing Maps.* Springer-Verlag

Kohonen, T., Hynninen, J., Kangas, J. and Laaksonen, J. (1996) *SOM PAK: The self-organizing map program package.* Laboratory of Computer and Information Science, Helsinki University of Technology, Technical Report A31.

Ripley, B. D. (1996) *Pattern Recognition and Neural Networks.* Cambridge.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

somgrid, batchSOM

Examples

```
data(crabs, package = "MASS")

lcrabs <- log(crabs[, 4:8])
crabs.grp <- factor(c("B", "b", "O", "o")[rep(1:4, rep(50,4))])
gr <- somgrid(topo = "hexagonal")
crabs.som <- SOM(lcrabs, gr)
plot(crabs.som)

## 2-phase training
crabs.som2 <- SOM(lcrabs, gr,
    alpha = list(seq(0.05, 0, len = 1e4), seq(0.02, 0, len = 1e5)),
    radii = list(seq(8, 1, len = 1e4), seq(4, 1, len = 1e5)))
plot(crabs.som2)
```

---

somgrid *Plot SOM Fits*

Description

Plotting functions for SOM results.

Usage

```
somgrid(xdim = 8, ydim = 6, topo = c("rectangular", "hexagonal"))

## S3 method for class 'somgrid':
plot(x, type = "p", ...)

## S3 method for class 'SOM':
plot(x, ...)
```

Arguments

`xdim, ydim`	dimensions of the grid
`topo`	the topology of the grid.
`x`	an object inheriting from class `"somgrid"` or `"SOM"`.
`type, ...`	graphical parameters.

Details

The class `"somgrid"` records the coordinates of the grid to be used for (batch or on-line) SOM: this has a plot method.

The plot method for class `"SOM"` plots a `stars` plot of the representative at each grid point.

Value

For `somgrid`, an object of class `"somgrid"`, a list with components

`pts`	a two-column matrix giving locations for the grid points.
`xdim, ydim, topo`	as in the arguments to `somgrid`.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`batchSOM`, `SOM`

Chapter 14

The `cluster` package

`agnes` *Agglomerative Nesting (Hierarchical Clustering)*

Description

Computes agglomerative hierarchical clustering of the dataset.

Usage

```
agnes(x, diss = inherits(x, "dist"), metric = "euclidean",
      stand = FALSE, method = "average", par.method,
      keep.diss = n < 100, keep.data = !diss)
```

Arguments

`x`	data matrix or data frame, or dissimilarity matrix, depending on the value of the `diss` argument. In case of a matrix or data frame, each row corresponds to an observation, and each column corresponds to a variable. All variables must be numeric. Missing values (NAs) are allowed. In case of a dissimilarity matrix, `x` is typically the output of `daisy` or `dist`. Also a vector with length n*(n-1)/2 is allowed (where n is the number of observations), and will be interpreted in the same way as the output of the above-mentioned functions. Missing values (NAs) are not allowed.
`diss`	logical flag: if TRUE (default for `dist` or `dissimilarity` objects), then `x` is assumed to be a dissimilarity matrix. If FALSE, then `x` is treated as a matrix of observations by variables.
`metric`	character string specifying the metric to be used for calculating dissimilarities between observations. The currently available options are "euclidean" and "manhattan". Euclidean distances are root sum-of-squares of differences, and manhattan distances are the sum of absolute differences. If `x` is already a dissimilarity matrix, then this argument will be ignored.
`stand`	logical flag: if TRUE, then the measurements in `x` are standardized before calculating the dissimilarities. Measurements are standardized for each variable

(column), by subtracting the variable's mean value and dividing by the variable's mean absolute deviation. If x is already a dissimilarity matrix, then this argument will be ignored.

method — character string defining the clustering method. The six methods implemented are "average" ([unweighted pair-]group average method, UPGMA), "single" (single linkage), "complete" (complete linkage), "ward" (Ward's method), "weighted" (weighted average linkage) and its generalization "flexible" which uses (a constant version of) the Lance-Williams formula and the par.method argument. Default is "average".

par.method — if method == "flexible", numeric vector of length 1, 3, or 4, see in the details section.

keep.diss, keep.data — logicals indicating if the dissimilarities and/or input data x should be kept in the result. Setting these to FALSE can give much smaller results and hence even save memory allocation *time*.

Details

agnes is fully described in chapter 5 of Kaufman and Rousseeuw (1990). Compared to other agglomerative clustering methods such as hclust, agnes has the following features: (a) it yields the agglomerative coefficient (see agnes.object) which measures the amount of clustering structure found; and (b) apart from the usual tree it also provides the banner, a novel graphical display (see plot.agnes).

The agnes-algorithm constructs a hierarchy of clusterings.
At first, each observation is a small cluster by itself. Clusters are merged until only one large cluster remains which contains all the observations. At each stage the two *nearest* clusters are combined to form one larger cluster.

For method="average", the distance between two clusters is the average of the dissimilarities between the points in one cluster and the points in the other cluster.
In method="single", we use the smallest dissimilarity between a point in the first cluster and a point in the second cluster (nearest neighbor method).
When method="complete", we use the largest dissimilarity between a point in the first cluster and a point in the second cluster (furthest neighbor method).

The method = "flexible" allows (and requires) more details: The Lance-Williams formula specifies how dissimilarities are computed when clusters are agglomerated (equation (32) in K.&R., p.237). If clusters C_1 and C_2 are agglomerated into a new cluster, the dissimilarity between their union and another cluster Q is given by

$$D(C_1 \cup C_2, Q) = \alpha_1 * D(C_1, Q) + \alpha_2 * D(C_2, Q) + \beta * D(C_1, C_2) + \gamma * |D(C_1, Q) - D(C_2, Q)|,$$

where the four coefficients $(\alpha_1, \alpha_2, \beta, \gamma)$ are specified by the vector par.method:

If par.method is of length 1, say $= \alpha$, par.method is extended to give the "Flexible Strategy" (K. & R., p.236 f) with Lance-Williams coefficients $(\alpha_1 = \alpha_2 = \alpha, \beta = 1 - 2\alpha, \gamma = 0)$.
If of length 3, $\gamma = 0$ is used.

Care and expertise is probably needed when using method = "flexible" particularly for the case when par.method is specified of longer length than one. The *weighted average* (method="weighted") is the same as method="flexible", par.method = 0.5.

Value

an object of class "agnes" representing the clustering. See agnes.object for details.

BACKGROUND

Cluster analysis divides a dataset into groups (clusters) of observations that are similar to each other.

Hierarchical methods like `agnes`, `diana`, and `mona` construct a hierarchy of clusterings, with the number of clusters ranging from one to the number of observations.

Partitioning methods like `pam`, `clara`, and `fanny` require that the number of clusters be given by the user.

References

Kaufman, L. and Rousseeuw, P.J. (1990). *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley, New York.

Anja Struyf, Mia Hubert & Peter J. Rousseeuw (1996): Clustering in an Object-Oriented Environment. *Journal of Statistical Software*, **1**. http://www.stat.ucla.edu/journals/jss/

Struyf, A., Hubert, M. and Rousseeuw, P.J. (1997). Integrating Robust Clustering Techniques in S-PLUS, *Computational Statistics and Data Analysis*, **26**, 17–37.

See Also

`agnes.object`, `daisy`, `diana`, `dist`, `hclust`, `plot.agnes`, `twins.object`.

Examples

```
data(votes.repub)
agn1 <- agnes(votes.repub, metric = "manhattan", stand = TRUE)
agn1
plot(agn1)

op <- par(mfrow=c(2,2))
agn2 <- agnes(daisy(votes.repub), diss = TRUE, method = "complete")
plot(agn2)
agnS <- agnes(votes.repub, method = "flexible", par.meth = 0.6)
plot(agnS)
par(op)

data(agriculture)
## Plot similar to Figure 7 in ref
## Not run:  plot(agnes(agriculture), ask = TRUE)
```

`agnes.object` *Agglomerative Nesting (AGNES) Object*

Description

The objects of class `"agnes"` represent an agglomerative hierarchical clustering of a dataset.

Value

A legitimate `agnes` object is a list with the following components:

`order`	a vector giving a permutation of the original observations to allow for plotting, in the sense that the branches of a clustering tree will not cross.
`order.lab`	a vector similar to `order`, but containing observation labels instead of observation numbers. This component is only available if the original observations were labelled.
`height`	a vector with the distances between merging clusters at the successive stages.
`ac`	the agglomerative coefficient, measuring the clustering structure of the dataset. For each observation i, denote by m(i) its dissimilarity to the first cluster it is merged with, divided by the dissimilarity of the merger in the final step of the algorithm. The `ac` is the average of all 1 - m(i). It can also be seen as the average width (or the percentage filled) of the banner plot. Because `ac` grows with the number of observations, this measure should not be used to compare datasets of very different sizes.
`merge`	an (n-1) by 2 matrix, where n is the number of observations. Row i of `merge` describes the merging of clusters at step i of the clustering. If a number j in the row is negative, then the single observation |j| is merged at this stage. If j is positive, then the merger is with the cluster formed at stage j of the algorithm.
`diss`	an object of class `"dissimilarity"` (see `dissimilarity.object`), representing the total dissimilarity matrix of the dataset.
`data`	a matrix containing the original or standardized measurements, depending on the `stand` option of the function `agnes`. If a dissimilarity matrix was given as input structure, then this component is not available.

GENERATION

This class of objects is returned from `agnes`.

METHODS

The `"agnes"` class has methods for the following generic functions: `print`, `summary`, `plot`.

INHERITANCE

The class `"agnes"` inherits from `"twins"`. Therefore, the generic functions `pltree` and `as.hclust` are available for `agnes` objects. After applying `as.hclust()`, all *its* methods are available, of course.

See Also

`agnes`, `diana`, `as.hclust`, `hclust`, `plot.agnes`, `twins.object`.

`agriculture` *European Union Agricultural Workforces*

Description

Gross National Product (GNP) per capita and percentage of the population working in agriculture for each country belonging to the European Union in 1993.

Usage

```
data(agriculture)
```

Format

A data frame with 12 observations on 2 variables:

[, 1]	`x`	numeric	per capita GNP
[, 2]	`y`	numeric	percentage in agriculture

The row names of the data frame indicate the countries.

Details

The data seem to show two clusters, the "more agricultural" one consisting of Greece, Portugal, Spain, and Ireland.

Source

Eurostat (European Statistical Agency, 1994): *Cijfers en feiten: Een statistisch portret van de Europese Unie*.

References

Anja Struyf, Mia Hubert & Peter J. Rousseeuw (1996): Clustering in an Object-Oriented Environment. *Journal of Statistical Software*, **1**. http://www.stat.ucla.edu/journals/jss/

See Also

`agnes`, `daisy`, `diana`.

Examples

```
data(agriculture)

## Compute the dissimilarities using Euclidean metric and without
## standardization
daisy(agriculture, metric = "euclidean", stand = FALSE)

## 2nd plot is similar to Figure 3 in Struyf et al (1996)
plot(pam(agriculture, 2))

## Plot similar to Figure 7 in Struyf et al (1996)
```

```
## Not run: plot(agnes(agriculture), ask = TRUE)


## Plot similar to Figure 8 in Struyf et al (1996)
## Not run: plot(diana(agriculture), ask = TRUE)
```

`animals` *Attributes of Animals*

Description

This data set considers 6 binary attributes for 20 animals.

Usage

```
data(animals)
```

Format

A data frame with 20 observations on 6 variables:

[, 1]	war	warm-blooded
[, 2]	fly	can fly
[, 3]	ver	vertebrate
[, 4]	end	endangered
[, 5]	gro	live in groups
[, 6]	hai	have hair

All variables are encoded as 1 = 'no', 2 = 'yes'.

Details

This dataset is useful for illustrating monothetic (only a single variable is used for each split) hierarchical clustering.

Source

Leonard Kaufman and Peter J. Rousseeuw (1990): *Finding Groups in Data* (pp 297ff). New York: Wiley.

References

Anja Struyf, Mia Hubert & Peter J. Rousseeuw (1996): Clustering in an Object-Oriented Environment. *Journal of Statistical Software*, **1**. http://www.stat.ucla.edu/journals/jss/

Examples

```
data(animals)
apply(animals,2, table) # simple overview

ma <- mona(animals)
```

```
ma
## Plot similar to Figure 10 in Struyf et al (1996)
plot(ma)
```

`bannerplot` *Plot Banner (of Hierarchical Clustering)*

Description

Draws a "banner", i.e. basically a horizontal `barplot` visualizing the (agglomerative or divisive) hierarchical clustering or an other binary dendrogram structure.

Usage

```
bannerplot(x, w = rev(x$height), fromLeft = TRUE,
           main=NULL, sub=NULL, xlab = "Height",  adj = 0,
           col = c(2, 0), border = 0, axes = TRUE, frame.plot = axes,
           rev.xax = !fromLeft, xax.pretty = TRUE,
           labels = NULL, nmax.lab = 35, max.strlen = 5,
           yax.do = axes && length(x$order) <= nmax.lab,
           yaxRight = fromLeft, y.mar = 2.4 + max.strlen/2.5, ...)
```

Arguments

`x`	a list with components `order`, `order.lab` and `height` when `w`, the next argument is not specified.
`w`	non-negative numeric vector of bar widths.
`fromLeft`	logical, indicating if the banner is from the left or not.
`main,sub`	main and sub titles, see `title`.
`xlab`	x axis label (with 'correct' default e.g. for `plot.agnes`).
`adj`	passed to `title(main,sub)` for string adjustment.
`col`	vector of length 2, for two horizontal segments.
`border`	color for bar border; now defaults to background (no border).
`axes`	logical indicating if axes (and labels) should be drawn at all.
`frame.plot`	logical indicating the banner should be framed; mainly used when `border = 0` (as per default).
`rev.xax`	logical indicating if the x axis should be reversed (as in `plot.diana`).
`xax.pretty`	logical or integer indicating if `pretty()` should be used for the x axis. `xax.pretty = FALSE` is mainly for back compatibility.
`labels`	labels to use on y-axis; the default is constructed from `x`.
`nmax.lab`	integer indicating the number of labels which is considered too large for single-name labelling the banner plot.
`max.strlen`	positive integer giving the length to which strings are truncated in banner plot labeling.
`yax.do`	logical indicating if a y axis and banner labels should be drawn.
`yaxRight`	logical indicating if the y axis is on the right or left.

y.mar	positive number specifying the margin width to use when banners are labeled (along a y-axis). The default adapts to the string width and optimally would also dependend on the font.
...	graphical parameters (see par) may also be supplied as arguments to this function.

Note

This is mainly a utility called from plot.agnes, plot.diana and plot.mona.

Author(s)

Martin Maechler (from original code of Kaufman and Rousseeuw).

Examples

```
data(agriculture)
bannerplot(agnes(agriculture), main = "Bannerplot")
```

---

chorSub *Subset of C-horizon of Kola Data*

Description

This is a small rounded subset of the C-horizon data chorizon from package **mvoutlier**.

Usage

```
data(chorSub)
```

Format

A data frame with 61 observations on 10 variables. The variables contain scaled concentrations of chemical elements.

Details

This data set was produced from chorizon via these statements:

```
data(chorizon, package = "mvoutlier")
chorSub <- round(100*scale(chorizon[,101:110]))[190:250,]
storage.mode(chorSub) <- "integer"
colnames(chorSub) <- gsub("_.*", '', colnames(chorSub))
```

Source

Kola Project (1993-1998)

See Also

chorizon in package **mvoutlier** and other Kola data in the same package.

Examples

```
data(chorSub)
summary(chorSub)
pairs(chorSub, gap= .1)# some outliers
```

---

clara *Clustering Large Applications*

Description

Computes a "clara" object, a list representing a clustering of the data into k clusters.

Usage

```
clara(x, k, metric = "euclidean", stand = FALSE, samples = 5,
      sampsize = min(n, 40 + 2 * k), trace = 0, medoids.x = TRUE,
      keep.data = medoids.x, rngR = FALSE)
```

Arguments

x	data matrix or data frame, each row corresponds to an observation, and each column corresponds to a variable. All variables must be numeric. Missing values (NAs) are allowed.
k	integer, the number of clusters. It is required that $0 < k < n$ where n is the number of observations (i.e., n = nrow(x)).
metric	character string specifying the metric to be used for calculating dissimilarities between observations. The currently available options are "euclidean" and "manhattan". Euclidean distances are root sum-of-squares of differences, and manhattan distances are the sum of absolute differences.
stand	logical, indicating if the measurements in x are standardized before calculating the dissimilarities. Measurements are standardized for each variable (column), by subtracting the variable's mean value and dividing by the variable's mean absolute deviation.
samples	integer, number of samples to be drawn from the dataset.
sampsize	integer, number of observations in each sample. sampsize should be higher than the number of clusters (k) and at most the number of observations (n = nrow(x)).
trace	integer indicating a *trace level* for diagnostic output during the algorithm.
medoids.x	logical indicating if the medoids should be returned, identically to some rows of the input data x. If FALSE, keep.data must be false as well, and the medoid indices, i.e., row numbers of the medoids will still be returned (i.med component), and the algorithm saves space by needing one copy less of x.
keep.data	logical indicating if the (*scaled* if stand is true) data should be kept in the result. Setting this to FALSE saves memory (and hence time), but disables clusplot() ing of the result. Use medoids.x = FALSE to save even more memory.
rngR	logical indicating if R's random number generator should be used instead of the primitive clara()-builtin one. If true, this also means that each call to clara() returns a different result – though only slightly different in good situations.

Details

clara is fully described in chapter 3 of Kaufman and Rousseeuw (1990). Compared to other partitioning methods such as pam, it can deal with much larger datasets. Internally, this is achieved by considering sub-datasets of fixed size (sampsize) such that the time and storage requirements become linear in n rather than quadratic.

Each sub-dataset is partitioned into k clusters using the same algorithm as in pam.
Once k representative objects have been selected from the sub-dataset, each observation of the entire dataset is assigned to the nearest medoid.

The sum of the dissimilarities of the observations to their closest medoid is used as a measure of the quality of the clustering. The sub-dataset for which the sum is minimal, is retained. A further analysis is carried out on the final partition.

Each sub-dataset is forced to contain the medoids obtained from the best sub-dataset until then. Randomly drawn observations are added to this set until sampsize has been reached.

Value

an object of class "clara" representing the clustering. See clara.object for details.

Note

By default, the random sampling is implemented with a *very* simple scheme (with period $2^{16} = 65536$) inside the Fortran code, independently of R's random number generation, and as a matter of fact, deterministically. Alternatively, we recommend setting rngR = TRUE which uses R's random number generators. Then, clara() results are made reproducible typically by using set.seed() before calling clara.

The storage requirement of clara computation (for small k) is about $O(n \times p) + O(j^2)$ where $j =$ sampsize, and $(n, p) =$ dim(x). The CPU computing time (again assuming small k) is about $O(n \times p \times j^2 \times N)$, where $N =$ samples.

For "small" datasets, the function pam can be used directly. What can be considered *small*, is really a function of available computing power, both memory (RAM) and speed. Originally (1990), "small" meant less than 100 observations; later, the authors said *"small (say with fewer than 200 observations)"...*

Author(s)

Kaufman and Rousseeuw (see agnes), originally. All arguments from trace on, and most R documentation and all tests by Martin Maechler.

See Also

agnes for background and references; clara.object, pam, partition.object, plot.partition.

Examples

```
## generate 500 objects, divided into 2 clusters.
x <- rbind(cbind(rnorm(200,0,8), rnorm(200,0,8)),
           cbind(rnorm(300,50,8), rnorm(300,50,8)))
clarax <- clara(x, 2)
clarax
clarax$clusinfo
plot(clarax)
```

```
## `xclara' is an artificial data set with 3 clusters of 1000 bivariate
## objects each.
data(xclara)
(clx3 <- clara(xclara, 3))
## Plot similar to Figure 5 in Struyf et al (1996)
## Not run: plot(clx3, ask = TRUE)


## Try 100 times *different* random samples -- for reliability:
nSim <- 100
nCl <- 3 # = no.classes
set.seed(421)# (reproducibility)
cl <- matrix(NA,nrow(xclara), nSim)
for(i in 1:nSim)
   cl[,i] <- clara(xclara, nCl, medoids.x = FALSE, rngR = TRUE)$cluster
tcl <- apply(cl,1, tabulate, nbins = nCl)
## those that are not always in same cluster (5 out of 3000 for this seed):
(iDoubt <- which(apply(tcl,2, function(n) all(n < nSim))))
if(length(iDoubt)) { # (not for all seeds)
  tabD <- tcl[,iDoubt, drop=FALSE]
  dimnames(tabD) <- list(cluster = paste(1:nCl), obs = format(iDoubt))
  t(tabD) # how many times in which clusters
}
```

`clara.object` *Clustering Large Applications (CLARA) Object*

Description

The objects of class `"clara"` represent a partitioning of a large dataset into clusters and are typically returned from `clara`.

Value

A legitimate `clara` object is a list with the following components:

`sample`	labels or case numbers of the observations in the best sample, that is, the sample used by the `clara` algorithm for the final partition.
`medoids`	the medoids or representative objects of the clusters. It is a matrix with in each row the coordinates of one medoid. Possibly `NULL`, namely when the object resulted from `clara(*, medoids.x=FALSE)`. Use the following `i.med` in that case.
`i.med`	the *indices* of the `medoids` above: `medoids <- x[i.med,]` where `x` is the original data matrix in `clara(x,*)`.
`clustering`	the clustering vector, see `partition.object`.
`objective`	the objective function for the final clustering of the entire dataset.
`clusinfo`	matrix, each row gives numerical information for one cluster. These are the cardinality of the cluster (number of observations), the maximal and average dissimilarity between the observations in the cluster and the cluster's medoid. The last column is the maximal dissimilarity between the observations in the

	cluster and the cluster's medoid, divided by the minimal dissimilarity between the cluster's medoid and the medoid of any other cluster. If this ratio is small, the cluster is well-separated from the other clusters.
diss	dissimilarity (maybe NULL), see partition.object.
silinfo	list with silhouette width information for the best sample, see partition.object.
call	generating call, see partition.object.
data	matrix, possibibly standardized, or NULL, see partition.object.

Methods, Inheritance

The "clara" class has methods for the following generic functions: print, summary.

The class "clara" inherits from "partition". Therefore, the generic functions plot and clusplot can be used on a clara object.

See Also

clara, dissimilarity.object, partition.object, plot.partition.

---

clusplot *Cluster Plot - Generic Function*

Description

Draws a 2-dimensional "clusplot" on the current graphics device. This is a generic function with a default and partition method.

Usage

```
clusplot(x, ...)
```

Arguments

x	an R object.
...	additional arguments for methods. Graphical parameters (see par) may also be supplied as arguments to this function.

Side Effects

a 2-dimensional clusplot is created on the current graphics device.

See Also

(for references and examples) clusplot.default, clusplot.partition, partition.object, pam, fanny, clara.

`clusplot.default` *Bivariate Cluster Plot (clusplot) Default Method*

Description

Creates a bivariate plot visualizing a partition (clustering) of the data. All observation are represented by points in the plot, using principal components or multidimensional scaling. Around each cluster an ellipse is drawn.

Usage

```
## Default S3 method:
clusplot(x, clus, diss = FALSE, cor = TRUE, stand = FALSE,
         lines = 2, shade = FALSE, color = FALSE,
         labels= 0, plotchar = TRUE,
         col.p = "dark green", col.txt = col.p,
         col.clus = if(color) c(2, 4, 6, 3) else 5,
         span = TRUE, xlim = NULL, ylim = NULL,
         main = paste("CLUSPLOT(", deparse(substitute(x)),")"),
         sub = paste("These two components explain",
            round(100 * var.dec, digits = 2), "% of the point variability."),
         verbose = getOption("verbose"),
         ...)
```

Arguments

`x`	matrix or data frame, or dissimilarity matrix, depending on the value of the `diss` argument. In case of a matrix (alike), each row corresponds to an observation, and each column corresponds to a variable. All variables must be numeric. Missing values (`NA`s) are allowed. They are replaced by the median of the corresponding variable. When some variables or some observations contain only missing values, the function stops with a warning message. In case of a dissimilarity matrix, `x` is the output of `daisy` or `dist` or a symmetric matrix. Also, a vector of length $n*(n-1)/2$ is allowed (where n is the number of observations), and will be interpreted in the same way as the output of the above-mentioned functions. Missing values (NAs) are not allowed.
`clus`	a vector of length n representing a clustering of `x`. For each observation the vector lists the number or name of the cluster to which it has been assigned. `clus` is often the clustering component of the output of `pam`, `fanny` or `clara`.
`diss`	logical indicating if `x` will be considered as a dissimilarity matrix or a matrix of observations by variables (see `x` arugment above).
`cor`	logical flag, only used when working with a data matrix (`diss = FALSE`). If TRUE, then the variables are scaled to unit variance.
`stand`	logical flag: if true, then the representations of the n observations in the 2-dimensional plot are standardized.
`lines`	integer out of `0, 1, 2`, used to obtain an idea of the distances between ellipses. The distance between two ellipses E1 and E2 is measured along the line connecting the centers $m1$ and $m2$ of the two ellipses.

In case E1 and E2 overlap on the line through $m1$ and $m2$, no line is drawn. Otherwise, the result depends on the value of `lines`: If

lines = 0, no distance lines will appear on the plot;

lines = 1, the line segment between $m1$ and $m2$ is drawn;

lines = 2, a line segment between the boundaries of E1 and E2 is drawn (along the line connecting $m1$ and $m2$).

`shade` logical flag: if TRUE, then the ellipses are shaded in relation to their density. The density is the number of points in the cluster divided by the area of the ellipse.

`color` logical flag: if TRUE, then the ellipses are colored with respect to their density. With increasing density, the colors are light blue, light green, red and purple. To see these colors on the graphics device, an appropriate color scheme should be selected (we recommend a white background).

`labels` integer code, currently one of 0,1,2,3,4 and 5. If

labels= 0, no labels are placed in the plot;

labels= 1, points and ellipses can be identified in the plot (see `identify`);

labels= 2, all points and ellipses are labelled in the plot;

labels= 3, only the points are labelled in the plot;

labels= 4, only the ellipses are labelled in the plot.

labels= 5, the ellipses are labelled in the plot, and points can be identified.

The levels of the vector `clus` are taken as labels for the clusters. The labels of the points are the rownames of `x` if `x` is matrix like. Otherwise (`diss = TRUE`), `x` is a vector, point labels can be attached to `x` as a "Labels" attribute (`attr(x,"Labels")`), as is done for the output of `daisy`.

A possible `names` attribute of `clus` will not be taken into account.

`plotchar` logical flag: if TRUE, then the plotting symbols differ for points belonging to different clusters.

`span` logical flag: if TRUE, then each cluster is represented by the ellipse with smallest area containing all its points. (This is a special case of the minimum volume ellipsoid.)
If FALSE, the ellipse is based on the mean and covariance matrix of the same points. While this is faster to compute, it often yields a much larger ellipse.

There are also some special cases: When a cluster consists of only one point, a tiny circle is drawn around it. When the points of a cluster fall on a straight line, `span=FALSE` draws a narrow ellipse around it and `span=TRUE` gives the exact line segment.

`col.p` color code(s) used for the observation points.

`col.txt` color code(s) used for the labels (if `labels >= 2`).

`col.clus` color code for the ellipses (and their labels); only one if color is false (as per default).

`xlim, ylim` numeric vectors of length 2, giving the x- and y- ranges as in `plot.default`.

`main` main title for the plot; by default, one is constructed.

`sub` sub title for the plot; by default, one is constructed.

`verbose` a logical indicating, if there should be extra diagnostic output; mainly for 'debugging'.

`...` Further graphical parameters may also be supplied, see `par`.

Details

clusplot uses the functions princomp and cmdscale. These functions are data reduction techniques. They will represent the data in a bivariate plot. Ellipses are then drawn to indicate the clusters. The further layout of the plot is determined by the optional arguments.

Value

An invisible list with components:

Distances	When lines is 1 or 2 we optain a k by k matrix (k is the number of clusters). The element in [i,j] is the distance between ellipse i and ellipse j. If lines = 0, then the value of this component is NA.
Shading	A vector of length k (where k is the number of clusters), containing the amount of shading per cluster. Let y be a vector where element i is the ratio between the number of points in cluster i and the area of ellipse i. When the cluster i is a line segment, y[i] and the density of the cluster are set to NA. Let z be the sum of all the elements of y without the NAs. Then we put shading = y/z *37 + 3 .

Side Effects

a visual display of the clustering is plotted on the current graphics device.

Note

When we have 4 or fewer clusters, then the color=TRUE gives every cluster a different color. When there are more than 4 clusters, clusplot uses the function pam to cluster the densities into 4 groups such that ellipses with nearly the same density get the same color. col.clus specifies the colors used.

The col.p and col.txt arguments, added for R, are recycled to have length the number of observations. If col.p has more than one value, using color = TRUE can be confusing because of a mix of point and ellipse colors.

References

Pison, G., Struyf, A. and Rousseeuw, P.J. (1999) Displaying a Clustering with CLUSPLOT, *Computational Statistics and Data Analysis*, **30**, 381–392.
A version of this is available as technical report from http://www.agoras.ua.ac.be/abstract/Disclu99.htm

Kaufman, L. and Rousseeuw, P.J. (1990). *Finding Groups in Data: An Introduction to Cluster Analysis.* Wiley, New York.

Struyf, A., Hubert, M. and Rousseeuw, P.J. (1997). Integrating Robust Clustering Techniques in S-PLUS, *Computational Statistics and Data Analysis*, **26**, 17-37.

See Also

princomp, cmdscale, pam, clara, daisy, par, identify, cov.mve, clusplot.partition.

Examples

```
## plotting votes.diss(dissimilarity) in a bivariate plot and
## partitioning into 2 clusters
data(votes.repub)
votes.diss <- daisy(votes.repub)
votes.clus <- pam(votes.diss, 2, diss = TRUE)$clustering
clusplot(votes.diss, votes.clus, diss = TRUE, shade = TRUE)
clusplot(votes.diss, votes.clus, diss = TRUE,
         col.p = votes.clus, labels = 4)# color points and label ellipses
clusplot(votes.diss, votes.clus, diss = TRUE, span = FALSE)# simple ellipses

if(interactive()) { #  uses identify() *interactively* :
  clusplot(votes.diss, votes.clus, diss = TRUE, shade = TRUE, labels = 1)
  clusplot(votes.diss, votes.clus, diss = TRUE, labels = 5)# ident. only points
}

## plotting iris (data frame) in a 2-dimensional plot and partitioning
## into 3 clusters.
data(iris)
iris.x <- iris[, 1:4]
cl3 <- pam(iris.x, 3)$clustering
op <- par(mfrow= c(2,2))
clusplot(iris.x, cl3, color = TRUE)
U <- par("usr")
## zoom in :
rect(0,-1, 2,1, border = "orange", lwd=2)
clusplot(iris.x, cl3, color = TRUE, xlim = c(0,2), ylim = c(-1,1))
box(col="orange",lwd=2); mtext("sub region", font = 4, cex = 2)
##  or zoom out :
clusplot(iris.x, cl3, color = TRUE, xlim = c(-4,4), ylim = c(-4,4))
mtext("`super' region", font = 4, cex = 2)
rect(U[1],U[3], U[2],U[4], lwd=2, lty = 3)

# reset graphics
par(op)
```

---

clusplot.partition *Bivariate Clusplot of a Partitioning Object*

Description

Clusplot (Clustering Plot) method for an object of class `partition`.

Usage

```
## S3 method for class 'partition':
clusplot(x, main = NULL, dist = NULL, ...)
```

Arguments

`x`	an object of class `"partition"`, e.g. created by the functions `pam`, `clara`, or `fanny`.
`main`	title for the plot; when `NULL` (by default), a title is constructed, using `x$call`.

dist — when x does not have a diss nor a data component, e.g., for pam(dist(*), keep.diss=FALSE), dist must specify the dissimilarity for the clusplot.

... — all optional arguments available for the clusplot.default function (except for the diss one) may also be supplied to this function. Graphical parameters (see par) may also be supplied as arguments to this function.

Details

This clusplot.partition() method relies on clusplot.default.

If the clustering algorithms pam, fanny and clara are applied to a data matrix of observations-by-variables then a clusplot of the resulting clustering can always be drawn. When the data matrix contains missing values and the clustering is performed with pam or fanny, the dissimilarity matrix will be given as input to clusplot. When the clustering algorithm clara was applied to a data matrix with NAs then clusplot will replace the missing values as described in clusplot.default, because a dissimilarity matrix is not available.

Value

An invisible list with components

Distances — When option lines is 1 or 2 we optain a k by k matrix (k is the number of clusters). The element at row j and column s is the distance between ellipse j and ellipse s. If lines=0, then the value of this component is NA.

Shading — A vector of length k (where k is the number of clusters), containing the amount of shading per cluster. Let y be a vector where element i is the ratio between the number of objects in cluster i and the area of ellipse i. When the cluster i is a line segment, y[i] and the density of the cluster are set to NA. Let z be the sum of all the elements of y without the NAs. Then we put shading = y/z *37 + 3.

See Also

clusplot.default for references; partition.object, pam, pam.object, clara, clara.object, fanny, fanny.object, par.

Examples

```
## generate 25 objects, divided into 2 clusters.
x <- rbind(cbind(rnorm(10,0,0.5), rnorm(10,0,0.5)),
           cbind(rnorm(15,5,0.5), rnorm(15,5,0.5)))
clusplot(pam(x, 2))
## add noise, and try again :
x4 <- cbind(x, rnorm(25), rnorm(25))
clusplot(pam(x4, 2))
```

`coef.hclust` *Agglomerative Coefficient for 'hclust' Objects*

Description

Computes the "agglomerative coefficient", measuring the clustering structure of the dataset.

For each observation i, denote by $m(i)$ its dissimilarity to the first cluster it is merged with, divided by the dissimilarity of the merger in the final step of the algorithm. The agglomerative coefficient is the average of all $1 - m(i)$. It can also be seen as the average width (or the percentage filled) of the banner plot.

Because it grows with the number of observations, this measure should not be used to compare datasets of very different sizes.

Usage

```
coef.hclust(object, ...)
## S3 method for class 'hclust':
coef(object, ...)
## S3 method for class 'twins':
coef(object, ...)
```

Arguments

`object`	an object of class `"hclust"` or `"twins"`, i.e., typically the result of `hclust(.)`,`agnes(.)`, or `diana(.)`. Since `coef.hclust` only uses `object$heights`, and `object$merge`, `object` can be any list-like object with appropriate `merge` and `heights` components.
`...`	currently unused potential further arguments

Value

a number specifying the *agglomerative* (or *divisive* for `diana` objects) coefficient as defined by Kaufman and Rousseeuw, see `agnes.object $ ac` or `diana.object $ dc`.

Examples

```
data(agriculture)
aa <- agnes(agriculture)
coef(aa) # really just extracts aa$ac
coef(as.hclust(aa))# recomputes
```

daisy *Dissimilarity Matrix Calculation*

Description

Compute all the pairwise dissimilarities (distances) between observations in the data set. The original variables may be of mixed types.

Usage

```
daisy(x, metric = c("euclidean", "manhattan", "gower"),
      stand = FALSE, type = list())
```

Arguments

x	numeric matrix or data frame. Dissimilarities will be computed between the rows of x. Columns of mode numeric (i.e. all columns when x is a matrix) will be recognized as interval scaled variables, columns of class factor will be recognized as nominal variables, and columns of class ordered will be recognized as ordinal variables. Other variable types should be specified with the type argument. Missing values (NAs) are allowed.
metric	character string specifying the metric to be used. The currently available options are "euclidean" (the default), "manhattan" and "gower". Euclidean distances are root sum-of-squares of differences, and manhattan distances are the sum of absolute differences. "Gower's distance" is chosen by metric "gower" or automatically if some columns of x are not numeric. Also known as Gower's coefficient (1971), expressed as a dissimilarity, this implies that a particular standardisation will be applied to each variable, and the "distance" between two units is the sum of all the variable-specific distances, see the details section.
stand	logical flag: if TRUE, then the measurements in x are standardized before calculating the dissimilarities. Measurements are standardized for each variable (column), by subtracting the variable's mean value and dividing by the variable's mean absolute deviation. If not all columns of x are numeric, stand will be ignored and Gower's standardization (based on the range) will be applied in any case, see argument metric, above, and the details section.
type	list for specifying some (or all) of the types of the variables (columns) in x. The list may contain the following components: "ordratio" (ratio scaled variables to be treated as ordinal variables), "logratio" (ratio scaled variables that must be logarithmically transformed), "asymm" (asymmetric binary) and "symm" (symmetric binary variables). Each component's value is a vector, containing the names or the numbers of the corresponding columns of x. Variables not mentioned in the type list are interpreted as usual (see argument x).

Details

daisy is fully described in chapter 1 of Kaufman and Rousseeuw (1990). Compared to dist whose input must be numeric variables, the main feature of daisy is its ability to handle other

variable types as well (e.g. nominal, ordinal, (a)symmetric binary) even when different types occur in the same data set.

The handling of nominal, ordinal, and (a)symmetric binary data is achieved by using the general dissimilarity coefficient of Gower (1971). If `x` contains any columns of these data-types, both arguments `metric` and `stand` will be ignored and Gower's coefficient will be used as the metric. This can also be activated for purely numeric data by `metric = "gower"`. With that, each variable (column) is first standardized by dividing each entry by the range of the corresponding variable.

Note that setting the type to `symm` (symmetric binary) gives the same dissimilarities as using *nominal* (which is chosen for non-ordered factors) only when no missing values are present, and more efficiently.

Note that `daisy` now gives a warning when 2-valued numerical variables don't have an explicit `type` specified, because the reference authors recommend to consider using `"asymm"`.

In the `daisy` algorithm, missing values in a row of x are not included in the dissimilarities involving that row. There are two main cases,

1. If all variables are interval scaled, the metric is "euclidean", and ng is the number of columns in which neither row i and j have NAs, then the dissimilarity d(i,j) returned is sqrt(ncol(x)/ng) times the Euclidean distance between the two vectors of length ng shortened to exclude NAs. The rule is similar for the "manhattan" metric, except that the coefficient is ncol(x)/ng. If ng is zero, the dissimilarity is NA.
2. When some variables have a type other than interval scaled, the dissimilarity between two rows is the weighted sum of the contributions of each variable.
   The weight becomes zero when that variable is missing in either or both rows, or when the variable is asymmetric binary and both values are zero. In all other situations, the weight of the variable is 1.
   The contribution of a nominal or binary variable to the total dissimilarity is 0 if both values are different, 1 otherwise. The contribution of other variables is the absolute difference of both values, divided by the total range of that variable. Ordinal variables are first converted to ranks.
   If `nok` is the number of nonzero weights, the dissimilarity is multiplied by the factor `1/nok` and thus ranges between 0 and 1. If `nok = 0`, the dissimilarity is set to `NA`.

Value

an object of class `"dissimilarity"` containing the dissimilarities among the rows of `x`. This is typically the input for the functions `pam`, `fanny`, `agnes` or `diana`. For more details, see `dissimilarity.object`.

Background

Dissimilarities are used as inputs to cluster analysis and multidimensional scaling. The choice of metric may have a large impact.

References

Gower, J. C. (1971) A general coefficient of similarity and some of its properties, *Biometrics* **27**, 623–637.

Kaufman, L. and Rousseeuw, P.J. (1990) *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley, New York.

Struyf, A., Hubert, M. and Rousseeuw, P.J. (1997) Integrating Robust Clustering Techniques in S-PLUS, *Computational Statistics and Data Analysis* **26**, 17–37.

See Also

dissimilarity.object, dist, pam, fanny, clara, agnes, diana.

Examples

```
data(agriculture)
## Example 1 in ref:
##  Dissimilarities using Euclidean metric and without standardization
d.agr <- daisy(agriculture, metric = "euclidean", stand = FALSE)
d.agr
as.matrix(d.agr)[,"DK"] # via as.matrix.dist(.)
## compare with
as.matrix(daisy(agriculture, metric = "gower"))

data(flower)
## Example 2 in ref
summary(dfl1 <- daisy(flower, type = list(asymm = 3)))
summary(dfl2 <- daisy(flower, type = list(asymm = c(1, 3), ordratio = 7)))
## this failed earlier:
summary(dfl3 <- daisy(flower,
        type = list(asymm = c("V1", "V3"), symm= 2,
                    ordratio= 7, logratio= 8)))
```

---

diana *DIvisive ANAlysis Clustering*

Description

Computes a divisive hierarchical clustering of the dataset returning an object of class diana.

Usage

```
diana(x, diss = inherits(x, "dist"), metric = "euclidean", stand = FALSE,
      keep.diss = n < 100, keep.data = !diss)
```

Arguments

x	data matrix or data frame, or dissimilarity matrix or object, depending on the value of the diss argument. In case of a matrix or data frame, each row corresponds to an observation, and each column corresponds to a variable. All variables must be numeric. Missing values (NAs) *are* allowed. In case of a dissimilarity matrix, x is typically the output of daisy or dist. Also a vector of length n*(n-1)/2 is allowed (where n is the number of observations), and will be interpreted in the same way as the output of the above-mentioned functions. Missing values (NAs) are *not* allowed.
diss	logical flag: if TRUE (default for dist or dissimilarity objects), then x will be considered as a dissimilarity matrix. If FALSE, then x will be considered as a matrix of observations by variables.

`metric` character string specifying the metric to be used for calculating dissimilarities between observations.
The currently available options are "euclidean" and "manhattan". Euclidean distances are root sum-of-squares of differences, and manhattan distances are the sum of absolute differences. If `x` is already a dissimilarity matrix, then this argument will be ignored.

`stand` logical; if true, the measurements in `x` are standardized before calculating the dissimilarities. Measurements are standardized for each variable (column), by subtracting the variable's mean value and dividing by the variable's mean absolute deviation. If `x` is already a dissimilarity matrix, then this argument will be ignored.

`keep.diss, keep.data`
logicals indicating if the dissimilarities and/or input data `x` should be kept in the result. Setting these to `FALSE` can give much smaller results and hence even save memory allocation *time*.

Details

`diana` is fully described in chapter 6 of Kaufman and Rousseeuw (1990). It is probably unique in computing a divisive hierarchy, whereas most other software for hierarchical clustering is agglomerative. Moreover, `diana` provides (a) the divisive coefficient (see `diana.object`) which measures the amount of clustering structure found; and (b) the banner, a novel graphical display (see `plot.diana`).

The `diana`-algorithm constructs a hierarchy of clusterings, starting with one large cluster containing all n observations. Clusters are divided until each cluster contains only a single observation.
At each stage, the cluster with the largest diameter is selected. (The diameter of a cluster is the largest dissimilarity between any two of its observations.)
To divide the selected cluster, the algorithm first looks for its most disparate observation (i.e., which has the largest average dissimilarity to the other observations of the selected cluster). This observation initiates the "splinter group". In subsequent steps, the algorithm reassigns observations that are closer to the "splinter group" than to the "old party". The result is a division of the selected cluster into two new clusters.

Value

an object of class `"diana"` representing the clustering. See `?diana.object` for details.

See Also

`agnes` also for background and references; `diana.object`, `daisy`, `dist`, `plot.diana`, `twins.object`.

Examples

```
data(votes.repub)
dv <- diana(votes.repub, metric = "manhattan", stand = TRUE)
print(dv)
plot(dv)

data(agriculture)
## Plot similar to Figure 8 in ref
## Not run: plot(diana(agriculture), ask = TRUE)
```

diana.object *Divisive Analysis (DIANA) Object*

Description

The objects of class "diana" represent a divisive hierarchical clustering of a dataset.

Value

A legitimate diana object is a list with the following components:

order	a vector giving a permutation of the original observations to allow for plotting, in the sense that the branches of a clustering tree will not cross.
order.lab	a vector similar to order, but containing observation labels instead of observation numbers. This component is only available if the original observations were labelled.
height	a vector with the diameters of the clusters prior to splitting.
dc	the divisive coefficient, measuring the clustering structure of the dataset. For each observation i, denote by $d(i)$ the diameter of the last cluster to which it belongs (before being split off as a single observation), divided by the diameter of the whole dataset. The dc is the average of all $1 - d(i)$. It can also be seen as the average width (or the percentage filled) of the banner plot. Because dc grows with the number of observations, this measure should not be used to compare datasets of very different sizes.
merge	an (n-1) by 2 matrix, where n is the number of observations. Row i of merge describes the split at step n-i of the clustering. If a number j in row r is negative, then the single observation $\|j\|$ is split off at stage n-r. If j is positive, then the cluster that will be splitted at stage n-j (described by row j), is split off at stage n-r.
diss	an object of class "dissimilarity", representing the total dissimilarity matrix of the dataset.
data	a matrix containing the original or standardized measurements, depending on the stand option of the function agnes. If a dissimilarity matrix was given as input structure, then this component is not available.

GENERATION

This class of objects is returned from diana.

METHODS

The "diana" class has methods for the following generic functions: print, summary, plot.

INHERITANCE

The class "diana" inherits from "twins". Therefore, the generic function pltree can be used on a diana object, and an as.hclust method is available.

See Also

agnes, diana, plot.diana, twins.object.

Examples

```
## really see example(diana) !   Additionally:
data(votes.repub)
dv0 <- diana(votes.repub, stand = TRUE)
## Cut into 2 groups:
dv2 <- cutree(as.hclust(dv0), k = 2)
table(dv2)
rownames(votes.repub)[dv2 == 1]
```

---

dissimilarity.object

Dissimilarity Matrix Object

Description

Objects of class "dissimilarity" representing the dissimilarity matrix of a dataset.

Value

The dissimilarity matrix is symmetric, and hence its lower triangle (column wise) is represented as a vector to save storage space. If the object, is called do, and n the number of observations, i.e., n <- attr(do, "Size"), then for $i < j <= n$, the dissimilarity between (row) i and j is do[n*(i-1) - i*(i-1)/2 + j-i]. The length of the vector is $n * (n-1)/2$, i.e., of order n^2.

"dissimilarity" objects also inherit from class dist and can use dist methods, in particular, as.matrix, such that d_{ij} from above is just as.matrix(do)[i,j].

The object has the following attributes:

Size	the number of observations in the dataset.
Metric	the metric used for calculating the dissimilarities. Possible values are "euclidean", "manhattan", "mixed" (if variables of different types were present in the dataset), and "unspecified".
Labels	optionally, contains the labels, if any, of the observations of the dataset.
NA.message	optionally, if a dissimilarity could not be computed, because of too many missing values for some observations of the dataset.
Types	when a mixed metric was used, the types for each variable as one-letter codes (as in the book, e.g. p.54): **A** Asymmetric binary **S** Symmetric binary **N** Nominal (factor) **O** Ordinal (ordered factor) **I** Interval scaled (numeric) **T** raTio to be log transformed (positive numeric) .

GENERATION

daisy returns this class of objects. Also the functions pam, clara, fanny, agnes, and diana return a dissimilarity object, as one component of their return objects.

METHODS

The `"dissimilarity"` class has methods for the following generic functions: `print`, `summary`.

See Also

`daisy`, `dist`, `pam`, `clara`, `fanny`, `agnes`, `diana`.

`ellipsoidhull` *Compute the Ellipsoid Hull or Spanning Ellipsoid of a Point Set*

Description

Compute the "ellipsoid hull" or "spanning ellipsoid", i.e. the ellipsoid of minimal volume ('area' in 2D) such that all given points lie just inside or on the boundary of the ellipsoid.

Usage

```
ellipsoidhull(x, tol=0.01, maxit=5000,
              ret.wt = FALSE, ret.sqdist = FALSE, ret.pr = FALSE)
## S3 method for class 'ellipsoid':
print(x, digits = max(1, getOption("digits") - 2), ...)
```

Arguments

`x`	the n p-dimensional points asnumeric $n \times p$ matrix.
`tol`	convergence tolerance for Titterington's algorithm. Setting this to much smaller values may drastically increase the number of iterations needed, and you may want to increas `maxit` as well.
`maxit`	integer giving the maximal number of iteration steps for the algorithm.
`ret.wt, ret.sqdist, ret.pr`	logicals indicating if additional information should be returned, `ret.wt` specifying the *weights*, `ret.sqdist` the ***squared distances*** and `ret.pr` the final **pr**obabilities in the algorithms.
`digits,...`	the usual arguments to `print` methods.

Details

The "spanning ellipsoid" algorithm is said to stem from Titterington(1976), in Pison et al(1999) who use it for `clusplot.default`.
The problem can be seen as a special case of the "Min.Vol." ellipsoid of which a more more flexible and general implementation is `cov.mve` in the `MASS` package.

Value

an object of class `"ellipsoid"`, basically a `list` with several components, comprising at least

`cov`	$p \times p$ *covariance* matrix description the ellipsoid.
`loc`	p-dimensional location of the ellipsoid center.

`d2`	average squared radius. Further, $d2 = t^2$, where t is "the value of a t-statistic on the ellipse boundary" (from `ellipse` in the **ellipse** package), and hence, more usefully, `d2 = qchisq(alpha, df = p)`, where `alpha` is the confidence level for p-variate normally distributed data with location and covariance `loc` and `cov` to lie inside the ellipsoid.
`wt`	the vector of weights iff `ret.wt` was true.
`sqdist`	the vector of squared distances iff `ret.sqdist` was true.
`prob`	the vector of algorithm probabilities iff `ret.pr` was true.
`it`	number of iterations used.
`tol, maxit`	just the input argument, see above.
`eps`	the achieved tolerance which is the maximal squared radius minus p.
`ierr`	error code as from the algorithm; `0` means *ok*.
`conv`	logical indicating if the converged. This is defined as `it < maxit && ierr == 0`.

Author(s)

Martin Maechler did the present class implementation; Rousseeuw et al did the underlying code.

References

Pison, G., Struyf, A. and Rousseeuw, P.J. (1999) Displaying a Clustering with CLUSPLOT, *Computational Statistics and Data Analysis*, **30**, 381–392.
A version of this is available as technical report from http://www.agoras.ua.ac.be/abstract/Disclu99.htm

D.N. Titterington. (1976) Algorithms for computing D-optimal design on finite design spaces. In *Proc. of the 1976 Conf. on Information Science and Systems*, 213–216; John Hopkins University.

See Also

`predict.ellipsoid` which is also the `predict` method for `ellipsoid` objects. `volume.ellipsoid` for an example of 'manual' `ellipsoid` object construction;
further `ellipse` from package **ellipse** and `ellipsePoints` from package **sfsmisc**.

`chull` for the convex hull, `clusplot` which makes use of this; `cov.mve`.

Examples

```
x <- rnorm(100)
xy <- unname(cbind(x, rnorm(100) + 2*x + 10))
exy <- ellipsoidhull(xy)
exy # >> calling print.ellipsoid()

plot(xy)
lines(predict(exy))
points(rbind(exy$loc), col = "red", cex = 3, pch = 13)

exy <- ellipsoidhull(xy, tol = 1e-7, ret.wt = TRUE, ret.sq = TRUE)
str(exy) # had small `tol', hence many iterations
(ii <- which(zapsmall(exy $ wt) > 1e-6)) # only about 4 to 6 points
round(exy$wt[ii],3); sum(exy$wt[ii]) # sum to 1
```

`fanny` *Fuzzy Analysis Clustering*

Description

Computes a fuzzy clustering of the data into `k` clusters.

Usage

```
fanny(x, k, diss = inherits(x, "dist"), memb.exp = 2,
      metric = c("euclidean", "manhattan", "SqEuclidean"),
      stand = FALSE, iniMem.p = NULL, cluster.only = FALSE,
      keep.diss = !diss && !cluster.only && n < 100,
      keep.data = !diss && !cluster.only,
      maxit = 500, tol = 1e-15, trace.lev = 0)
```

Arguments

`x`	data matrix or data frame, or dissimilarity matrix, depending on the value of the `diss` argument. In case of a matrix or data frame, each row corresponds to an observation, and each column corresponds to a variable. All variables must be numeric. Missing values (NAs) are allowed. In case of a dissimilarity matrix, `x` is typically the output of `daisy` or `dist`. Also a vector of length n*(n-1)/2 is allowed (where n is the number of observations), and will be interpreted in the same way as the output of the above-mentioned functions. Missing values (NAs) are not allowed.
`k`	integer giving the desired number of clusters. It is required that $0 < k < n/2$ where n is the number of observations.
`diss`	logical flag: if TRUE (default for `dist` or `dissimilarity` objects), then `x` is assumed to be a dissimilarity matrix. If FALSE, then `x` is treated as a matrix of observations by variables.
`memb.exp`	number r strictly larger than 1 specifying the *membership exponent* used in the fit criterion; see the 'Details' below. Default: `2` which used to be hardwired inside FANNY.
`metric`	character string specifying the metric to be used for calculating dissimilarities between observations. Options are `"euclidean"` (default), `"manhattan"`, and `"SqEuclidean"`. Euclidean distances are root sum-of-squares of differences, and manhattan distances are the sum of absolute differences, and `"SqEuclidean"`, the *squared* euclidean distances are sum-of-squares of differences. Using this last option is equivalent (but somewhat slower) to computing so called "fuzzy C-means". If `x` is already a dissimilarity matrix, then this argument will be ignored.
`stand`	logical; if true, the measurements in `x` are standardized before calculating the dissimilarities. Measurements are standardized for each variable (column), by subtracting the variable's mean value and dividing by the variable's mean absolute deviation. If `x` is already a dissimilarity matrix, then this argument will be ignored.

`iniMem.p` numeric $n \times k$ matrix or `NULL` (by default); can be used to specify a starting `membership` matrix, i.e., a matrix of non-negative numbers, each row summing to one.

`cluster.only` logical; if true, no silhouette information will be computed and returned, see details.

`keep.diss, keep.data` logicals indicating if the dissimilarities and/or input data `x` should be kept in the result. Setting these to `FALSE` can give smaller results and hence also save memory allocation *time*.

`maxit, tol` maximal number of iterations and default tolerance for convergence (relative convergence of the fit criterion) for the FANNY algorithm. The defaults `maxit = 500` and `tol = 1e-15` used to be hardwired inside the algorithm.

`trace.lev` integer specifying a trace level for printing diagnostics during the C-internal algorithm. Default `0` does not print anything; higher values print increasingly more.

Details

In a fuzzy clustering, each observation is "spread out" over the various clusters. Denote by u_{iv} the membership of observation i to cluster v.

The memberships are nonnegative, and for a fixed observation i they sum to 1. The particular method `fanny` stems from chapter 4 of Kaufman and Rousseeuw (1990) (see the references in `daisy`) and has been extended by Martin Maechler to allow user specified `memb.exp`, `iniMem.p`, `maxit`, `tol`, etc.

Fanny aims to minimize the objective function

$$\sum_{v=1}^{k} \frac{\sum_{i=1}^{n} \sum_{j=1}^{n} u_{iv}^r u_{jv}^r d(i,j)}{2 \sum_{j=1}^{n} u_{jv}^r}$$

where n is the number of observations, k is the number of clusters, r is the membership exponent `memb.exp` and $d(i,j)$ is the dissimilarity between observations i and j.
Note that $r \to 1$ gives increasingly crisper clusterings whereas $r \to \infty$ leads to complete fuzzyness. K&R(1990), p.191 note that values too close to 1 can lead to slow convergence. Further note that even the default, $r = 2$ can lead to complete fuzzyness, i.e., memberships $u_{iv} \equiv 1/k$. In that case a warning is signalled and the user is advised to chose a smaller `memb.exp` ($= r$).

Compared to other fuzzy clustering methods, `fanny` has the following features: (a) it also accepts a dissimilarity matrix; (b) it is more robust to the `spherical cluster` assumption; (c) it provides a novel graphical display, the silhouette plot (see `plot.partition`).

Value

an object of class `"fanny"` representing the clustering. See `fanny.object` for details.

See Also

`agnes` for background and references; `fanny.object`, `partition.object`, `plot.partition`, `daisy`, `dist`.

Examples

```
## generate 10+15 objects in two clusters, plus 3 objects lying
## between those clusters.
x <- rbind(cbind(rnorm(10, 0, 0.5), rnorm(10, 0, 0.5)),
           cbind(rnorm(15, 5, 0.5), rnorm(15, 5, 0.5)),
           cbind(rnorm( 3,3.2,0.5), rnorm( 3,3.2,0.5)))
fannyx <- fanny(x, 2)
## Note that observations 26:28 are "fuzzy" (closer to # 2):
fannyx
summary(fannyx)
plot(fannyx)

(fan.x.15 <- fanny(x, 2, memb.exp = 1.5)) # 'crispier' for obs. 26:28
(fanny(x, 2, memb.exp = 3))               # more fuzzy in general

data(ruspini)
f4 <- fanny(ruspini, 4)
stopifnot(rle(f4$clustering)$lengths == c(20,23,17,15))
plot(f4, which = 1)
## Plot similar to Figure 6 in Stryuf et al (1996)
plot(fanny(ruspini, 5))
```

`fanny.object` *Fuzzy Analysis (FANNY) Object*

Description

The objects of class `"fanny"` represent a fuzzy clustering of a dataset.

Value

A legitimate `fanny` object is a list with the following components:

`membership`	matrix containing the memberships for each pair consisting of an observation and a cluster.
`memb.exp`	the membership exponent used in the fitting criterion.
`coeff`	Dunn's partition coefficient $F(k)$ of the clustering, where k is the number of clusters. $F(k)$ is the sum of all *squared* membership coefficients, divided by the number of observations. Its value is between $1/k$ and 1. The normalized form of the coefficient is also given. It is defined as $(F(k) - 1/k)/(1 - 1/k)$, and ranges between 0 and 1. A low value of Dunn's coefficient indicates a very fuzzy clustering, whereas a value close to 1 indicates a near-crisp clustering.
`clustering`	the clustering vector of the nearest crisp clustering, see `partition.object`.
`k.crisp`	integer ($\leq k$) giving the number of *crisp* clusters; can be less than k, where it's recommended to decrease `memb.exp`.
`objective`	named vector containing the minimal value of the objective function reached by the FANNY algorithm and the relative convergence tolerance `tol` used.
`convergence`	named vector with `iterations`, the number of iterations needed and `converged` indicating if the algorithm converged (in `maxit` iterations within convergence tolerance `tol`).

`diss`	an object of class `"dissimilarity"`, see `partition.object`.
`call`	generating call, see `partition.object`.
`silinfo`	list with silhouette information of the nearest crisp clustering, see `partition.object`.
`data`	matrix, possibibly standardized, or NULL, see `partition.object`.

GENERATION

These objects are returned from `fanny`.

METHODS

The `"fanny"` class has methods for the following generic functions: `print`, `summary`.

INHERITANCE

The class `"fanny"` inherits from `"partition"`. Therefore, the generic functions `plot` and `clusplot` can be used on a `fanny` object.

See Also

`fanny`, `print.fanny`, `dissimilarity.object`, `partition.object`, `plot.partition`.

`flower` *Flower Characteristics*

Description

8 characteristics for 18 popular flowers.

Usage

```
data(flower)
```

Format

A data frame with 18 observations on 8 variables:

[, "V1"]	factor	winters
[, "V2"]	factor	shadow
[, "V3"]	factor	tubers
[, "V4"]	factor	color
[, "V5"]	ordered	soil
[, "V6"]	ordered	preference
[, "V7"]	numeric	height
[, "V8"]	numeric	distance

V1 winters, is binary and indicates whether the plant may be left in the garden when it freezes.

V2 shadow, is binary and shows whether the plant needs to stand in the shadow.

V3 tubers, is asymmetric binary and distinguishes between plants with tubers and plants that grow

in any other way.

V4 color, is nominal and specifies the flower's color (1 = white, 2 = yellow, 3 = pink, 4 = red, 5 = blue).

V5 soil, is ordinal and indicates whether the plant grows in dry (1), normal (2), or wet (3) soil.

V6 preference, is ordinal and gives someone's preference ranking going from 1 to 18.

V7 height, is interval scaled, the plant's height in centimeters.

V8 distance, is interval scaled, the distance in centimeters that should be left between the plants.

Source

The reference below.

References

Anja Struyf, Mia Hubert & Peter J. Rousseeuw (1996): Clustering in an Object-Oriented Environment. *Journal of Statistical Software*, **1**. http://www.stat.ucla.edu/journals/jss/

Examples

```
data(flower)
## Example 2 in ref
daisy(flower, type = list(asymm = 3))
daisy(flower, type = list(asymm = c(1, 3), ordratio = 7))
```

---

lower.to.upper.tri.inds

Permute Indices for Triangular Matrices

Description

Compute index vectors for extracting or reordering of lower or upper triangular matrices that are stored as contiguous vectors.

Usage

```
lower.to.upper.tri.inds(n)
upper.to.lower.tri.inds(n)
```

Arguments

`n`	integer larger than 1.

Value

integer vector containing a permutation of `1:N` where $N = n(n-1)/2$.

Note

these functions are mainly for internal use in the cluster package, and may not remain available (unless we see a good reason).

See Also

upper.tri, lower.tri with a related purpose.

Examples

```
m5 <- matrix(NA,5,5)
m <- m5; m[lower.tri(m)] <- upper.to.lower.tri.inds(5); m
m <- m5; m[upper.tri(m)] <- lower.to.upper.tri.inds(5); m

stopifnot(lower.to.upper.tri.inds(2) == 1,
          lower.to.upper.tri.inds(3) == 1:3,
          upper.to.lower.tri.inds(3) == 1:3,
     sort(upper.to.lower.tri.inds(5)) == 1:10,
     sort(lower.to.upper.tri.inds(6)) == 1:15)
```

---

mona *MONothetic Analysis Clustering of Binary Variables*

Description

Returns a list representing a divisive hierarchical clustering of a dataset with binary variables only.

Usage

```
mona(x)
```

Arguments

x	data matrix or data frame in which each row corresponds to an observation, and each column corresponds to a variable. All variables must be binary. A limited number of missing values (NAs) is allowed. Every observation must have at least one value different from NA. No variable should have half of its values missing. There must be at least one variable which has no missing values. A variable with all its non-missing values identical, is not allowed.

Details

mona is fully described in chapter 7 of Kaufman and Rousseeuw (1990). It is "monothetic" in the sense that each division is based on a single (well-chosen) variable, whereas most other hierarchical methods (including agnes and diana) are "polythetic", i.e. they use all variables together.

The mona-algorithm constructs a hierarchy of clusterings, starting with one large cluster. Clusters are divided until all observations in the same cluster have identical values for all variables.
At each stage, all clusters are divided according to the values of one variable. A cluster is divided into one cluster with all observations having value 1 for that variable, and another cluster with all observations having value 0 for that variable.

The variable used for splitting a cluster is the variable with the maximal total association to the other variables, according to the observations in the cluster to be splitted. The association between variables f and g is given by a(f,g)*d(f,g) - b(f,g)*c(f,g), where a(f,g), b(f,g), c(f,g), and d(f,g) are the numbers in the contingency table of f and g. [That is, a(f,g) (resp. d(f,g)) is the number of observations for which f and g both have value 0 (resp. value 1); b(f,g) (resp. c(f,g)) is the number

of observations for which f has value 0 (resp. 1) and g has value 1 (resp. 0).] The total association of a variable f is the sum of its associations to all variables.

This algorithm does not work with missing values, therefore the data are revised, e.g. all missing values are filled in. To do this, the same measure of association between variables is used as in the algorithm. When variable f has missing values, the variable g with the largest absolute association to f is looked up. When the association between f and g is positive, any missing value of f is replaced by the value of g for the same observation. If the association between f and g is negative, then any missing value of f is replaced by the value of 1-g for the same observation.

Value

an object of class `"mona"` representing the clustering. See `mona.object` for details.

See Also

`agnes` for background and references; `mona.object`, `plot.mona`.

Examples

```
data(animals)
ma <- mona(animals)
ma
## Plot similar to Figure 10 in Struyf et al (1996)
plot(ma)
```

---

mona.object *Monothetic Analysis (MONA) Object*

Description

The objects of class `"mona"` represent the divisive hierarchical clustering of a dataset with only binary variables (measurements). This class of objects is returned from `mona`.

Value

A legitimate `mona` object is a list with the following components:

`data`	matrix with the same dimensions as the original data matrix, but with factors coded as 0 and 1, and all missing values replaced.
`order`	a vector giving a permutation of the original observations to allow for plotting, in the sense that the branches of a clustering tree will not cross.
`order.lab`	a vector similar to `order`, but containing observation labels instead of observation numbers. This component is only available if the original observations were labelled.
`variable`	vector of length n-1 where n is the number of observations, specifying the variables used to separate the observations of `order`.
`step`	vector of length n-1 where n is the number of observations, specifying the separation steps at which the observations of `order` are separated.

METHODS

The "mona" class has methods for the following generic functions: print, summary, plot.

See Also

mona for examples etc, plot.mona.

---

pam *Partitioning Around Medoids*

Description

Partitioning (clustering) of the data into k clusters "around medoids", a more robust version of K-means.

Usage

```
pam(x, k, diss = inherits(x, "dist"), metric = "euclidean",
    medoids = NULL, stand = FALSE, cluster.only = FALSE,
    keep.diss = !diss && !cluster.only && n < 100,
    keep.data = !diss && !cluster.only, trace.lev = 0)
```

Arguments

x	data matrix or data frame, or dissimilarity matrix or object, depending on the value of the diss argument. In case of a matrix or data frame, each row corresponds to an observation, and each column corresponds to a variable. All variables must be numeric. Missing values (NAs) *are* allowed—as long as every pair of observations has at least one case not missing. In case of a dissimilarity matrix, x is typically the output of daisy or dist. Also a vector of length n*(n-1)/2 is allowed (where n is the number of observations), and will be interpreted in the same way as the output of the above-mentioned functions. Missing values (NAs) are *not* allowed.
k	positive integer specifying the number of clusters, less than the number of observations.
diss	logical flag: if TRUE (default for dist or dissimilarity objects), then x will be considered as a dissimilarity matrix. If FALSE, then x will be considered as a matrix of observations by variables.
metric	character string specifying the metric to be used for calculating dissimilarities between observations. The currently available options are "euclidean" and "manhattan". Euclidean distances are root sum-of-squares of differences, and manhattan distances are the sum of absolute differences. If x is already a dissimilarity matrix, then this argument will be ignored.
medoids	NULL (default) or length-k vector of integer indices (in 1:n) specifying initial medoids instead of using the '*build*' algorithm.

`stand` logical; if true, the measurements in `x` are standardized before calculating the dissimilarities. Measurements are standardized for each variable (column), by subtracting the variable's mean value and dividing by the variable's mean absolute deviation. If `x` is already a dissimilarity matrix, then this argument will be ignored.

`cluster.only` logical; if true, only the clustering will be computed and returned, see details.

`keep.diss, keep.data` logicals indicating if the dissimilarities and/or input data `x` should be kept in the result. Setting these to `FALSE` can give much smaller results and hence even save memory allocation *time*.

`trace.lev` integer specifying a trace level for printing diagnostics during the build and swap phase of the algorithm. Default `0` does not print anything; higher values print increasingly more.

Details

`pam` is fully described in chapter 2 of Kaufman and Rousseeuw (1990). Compared to the k-means approach in `kmeans`, the function `pam` has the following features: (a) it also accepts a dissimilarity matrix; (b) it is more robust because it minimizes a sum of dissimilarities instead of a sum of squared euclidean distances; (c) it provides a novel graphical display, the silhouette plot (see `plot.partition`) (d) it allows to select the number of clusters using `mean(silhouette(pr))` on the result `pr <- pam(..)`, or directly its component `pr$silinfo$avg.width`, see also `pam.object`.

When `cluster.only` is true, the result is simply a (possibly named) integer vector specifying the clustering, i.e.,
`pam(x,k, cluster.only=TRUE)` is the same as
`pam(x,k)$clustering` but computed more efficiently.

The `pam`-algorithm is based on the search for `k` representative objects or medoids among the observations of the dataset. These observations should represent the structure of the data. After finding a set of `k` medoids, `k` clusters are constructed by assigning each observation to the nearest medoid. The goal is to find `k` representative objects which minimize the sum of the dissimilarities of the observations to their closest representative object.
By default, when `medoids` are not specified, the algorithm first looks for a good initial set of medoids (this is called the **build** phase). Then it finds a local minimum for the objective function, that is, a solution such that there is no single switch of an observation with a medoid that will decrease the objective (this is called the **swap** phase).

When the `medoids` are specified, their order does *not* matter; in general, the algorithms have been designed to not depend on the order of the observations.

Value

an object of class `"pam"` representing the clustering. See `?pam.object` for details.

Note

For datasets larger than (say) 200 observations, `pam` will take a lot of computation time. Then the function `clara` is preferable.

See Also

`agnes` for background and references; `pam.object`, `clara`, `daisy`, `partition.object`, `plot.partition`, `dist`.

Examples

```
## generate 25 objects, divided into 2 clusters.
x <- rbind(cbind(rnorm(10,0,0.5), rnorm(10,0,0.5)),
           cbind(rnorm(15,5,0.5), rnorm(15,5,0.5)))
pamx <- pam(x, 2)
pamx
summary(pamx)
plot(pamx)
## use obs. 1 & 16 as starting medoids -- same result (typically)
(p2m <- pam(x, 2, medoids = c(1,16)))

p3m <- pam(x, 3, trace = 2)
## rather stupid initial medoids:
(p3m. <- pam(x, 3, medoids = 3:1, trace = 1))


pam(daisy(x, metric = "manhattan"), 2, diss = TRUE)

data(ruspini)
## Plot similar to Figure 4 in Stryuf et al (1996)
## Not run: plot(pam(ruspini, 4), ask = TRUE)
```

---

pam.object *Partitioning Around Medoids (PAM) Object*

Description

The objects of class "pam" represent a partitioning of a dataset into clusters.

Value

A legitimate pam object is a list with the following components:

medoids	the medoids or representative objects of the clusters. If a dissimilarity matrix was given as input to pam, then a vector of numbers or labels of observations is given, else medoids is a matrix with in each row the coordinates of one medoid.
id.med	integer vector of *indices* giving the medoid observation numbers.
clustering	the clustering vector, see partition.object.
objective	the objective function after the first and second step of the pam algorithm.
isolation	vector with length equal to the number of clusters, specifying which clusters are isolated clusters (L- or L*-clusters) and which clusters are not isolated. A cluster is an L*-cluster iff its diameter is smaller than its separation. A cluster is an L-cluster iff for each observation i the maximal dissimilarity between i and any other observation of the cluster is smaller than the minimal dissimilarity between i and any observation of another cluster. Clearly each L*-cluster is also an L-cluster.
clusinfo	matrix, each row gives numerical information for one cluster. These are the cardinality of the cluster (number of observations), the maximal and average dissimilarity between the observations in the cluster and the cluster's medoid,

	the diameter of the cluster (maximal dissimilarity between two observations of the cluster), and the separation of the cluster (minimal dissimilarity between an observation of the cluster and an observation of another cluster).
`silinfo`	list with silhouette width information, see `partition.object`.
`diss`	dissimilarity (maybe NULL), see `partition.object`.
`call`	generating call, see `partition.object`.
`data`	(possibibly standardized) see `partition.object`.

GENERATION

These objects are returned from `pam`.

METHODS

The `"pam"` class has methods for the following generic functions: `print`, `summary`.

INHERITANCE

The class `"pam"` inherits from `"partition"`. Therefore, the generic functions `plot` and `clusplot` can be used on a `pam` object.

See Also

`pam`, `dissimilarity.object`, `partition.object`, `plot.partition`.

Examples

```
## Use the silhouette widths for assessing the best number of clusters,
## following a one-dimensional example from Christian Hennig :
##
x <- c(rnorm(50), rnorm(50,mean=5), rnorm(30,mean=15))
asw <- numeric(20)
## Note that "k=1" won't work!
for (k in 2:20)
  asw[k] <- pam(x, k) $ silinfo $ avg.width
k.best <- which.max(asw)
cat("silhouette-optimal number of clusters:", k.best, "\n")

plot(1:20, asw, type= "h", main = "pam() clustering assessment",
     xlab= "k  (# clusters)", ylab = "average silhouette width")
axis(1, k.best, paste("best",k.best,sep="\n"), col = "red", col.axis = "red")
```

`partition.object` *Partitioning Object*

Description

The objects of class `"partition"` represent a partitioning of a dataset into clusters.

Value

a "partition" object is a list with the following (and typically more) components:

`clustering` the clustering vector. An integer vector of length n, the number of observations, giving for each observation the number ('id') of the cluster to which it belongs.

`call` the matched `call` generating the object.

`silinfo` a list with all *silhouette* information, only available when the number of clusters is non-trivial, i.e., $1 < k < n$ and then has the following components, see `silhouette`

widths an (n x 3) matrix, as returned by `silhouette()`, with for each observation i the cluster to which i belongs, as well as the neighbor cluster of i (the cluster, not containing i, for which the average dissimilarity between its observations and i is minimal), and the silhouette width $s(i)$ of the observation.

clus.avg.widths the average silhouette width per cluster.

avg.width the average silhouette width for the dataset, i.e., simply the average of $s(i)$ over all observations i.

This information is also needed to construct a *silhouette plot* of the clustering, see `plot.partition`.

Note that `avg.width` can be maximized over different clusterings (e.g. with varying number of clusters) to choose an *optimal* clustering.

`objective` value of criterion maximized during the partitioning algorithm, may more than one entry for different stages.

`diss` an object of class "dissimilarity", representing the total dissimilarity matrix of the dataset (or relevant subset, e.g. for `clara`).

`data` a matrix containing the original or standardized data. This might be missing to save memory or when a dissimilarity matrix was given as input structure to the clustering method.

GENERATION

These objects are returned from `pam`, `clara` or `fanny`.

METHODS

The "partition" class has a method for the following generic functions: `plot`, `clusplot`.

INHERITANCE

The following classes inherit from class "partition" : "pam", "clara" and "fanny".

See `pam.object`, `clara.object` and `fanny.object` for details.

See Also

`pam`, `clara`, `fanny`.

plantTraits *Plant Species Traits Data*

Description

This dataset constitutes a description of 136 plant species according to biological attributes (morphological or reproductive)

Usage

```
data(plantTraits)
```

Format

A data frame with 136 observations on the following 31 variables.

pdias Diaspore mass (mg)

longindex Seed bank longevity

durflow Flowering duration

height Plant height, an ordered factor with levels 1 < 2 < ... < 8.

begflow Time of first flowering, an ordered factor with levels 1 < 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9

mycor Mycorrhizas, an ordered factor with levels 0never < 1 sometimes< 2always

vegaer aerial vegetative propagation, an ordered factor with levels 0never < 1 present but limited< 2important.

vegsout underground vegetative propagation, an ordered factor with 3 levels identical to vegaer above.

autopoll selfing pollination, an ordered factor with levels 0never < 1rare < 2 often< the rule3

insects insect pollination, an ordered factor with 5 levels 0 < ... < 4.

wind wind pollination, an ordered factor with 5 levels 0 < ... < 4.

lign a binary factor with levels 0:1, indicating if plant is woody.

piq a binary factor indicating if plant is thorny.

ros a binary factor indicating if plant is rosette.

semiros semi-rosette plant, a binary factor (0: no; 1: yes).

leafy leafy plant, a binary factor.

suman summer annual, a binary factor.

winan winter annual, a binary factor.

monocarp monocarpic perennial, a binary factor.

polycarp polycarpic perennial, a binary factor.

seasaes seasonal aestival leaves, a binary factor.

seashiv seasonal hibernal leaves, a binary factor.

seasver seasonal vernal leaves, a binary factor.

everalw leaves always evergreen, a binary factor.

everparti leaves partially evergreen, a binary factor.

elaio fruits with an elaiosome (dispersed by ants), a binary factor.

endozoo endozoochorous fruits, a binary factor.

epizoo epizoochorous fruits, a binary factor.

aquat aquatic dispersal fruits, a binary factor.

windgl wind dispersed fruits, a binary factor.

unsp unspecialized mechanism of seed dispersal, a binary factor.

Details

Most of factor attributes are not disjunctive. For example, a plant can be usually pollinated by insects but sometimes self-pollination can occured.

Source

Vallet, Jeanne (2005) *Structuration de communautés végétales et analyse comparative de traits biologiques le long d'un gradient d'urbanisation*. Mémoire de Master 2 'Ecologie-Biodiversité-Evolution'; Université Paris Sud XI, 30p.+ annexes (in french)

Examples

```
data(plantTraits)

## Calculation of a dissimilarity matrix
library(cluster)
dai.b <- daisy(plantTraits,
               type = list(ordratio = 4:11, symm = 12:13, asymm = 14:31))

## Hierarchical classification
agn.trts <- agnes(dai.b, method="ward")
plot(agn.trts, which.plots = 2, cex= 0.6)
plot(agn.trts, which.plots = 1)
cutree6 <- cutree(agn.trts, k=6)
cutree6

## Principal Coordinate Analysis
cmdsdai.b <- cmdscale(dai.b, k=6)
plot(cmdsdai.b[, 1:2], asp = 1, col = cutree6)
```

---

plot.agnes *Plots of an Agglomerative Hierarchical Clustering*

Description

Creates plots for visualizing an `agnes` object.

Usage

```
## S3 method for class 'agnes':
plot(x, ask = FALSE, which.plots = NULL, main = NULL,
           sub = paste("Agglomerative Coefficient = ",round(x$ac, digits = 2)),
           adj = 0, nmax.lab = 35, max.strlen = 5, xax.pretty = TRUE, ...)
```

Arguments

x	an object of class "agnes", typically created by agnes(.).
ask	logical; if true and which.plots is NULL, plot.agnes operates in interactive mode, via menu.
which.plots	integer vector or NULL (default), the latter producing both plots. Otherwise, which.plots must contain integers of 1 for a *banner* plot or 2 for a dendrogramm or "clustering tree".
main, sub	main and sub title for the plot, with convenient defaults. See documentation for these arguments in plot.default.
adj	for label adjustment in bannerplot().
nmax.lab	integer indicating the number of labels which is considered too large for single-name labelling the banner plot.
max.strlen	positive integer giving the length to which strings are truncated in banner plot labeling.
xax.pretty	logical or integer indicating if pretty(*, n = xax.pretty) should be used for the x axis. xax.pretty = FALSE is for back compatibility.
...	graphical parameters (see par) may also be supplied and are passed to bannerplot() or pltree(), respectively.

Details

When ask = TRUE, rather than producing each plot sequentially, plot.agnes displays a menu listing all the plots that can be produced. If the menu is not desired but a pause between plots is still wanted one must set par(ask= TRUE) before invoking the plot command.

The banner displays the hierarchy of clusters, and is equivalent to a tree. See Rousseeuw (1986) or chapter 5 of Kaufman and Rousseeuw (1990). The banner plots distances at which observations and clusters are merged. The observations are listed in the order found by the agnes algorithm, and the numbers in the height vector are represented as bars between the observations.

The leaves of the clustering tree are the original observations. Two branches come together at the distance between the two clusters being merged.

For more customization of the plots, rather call bannerplot and pltree directly with corresponding arguments, e.g., xlab or ylab.

Side Effects

Appropriate plots are produced on the current graphics device. This can be one or both of the following choices:
Banner
Clustering tree

Note

In the banner plot, observation labels are only printed when the number of observations is limited less than nmax.lab (35, by default), for readability. Moreover, observation labels are truncated to maximally max.strlen (5) characters.

References

Kaufman, L. and Rousseeuw, P.J. (1990) *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley, New York.

Rousseeuw, P.J. (1986). A visual display for hierarchical classification, in *Data Analysis and Informatics 4*; edited by E. Diday, Y. Escoufier, L. Lebart, J. Pages, Y. Schektman, and R. Tomassone. North-Holland, Amsterdam, 743–748.

Struyf, A., Hubert, M. and Rousseeuw, P.J. (1997) Integrating Robust Clustering Techniques in S-PLUS, *Computational Statistics and Data Analysis*, **26**, 17-37.

See Also

`agnes` and `agnes.object`; `bannerplot`, `pltree`, and `par`.

Examples

```
## Can also pass `labels' to pltree() and bannerplot():
data(iris)
cS <- as.character(Sp <- iris$Species)
cS[Sp == "setosa"] <- "S"
cS[Sp == "versicolor"] <- "V"
cS[Sp == "virginica"] <- "g"
ai <- agnes(iris[, 1:4])
plot(ai, labels = cS, nmax = 150)# bannerplot labels are mess
```

---

plot.diana *Plots of a Divisive Hierarchical Clustering*

Description

Creates plots for visualizing a `diana` object.

Usage

```
## S3 method for class 'diana':
plot(x, ask = FALSE, which.plots = NULL, main = NULL,
           sub = paste("Divisive Coefficient = ", round(x$dc, digits = 2)),
           adj = 0, nmax.lab = 35, max.strlen = 5, xax.pretty = TRUE, ...)
```

Arguments

`x`	an object of class `"diana"`, typically created by `diana(.)`.
`ask`	logical; if true and `which.plots` is `NULL`, `plot.diana` operates in interactive mode, via `menu`.
`which.plots`	integer vector or NULL (default), the latter producing both plots. Otherwise, `which.plots` must contain integers of `1` for a *banner* plot or `2` for a dendrogramm or "clustering tree".
`main, sub`	main and sub title for the plot, each with a convenient default. See documentation for these arguments in `plot.default`.
`adj`	for label adjustment in `bannerplot()`.

nmax.lab	integer indicating the number of labels which is considered too large for single-name labelling the banner plot.
max.strlen	positive integer giving the length to which strings are truncated in banner plot labeling.
xax.pretty	logical or integer indicating if pretty(*, n = xax.pretty) should be used for the x axis. xax.pretty = FALSE is for back compatibility.
...	graphical parameters (see par) may also be supplied and are passed to bannerplot() or pltree(), respectively.

Details

When ask = TRUE, rather than producing each plot sequentially, plot.diana displays a menu listing all the plots that can be produced. If the menu is not desired but a pause between plots is still wanted one must set par(ask= TRUE) before invoking the plot command.

The banner displays the hierarchy of clusters, and is equivalent to a tree. See Rousseeuw (1986) or chapter 6 of Kaufman and Rousseeuw (1990). The banner plots the diameter of each cluster being splitted. The observations are listed in the order found by the diana algorithm, and the numbers in the height vector are represented as bars between the observations.

The leaves of the clustering tree are the original observations. A branch splits up at the diameter of the cluster being splitted.

Side Effects

An appropriate plot is produced on the current graphics device. This can be one or both of the following choices:
Banner
Clustering tree

Note

In the banner plot, observation labels are only printed when the number of observations is limited less than nmax.lab (35, by default), for readability. Moreover, observation labels are truncated to maximally max.strlen (5) characters.

References

see those in plot.agnes.

See Also

diana, diana.object, twins.object, par.

Examples

```
example(diana)# -> dv <- diana(....)

plot(dv, which = 1, nmax.lab = 100)

## wider labels :
op <- par(mar = par("mar") + c(0, 2, 0,0))
plot(dv, which = 1, nmax.lab = 100, max.strlen = 12)
par(op)
```

plot.mona *Banner of Monothetic Divisive Hierarchical Clusterings*

Description

Creates the banner of a mona object.

Usage

```
## S3 method for class 'mona':
plot(x, main = paste("Banner of ", deparse(x$call)),
          sub = NULL, xlab = "Separation step",
          col = c(2,0), axes = TRUE, adj = 0,
          nmax.lab = 35, max.strlen = 5, ...)
```

Arguments

x	an object of class "mona", typically created by mona(.).
main,sub	main and sub titles for the plot, with convenient defaults. See documentation in plot.default.
xlab	x axis label, see title.
col,adj	graphical parameters passed to bannerplot().
axes	logical, indicating if (labeled) axes should be drawn.
nmax.lab	integer indicating the number of labels which is considered too large for labeling.
max.strlen	positive integer giving the length to which strings are truncated in labeling.
...	further graphical arguments are passed to bannerplot() and text.

Details

Plots the separation step at which clusters are splitted. The observations are given in the order found by the mona algorithm, the numbers in the step vector are represented as bars between the observations.

When a long bar is drawn between two observations, those observations have the same value for each variable. See chapter 7 of Kaufman and Rousseeuw (1990).

Side Effects

A banner is plotted on the current graphics device.

Note

In the banner plot, observation labels are only printed when the number of observations is limited less than nmax.lab (35, by default), for readability. Moreover, observation labels are truncated to maximally max.strlen (5) characters.

References

see those in plot.agnes.

See Also

mona, mona.object, par.

---

plot.partition *Plot of a Partition of the Data Set*

Description

Creates plots for visualizing a partition object.

Usage

```
## S3 method for class 'partition':
plot(x, ask = FALSE, which.plots = NULL,
     nmax.lab = 40, max.strlen = 5, data = x$data, dist = NULL,
     cor = TRUE, stand = FALSE, lines = 2,
     shade = FALSE, color = FALSE, labels = 0, plotchar = TRUE,
     span = TRUE, xlim = NULL, ylim = NULL, main = NULL, ...)
```

Arguments

x	an object of class "partition", typically created by the functions pam, clara, or fanny.
ask	logical; if true and which.plots is NULL, plot.partition operates in interactive mode, via menu.
which.plots	integer vector or NULL (default), the latter producing both plots. Otherwise, which.plots must contain integers of 1 for a *clusplot* or 2 for *silhouette*.
nmax.lab	integer indicating the number of labels which is considered too large for single-name labeling the silhouette plot.
max.strlen	positive integer giving the length to which strings are truncated in silhouette plot labeling.
data	numeric matrix with the scaled data; per default taken from the partition object x, but can be specified explicitly.
dist	when x does not have a diss component as for pam(*, keep.diss=FALSE), dist must be the dissimilarity if a clusplot is desired.
cor, stand, lines, shade, color, labels, plotchar, span, xlim, ylim, main, ...	All optional arguments available for the clusplot.default function (except for the diss one) and graphical parameters (see par) may also be supplied as arguments to this function.

Details

When ask= TRUE, rather than producing each plot sequentially, plot.partition displays a menu listing all the plots that can be produced. If the menu is not desired but a pause between plots is still wanted, call par(ask= TRUE) before invoking the plot command.

The *clusplot* of a cluster partition consists of a two-dimensional representation of the observations, in which the clusters are indicated by ellipses (see clusplot.partition for more details).

The *silhouette plot* of a nonhierarchical clustering is fully described in Rousseeuw (1987) and in chapter 2 of Kaufman and Rousseeuw (1990). For each observation i, a bar is drawn, representing its silhouette width s(i), see `silhouette` for details. Observations are grouped per cluster, starting with cluster 1 at the top. Observations with a large s(i) (almost 1) are very well clustered, a small s(i) (around 0) means that the observation lies between two clusters, and observations with a negative s(i) are probably placed in the wrong cluster.

A clustering can be performed for several values of `k` (the number of clusters). Finally, choose the value of `k` with the largest overall average silhouette width.

Side Effects

An appropriate plot is produced on the current graphics device. This can be one or both of the following choices:
Clusplot
Silhouette plot

Note

In the silhouette plot, observation labels are only printed when the number of observations is less than `nmax.lab` (40, by default), for readability. Moreover, observation labels are truncated to maximally `max.strlen` (5) characters.
For more flexibility, use `plot(silhouette(x), ...)`, see `plot.silhouette`.

References

Rousseeuw, P.J. (1987) Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *J. Comput. Appl. Math.*, **20**, 53–65.

Further, the references in `plot.agnes`.

See Also

`partition.object`, `clusplot.partition`, `clusplot.default`, `pam`, `pam.object`, `clara`, `clara.object`, `fanny`, `fanny.object`, `par`.

Examples

```
## generate 25 objects, divided into 2 clusters.
x <- rbind(cbind(rnorm(10,0,0.5), rnorm(10,0,0.5)),
           cbind(rnorm(15,5,0.5), rnorm(15,5,0.5)))
plot(pam(x, 2))

## Save space not keeping data in clus.object, and still clusplot() it:
data(xclara)
cx <- clara(xclara, 3, keep.data = FALSE)
cx$data # is NULL
plot(cx, data = xclara)
```

pltree *Clustering Trees - Generic Function*

Description

Generic function drawing a clustering tree ("dendrogram") on the current graphics device. There is a `twins` method, see `pltree.twins` for usage and examples.

Usage

```
pltree(x, ...)
```

Arguments

`x`	an R object (for which a `pltree` method is defined).
`...`	methods may have additional arguments. Graphical parameters (see `par`) may also be supplied as arguments to this function.

See Also

`pltree.twins`, `twins.object`.

pltree.twins *Clustering Tree of a Hierarchical Clustering*

Description

Draws a clustering tree (dendrogramm) of a `twins` object, i.e., hierarchical clustering, typically resulting from `agnes()` or `diana()`.

Usage

```
## S3 method for class 'twins':
pltree(x, main = paste("Dendrogram of ", deparse(x$call)),
             labels = NULL, ylab = "Height", ...)
```

Arguments

`x`	an object of class `"twins"`, typically created by either `agnes()` or `diana()`.
`main`	main title with a sensible default.
`labels`	labels to use; the default is constructed from `x`.
`ylab`	label for y-axis.
`...`	graphical parameters (see `par`) may also be supplied as arguments to this function.

Details

Creates a plot of a clustering tree given a `twins` object. The leaves of the tree are the original observations. In case of an agglomerative clustering, two branches come together at the distance between the two clusters being merged. For a divisive clustering, a branch splits up at the diameter of the cluster being splitted.

Note that currently the method function simply calls `plot(as.hclust(x), ...)`, which dispatches to `plot.hclust(..)`. If more flexible plots are needed, consider `xx <- as.dendrogram(as.dendrogram(x))` and plotting `xx`, see `plot.dendrogram`.

Value

a NULL value is returned.

See Also

`agnes`, `agnes.object`, `diana`, `diana.object`, `hclust`, `par`, `plot.agnes`, `plot.diana`.

Examples

```
data(votes.repub)
agn <- agnes(votes.repub)
pltree(agn)

dagn  <- as.dendrogram(as.hclust(agn))
dagn2 <- as.dendrogram(as.hclust(agn), hang = 0.2)
op <- par(mar = par("mar") + c(0,0,0, 2)) # more space to the right
plot(dagn2, horiz = TRUE)
plot(dagn, horiz = TRUE, center = TRUE,
     nodePar = list(lab.cex = 0.6, lab.col = "forest green", pch = NA),
     main = deparse(agn$call))
par(op)
```

---

pluton *Isotopic Composition Plutonium Batches*

Description

The `pluton` data frame has 45 rows and 4 columns, containing percentages of isotopic composition of 45 Plutonium batches.

Usage

```
data(pluton)
```

Format

This data frame contains the following columns:

Pu238 the percentages of ^{238}Pu, always less than 2 percent.

Pu239 the percentages of ^{239}Pu, typically between 60 and 80 percent (from neutron capture of Uranium, ^{238}U).

Pu240 percentage of the plutonium 240 isotope.

Pu241 percentage of the plutonium 241 isotope.

Details

Note that the percentage of plutonium 242 can be computed from the other four percentages, see the examples.

In the reference below it is explained why it is very desirable to combine these plutonium patches in three groups of similar size.

Source

Available as 'pluton.dat' from the archive currently http://www.agoras.ua.ac.be/datasets/clusplot-examples.tar.gz.

References

Rousseeuw, P.J. and Kaufman, L and Trauwaert, E. (1996) Fuzzy clustering using scatter matrices, *Computational Statistics and Data Analysis* **23**(1), 135–151.

Examples

```
data(pluton)

hist(apply(pluton,1,sum), col = "gray") # between 94% and 100%
pu5 <- pluton
pu5$Pu242 <- 100 - apply(pluton,1,sum) # the remaining isotope.
pairs(pu5)
```

---

predict.ellipsoid *Predict Method for Ellipsoid Objects*

Description

Compute points on the ellipsoid boundary, mostly for drawing.

Usage

```
predict.ellipsoid(object, n.out=201, ...)
## S3 method for class 'ellipsoid':
predict(object, n.out=201, ...)
ellipsoidPoints(A, d2, loc, n.half = 201)
```

Arguments

`object`	an object of class `ellipsoid`, typically from `ellipsoidhull()`; alternatively any list-like object with proper components, see details below.
`n.out, n.half`	half the number of points to create.
`A, d2, loc`	arguments of the auxilary `ellipsoidPoints`, see below.
`...`	passed to and from methods.

Details

Note `ellipsoidPoints` is the workhorse function of `predict.ellipsoid` a standalone function and method for `ellipsoid` objects, see `ellipsoidhull`. The class of `object` is not checked; it must solely have valid components `loc` (length p), the $p \times p$ matrix `cov` (corresponding to `A`) and `d2` for the center, the shape ("covariance") matrix and the squared average radius (or distance) or `qchisq(*, p)` quantile.

Value

a numeric matrix of dimension `2*n.out` times p.

See Also

`ellipsoidhull`, `volume.ellipsoid`.

Examples

```
 ## see also  example(ellipsoidhull)

## Robust vs. L.S. covariance matrix
set.seed(143)
x <- rt(200, df=3)
y <- 3*x + rt(200, df=2)
plot(x,y, main="non-normal data (N=200)")
mtext("with classical and robust cov.matrix ellipsoids")
X <- cbind(x,y)
C.ls <- cov(X) ; m.ls <- colMeans(X)
d2.99 <- qchisq(0.99, df = 2)
lines(ellipsoidPoints(C.ls, d2.99, loc=m.ls), col="green")
if(require(MASS)) {
  Cxy <- cov.rob(cbind(x,y))
  lines(ellipsoidPoints(Cxy$cov, d2 = d2.99, loc=Cxy$center), col="red")
}# MASS
```

`print.agnes` *Print Method for AGNES Objects*

Description

Prints the call, agglomerative coefficient, ordering of objects and distances between merging clusters ('Height') of an `agnes` object.

This is a method for the generic `print()` function for objects inheriting from class `agnes`, see `agnes.object`.

Usage

```
## S3 method for class 'agnes':
print(x, ...)
```

Arguments

`x`	an agnes object.
`...`	potential further arguments (required by generic).

See Also

summary.agnes producing more output; agnes, agnes.object, print, print.default.

---

print.clara *Print Method for CLARA Objects*

Description

Prints the best sample, medoids, clustering vector and objective function of clara object.

This is a method for the function print() for objects inheriting from class clara.

Usage

```
## S3 method for class 'clara':
print(x, ...)
```

Arguments

x	a clara object.
...	potential further arguments (require by generic).

See Also

summary.clara producing more output; clara, clara.object, print, print.default.

---

print.diana *Print Method for DIANA Objects*

Description

Prints the ordering of objects, diameters of splitted clusters, and divisive coefficient of a diana object.

This is a method for the function print() for objects inheriting from class diana.

Usage

```
## S3 method for class 'diana':
print(x, ...)
```

Arguments

x	a diana object.
...	potential further arguments (require by generic).

See Also

diana, diana.object, print, print.default.

`print.dissimilarity`
Print and Summary Methods for Dissimilarity Objects

Description

Print or summarize the distances and the attributes of a `dissimilarity` object.

These are methods for the functions `print()` and `summary()` for `dissimilarity` objects. See `print`, `print.default`, or `summary` for the general behavior of these.

Usage

```
## S3 method for class 'dissimilarity':
print(x, diag = NULL, upper = NULL,
      digits = getOption("digits"), justify = "none", right = TRUE, ...)
## S3 method for class 'dissimilarity':
summary(object,
        digits = max(3, getOption("digits") - 2), ...)
## S3 method for class 'summary.dissimilarity':
print(x, ...)
```

Arguments

`x, object`	a `dissimilarity` object or a `summary.dissimilarity` one for `print.summary.dissimilarity()`.
`digits`	the number of digits to use, see `print.default`.
`diag, upper, justify, right`	optional arguments specifying how the triangular dissimilarity matrix is printed; see `print.dist`.
`...`	potential further arguments (require by generic).

See Also

`daisy`, `dissimilarity.object`, `print`, `print.default`, `print.dist`.

Examples

```
## See  example(daisy)

sd <- summary(daisy(matrix(rnorm(100), 20,5)))
sd # -> print.summary.dissimilarity(.)
str(sd)
```

print.fanny *Print and Summary Methods for FANNY Objects*

Description

Prints the objective function, membership coefficients and clustering vector of `fanny` object.

This is a method for the function `print()` for objects inheriting from class `fanny`.

Usage

```
## S3 method for class 'fanny':
print(x, digits = getOption("digits"), ...)
## S3 method for class 'fanny':
summary(object, ...)
## S3 method for class 'summary.fanny':
print(x, digits = getOption("digits"), ...)
```

Arguments

`x, object`	a `fanny` object.
`digits`	number of significant digits for printing, see `print.default`.
`...`	potential further arguments (required by generic).

See Also

`fanny`, `fanny.object`, `print`, `print.default`.

print.mona *Print Method for MONA Objects*

Description

Prints the ordering of objects, separation steps, and used variables of a `mona` object.

This is a method for the function `print()` for objects inheriting from class `mona`.

Usage

```
## S3 method for class 'mona':
print(x, ...)
```

Arguments

`x`	a mona object.
`...`	potential further arguments (require by generic).

See Also

`mona`, `mona.object`, `print`, `print.default`.

print.pam *Print Method for PAM Objects*

Description

Prints the medoids, clustering vector and objective function of pam object.

This is a method for the function print() for objects inheriting from class pam.

Usage

```
## S3 method for class 'pam':
print(x, ...)
```

Arguments

x	a pam object.
...	potential further arguments (require by generic).

See Also

pam, pam.object, print, print.default.

ruspini *Ruspini Data*

Description

The Ruspini data set, consisting of 75 points in four groups that is popular for illustrating clustering techniques.

Usage

```
data(ruspini)
```

Format

A data frame with 75 observations on 2 variables giving the x and y coordinates of the points, respectively.

Source

E. H. Ruspini (1970): Numerical methods for fuzzy clustering. *Inform. Sci.*, **2**, 319–350.

References

Anja Struyf, Mia Hubert & Peter J. Rousseeuw (1996): Clustering in an Object-Oriented Environment. *Journal of Statistical Software*, **1**. http://www.stat.ucla.edu/journals/jss/

Examples

```
data(ruspini)

## Plot similar to Figure 4 in Stryuf et al (1996)
## Not run: plot(pam(ruspini, 4), ask = TRUE)


## Plot similar to Figure 6 in Stryuf et al (1996)
plot(fanny(ruspini, 5))
```

---

silhouette *Compute or Extract Silhouette Information from Clustering*

Description

Compute silhouette information according to a given clustering in k clusters.

Usage

```
silhouette(x, ...)
## Default S3 method:
silhouette  (x, dist, dmatrix, ...)
## S3 method for class 'partition':
silhouette(x, ...)
## S3 method for class 'clara':
silhouette(x, full = FALSE, ...)

sortSilhouette(object, ...)
## S3 method for class 'silhouette':
summary(object, FUN = mean, ...)
## S3 method for class 'silhouette':
plot(x, nmax.lab = 40, max.strlen = 5,
     main = NULL, sub = NULL, xlab = expression("Silhouette width "* s[i]),
     col = "gray",  do.col.sort = length(col) > 1, border = 0,
     cex.names = par("cex.axis"), do.n.k = TRUE, do.clus.stat = TRUE, ...)
```

Arguments

`x`	an object of appropriate class; for the `default` method an integer vector with k different integer cluster codes or a list with such an `x$clustering` component. Note that silhouette statistics are only defined if $2 \leq k \leq n-1$.
`dist`	a dissimilarity object inheriting from class `dist` or coercible to one. If not specified, `dmatrix` must be.
`dmatrix`	a symmetric dissimilarity matrix ($n \times n$), specified instead of `dist`, which can be more efficient.
`full`	logical specifying if a *full* silhouette should be computed for `clara` object. Note that this requires $O(n^2)$ memory, since the full dissimilarity (see `daisy`) is needed internally.
`object`	an object of class `silhouette`.
`...`	further arguments passed to and from methods.

FUN	function used to summarize silhouette widths.
nmax.lab	integer indicating the number of labels which is considered too large for single-name labeling the silhouette plot.
max.strlen	positive integer giving the length to which strings are truncated in silhouette plot labeling.
main, sub, xlab	arguments to title; have a sensible non-NULL default here.
col, border, cex.names	arguments passed barplot(); note that the default used to be col = heat.colors(n), border = par("fg") instead. col can also be a color vector of length k for clusterwise coloring, see also do.col.sort:
do.col.sort	logical indicating if the colors col should be sorted "along" the silhouette; this is useful for casewise or clusterwise coloring.
do.n.k	logical indicating if n and k "title text" should be written.
do.clus.stat	logical indicating if cluster size and averages should be written right to the silhouettes.

Details

For each observation i, the *silhouette width* $s(i)$ is defined as follows:
Put a(i) = average dissimilarity between i and all other points of the cluster to which i belongs (if i is the *only* observation in its cluster, $s(i) := 0$ without further calculations). For all *other* clusters C, put $d(i, C)$ = average dissimilarity of i to all observations of C. The smallest of these $d(i, C)$ is $b(i) := \min_C d(i, C)$, and can be seen as the dissimilarity between i and its "neighbor" cluster, i.e., the nearest one to which it does *not* belong. Finally,

$$s(i) := \frac{b(i) - a(i)}{max(a(i), b(i))}.$$

silhouette.default() is now based on C code donated by Romain Francois (the R version being still available as cluster:::silhouette.default.R).

Observations with a large $s(i)$ (almost 1) are very well clustered, a small $s(i)$ (around 0) means that the observation lies between two clusters, and observations with a negative $s(i)$ are probably placed in the wrong cluster.

Value

silhouette() returns an object, sil, of class silhouette which is an [n x 3] matrix with attributes. For each observation i, sil[i,] contains the cluster to which i belongs as well as the neighbor cluster of i (the cluster, not containing i, for which the average dissimilarity between its observations and i is minimal), and the silhouette width $s(i)$ of the observation. The colnames correspondingly are c("cluster", "neighbor", "sil_width").

summary(sil) returns an object of class summary.silhouette, a list with components

si.summary	numerical summary of the individual silhouette widths $s(i)$.
clus.avg.widths	numeric (rank 1) array of clusterwise *means* of silhouette widths where mean = FUN is used.
avg.width	the total mean FUN(s) where s are the individual silhouette widths.

`clus.sizes` `table` of the k cluster sizes.

`call` if available, the call creating `sil`.

`Ordered` logical identical to `attr(sil, "Ordered")`, see below.

`sortSilhouette(sil)` orders the rows of `sil` as in the silhouette plot, by cluster (increasingly) and decreasing silhouette width $s(i)$.
`attr(sil, "Ordered")` is a logical indicating if `sil` *is* ordered as by `sortSilhouette()`. In that case, `rownames(sil)` will contain case labels or numbers, and
`attr(sil, "iOrd")` the ordering index vector.

Note

While `silhouette()` is *intrinsic* to the `partition` clusterings, and hence has a (trivial) method for these, it is straightforward to get silhouettes from hierarchical clusterings from `silhouette.default()` with `cutree()` and distance as input.

By default, for `clara()` partitions, the silhouette is just for the best random *subset* used. Use `full = TRUE` to compute (and later possibly plot) the full silhouette.

References

Rousseeuw, P.J. (1987) Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *J. Comput. Appl. Math.*, **20**, 53–65.

chapter 2 of Kaufman, L. and Rousseeuw, P.J. (1990), see the references in `plot.agnes`.

See Also

`partition.object`, `plot.partition`.

Examples

```
data(ruspini)
pr4 <- pam(ruspini, 4)
str(si <- silhouette(pr4))
(ssi <- summary(si))
plot(si) # silhouette plot
plot(si, col = c("red", "green", "blue", "purple"))# with cluster-wise coloring

si2 <- silhouette(pr4$clustering, dist(ruspini, "canberra"))
summary(si2) # has small values: "canberra"'s fault
plot(si2, nmax= 80, cex.names=0.6)

op <- par(mfrow= c(3,2), oma= c(0,0, 3, 0),
          mgp= c(1.6,.8,0), mar= .1+c(4,2,2,2))
for(k in 2:6)
   plot(silhouette(pam(ruspini, k=k)), main = paste("k = ",k), do.n.k=FALSE)
mtext("PAM(Ruspini) as in Kaufman & Rousseeuw, p.101",
      outer = TRUE, font = par("font.main"), cex = par("cex.main"))
par(op)

## clara(): standard silhouette is just for the best random subset
data(xclara)
set.seed(7)
str(xc1k <- xclara[sample(nrow(xclara), size = 1000) ,])
```

```
cl3 <- clara(xc1k, 3)
plot(silhouette(cl3))# only of the "best" subset of 46
## The full silhouette: internally needs large (36 MB) dist object:
sf <- silhouette(cl3, full = TRUE) ## this is the same as
s.full <- silhouette(cl3$clustering, daisy(xc1k))
if(paste(R.version$major, R.version$minor, sep=".") >= "2.3.0")
   stopifnot(all.equal(sf, s.full, check.attributes = FALSE, tol = 0))
## color dependent on original "3 groups of each 1000":
plot(sf, col = 2+ as.integer(names(cl3$clustering) ) %/% 1000,
     main ="plot(silhouette(clara(.), full = TRUE))")

## Silhouette for a hierarchical clustering:
ar <- agnes(ruspini)
si3 <- silhouette(cutree(ar, k = 5), # k = 4 gave the same as pam() above
                  daisy(ruspini))
plot(si3, nmax = 80, cex.names = 0.5)
## 2 groups: Agnes() wasn't too good:
si4 <- silhouette(cutree(ar, k = 2), daisy(ruspini))
plot(si4, nmax = 80, cex.names = 0.5)
```

`sizeDiss` *Sample Size of Dissimilarity Like Object*

Description

Returns the number of observations (*sample size*) corresponding to a dissimilarity like object, or equivalently, the number of rows or columns of a matrix when only the lower or upper triangular part (without diagonal) is given.

It is nothing else but the inverse function of $f(n) = n(n-1)/2$.

Usage

```
sizeDiss(d)
```

Arguments

`d`	any R object with length (typically) $n(n-1)/2$.

Value

a number; n if `length(d) == n(n-1)/2`, `NA` otherwise.

See Also

`dissimilarity.object` and also `as.dist` for class `dissimilarity` and `dist` objects which have a `Size` attribute.

Examples

```
sizeDiss(1:10)# 5, since 10 == 5 * (5 - 1) / 2
sizeDiss(1:9) # NA

n <- 1:100
stopifnot(n == sapply( n*(n-1)/2, function(n) sizeDiss(logical(n))))
```

summary.agnes *Summary Method for 'agnes' Objects*

Description

Returns (and prints) a summary list for an agnes object. Printing gives more output than the corresponding print.agnes method.

Usage

```
## S3 method for class 'agnes':
summary(object, ...)
## S3 method for class 'summary.agnes':
print(x, ...)
```

Arguments

x, object	a agnes object.
...	potential further arguments (require by generic).

See Also

agnes, agnes.object.

Examples

```
data(agriculture)
summary(agnes(agriculture))
```

summary.clara *Summary Method for 'clara' Objects*

Description

Returns (and prints) a summary list for a clara object. Printing gives more output than the corresponding print.clara method.

Usage

```
## S3 method for class 'clara':
summary(object, ...)
## S3 method for class 'summary.clara':
print(x, ...)
```

Arguments

x, object	a clara object.
...	potential further arguments (require by generic).

See Also

clara.object

Examples

```
## generate 2000 objects, divided into 5 clusters.
set.seed(47)
x <- rbind(cbind(rnorm(400, 0,4), rnorm(400, 0,4)),
           cbind(rnorm(400,10,8), rnorm(400,40,6)),
           cbind(rnorm(400,30,4), rnorm(400, 0,4)),
           cbind(rnorm(400,40,4), rnorm(400,20,2)),
           cbind(rnorm(400,50,4), rnorm(400,50,4))
)
clx5 <- clara(x, 5)
## Mis`classification' table:
table(rep(1:5, rep(400,5)), clx5$clust) # -> 1 "error"
summary(clx5)

## Graphically:
par(mfrow = c(3,1), mgp = c(1.5, 0.6, 0), mar = par("mar") - c(0,0,2,0))
plot(x, col = rep(2:6, rep(400,5)))
plot(clx5)
```

---

summary.diana *Summary Method for 'diana' Objects*

Description

Returns (and prints) a summary list for a diana object.

Usage

```
## S3 method for class 'diana':
summary(object, ...)
## S3 method for class 'summary.diana':
print(x, ...)
```

Arguments

x, object	a diana object.
...	potential further arguments (require by generic).

See Also

diana, diana.object.

summary.mona *Summary Method for 'mona' Objects*

Description

Returns (and prints) a summary list for a mona object.

Usage

```
## S3 method for class 'mona':
summary(object, ...)
## S3 method for class 'summary.mona':
print(x, ...)
```

Arguments

x, object	a mona object.
...	potential further arguments (require by generic).

See Also

mona, mona.object.

summary.pam *Summary Method for PAM Objects*

Description

Summarize a pam object and return an object of class summary.pam. There's a print method for the latter.

Usage

```
## S3 method for class 'pam':
summary(object, ...)
## S3 method for class 'summary.pam':
print(x, ...)
```

Arguments

x, object	a pam object.
...	potential further arguments (require by generic).

See Also

pam, pam.object.

twins.object *Hierarchical Clustering Object*

Description

The objects of class "twins" represent an agglomerative or divisive (polythetic) hierarchical clustering of a dataset.

Value

See agnes.object and diana.object for details.

GENERATION

This class of objects is returned from agnes or diana.

METHODS

The "twins" class has a method for the following generic function: pltree.

INHERITANCE

The following classes inherit from class "twins" : "agnes" and "diana".

See Also

agnes,diana.

volume.ellipsoid *Compute the Volume of Planar Object*

Description

Compute the volume of a planar object. This is a generic function and a method for ellipsoid objects.

Usage

```
## S3 method for class 'ellipsoid':
volume(object)
```

Arguments

object	an R object the volume of which is wanted; for the ellipsoid method, an object of that class (see ellipsoidhull or the example below).

Value

a number, the volume of the given object.

See Also

ellipsoidhull for spanning ellipsoid computation.

Examples

```
## example(ellipsoidhull) # which defines `ellipsoid' object <namefoo>

myEl <- structure(list(cov = rbind(c(3,1),1:2), loc = c(0,0), d2 = 10),
                  class = "ellipsoid")
volume(myEl)# i.e. "area" here (d = 2)
myEl # also mentions the "volume"
```

---

votes.repub *Votes for Republican Candidate in Presidential Elections*

Description

A data frame with the percents of votes given to the republican candidate in presidential elections from 1856 to 1976. Rows represent the 50 states, and columns the 31 elections.

Usage

```
data(votes.repub)
```

Source

S. Peterson (1973): *A Statistical History of the American Presidential Elections*. New York: Frederick Ungar Publishing Co.

Data from 1964 to 1976 is from R. M. Scammon, *American Votes 12*, Congressional Quarterly.

---

xclara *Bivariate Data Set with 3 Clusters*

Description

An artificial data set consisting of 3000 points in 3 well-separated clusters of size 1000 each.

Usage

```
data(xclara)
```

Format

A data frame with 3000 observations on 2 numeric variables giving the x and y coordinates of the points, respectively.

Source

Sample data set accompanying the reference below, obtained from http://www.stat.ucla.edu/journals/jss/v01/i04/data/.

References

Anja Struyf, Mia Hubert & Peter J. Rousseeuw (1996): Clustering in an Object-Oriented Environment. *Journal of Statistical Software*, **1**. http://www.stat.ucla.edu/journals/jss/

Chapter 15

The `foreign` package

`lookup.xport` *Lookup Information on a SAS XPORT Format Library*

Description

Scans a file as a SAS XPORT format library and returns a list containing information about the SAS library.

Usage

```
lookup.xport(file)
```

Arguments

`file`	character variable with the name of the file to read. The file must be in SAS XPORT format.

Value

A list with one component for each dataset in the XPORT format library.

Author(s)

Saikat DebRoy

References

SAS Technical Support document TS-140: "The Record Layout of a Data Set in SAS Transport (XPORT) Format" available as http://ftp.sas.com/techsup/download/technote/ts140.html.

See Also

`read.xport`

Examples

```
## Not run:
lookup.xport("transport")
## End(Not run)
```

---

read.dbf *Read a DBF File*

Description

The function reads a DBF file into a data frame, converting character fields to factors, and trying to respect NULL fields.

Usage

```
read.dbf(file, as.is = FALSE)
```

Arguments

file	name of input file
as.is	should character vectors not be converted to factors?

Details

DBF is the extension used for files written for the 'XBASE' family of database languages, 'covering the dBase, Clipper, FoxPro, and their Windows equivalents Visual dBase, Visual Objects, and Visual FoxPro, plus some older products' (http://www.clicketyclick.dk/databases/xbase/format/). Most of these follow the file structure used by Ashton-Tate's dBase II, III or 4 (later owned by Borland).

read.dbf is based on C code from http://shapelib.maptools.org/ which implements the 'XBASE' specification. It can convert fields of type "L" (logical), "N" and "F" (numeric and float) and "D" (dates): all other field types are read as-is as character vectors. A numeric field are read as an R integer vector if it is encoded to have no decimals, otherwise as a numeric vector. However, if the numbers are too large to fit into an integer vector, it is changed to numeric. Note that is possible to read integers that cannot be represented exactly even as doubles: this sometimes occurs if IDs are incorrectly coded as numeric.

Value

A data frame of data from the DBF file; note that the field names are adjusted to use in R using make.names(unique=TRUE).

There is an attribute "data_type" giving the single-character dBase types for each field.

Author(s)

Nicholas Lewin-Koh and Roger Bivand; shapelib by Frank Warmerdam

References

http://shapelib.maptools.org/.

The Borland file specification *via* http://www.wotsit.org/search.asp?s=database.

See Also

write.dbf

Examples

```
x <- read.dbf(system.file("files/sids.dbf", package="foreign")[1])
str(x)
summary(x)
```

`read.dta` *Read Stata Binary Files*

Description

Reads a file in Stata version 5-9 binary format into a data frame.

Usage

```
read.dta(file, convert.dates = TRUE, tz = NULL,
         convert.factors = TRUE, missing.type = FALSE,
         convert.underscore=TRUE, warn.missing.labels=TRUE)
```

Arguments

`file`	a filename as a character string.
`convert.dates`	Convert Stata dates to `Date` class?
`tz`	Previously used to specify time zone. Now deprecated.
`convert.factors`	Use Stata value labels to create factors? (version 6.0 or later).
`missing.type`	For version 8,9 only, store information about different types of missing data?
`convert.underscore`	Convert `"_"` in Stata variable names to `"."` in R names?
`warn.missing.labels`	Warn if a variable is specified with value labels and those value labels are not present in the file.

Details

The variables in the Stata data set become the columns of the data frame. Missing values are correctly handled. The data label, variable labels, and timestamp are stored as attributes of the data frame. Nothing is done with variable characteristics.

Optionally, Stata dates (%d formats) are converted to R's `Date` class and variables with Stata value labels are converted to factors. Ordinarily, `read.dta` will not convert a variable to a factor unless a label is present for every level. Use `convert.factors=NA` to override this. In any case the value label and format information is stored as attributes on the returned data frame.

Stata 8.0 introduced a system of 27 different missing data values. If `missing.type` is `TRUE` a separate list is created with the same variable names as the loaded data. For string variables the list value is `NULL`. For other variables the value is `NA` where the observation is not missing and 0-26

when the observation is missing. This is attached as the code"missing" attribute of the returned value.

The option to allow underscores in variable names may become the default in future versions now that R supports their use.

Value

a data frame

Author(s)

Thomas Lumley

References

Stata Users Manual (versions 5 & 6), Programming manual (version 7), or online help (version 8,9) describe the format of the files.

See Also

`write.dta`, `attributes` `Date` `factor`

Examples

```
data(swiss)
write.dta(swiss,swissfile<-tempfile())
read.dta(swissfile)
```

---

read.epiinfo *Read Epi Info Data Files*

Description

Reads data files in the `.REC` format used by Epi Info versions 6 and earlier and by EpiData. Epi Info is a public domain database and statistics package produced by the US Centers for Disease Control and EpiData is a freely available data entry and validation system.

Usage

```
read.epiinfo(file, read.deleted = FALSE, guess.broken.dates = FALSE,
             thisyear = NULL, lower.case.names = FALSE)
```

Arguments

`file`	A filename, URL, or connnection.
`read.deleted`	Deleted records are read if `TRUE`, omitted if `FALSE` or replaced with `NA` if `NA`.
`guess.broken.dates`	Attempt to convert dates with 0 or 2 digit year information (see Details below).
`thisyear`	A 4-digit year to use for dates with no year. Defaults to the current year.
`lower.case.names`	Convert variable names to lowercase?

Details

Epi Info allows dates to be specified with no year or with a 2 or 4 digits. Dates with four-digit years are always converted to `Date` class. With the `guess.broken.dates` option the function will attempt to convert two-digit years using the operating system's default method (see Date) and will use the current year or the `thisyear` argument for dates with no year information.

If `read.deleted` is `TRUE` the `"deleted"` attribute of the data frame indicates the deleted records.

Value

A data frame.

Note

Epi Info 2000, the current version, uses the Microsoft Access file format to store data. This may be readable with the `RODBC` or `RDCOM` packages.

References

http://www.cdc.gov/epiinfo/, http://www.epidata.dk

See Also

DateTimeClasses

Examples

```
## Not run:
data<-read.epiinfo("oswego.rec",guess.broken.dates=TRUE,thisyear="1972")
## End(Not run)
```

`read.mtp` *Read a Minitab Portable Worksheet*

Description

Return a list with the data stored in a file as a Minitab Portable Worksheet.

Usage

```
read.mtp(file)
```

Arguments

`file`	character variable with the name of the file to read. The file must be in Minitab Portable Worksheet format.

Value

A list with one component for each column, matrix, or constant stored in the Minitab worksheet.

Author(s)

Douglas M. Bates

References

http://www.minitab.com/

Examples

```
## Not run:
read.mtp("ex1-10.mtp")
## End(Not run)
```

---

read.octave *Read Octave Text Data Files*

Description

Read a file in Octave text data format into a list.

Usage

```
read.octave(file)
```

Arguments

file	a character string with the name of the file to read.

Details

This function is used to read in files in Octave text data format, as created by `save -ascii` in Octave. It knows about most of the common types of variables, including the standard atomic (real and complex scalars, matrices, and N-d arrays, strings, ranges, and boolean scalars and matrices) and recursive (structs, cells, and lists) ones, but has no guarantee to read all types. If a type is not recognized, a warning indicating the unknown type is issued, it is attempted to skip the unknown entry, and `NULL` is used as its value. Note that this will give incorrect results, and maybe even errors, in the case of unknown recursive data types.

As Octave can read MATLAB binary files, one can make the contents of such files available to R by using Octave's load and save (as text) facilities as an intermediary step.

Value

A list with one named component for each variable in the file.

Author(s)

Stephen Eglen ⟨stephen@gnu.org⟩ and Kurt Hornik

References

http://www.octave.org/

`read.spss` *Read an SPSS Data File*

Description

`read.spss` reads a file stored by the SPSS `save` and `export` commands and returns a list.

Usage

```
read.spss(file, use.value.labels=TRUE, to.data.frame=FALSE,
          max.value.labels=Inf, trim.factor.names=FALSE)
```

Arguments

`file`	character variable with the name of the file to read.
`use.value.labels`	Convert variables with value labels into R factors with those levels?
`to.data.frame`	return a data frame?
`max.value.labels`	Only variables with at most this many unique values will be converted to factors
`trim.factor.names`	Trim trailing spaces from factor levels?

Details

This uses modified code from the PSPP project for reading the SPSS formats.

Occasionally in SPSS value labels will be added to some values of a continuous variable (eg to distinguish different types of missing data), and you will not want these variables converted to factors. By setting `max.val.labels` you can specify that variables with a large number of distinct values are not converted to factors even if they have value labels. In addition, variables will not be converted to factors if there are non-missing values that have no value label. The value labels are then returned in the `"value.labels"` attribute of the variable.

If SPSS variable labels are present, they are returned as the `"variable.labels"` attribute of the answer.

Fixed length strings (including value labels) are padded on the right with spaces by SPSS, and so are read that way by R. See the examples for `sub` for ways to remove trailing spaces in string data.

Value

A list (or data frame) with one component for each variable in the saved data set.

Note

If SPSS value labels are converted to factors the underlying numerical codes will not in general be the same as the SPSS numerical values, since the numerical codes in R are always 1,2,3,...

Author(s)

Saikat DebRoy

Examples

```
## Not run:
read.spss("datafile")
## don't convert value labels to factor levels
read.spss("datafile",use.value.labels=FALSE)
## convert value labels to factors for variables with at most
## ten distinct values.
read.spss("datafile",max.val.labels=10)
## End(Not run)
```

---

read.ssd *Obtain a Data Frame from a SAS Permanent Dataset, via read.xport*

Description

Generates a SAS program to convert the ssd contents to SAS transport format and then uses `read.xport` to obtain a dataframe.

Usage

```
read.ssd(libname, sectionnames,
   tmpXport=tempfile(), tmpProgLoc=tempfile(), sascmd="sas")
```

Arguments

`libname`	character string defining the SAS library (usually a directory reference)
`sectionnames`	character vector giving member names. These are files in the `libname` directory. They will usually have a `.ssd0x` or `.sas7bdat` extension, which should be omitted.
`tmpXport`	character string: location where temporary xport format archive should reside – defaults to a randomly named file in the session temporary directory, which will be removed.
`tmpProgLoc`	character string: location where temporary conversion SAS program should reside – defaults to a randomly named file in session temporary directory, which will be removed on successful operation.
`sascmd`	character string giving full path to SAS executable.

Details

Creates a SAS program and runs it.

Value

A data frame if all goes well, or NULL with warnings and some enduring side effects (log file for auditing)

Note

error handling is primitive

Author(s)

For Unix: VJ Carey ⟨<stvjc@channing.harvard.edu>⟩

See Also

read.xport

Examples

```
## if there were some files on the web we could get a real
## runnable example
## Not run:
R> list.files("trialdata")
 [1] "baseline.sas7bdat" "form11.sas7bdat"   "form12.sas7bdat"
 [4] "form13.sas7bdat"   "form22.sas7bdat"   "form23.sas7bdat"
 [7] "form3.sas7bdat"    "form4.sas7bdat"    "form48.sas7bdat"
[10] "form50.sas7bdat"   "form51.sas7bdat"   "form71.sas7bdat"
[13] "form72.sas7bdat"   "form8.sas7bdat"    "form9.sas7bdat"
[16] "form90.sas7bdat"   "form91.sas7bdat"
R> baseline<-read.ssd("trialdata","baseline")
R> form90<-read.ssd("trialdata","form90")

## Or for a Windows example
sashome <- "/Program Files/SAS/SAS 9.1"
read.ssd(file.path(sashome, "core", "sashelp"), "retail",
         sascmd = file.path(sashome, "sas.exe"))
## End(Not run)
```

---

read.systat *Obtain a Data Frame from a Systat File*

Description

read.systat reads a rectangular data file stored by the Systat SAVE command as (legacy) *.sys or more recently *.syd files.

Usage

```
read.systat(file, to.data.frame = TRUE)
```

Arguments

file	character variable with the name of the file to read
to.data.frame	return a data frame (otherwise a list)

Details

The function only reads those Systat files that are rectangular data files (mtype = 1), and warns when files have non-standard variable name codings. The files tested were produced on MS-DOS and Windows: files for the Mac version of Systat have a completely different format.

The C code was originally written for an add-on module for Systat described in Bivand (1992 paper). Variable names retain the trailing dollar in the list returned when to.data.frame is FALSE, and

in that case character variables are returned as is and filled up to 12 characters with blanks on the right. The original function was limited to reading Systat files with up to 256 variables (a Systat limitation); it will now read up to 8192 variables.

If there is a user comment in the header this is returned as attribute `"comment"`. Such comments are always a multiple of 72 characters (with a maximum of 720 chars returned), normally padded with trailing spaces.

Value

A data frame (or list) with one component for each variable in the saved data set.

Author(s)

Roger Bivand

References

Systat Manual, 1987, 1989

Bivand, R. S. (1992) SYSTAT-compatible software for modelling spatial dependence among observations. *Computers and Geosciences* **18**, 951–963.

Examples

```
summary(iris)
iris.s <- read.systat(system.file("files/Iris.syd", package="foreign")[1])
str(iris.s)
summary(iris.s)
```

---

read.xport *Read a SAS XPORT Format Library*

Description

Reads a file as a SAS XPORT format library and returns a list of data.frames.

Usage

```
read.xport(file)
```

Arguments

`file`	character variable with the name of the file to read. The file must be in SAS XPORT format.

Value

If there is a more than one dataset in the XPORT format library, a named list of data frames, otherwise a data frame. The columns of the data frames will be either numeric (corresponding to numeric in SAS) or factor (corresponding to character in SAS). All SAS numeric missing values (including special missing values represented by `._`, `.A` to `.Z` by SAS) are mapped to R `NA`.

Trailing blanks are removed from character columns before conversion to a factor. Some sources claim that character missing values in SAS are represented by `' '` or `""`: these are not treated as R missing values.

Author(s)

Saikat DebRoy ⟨<saikat@stat.wisc.edu>⟩

References

SAS Technical Support document TS-140: "The Record Layout of a Data Set in SAS Transport (XPORT) Format" available at http://ftp.sas.com/techsup/download/technote/ts140.html.

See Also

`lookup.xport`

Examples

```
## Not run:
read.xport("transport")
## End(Not run)
```

S3 read functions *Read an S3 Binary or data.dump File*

Description

Reads binary data files or `data.dump` files that were produced in S version 3.

Usage

```
data.restore(file, print = FALSE, verbose = FALSE, env = .GlobalEnv)
read.S(file)
```

Arguments

`file`	the filename of the S-PLUS `data.dump` or binary file.
`print`	whether to print the name of each object as read from the file.
`verbose`	whether to print the name of every subitem within each object.
`env`	environment within which to create the restored object(s).

Details

`read.S` can read the binary files produced in some older versions of S-PLUS on either Windows (versions 3.x, 4.x, 2000) or Unix (version 3.x with 4 byte integers). It automatically detects whether the file was produced on a big- or little-endian machine and adapts itself accordingly.

`data.restore` can read a similar range of files produced by `data.dump` and for newer versions of S-PLUS, those from `data.dump(....., oldStyle=TRUE)`.

Not all S3 objects can be handled in the current version. The most frequently encountered exceptions are functions and expressions; you will also have trouble with objects that contain model formulas. In particular, comments will be lost from function bodies, and the argument lists of functions will often be changed.

Value

for `read.S`, an R version of the S3 object.

for `data.restore`, the name of the file.

Author(s)

Duncan Murdoch

Examples

```
## Not run:
read.S(file.path("_Data", "myobj"))
data.restore("dumpdata", print = TRUE)
## End(Not run)
```

---

write.dbf *Write a DBF File*

Description

The function tries to write a data frame to a DBF file.

Usage

```
write.dbf(dataframe, file, factor2char = TRUE, max_nchar = 254)
```

Arguments

`dataframe`	a data frame object.
`file`	a file name to be written to.
`factor2char`	logical, default `TRUE`, convert factor columns to character: otherwise they are written as the internal integer codes.
`max_nchar`	The maximum number of characters allowed in a character field. Strings which exceed this will be truncated with a warning. See Details.

Details

Dots in column names are replaced by underlines in the DBF file, and names are truncated to 11 characters.

Only vector columns of classes `"logical"`, `"numeric"`, `"integer"`, `"character"`, `"factor"` and `"Date"` can be written. Other columns should be converted to one of these.

Maximum precision (number of digits including minus sign and decimal sign) for numeric is 19 - scale (digits after the decimal sign) which is calculated internally based on the number of digits before the decimal sign.

The original DBASE format limited character fields to 254 bytes. It is said that Clipper and FoxPro can read up to 32K, and it is possible to write a reader that could accept up to 65535 bytes. (The documentation suggests that only ASCII characters can be assumed to be supported.) Readers expecting the older standard (which includes Excel 2003, Access 2003 and OpenOffice 2.0) will truncate the field to the maximum width modulo 256, so increase `max_nchar` only if you are sure the intended reader supports wider character fields.

Value

Invisible `NULL`.

Note

Other applications have varying abilities to read the data types used here. Microsoft Access reads `"numeric"`, `"integer"`, `"character"` and `"Date"` fields, including recognizing missing values, but not `"logical"` (read as `0,-1`). Microsoft Excel understood all possible types but did not interpret missing values in character fields correctly (showing them as character nuls).

Author(s)

Nicholas J. Lewin-Koh, modified by Roger Bivand and Brian Ripley; shapelib by Frank Warmerdam.

References

http://shapelib.maptools.org/

http://www.clicketyclick.dk/databases/xbase/format/data_types.html

See Also

`read.dbf`

Examples

```
str(warpbreaks)
try1 <- paste(tempfile(), ".dbf", sep = "")
write.dbf(warpbreaks, try1, factor2char = FALSE)
in1 <- read.dbf(try1)
str(in1)
try2 <- paste(tempfile(), ".dbf", sep = "")
write.dbf(warpbreaks, try2, factor2char = TRUE)
in2 <- read.dbf(try2)
str(in2)
unlink(c(try1, try2))
```

write.dta *Write Files in Stata Binary Format*

Description

Writes the data frame to file in the Stata version 6.0 or 7.0 binary format. Does not write matrix variables.

Usage

```
write.dta(dataframe, file, version = 6,
          convert.dates = TRUE, tz = "GMT",
          convert.factors = c("labels", "string", "numeric", "codes"))
```

Arguments

dataframe	a data frame.
file	character string giving filename.
version	Stata version: 6 and 7 are supported.
convert.dates	Convert POSIXt objects to Stata dates?
tz	timezone for date conversion
convert.factors	how to handle factors

Details

The major differences between Stata versions is that 7.0 allows 32-character variable names. The abbreviate function is used to trim long variables to the permitted length. A warning is given if this is needed and it is an error for the abbreviated names not to be unique.

The columns in the data frame become variables in the Stata data set. Missing values are correctly handled. Optionally, R date/time objects (POSIXt classes) are converted into the Stata format. This loses information – Stata dates are in days since 1960-1-1. POSIXct objects can be written without conversion but will not be understood as dates by Stata; POSIXlt objects cannot be written without conversion.

There are four options for handling factors. The default is to use Stata value labels for the factor levels. With convert.factors="string", the factor levels are written as strings. With convert.factors="numeric" the numeric values of the levels are written, or NA if they cannot be coerced to numeric. Finally, convert.factors="codes" writes the underlying integer codes of the factors. This last used to be the only available method and is provided largely for backwards compatibility.

For Stata 8 or 9 use version=7 – the only advantage of Stata 8 format is that it can represent multiple different missing value types, and R doesn't have them.

Value

NULL

Author(s)

Thomas Lumley

References

Stata 6.0 Users Manual, Stata 7.0 Programming manual, Stata 8.0, 9.0 online help describe the file formats.

See Also

read.dta, attributes, DateTimeClasses, abbreviate

Examples

```
write.dta(swiss, swissfile <- tempfile())
read.dta(swissfile)
```

---

write.foreign *Write Text Files and Code to Read Them*

Description

This function exports simple data frames to other statistical packages by writing the data as free-format text and writing a separate file of instructions for the other package to read the data.

Usage

```
write.foreign(df, datafile, codefile,
              package = c("SPSS", "Stata", "SAS"), ...)
```

Arguments

df	A data frame
datafile	Name of file for data output
codefile	Name of file for code output
package	Name of package
...	Other arguments for the individual writeForeign functions

Details

The work for this function is done by foreign:::writeForeignStata, foreign:::writeForeignSAS and foreign:::writeForeignSPSS. To add support for another package, eg Systat, create a function writeForeignSystat with the same first three arguments as write.foreign. This will be called from write.foreign when package="Systat".

Numeric variables and factors are supported for all packages, dates and times (Date, dates, date, and POSIXt classes) are also supported for SAS and characters are supported for SPSS.

For package="SAS" there are optional arguments dataname="rdata" taking a string that will be the SAS data set name and validvarname taking either "V6" or "V7".

Value

None

Author(s)

Thomas Lumley and Stephen Weigand

Examples

```
## Not run:
datafile<-tempfile()
codefile<-tempfile()
write.foreign(esoph,datafile,codefile,package="SPSS")
file.show(datafile)
file.show(codefile)
unlink(datafile)
unlink(codefile)
## End(Not run)
```

Chapter 16

The `lattice` package

`axis.default` *Default axis annotation utilities*

Description

Lattice funtions provide control over how the plot axes are annotated through a common interface. There are two levels of control. The `xscale.components` and `yscale.components` arguments can be functions that determine tick mark locations and labels given a packet. For more direct control, the `axis` argument can be a function that actually draws the axes. The functions documented here are the defaults for these arguments. They can additonally be used as components of user written replacements.

Usage

```
xscale.components.default(lim,
                          packet.number = 0,
                          packet.list = NULL,
                          top = TRUE,
                          ...)
yscale.components.default(lim,
                          packet.number = 0,
                          packet.list = NULL,
                          right = TRUE,
                          ...)
axis.default(side = c("top", "bottom", "left", "right"),
             scales, components, as.table,
             labels = c("default", "yes", "no"),
             ticks = c("default", "yes", "no"),
             ...)
```

Arguments

`lim`	the range of the data in that packet (data subset corresponding to a combination of levels of the conditioning variable). The range is not necessarily numeric; e.g. for factors, they could be character vectors representing levels, and for the

various date-time representations, they could be vectors of length 2 with the corresponding class.

`packet.number`
: which packet (counted according to the packet order, described in `print.trellis`) is being processed. In cases where all panels have the same limits, this function is called only once (rather than once for each packet), in which case this argument will have the value `0`.

`packet.list`
: list, as long as the number of packets, giving all the actual packets. Specifically, each component is the list of arguments given to the panel function when and if that packet is drawn in a panel. (This has not yet been implemented.)

`top, right`
: the value of the `top` and `right` components of the result, as appropriate. See below for interpretation.

`side`
: on which side the axis is to be drawn. The usual partial matching rules apply.

`scales`
: the appropriate component of the `scales` argument supplied to the high level function, suitably standardized.

`components`
: list, similar to those produced by `xscale.components.default` and `yscale.components.default`.

`as.table`
: the `as.table` argument in the high level function.

`labels`
: whether labels are to be drawn. By default, the rules determined by `scales` are used.

`ticks`
: whether labels are to be drawn. By default, the rules determined by `scales` are used.

`...`
: many other arguments may be supplied, and are passed on to other internal functions.

Details

These functions are part of a new API introduced in lattice 0.14 to provide the user more control over how axis annotation is done. While the API has been designed in anticipation of use that was previously unsupported, the implementation has initially focused on reproducing existing capabilities, rather then test new features. At the time of writing, several features are unimplemented. If you require them, please contact the maintainer.

Value

`xscale.components.default` and `yscale.components.default` return lists in a specific format expected by `axis.default`. The exact form will be documented later.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

`Lattice`, `xyplot`, `print.trellis`

Examples

```
str(xscale.components.default(c(0, 1)))

set.seed(36872)
rln <- rlnorm(100)

densityplot(rln,
            scales = list(x = list(log = 2), alternating = 3),
            xlab = "Simulated lognormal variates",
            xscale.components = function(...) {
                ans <- xscale.components.default(...)
                ans$top <- ans$bottom
                ans$bottom$labels$labels <- parse(text = ans$bottom$labels$labels)
                ans$top$labels$labels <-
                    if (require(MASS))
                        fractions(2^(ans$top$labels$at))
                    else
                        2^(ans$top$labels$at)
                ans
            })
```

---

banking *Banking*

Description

Calculates banking slope

Usage

```
banking(dx, dy)
```

Arguments

dx, dy	vector of consecutive x, y differences.

Details

`banking` is a crude banking function used when `aspect="xy"` in high level Trellis functions. Its usually not very menaingful except for `xyplot`. It just orders the absolute slopes and returns a value which when adjusted by the panel scale limits will make the median of the above absolute slopes a 45 degree line.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

Lattice, xyplot

`barchart.table` *table methods for barchart and dotplot*

Description

Contingency tables are often displayed using barcharts and dotplots. These methods are provided for convenience and operate directly on tables. Arrays and matrices are simply coerced to be a table.

Usage

```
## S3 method for class 'table':
barchart(x, data, groups = TRUE,
         origin = 0, stack = TRUE, ...)

## S3 method for class 'array':
barchart(x, data, ...)

## S3 method for class 'matrix':
barchart(x, data, ...)

## S3 method for class 'table':
dotplot(x, data, groups = TRUE, ...)

## S3 method for class 'array':
dotplot(x, data, ...)

## S3 method for class 'matrix':
dotplot(x, data, ...)
```

Arguments

`x`	a `table`, `array` or `matrix` object.
`data`	should not be specified. If specified, will be ignored with a warning.
`groups`	logical, whether to use the last dimension as the grouping variable in the display.
`origin, stack`	arguments to `panel.barchart` controlling the display. The defaults for the `table` method are diffrent.
`...`	other arguments, passed to the underlying `formula` method.

Details

The first dimension is used as the variable on the vertical axis. The last dimension is optionally used as a grouping variable (to produce stacked barcharts by default). All other dimensions are used as conditioning variables. The order of these variables cannot be altered (except by permuting the original argument using `t` or `aperm`). For more flexibility, use the formula method after converting the table to a data frame using the relevant `as.data.frame` method.

Value

An object of class `"trellis"`. The `update` method can be used to update components of the object and the `print` method (usually called by default) will plot it on an appropriate plotting device.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

barchart, t, aperm, table, panel.barchart, Lattice

Examples

```
barchart(Titanic, scales = list(x = "free"),
         auto.key = list(title = "Survived"))
```

---

barley *Yield data from a Minnesota barley trial*

Description

Total yield in bushels per acre for 10 varieties at 6 sites in each of two years.

Usage

```
barley
```

Format

A data frame with 120 observations on the following 4 variables.

yield Yield (averaged across three blocks) in bushels/acre.

variety Factor with levels `"Svansota"`, `"No. 462"`, `"Manchuria"`, `"No. 475"`, `"Velvet"`, `"Peatland"`, `"Glabron"`, `"No. 457"`, `"Wisconsin No. 38"`, `"Trebi"`.

year Factor with levels `1932`, `1931`

site Factor with 6 levels: `"Grand Rapids"`, `"Duluth"`, `"University Farm"`, `"Morris"`, `"Crookston"`, `"Waseca"`

Details

These data are yields in bushels per acre, of 10 varieties of barley grown in 1/40 acre plots at University Farm, St. Paul, and at the five branch experiment stations located at Waseca, Morris, Crookston, Grand Rapids, and Duluth (all in Minnesota). The varieties were grown in three randomized blocks at each of the six stations during 1931 and 1932, different land being used each year of the test.

Immer et al. (1934) present the data for each Year*Site*Variety*Block. The data here is the average yield across the three blocks.

Immer et al. (1934) refer (once) to the experiment as being conducted in 1930 and 1931, then later refer to it (repeatedly) as being conducted in 1931 and 1932. Later authors have continued the confusion.

Cleveland (1993) suggests that the data for the Morris site may have had the years switched.

Author(s)

Documentation contributed by Kevin Wright.

Source

Immer, R. F., H. K. Hayes, and LeRoy Powers. (1934). Statistical Determination of Barley Varietal Adaptation. *Journal of the American Society of Agronomy*, **26**, 403–419.

References

Cleveland, William S. (1993). *Visualizing Data*. Hobart Press, Summit, New Jersey.

Fisher, R. A. (1971). *The Design of Experiments*. Hafner, New York, 9th edition.

See Also

`immer` in the MASS package for data from the same experiment (expressed as total yield for 3 blocks) for a subset of varieties.

Examples

```
# Graphic suggesting the Morris data switched the years 1931 and 1932
# Figure 1.1 from Cleveland
dotplot(variety ~ yield | site, data = barley, groups = year,
        key = simpleKey(levels(barley$year), space = "right"),
        xlab = "Barley Yield (bushels/acre) ",
        aspect=0.5, layout = c(1,6), ylab=NULL)
```

---

cloud *3d Scatter Plot and Wireframe Surface Plot*

Description

Generic functions to draw 3d scatter plots and surfaces. The `"formula"` methods do most of the actual work.

Usage

```
cloud(x, data, ...)
wireframe(x, data, ...)

## S3 method for class 'formula':
cloud(x,
      data,
      allow.multiple,
      outer = FALSE,
      auto.key = FALSE,
      aspect = c(1,1),
      panel = "panel.cloud",
      prepanel = NULL,
      scales = list(),
      strip = TRUE,
      groups = NULL,
      xlab,
      xlim,
      ylab,
```

```
       ylim,
       zlab,
       zlim,
       at,
       drape = FALSE,
       pretty = FALSE,
       drop.unused.levels,
       ...,
       default.scales =
       list(distance = c(1, 1, 1),
            arrows = TRUE,
            axs = axs.default),
       colorkey,
       col.regions,
       alpha.regions,
       cuts = 70,
       subset = TRUE,
       axs.default = "r")

## S3 method for class 'formula':
wireframe(x,
          data,
          panel = "panel.wireframe",
          ...)

## S3 method for class 'matrix':
cloud(x, data = NULL, type = "h",
      zlab = deparse(substitute(x)), ...)
## S3 method for class 'matrix':
wireframe(x, data = NULL, zlab = deparse(substitute(x)), ...)
```

Arguments

x	The object on which method dispatch is carried out. For the "formula" methods, a formula of the form z ~ x * y | g1 * g2 * ..., where z is a numeric response, and x, y are numeric values. g1, g2, ..., if present, are conditioning variables used for conditioning, and must be either factors or shingles. In the case of wireframe, calculations are based on the assumption that the x and y values are evaluated on a rectangular grid defined by their unique values. The grid points need not be equally spaced. For wireframe, x, y and z may also be matrices (of the same dimension), in which case they are taken to represent a 3-D surface parametrized on a 2-D grid (e.g., a sphere). Conditioning is not possible with this feature. See details below. Missing values are allowed, either as NA values in the z vector, or missing rows in the data frame (note however that in that case the X and Y grids will be determined only by the available values). For a grouped display (producing multiple surfaces), missing rows are not allowed, but NA-s in z are. Both wireframe and cloud have methods for matrix objects, in which case x provides the z vector described above, while its rows and columns are interpreted as the x and y vectors respectively. This is similar to the form used in persp.

data — for the "formula" methods, an optional data frame in which variables in the formula (as well as groups and subset, if any) are to be evaluated. data should not be specified except when using the "formula" method.

allow.multiple, outer, auto.key, prepanel, strip, groups, xlab, xlim, ylab, ylim — These arguments are documented in the help page for xyplot.

type — type of display in cloud (see panel.3dscatter for details). Defaults to "h" for the matrix method.

aspect — vector of length 2, giving the relative aspects of the y-size/x-size and z-size/x-size of the enclosing cube.

panel — panel function used to create the display. See panel.cloud for (non-trivial) details.

scales — a list describing the scales. As with other high level functions (see xyplot for details), this list can contain parameters in name=value form. It can also contain components with the special names x, y and z, which can be similar lists with axis-specific values overriding the ones specified in scales.

The most common use for this argument is to set arrows=FALSE, which causes tick marks and labels to be used instead of arrows being drawn (the default). Both can be suppressed by draw=FALSE. Another special component is distance, which specifies the relative distance of the axis label from the bounding box. If specified as a component of scales (as opposed to one of scales$z etc), this can be (and is recycled if not) a vector of length 3, specifying distances for the x, y and z labels respectively.

Other components that work in the scales argument of xyplot etc. should also work here (as long as they make sense), including explicit specification of tick mark locations and labels. (Not everything is implemented yet, but if you find something that should work but does not, feel free to bug the maintainer.)

Note, however, that for these functions scales cannot contain information that is specific to particular panels. If you really need that, consider using the scales.3d argument of panel.cloud.

axs.default — Unlike 2-D display functions, cloud does not expand the bounding box to slightly beyound the range of the data, even though it should. This is primarily because this is the natural behaviour in wireframe, which uses the same code. axs.default is intended to provide a different default for cloud. However, this feature has not yet been implemented.

zlab — Specifies a label describing the z variable in ways similar to xlab and ylab (i.e. "grob", character string, expression or list) in other high level functions. Additionally, if zlab (and xlab and ylab) is a list, it can contain a component called rot, controlling the rotation for the label

zlim — limits for the z-axis. Similar to xlim and ylim in other high level functions

drape — logical, whether the wireframe is to be draped in color. If TRUE, the height of a facet is used to determine its color in a manner similar to the coloring scheme used in levelplot. Otherwise, the background color is used to color the facets. This argument is ignored if shade = TRUE (see panel.3dwire).

at, col.regions, alpha.regions — these arguments are analogous to those in levelplot. if drape=TRUE, at gives the vector of cutpoints where the colors change, and col.regions the vector of colors to be used in that case. alpha.regions determines the alpha-transparency on supporting devices. These are passed down to the panel function, and also used in the colorkey if appropriate. The default

for col.regions and alpha.regions is derived from the Trellis setting "regions"

cuts — if at is unspecified, the approximate number of cutpoints if drape=TRUE

pretty — whether automatic choice of cutpoints should be prettfied

colorkey — logical indicating whether a color key should be drawn alongside, or a list describing such a key. See levelplot for details.

... — Any number of other arguments can be specified, and are passed to the panel function. In particular, the arguments distance, perspective, screen and R.mat are very important in determining the 3-D display. The argument shade can be useful for wireframe calls, and controls shading of the rendered surface. These arguments are described in detail in the help page for panel.cloud.

Additionally, an argument called zoom may be specified, which should be a numeric scalar to be interpreted as a scale factor by which the projection is magnified. This can be useful to get the variable names into the plot. This argument is actually only used by the default prepanel function.

Details

These functions produce three dimensional plots in each panel (as long as the default panel functions are used). The orientation is obtained as follows: the data are scaled to fall within a bounding box that is contained in the [-0.5, 0.5] cube (even smaller for non-default values of aspect). The viewing direction is given by a sequence of rotations specified by the screen argument, starting from the positive Z-axis. The viewing point (camera) is located at a distance of 1/distance from the origin. If perspective=FALSE, distance is set to 0 (i.e., the viewing point is at an infinite distance).

cloud draws a 3-D Scatter Plot, while wireframe draws a 3-D surface (usually evaluated on a grid). Multiple surfaces can be drawn by wireframe using the groups argument (although this is of limited use because the display is incorrect when the surfaces intersect). Specifying groups with cloud results in a panel.superpose-like effect (via panel.3dscatter).

wireframe can optionally render the surface as being illuminated by a light source (no shadows though). Details can be found in the help page for panel.3dwire. Note that although arguments controlling these are actually arguments for the panel function, they can be supplied to cloud and wireframe directly.

For single panel plots, wireframe can also plot parametrized 3-D surfaces (i.e., functions of the form f(u,v) = (x(u,v), y(u,v), z(u,v)), where values of (u,v) lie on a rectangle. The simplest example of this sort of surface is a sphere parametrized by latitude and longitude. This can be achieved by calling wireframe with a formula x of the form z~x*y, where x, y and z are all matrices of the same dimension, representing the values of x(u,v), y(u,v) and z(u,v) evaluated on a discrete rectangular grid (the actual values of (u,v) are irrelevant).

When this feature is used, the heights used to calculate drape colors or shading colors are no longer the z values, but the distances of (x,y,z) from the origin.

Note that this feature does not work with groups, subscripts, subset, etc. Conditioning variables are also not supported in this case.

The algorithm for identifying which edges of the bounding box are 'behind' the points doesn't work in some extreme situations. Also, panel.cloud tries to figure out the optimal location of the arrows and axis labels automatically, but can fail on occasion (especially when the view is from 'below' the data). This can be manually controlled by the scpos argument in panel.cloud.

These and all other high level Trellis functions have several other arguments in common. These are extensively documented only in the help page for xyplot, which should be consulted to learn more detailed usage.

Value

An object of class "trellis". The update method can be used to update components of the object and the print method (usually called by default) will plot it on an appropriate plotting device.

Note

There is a known problem with grouped wireframe displays when the (x, y) coordinates represented in the data do not represent the full evaluation grid. The problem occurs whether the grouping is specified through the groups argument or through the formula interface, and currently causes memory access violations. Depending on the circumstances, this is manifested either as a meaningless plot or a crash. To work around the problem, it should be enough to have a row in the data frame for each grid point, with an NA response (z) in rows that were previously missing.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

Lattice for an overview of the package, as well as xyplot, levelplot, panel.cloud

Examples

```
## volcano  ## 87 x 61 matrix
wireframe(volcano, shade = TRUE,
          aspect = c(61/87, 0.4),
          light.source = c(10,0,10))

g <- expand.grid(x = 1:10, y = 5:15, gr = 1:2)
g$z <- log((g$x^g$g + g$y^2) * g$gr)
wireframe(z ~ x * y, data = g, groups = gr,
          scales = list(arrows = FALSE),
          drape = TRUE, colorkey = TRUE,
          screen = list(z = 30, x = -60))

cloud(Sepal.Length ~ Petal.Length * Petal.Width | Species, data = iris,
      screen = list(x = -90, y = 70), distance = .4, zoom = .6)

par.set <-
    list(axis.line = list(col = "transparent"), clip = list(panel = FALSE))
print(cloud(Sepal.Length ~ Petal.Length * Petal.Width,
            data = iris, cex = .8,
            groups = Species,
            main = "Stereo",
            screen = list(z = 20, x = -70, y = 3),
            par.settings = par.set),
      split = c(1,1,2,1), more = TRUE)
print(cloud(Sepal.Length ~ Petal.Length * Petal.Width,
            data = iris, cex = .8,
            groups = Species,
```

```
            main = "Stereo",
            screen = list(z = 20, x = -70, y = 0),
            par.settings = par.set),
      split = c(2,1,2,1))
```

`draw.colorkey` *Produce a Colorkey for levelplot*

Description

Produces (and possibly draws) a Grid frame grob which is a colorkey that can be placed in other Grid plots. Used in levelplot

Usage

```
draw.colorkey(key, draw=FALSE, vp=NULL)
```

Arguments

`key`	A list determining the key. See documentation for `levelplot`, in particular the section describing the `colorkey` argument, for details.
`draw`	logical, whether the grob is to be drawn.
`vp`	viewport

Value

A Grid frame object (that inherits from `"grob"`)

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

`xyplot`

`draw.key` *Produce a Legend or Key*

Description

Produces (and possibly draws) a Grid frame grob which is a legend (aka key) that can be placed in other Grid plots.

Usage

```
draw.key(key, draw=FALSE, vp=NULL)
```

Arguments

key	A list determining the key. See documentation for xyplot, in particular the section describing the key argument, for details.
draw	logical, whether the grob is to be drawn.
vp	viewport

Value

A Grid frame object (that inherits from 'grob').

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

xyplot

---

environmental *Atmospheric environmental conditions in New York City*

Description

Daily measurements of ozone concentration, wind speed, temperature and solar radiation in New York City from May to September of 1973.

Usage

```
environmental
```

Format

A data frame with 111 observations on the following 4 variables.

ozone Average ozone concentration (of hourly measurements) of in parts per billion.

radiation Solar radiation (from 08:00 to 12:00) in langleys.

temperature Maximum daily emperature in degrees Fahrenheit.

wind Average wind speed (at 07:00 and 10:00) in miles per hour.

Author(s)

Documentation contributed by Kevin Wright.

Source

Bruntz, S. M., W. S. Cleveland, B. Kleiner, and J. L. Warner. (1974). The Dependence of Ambient Ozone on Solar Radiation, Wind, Temperature, and Mixing Height. In *Symposium on Atmospheric Diffusion and Air Pollution*, pages 125–128. American Meterological Society, Boston.

References

Cleveland, William S. (1993). *Visualizing Data*. Hobart Press, Summit, New Jersey.

Examples

```
# Scatter plot matrix with loess lines
splom(~environmental,
  panel=function(x,y){
    panel.xyplot(x,y)
    panel.loess(x,y)
  }
)

# Conditioned plot similar to figure 5.3 from Cleveland
attach(environmental)
Temperature <- equal.count(temperature, 4, 1/2)
Wind <- equal.count(wind, 4, 1/2)
xyplot((ozone^(1/3)) ~ radiation | Temperature * Wind,
  aspect=1,
        prepanel = function(x, y)
                prepanel.loess(x, y, span = 1),
        panel = function(x, y){
                panel.grid(h = 2, v = 2)
                panel.xyplot(x, y, cex = .5)
                panel.loess(x, y, span = 1)
        },
        xlab = "Solar radiation (langleys)",
        ylab = "Ozone (cube root ppb)")
detach()

# Similar display using the coplot function
with(environmental,{
  coplot((ozone^.33) ~ radiation | temperature * wind,
  number=c(4,4),
  panel = function(x, y, ...) panel.smooth(x, y, span = .8, ...),
  xlab="Solar radiation (langleys)",
  ylab="Ozone (cube root ppb)")
})
```

---

ethanol *Engine exhaust fumes from burning ethanol*

Description

Ethanol fuel was burned in a single-cylinder engine. For various settings of the engine compression and equivalence ratio, the emissions of nitrogen oxides were recorded.

Usage

```
ethanol
```

Format

A data frame with 88 observations on the following 3 variables.

NOx Concentration of nitrogen oxides (NO and NO2) in micrograms/J.

C Compression ratio of the engine.

E Equivalence ratio–a measure of the richness of the air and ethanol fuel mixture.

Author(s)

Documentation contributed by Kevin Wright.

Source

Brinkman, N.D. (1981) Ethanol Fuel—A Single-Cylinder Engine Study of Efficiency and Exhaust Emissions. *SAE transactions*, **90**, 1410–1424.

References

Cleveland, William S. (1993). *Visualizing Data*. Hobart Press, Summit, New Jersey.

Examples

```
## Constructing panel functions on the fly
EE <- equal.count(ethanol$E, number=9, overlap=1/4)
xyplot(NOx ~ C | EE, data = ethanol,
       prepanel = function(x, y) prepanel.loess(x, y, span = 1),
       xlab = "Compression ratio", ylab = "NOx (micrograms/J)",
       panel = function(x, y) {
           panel.grid(h=-1, v= 2)
           panel.xyplot(x, y)
           panel.loess(x,y, span=1)
       },
       aspect = "xy")

# Wireframe loess surface fit.  See Figure 4.61 from Cleveland.
require(stats)
with(ethanol, {
  eth.lo <- loess(NOx ~ C * E, span = 1/3, parametric = "C",
                        drop.square = "C", family="symmetric")
  eth.marginal <- list(C = seq(min(C), max(C), length = 25),
                               E = seq(min(E), max(E), length = 25))
  eth.grid <- expand.grid(eth.marginal)
  eth.fit <- predict(eth.lo, eth.grid)
  wireframe(eth.fit ~ eth.grid$C * eth.grid$E,
            shade=TRUE,
            screen = list(z = 40, x = -60, y=0),
            distance = .1,
            xlab = "C", ylab = "E", zlab = "NOx")
})
```

`histogram` *Histograms and Kernel Density Plots*

Description

Draw Histograms and Kernel Density Plots, possibly conditioned on other variables.

Usage

```
histogram(x, data, ...)
densityplot(x, data, ...)
## S3 method for class 'formula':
histogram(x,
          data,
          allow.multiple, outer = TRUE,
          auto.key = FALSE,
          aspect = "fill",
          panel = "panel.histogram",
          prepanel, scales, strip, groups,
          xlab, xlim, ylab, ylim,
          type = c("percent", "count", "density"),
          nint = if (is.factor(x)) nlevels(x)
          else round(log2(length(x)) + 1),
          endpoints = extend.limits(range(x, finite = TRUE), prop = 0.04),
          breaks,
          equal.widths = TRUE,
          drop.unused.levels = lattice.getOption("drop.unused.levels"),
          ...,
          default.scales = list(),
          subscripts,
          subset)

## S3 method for class 'numeric':
histogram(x, data, xlab, ...)
## S3 method for class 'factor':
histogram(x, data, xlab, ...)

## S3 method for class 'formula':
densityplot(x,
            data,
            allow.multiple = is.null(groups) || outer,
            outer = !is.null(groups),
            auto.key = FALSE,
            aspect = "fill",
            panel = "panel.densityplot",
            prepanel, scales, strip, groups,
            xlab, xlim, ylab, ylim,
            bw, adjust, kernel, window, width, give.Rkern,
            n = 50, from, to, cut, na.rm,
            drop.unused.levels = lattice.getOption("drop.unused.levels"),
            ...,
```

```
            default.scales = list(),
            subscripts,
            subset)
## S3 method for class 'numeric':
densityplot(x, data, xlab, ...)

do.breaks(endpoints, nint)
```

Arguments

x	The object on which method dispatch is carried out. For the formula method, a formula of the form ~ x | g1 * g2 * ... indicates that histograms or Kernel Density estimates of x should be produced conditioned on the levels of the (optional) variables g1, g2, ... x can be numeric (or factor for histogram), and each of g1, g2, ... must be either factors or shingles. As a special case, the right hand side of the formula can contain more than one variable separated by a + sign. What happens in this case is described in details in the documentation for xyplot. Note that in either form, all the variables involved in the formula have to have same length. For the numeric and factor methods, x replaces the x vector described above. Conditioning is not allowed in these cases.
data	For the formula method, an optional data frame in which variables are to be evaluated. Ignored with a warning in other cases.
type	Character string indicating type of histogram to be drawn. "percent" and "count" give relative frequency and frequency histograms, and can be misleading when breakpoints are not equally spaced. "density" produces a density scale histogram. type defaults to "percent", except when the breakpoints are unequally spaced or breaks = NULL, when it defaults to "density".
nint	Number of bins. Applies only when breaks is unspecified or NULL in the call. Not applicable when the variable being plotted is a factor.
endpoints	vector of length 2 indicating the range of x-values that is to be covered by the histogram. This applies only when breaks is unspecified and the variable being plotted is not a factor. In do.breaks, this specifies the interval that is to be divided up.

breaks	usually a numeric vector of length (number of bins + 1) defining the breakpoints of the bins. Note that when breakpoints are not equally spaced, the only value of type that makes sense is density. When unspecified, the default is to use
	`breaks = seq_len(1 + nlevels(x)) - 0.5`
	when x is a factor, and
	`breaks = do.breaks(endpoints, nint)`
	otherwise. Breakpoints calculated in such a manner are used in all panels.
	Other values of breaks are possible, in which case they affect the display in each panel differently. A special value of breaks is NULL, in which case the number of bins is determined by nint and then breakpoints are chosen according to the value of equal.widths. Other valid values of breaks are those of the breaks argument in hist. This allows specification of breaks as an integer giving the number of bins (similar to nint), as a character string denoting a method, and as a function.

`equal.widths` logical, relevant only when `breaks=NULL`. If `TRUE`, equally spaced bins will be selected, otherwise, approximately equal area bins will be selected (this would mean that the breakpoints will **not** be equally spaced).

`n` number of points at which density is to be evaluated

`panel` The function that uses the packet (subset of display variables) corresponding to a panel to create a display. Default panel functions are documented separately, and often have arguments that can be used to customize its display in various ways. Such arguments can usually be directly supplied to the high level function.

`allow.multiple, outer, auto.key, aspect, prepanel, scales, strip, groups, xlab,` See `xyplot`

`bw, adjust, kernel, window, width, give.Rkern, from, to, cut, na.rm` arguments to `density`, passed on as appropriate

`...` Further arguments. See corresponding entry in `xyplot` for non-trivial details.

Details

`histogram` draws Conditional Histograms, while `densityplot` draws Conditional Kernel Density Plots. The density estimate in `densityplot` is actually calculated using the function `density`, and all arguments accepted by it can be passed (as `...`) in the call to `densityplot` to control the output. See documentation of `density` for details. (Note: The default value of the argument `n` of `density` is changed to 50.)

These and all other high level Trellis functions have several arguments in common. These are extensively documented only in the help page for `xyplot`, which should be consulted to learn more detailed usage.

`do.breaks` is an utility function that calculates breakpoints given an interval and the number of pieces to break it into.

Value

An object of class `"trellis"`. The `update` method can be used to update components of the object and the `print` method (usually called by default) will plot it on an appropriate plotting device.

Note

The form of the arguments accepted by the default panel function `panel.histogram` is different from that in S-PLUS. Whereas S-PLUS calculates the heights inside `histogram` and passes only the breakpoints and the heights to the panel function, here the original variable `x` is passed along with the breakpoints. This allows plots as in the second example below.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

`xyplot`, `panel.histogram`, `density`, `panel.densityplot`, `panel.mathdensity`, `Lattice`

Examples

```
require(stats)
histogram( ~ height | voice.part, data = singer, nint = 17,
          endpoints = c(59.5, 76.5), layout = c(2,4), aspect = 1,
          xlab = "Height (inches)")

histogram( ~ height | voice.part, data = singer,
          xlab = "Height (inches)", type = "density",
          panel = function(x, ...) {
              panel.histogram(x, ...)
              panel.mathdensity(dmath = dnorm, col = "black",
                                args = list(mean=mean(x),sd=sd(x)))
          } )

densityplot( ~ height | voice.part, data = singer, layout = c(2, 4),
            xlab = "Height (inches)", bw = 5)
```

---

interaction *Functions to Interact with Lattice Plots*

Description

The classic Trellis paradigm is to plot the whole object at once, without the possibility of interacting with it afterwards. However, by keeping track of the grid viewports where the panels and strips are drawn, it is possible to go back to them afterwards and enhance them one panel at a time. These functions provide convenient interfaces to help in this. Note that these are still experimental and the exact details may change in future.

Usage

```
panel.identify(x, y = NULL,
               subscripts = seq_along(x),
               labels = subscripts,
               n = length(x), offset = 0.5,
               threshold = 18, ## in points, roughly 0.25 inches
               panel.args = trellis.panelArgs(),
               ...)
trellis.vpname(name = c("position", "split", "split.location", "toplevel",
               "panel", "strip", "strip.left", "legend",
               "main", "sub", "xlab", "ylab", "page"),
               column, row,
               side = c("left", "top", "right", "bottom", "inside"),
               clip.off = FALSE, prefix)
trellis.grobname(name, prefix)
trellis.focus(name, column, row, side, clip.off,
              highlight = interactive(), ...)
trellis.switchFocus(name, side, clip.off, highlight, ...)
trellis.unfocus()
trellis.panelArgs(x, packet.number)
```

Arguments

`x, y`	variables defining the contents of the panel. In the case of `trellis.panelArgs`, a `"trellis"` object.
`n`	the number of points to identify by default (overridden by a right click)
`subscripts`	an optional vector of integer indices associated with each point. See details below.
`labels`	an optional vector of labels associated with each point. Defaults to `subscripts`
`offset`	the labels are printed either below, above, to the left or to the right of the identified point, depending on the relative location of the mouse click. The `offset` specifies (in "char" units) how far from the identified point the labels should be printed.
`threshold`	threshold in grid's `"points"` units. Points further than these from the mouse click position are not considered
`panel.args`	list that contains components names `x` (and usually `y`), to be used if `x` is missing. Typically, when called after `trellis.focus`, this would appropriately be the arguments passed to that panel.
`name`	character string indicating which viewport or grob we are looking for. Although these do not necessarily provide access to all viewports and grobs created by a lattice plot, they cover most that users might find interesting. `trellis.vpname` and `trellis.focus` deal with viewport names only, and only accept the values explicitly listed above. `trellis.grobname` is meant to create names for grobs, and can currently accept any value.
`column, row`	integers, indicating position of the panel or strip that should be assigned focus in the Trellis layout. Rows are usually calculated from the bottom up, unless the plot was created with `as.table=TRUE`
`side`	character string, relevant only for legends (i.e. when `name="legend"`), indicating their position. Partial specification is allowed, as long as it is unambiguous.
`clip.off`	logical, whether clipping should be off, relevant when `name` is `"panel"` or `"strip"`. This is necessary if axes are to be drawn outside the panel or strip. Note that setting `clip.off=FALSE` does not necessarily mean that clipping is on; that is determined by conditions in effect during printing.
`prefix`	character string acting as a prefix, meant to distinguish otherwise equivalent viewports in different plots. This only becomes relevant when a particular page is occupied by more than one plot. Defaults to the value appropriate for the last `"trellis"` object printed, as determined by the `prefix` argument in `print.trellis`. Users should not usually need to supply a value for this argument (see note below), however, if supplied explicitly, this has to be a valid R symbol name (briefly, it must start with a letter or a period followed by a letter) and must not contain the grid path separator (currently `"::"`)
`highlight`	logical, whether the viewport being assigned focus should be highlighted. For `trellis.focus`, the default is `TRUE` in interactive mode, and `trellis.switchFocus` by default preserves the setting currently active.
`packet.number`	integer, which panel to get data from. See `packet.number` for details on how this is calculated

`...` graphical parameters. For `panel.identify` these are used for labelling. For `trellis.focus` and `trellis.switchFocus`, these are used (in combination with `lattice.options`) for highlighting the chosen viewport if so requested.

Details

`panel.identify` is similar to `identify`. When called, it waits for the user to identify points (in the panel being drawn) via mouse clicks. Clicks other than left-clicks terminate the procedure. Although it is possible to call it as part of the panel function, it is more typical to use it to identify points after plotting the whole object, in which case a call to `trellis.focus` first is necessary.

One way in which `panel.identify` is different from `identify` is in how it uses the `subscripts` argument. In general, when one identifies points in a panel, one wants to identify the origin in the data frame used to produce the plot, and not within that particular panel. This information is available to the panel function, but only in certain situations. One way to ensure that `subscripts` is available is to specify `subscripts = TRUE` in the high level call such as `xyplot`. If `subscripts` is not explicitly specified in the call to `panel.identify`, but is available in `panel.args`, then those values will be used. Otherwise, they default to `seq_along(x)`. In either case, the final return value will be the subscripts that were marked.

The process of printing (plotting) a Trellis object builds up a grid layout with named viewports which can then be accessed to modify the plot further. While full flexibility can only be obtained by using grid functions directly, a few lattice functions are available for the more common tasks.

`trellis.focus` can be used to move to a particular panel or strip, identified by its position in the array of panels. It can also be used to focus on the viewport corresponding to one of the labels or a legend, though such usage would be less useful. The exact viewport is determined by the `name` along with the other arguments, not all of which are relevant for all names. Note that when more than one object is plotted on a page, `trellis.focus` will always go to the plot that was created last. For more flexibility, use grid functions directly (see note below).

After a successful call to `trellis.focus`, the desired viewport (typically panel or strip area) will be made the 'current' viewport (plotting area), which can then be enhanced by calls to standard lattice panel functions as well as grid functions.

It is quite common to have the layout of panels chosen when a `"trellis"` object is drawn, and not before then. Information on the layout (specifically, how many rows and columns, and which packet belongs in which position in this layout) is retained for the last `"trellis"` object plotted, and is available through `trellis.currentLayout`.

`trellis.unfocus` unsets the focus, and makes the top level viewport the current viewport.

`trellis.switchFocus` is a convenience function to switch from one viewport to another, while preserving the current `row` and `column`. Although the rows and columns only make sense for panels and strips, they would be preserved even when the user switches to some other viewport (where row/column is irrelevant) and then switches back.

Once a panel or strip is in focus, `trellis.panelArgs` can be used to retrieve the arguments that were available to the panel function at that position. In this case, it can be called without arguments as

```
trellis.panelArgs()
```

This usage is also allowed when a `"trellis"` object is being printed, e.g. inside the panel functions or the axis function (but not inside the prepanel function). `trellis.panelArgs` can also retrieve the panel arguments from any `"trellis"` object. Note that for this usage, one needs

to specify the `packet.number` (as described under the `panel` entry in `xyplot`) and not the position in the layout, because a layout determines the panel only **after** the object has been printed.

It is usually not necessary to call `trellis.vpname` and `trellis.grobname` directly. However, they can be useful in generating appropriate names in a portable way when using grid functions to interact with the plots directly, as described in the note below.

Value

`panel.identify` returns an integer vector containing the subscripts of the identified points (see details above). The equivalent of `identify` with `pos=TRUE` is not yet implemented, but can be considered for addition if requested.

`trellis.panelArgs` returns a named list of arguments that were available to the panel function for the chosen panel.

`trellis.vpname` and `trellis.grobname` return character strings.

Note

The viewports created by lattice is accessible to the user only up to a certain extent, as described above. In particular, `trellis.focus` can only be used to manipulate the last plot drawn. For full flexibility, use appropriate functions from the grid package directly. For example, `current.vpTree` can be used to inspect the current viewport tree and `seekViewport` or `downViewport` can be used to navigate to these viewports. For such usage, `trellis.vpname` and `trellis.grobname` (with a non-default `prefix` argument) provides a portable way to access the appropriate viewports and grobs by name.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

`identify`, `Lattice`, `print.trellis`, `trellis.currentLayout`, `current.vpTree`, `viewports`

Examples

```
## Not run:
xyplot(1:10 ~ 1:10)
trellis.focus("panel", 1, 1)
panel.identify()
## End(Not run)

xyplot(Petal.Length ~ Sepal.Length | Species, iris, layout = c(2, 2))
Sys.sleep(1)

trellis.focus("panel", 1, 1)
do.call("panel.lmline", trellis.panelArgs())
Sys.sleep(0.5)
trellis.unfocus()

trellis.focus("panel", 2, 1)
do.call("panel.lmline", trellis.panelArgs())
Sys.sleep(0.5)
trellis.unfocus()
```

```
trellis.focus("panel", 1, 2)
do.call("panel.lmline", trellis.panelArgs())
Sys.sleep(0.5)
trellis.unfocus()

## choosing loess smoothing parameter

p <- xyplot(dist ~ speed, cars)

panel.loessresid <-
    function(x = panel.args$x,
             y = panel.args$y,
             span,
             panel.args = trellis.panelArgs())
{
    fm <- loess(y ~ x, span = span)
    xgrid <- do.breaks(current.panel.limits()$xlim, 50)
    ygrid <- predict(fm, newdata = data.frame(x = xgrid))
    panel.lines(xgrid, ygrid)
    pred <- predict(fm)
    ## center residuals so that they fall inside panel
    resids <- y - pred + mean(y)
    fm.resid <- loess.smooth(x, resids, span = span)
    ##panel.points(x, resids, col = 1, pch = 4)
    panel.lines(fm.resid, col = 1)
}

spans <- c(0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8)
update(p, index.cond = list(rep(1, length(spans))))
panel.locs <- trellis.currentLayout()

i <- 1

for (row in 1:nrow(panel.locs))
    for (column in 1:ncol(panel.locs))
    if (panel.locs[row, column] > 0)
{
    trellis.focus("panel", row = row, column = column,
                  highlight = FALSE)
    panel.loessresid(span = spans[i])
    grid::grid.text(paste("span = ", spans[i]),
                    x = 0.25,
                    y = 0.75,
                    default.units = "npc")
    trellis.unfocus()
    i <- i + 1
}
```

`Lattice` *Lattice Graphics*

Description

Implementation of Trellis Graphics in R

Details

Trellis Graphics is a framework for data visualization developed at the Bell Labs by Rick Becker, Bill Cleveland et al, extending ideas presented in Bill Cleveland's 1993 book *Visualizing Data*.

Lattice is best thought of as an implementation of Trellis Graphics for R. Its interface is based on the implementation in S-PLUS, but there are several differences. To the extent possible, care has been taken to ensure that existing Trellis code written for S-PLUS works unchanged (or with minimal change) in Lattice. If you are having problems porting S-PLUS code, read the entry for `panel` in the documentation for `xyplot`. Most high level Trellis functions in S-PLUS are implemented, with the exception of `piechart`.

Lattice is built upon the Grid Graphics engine for R being developed by Paul Murrell and requires the `grid` add-on package.

Type `help(package = lattice)` to see a list of (public) Lattice graphics functions for which further documentation is available. The 'See Also' section below has list of specific areas of possible interest and pointers to the help pages with respective details. Apart from the documentation accompanying this package, several documents outlining the use of Trellis graphics is available from Bell Lab's website that might provide a holistic introduction to the Trellis paradigm. The example section shows how to bring up a brief history of changes to the lattice package, which provides a summary of new features.

Note

High level Lattice functions (like `xyplot`) are different from conventional R graphics functions because they don't actually draw anything. Instead, they return an object of class `"trellis"` which has to be then `print`ed or `plot`ted to create the actual plot. This is normally done automatically, but not when the high level functions are called inside another function (most often `source`) or other contexts where automatic printing is suppressed (e.g. `for` or `while` loops). In such situations, an explicit call to `print` or `plot` is required.

Lattice plots are highly customizable via user-modifiable settings. However, these are completely unrelated to base graphics settings; in particular, changing `par()` settings usually have no effect on lattice plots.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

References

Bell Lab's Trellis Page: `http://cm.bell-labs.com/cm/ms/departments/sia/project/trellis/`

Cleveland, W.S. (1993) *Visualizing Data*.

Becker, R.A., Cleveland, W.S. and Shyu, M. "The Visual Design and Control of Trellis Display", *Journal of Computational and Graphical Statistics*

See Also

The Lattice user interface primarily consists of several 'high level' generic functions (listed below), each designed to create a particular type of statistical display by default. While each function does different things, they share several common features, reflected in several common arguments that affect the resulting displays in similar ways. These arguments are extensively (sometimes only) documented in the help page for `xyplot`. This includes a discussion of *conditioning* and control of the Trellis layout.

Lattice employs an extensive system of user-controllable parameters to determine the look and feel of the displays it produces. To learn how to use and customise the Graphical parameters used by the Lattice functions, see `trellis.par.set`. For other settings, see `lattice.options`. The default graphical settings are different for different graphical devices. To learn how to initialise new devices with the desired settings or change the settings of the current device, see `trellis.device`.

To learn about sophisticated (non-default) printing capabilities, see `print.trellis`. See `update.trellis` to learn about manipulating a `"trellis"` object. Tools to augment lattice plots after they are drawn (including `locator`-like functionality) is described in the `interaction` help page.

The following is a list of 'high level' functions in the Lattice package with a brief description of what they do. In all cases, the actual display is produced by the so-called `panel` function, which has a suitable default, but can be substituted by an user defined function to create custom displays. The user will most often be interested in the default panel functions, which have a separate help page, linked to from the help pages of the corresponding high level function. Although documented separately, arguments to these panel functions can be supplied directly to the high level functions, which will forward the arguments as appropriate.

Univariate:

`barchart` bar plots

`bwplot` box and whisker plots

`densityplot` kernel density plots

`dotplot` dot plots

`histogram` histograms

`qqmath` quantile plots against mathematical distributions

`stripplot` 1-dimensional scatterplot

Bivariate:

`qq` q-q plot for comparing two distributions

`xyplot` scatter plot (and possibly a lot more)

Trivariate:

`levelplot` level plots (similar to image plots in R)

`contourplot` contour plots

`cloud` 3-D scatter plots

`wireframe` 3-D surfaces (similar to persp plots in R)

Hypervariate:

`splom` scatterplot matrix

`parallel` parallel coordinate plots

Miscellaneous:

`rfs` residual and fitted value plot (also see `oneway`)

`tmd` Tukey Mean-Difference plot

Additionally, there are several panel functions that do little by themselves, but can be useful components of custom panel functions. These are documented in `panel.functions`. Lattice also has a collection of convenience functions that correspond to the base graphics primitives `lines`, `points`, etc. They are implemented using Grid graphics, but try to be as close to the base versions as possible in terms of their argument list. These functions have imaginative names like `llines` or `panel.lines` and are often useful when writing (or porting from S-PLUS code) nontrivial panel functions.

Examples

```
## Not run:
## brief history of changes
file.show(system.file("Changes", package = "lattice"))
## End(Not run)
```

`lattice.options` *Low-level Options Controlling Behaviour of Lattice*

Description

Functions to handle settings used by lattice. Their main purpose is to make code maintainance easier, and users normally should not need to use these functions. However, fine control at this level maybe useful in certain cases.

Usage

```
lattice.options(...)
lattice.getOption(name)
```

Arguments

`name`	character giving the name of a setting
`...`	new options can be defined, or existing ones modified, using one or more arguments of the form `name = value` or by passing a list of such tagged values. Existing values can be retrieved by supplying the names (as character strings) of the components as unnamed arguments.

Details

These functions are modeled on `options` and `getOption`, and behave similarly for the most part. The components currently used are not documented here, but are fairly self-explanatory.

Value

`lattice.getOption` returns the value of a single component, whereas `lattice.options` always returns a list with one or more named components. When changing the values of components, the old values of the modified components are returned by `lattice.options`. If called without any arguments, the full list is returned.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

`options`, `trellis.device`, `trellis.par.get`, `Lattice`

Examples

```
str(lattice.options())
lattice.getOption("save.object")
```

`latticeParseFormula` *Parse Trellis formula*

Description

this function is used by high level Lattice functions like `xyplot` to parse the formula argument and evaluate various components of the data.

Usage

```
latticeParseFormula(model, data, dimension = 2,
                    subset = TRUE, groups = NULL,
                    multiple, outer,
                    subscripts,
                    drop)
```

Arguments

`model`	the model/formula to be parsed. This can be in either of two possible forms, one for 2d and one for 3d formulas, determined by the `dimension` argument. The 2d formulas are of the form `y ~ x| g1 * ... *gn`, and the 3d formulas are of the form `z ~ x * y	g1 * ...* gn`. In the first form, `y` may be omitted. The conditioning variables `g1, ...,gn` can be omitted in either case.
`data`	the environment/dataset where the variables in the formula are evaluated.
`dimension`	dimension of the model, see above
`subset`	index for choosing a subset of the data frame
`groups`	the grouping variable, if present
`multiple, outer`	logicals, determining how a '+' in the y and x components of the formula are processed. See `xyplot` for details
`subscripts`	logical, whether subscripts are to be calculated
`drop`	logical or list, similar to the `drop.unused.levels` argument in `xyplot`, indicating whether unused levels of conditioning factors and data variables that are factors are to be dropped.

Value

returns a list with several components, including `left, right, left.name, right.name, condition` for 2-D, and `left, right.x, right.y, left.name, right.x.name, right.y.name, condition` for 3-D. Other possible components are groups, subscr

Author(s)

Saikat DebRoy, Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

`xyplot`, `Lattice`

`levelplot` *Level Plots*

Description

Draw Level Plots and Contour plots.

Usage

```
levelplot(x, data, ...)
contourplot(x, data, ...)

## S3 method for class 'formula':
levelplot(x,
              data,
              allow.multiple = is.null(groups) || outer,
              outer = TRUE,
              aspect = "fill",
              panel = "panel.levelplot",
              prepanel = NULL,
              scales = list(),
              strip = TRUE,
              groups = NULL,
              xlab,
              xlim,
              ylab,
              ylim,
              at,
              cuts = 15,
              pretty = FALSE,
              region = TRUE,
              drop.unused.levels = lattice.getOption("drop.unused.levels"),
              ...,
              default.scales = list(),
              colorkey = region,
              col.regions,
              alpha.regions,
              subset = TRUE)

## S3 method for class 'formula':
contourplot(x,
              data,
              panel = "panel.contourplot",
              cuts = 7,
              labels = TRUE,
              contour = TRUE,
              pretty = TRUE,
              region = FALSE,
              ...)
## S3 method for class 'matrix':
levelplot(x, data, aspect = "iso", ...)
```

```
## S3 method for class 'matrix':
contourplot(x, data, aspect = "iso", ...)
```

Arguments

x	for the formula method, a formula of the form z ~ x * y | g1 * g2 * ..., where z is a numeric response, and x, y are numeric values evaluated on a rectangular grid. g1, g2, ... are optional conditional variables, and must be either factors or shingles if present. Calculations are based on the assumption that all x and y values are evaluated on a grid (defined by their unique values). The function will not return an error if this is not true, but the display might not be meaningful. However, the x and y values need not be equally spaced. Both levelplot and wireframe have methods for matrix objects, in which case x provides the z vector described above, while its rows and columns are interpreted as the x and y vectors respectively. This is similar to the form used in filled.contour and image.
data	For the formula methods, an optional data frame in which variables in the formula (as well as groups and subset, if any) are to be evaluated. Usually ignored with a warning in other cases.
panel	panel function used to create the display, as described in xyplot
aspect	For the matrix methods, the default aspect ratio is chosen to make each cell square. The usual default is aspect="fill", as described in xyplot.
at	numeric vector giving breaks along the range of z. Contours (if any) will be drawn at these heights, and the regions in between would be colored using col.regions.
col.regions	color vector to be used if regions is TRUE. The general idea is that this should be a color vector of moderately large length (longer than the number of regions. By default this is 100). It is expected that this vector would be gradually varying in color (so that nearby colors would be similar). When the colors are actually chosen, they are chosen to be equally spaced along this vector. When there are more regions than col.regions, the colors are recycled.
alpha.regions	numeric, specifying alpha transparency (works only on some devices)
colorkey	logical specifying whether a color key is to be drawn alongside the plot, or a list describing the color key. The list may contain the following components: **space:** location of the colorkey, can be one of "left", "right", "top" and "bottom". Defaults to "right". **x, y:** location, currently unused **col:** vector of colors **at:** numeric vector specifying where the colors change. must be of length 1 more than the col vector. **labels:** a character vector for labelling the at values, or more commonly, a list of components labels, at, cex, col, font describing label positions. **tick.number:** approximate number of ticks. **corner:** interacts with x, y; unimplemented

width: width of the key
height: length of key w.r.t side of plot.

`contour` logical, whether to draw contour lines.

`cuts` number of levels the range of `z` would be divided into

`labels` logical specifying whether contour lines should be labelled, or character vector of labels for contour lines. The type of labelling can be controlled by the `label.style` argument, which is passed on to `panel.levelplot`

`pretty` logical, whether to use pretty cut locations and labels

`region` logical, whether regions between contour lines should be filled

`allow.multiple, outer, prepanel, scales, strip, groups, xlab, xlim, ylab, ylim,` these arguments are described in the help page for `xyplot`.

`...` other arguments

Details

These and all other high level Trellis functions have several arguments in common. These are extensively documented only in the help page for `xyplot`, which should be consulted to learn more detailed usage.

Other useful arguments are mentioned in the help page for the default panel function `panel.levelplot` (these are formally arguments to the panel function, but can be specified in the high level calls directly).

Value

An object of class `"trellis"`. The `update` method can be used to update components of the object and the `print` method (usually called by default) will plot it on an appropriate plotting device.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

`xyplot`, `Lattice`, `panel.levelplot`

Examples

```
x <- seq(pi/4, 5 * pi, length = 100)
y <- seq(pi/4, 5 * pi, length = 100)
r <- as.vector(sqrt(outer(x^2, y^2, "+")))
grid <- expand.grid(x=x, y=y)
grid$z <- cos(r^2) * exp(-r/(pi^3))
levelplot(z~x*y, grid, cuts = 50, scales=list(log="e"), xlab="",
          ylab="", main="Weird Function", sub="with log scales",
          colorkey = FALSE, region = TRUE)

#S-PLUS example
require(stats)
attach(environmental)
ozo.m <- loess((ozone^(1/3)) ~ wind * temperature * radiation,
       parametric = c("radiation", "wind"), span = 1, degree = 2)
```

```
w.marginal <- seq(min(wind), max(wind), length = 50)
t.marginal <- seq(min(temperature), max(temperature), length = 50)
r.marginal <- seq(min(radiation), max(radiation), length = 4)
wtr.marginal <- list(wind = w.marginal, temperature = t.marginal,
       radiation = r.marginal)
grid <- expand.grid(wtr.marginal)
grid[, "fit"] <- c(predict(ozo.m, grid))
contourplot(fit ~ wind * temperature | radiation, data = grid,
            cuts = 10, region = TRUE,
            xlab = "Wind Speed (mph)",
            ylab = "Temperature (F)",
            main = "Cube Root Ozone (cube root ppb)")
detach()
```

`llines` *Replacements of traditional graphics functions*

Description

These functions are intended to replace common low level traditional graphics functions, primarily for use in panel functions. The originals can not be used (at least not easily) because lattice panel functions need to use grid graphics. Low level drawing functions in grid can be used directly as well, and is often more flexible. These functions are provided for convenience and portability.

Usage

```
lplot.xy(xy, type, pch, lty, col, cex, lwd,
         font, fontfamily, fontface,
         col.line, alpha, fill,
         origin = 0, ...)

llines(x, ...)
lpoints(x, ...)
ltext(x, ...)

## Default S3 method:
llines(x, y = NULL, type = "l",
       col, alpha, lty, lwd, ...)
## Default S3 method:
lpoints(x, y = NULL, type = "p", col, pch, alpha, fill,
        font, fontfamily, fontface, cex, ...)
## Default S3 method:
ltext(x, y = NULL, labels = seq_along(x),
      col, alpha, cex, srt = 0,
      lineheight, font, fontfamily, fontface,
      adj = c(0.5, 0.5), pos = NULL, offset = 0.5, ...)

lsegments(x0, y0, x1, y1, x2, y2,
          col, alpha, lty, lwd, ...)
lrect(xleft, ybottom, xright, ytop,
      x = (xleft + xright) / 2,
      y = (ybottom + ytop) / 2,
```

```
       width = xright - xleft,
       height = ytop - ybottom,
       col = "transparent",
       border = "black",
       lty = 1, lwd = 1, alpha = 1,
       just = "center",
       hjust = NULL, vjust = NULL,
       ...)
larrows(x0 = NULL, y0 = NULL, x1, y1, x2 = NULL, y2 = NULL,
        angle = 30, code = 2, length = 0.25, unit = "inches",
        ends = switch(code, "first", "last", "both"),
        type = "open",
        col = add.line$col,
        alpha = add.line$alpha,
        lty = add.line$lty,
        lwd = add.line$lwd,
        fill = NULL, ...)
lpolygon(x, y = NULL,
         border = "black", col = "transparent", ...)

panel.lines(...)
panel.points(...)
panel.segments(...)
panel.text(...)
panel.rect(...)
panel.arrows(...)
panel.polygon(...)
```

Arguments

`x, y, x0, y0, x1, y1, x2, y2, xy`
: locations. `x2` and `y2` are available for for S compatibility.

`length, unit`
: determines extent of arrow head. `length` specifies the length in terms of `unit`, which can be any valid grid unit as long as it doesn't need a `data` argument. `unit` defaults to inches, which is the only option in the base version of the function, `arrows`.

`angle, code, type, labels, srt, adj, pos, offset`
: arguments controlling behaviour. See respective base functions for details. For `larrows` and `panel.larrows`, `type` is either `"open"` or `"closed"`, indicating the type of arrowhead.

`ends`
: serves the same function as `code`, using descriptive names rather than integer codes. If specified, this overrides `code`

`col, alpha, lty, lwd, fill, pch, cex, lineheight, font, fontfamily, fontface, co`
: graphical parameters. `fill` applies to points when `pch` is in `21:25` and specifies the fill color, similar to the `bg` argument in the base graphics function `points`. For devices that support alpha-transparency, a numeric argument `alpha` between 0 and 1 can controls transparency. Be careful with this, since for devices that do not support alpha-transparency, nothing will be drawn at all if this is set to anything other than 0.

`origin`
: for `type="h"` or `type="H"`, the value to which lines drop down.

`xleft, ybottom, xright, ytop`
: see `rect`

`width, height, just, hjust, vjust`
: finer control over rectangles, see `grid.rect`

`...`
: extra arguments, passed on to lower level functions as appropriate.

Details

These functions are meant to be grid replacements of the corresponding base R graphics functions, to allow existing Trellis code to be used with minimal modification. The functions `panel.*` are essentially identical to the `l*` versions, are recommended for use in new code (as opposed to ported code) as they have more readable names.

See the documentation of the base functions for usage. Not all arguments are always supported. All these correspond to the default methods only.

Note

There is a new `type="H"` option wherever appropriate, which is similar to `type="h"`, but with horizontal lines.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

`points`, `lines`, `rect`, `text`, `segments`, `arrows`, `Lattice`

---

lset *Interface to modify Trellis Settings - Deprecated*

Description

A (hopefully) simpler alternative to `trellis.par.get/set`. This is deprecated, and the same functionality is now available with `trellis.par.set`

Usage

```
lset(theme = col.whitebg())
```

Arguments

`theme`
: a list decribing how to change the settings of the current active device. Valid components are those in the list returned by `trellis.par.get()`. Each component must itself be a list, with one or more of the appropriate components (need not have all components). Changes are made to the settings for the currently active device only.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

make.groups *Grouped data from multiple vectors*

Description

Combines two or more vectors, possibly of different lengths, producing a data frame with a second column indicating which of these vectors that row came from. This is mostly useful for getting data into a form suitable for use in high level Lattice functions.

Usage

```
make.groups(...)
```

Arguments

...	one or more vectors of the same type (coercion is attempted if not), or one or more data frames with similar columns, with possibly differing number of rows.

Value

When all the input arguments are vectors, a data frame with two columns

this-is-escaped-codenormal-bracket9bracket-normal
: all the vectors supplied, concatenated

this-is-escaped-codenormal-bracket12bracket-normal
: factor indicating which vector the corresponding data value came from

When all the input arguments are data frames, the result of cbind applied to them, along with an additional which column as described above.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

Lattice

Examples

```
sim.dat <-
    make.groups(uniform = runif(200),
                exponential = rexp(175),
                lognormal = rlnorm(150),
                normal = rnorm(125))
qqmath( ~ data | which, sim.dat, scales = list(y = "free"))
```

`melanoma` *Melanoma skin cancer incidence*

Description

These data from the Connecticut Tumor Registry present age-adjusted numbers of melanoma skin-cancer incidences per 100,000 people in Connectict for the years from 1936 to 1972.

Usage

```
melanoma
```

Format

A data frame with 37 observations on the following 2 variables.

year Years 1936 to 1972.

incidence Rate of melanoma cancer per 100,000 population.

Note

This dataset is not related to the `melanoma` dataset in the **boot** package with the same name.

The S-PLUS 6.2 help for the melanoma data says that the incidence rate is per *million*, but this is not consistent with data found at the National Cancer Institute (`http://www.nci.nih.gov`).

Author(s)

Documentation contributed by Kevin Wright.

Source

Houghton, A., E. W. Munster, and M. V. Viola. (1978). Increased Incidence of Malignant Melanoma After Peaks of Sunspot Activity. *The Lancet*, **8**, 759–760.

References

Cleveland, William S. (1993). *Visualizing Data*. Hobart Press, Summit, New Jersey.

Examples

```
# Time-series plot.  Figure 3.64 from Cleveland.
xyplot(incidence ~ year,
  data = melanoma,
        aspect = "xy",
        panel = function(x, y)
                panel.xyplot(x, y, type="o", pch = 16),
        ylim = c(0, 6),
        xlab = "Year",
        ylab = "Incidence")
```

`oneway` *Fit One-way Model*

Description

Fits a One-way model to univariate data grouped by a factor, the result often being displayed using `rfs`

Usage

```
oneway(formula, data, location=mean, spread=function(x) sqrt(var(x)))
```

Arguments

`formula`	formula of the form `y ~ x` where `y` is the numeric response and `x` is the grouping factor
`data`	data frame in which the model is to be evaluated
`location`	function or numeric giving the location statistic to be used for centering the observations, e.g. `median`, 0 (to avoid centering).
`spread`	function or numeric giving the spread statistic to be used for scaling the observations, e.g. `sd`, 1 (to avoid scaling).

Value

A list with components

`location`	vector of locations for each group.
`spread`	vector of spreads for each group.
`fitted.values`	vector of locations for each observation.
`residuals`	residuals (`y - fitted.values`).
`scaled.residuals`	residuals scaled by `spread` for their group

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

`rfs`, `Lattice`

`packet.panel.default` *Associating Packets with Panels*

Description

When a `"trellis"` object is plotted, panels are always drawn in an order such that columns vary the fastest, then rows and then pages. An optional function can be specified that determines, given the column, row and page and other relevant information, the packet (if any) which should be used in that panel. The function documented here implements the default behaviour, which is to match panel order with packet order, determined by varying the first conditioning variable the fastest, then the second, and so on. This matching is performed after any reordering and/or permutation of the conditioning variables.

Usage

```
packet.panel.default(layout, condlevels, page, row, column,
                     skip, all.pages.skip = TRUE)
```

Arguments

`layout`	the `layout` argument in high level functions, suitably standardized.
`condlevels`	a list of levels of conditioning variables, after relevant permutations and/or reordering of levels
`page, row, column`	the location of the panel in the coordinate system of pages, rows and columns.
`skip`	the `skip` argument in high level functions
`all.pages.skip`	whether `skip` should be replicated over all pages. If `FALSE`, `skip` will be replicated to be only as long as the number of positions on a page, and that template will be used for all pages.

Value

A suitable combination of levels of the conditioning variables in the form of a numeric vector as long as the number of conditioning variables, with each element an integer indexing the levels of the corresponding variable. Specifically, if the return value is `p`, then the `i`-th conditioning variable will have level `condlevels[[i]][p[i]]`.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

`Lattice`, `xyplot`

panel.axis *Panel Function for Drawing Axis Ticks and Labels*

Description

`panel.axis` is the function used by lattice to draw axes. It is typically not used by users, except those wishing to create advanced annotation. Keep in mind issues of clipping when trying to use it as part of the panel function. `current.panel.limits` can be used to retrieve a panel's x and y limits.

Usage

```
panel.axis(side = c("bottom", "left", "top", "right"),
           at,
           labels = TRUE,
           draw.labels = TRUE,
           check.overlap = FALSE,
           outside = FALSE,
           ticks = TRUE,
           half = !outside,
           which.half,
           tck = as.numeric(ticks),
           rot = if (is.logical(labels)) 0 else c(90, 0),
           text.col, text.alpha, text.cex, text.font,
           text.fontfamily, text.fontface,
           line.col, line.lty, line.lwd, line.alpha)

current.panel.limits(unit = "native")
```

Arguments

`side`	character string indicating which side axes are to be drawn on. Partial specification is allowed.
`at`	location of labels
`labels`	the labels to go along with `at`. The labels can be a character vector or a vector of expressions. Alternatively, this can be a logical. If TRUE, the labels are derived from `at`, otherwise, labels are empty.
`draw.labels`	logical indicating whether labels are to be drawn
`check.overlap`	logical, whether to check for overlapping of labels. This also has the effect of removing `at` values that are 'too close' to the limits.
`outside`	logical, whether to the labels draw outside the panel or inside. Note that `outside=TRUE` will only have a visible effect if clipping is disabled for the viewport (panel).
`ticks`	logical, whether to draw the tickmarks
`half`	logical, whether only half of scales will be drawn for each side
`which.half`	character string, one of `"lower"` and `"upper"`. Indicates which half is to be used for tick locations if `half=TRUE`. Defaults to whatever is suitable for `splom`

tck	numeric scalar, multiplier for tick length. Can be negative.
rot	rotation angles for labels in degrees. Can be a vector of length 2 for x- and y-axes respectively
text.col, text.alpha, text.cex, text.font, text.fontfamily, text.fontface, line.	graphical parameters
unit	which grid unit the values should be in

Details

panel.axis can draw axis tick marks inside or outside a panel (more precisely, a grid viewport). It honours the (native) axis scales. Used in panel.pairs for splom, as well as for all the usual axis drawing by the print method for "trellis" objects. It can also be used to enhance plots 'after the fact' by adding axes.

Value

current.panel.limits returns a list with components xlim and ylim, which are both numeric vectors of length 2, giving the scales of the current panel (viewport). The values correspond to the unit system specified by unit, by default "native".

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

Lattice, xyplot, trellis.focus, unit

---

panel.barchart *Default Panel Function for barchart*

Description

Default panel function for barchart.

Usage

```
panel.barchart(x, y, box.ratio = 1,
               horizontal = TRUE,
               origin = NULL, reference = TRUE,
               stack = FALSE,
               groups = NULL,
               col = if (is.null(groups)) plot.polygon$col else superpose.polygo
               border = if (is.null(groups)) plot.polygon$border else superpose.
               lty = if (is.null(groups)) plot.polygon$lty else superpose.polygo
               lwd = if (is.null(groups)) plot.polygon$lwd else superpose.polygo
               ...)
```

Arguments

x	Extent of Bars. By default, bars start at left of panel, unless origin is specified, in which case they start there
y	Horizontal location of bars, possibly factor
box.ratio	ratio of bar width to inter-bar space
horizontal	logical. If FALSE, the plot is 'transposed' in the sense that the behaviours of x and y are switched. x is now the 'factor'. Interpretation of other arguments change accordingly. See documentation of bwplot for a fuller explanation.
origin	origin of the bars. For grouped displays with stack = TRUE, this argument is ignored and the origin set to 0. Otherwise, defaults to NULL, in which case bars start at leftmost end of panel.
reference	logical, whether a reference line is to be drawn at the origin
stack	logical, relevant when groups is non-null. If FALSE (the default), bars for different values of the grouping variable are drawn side by side, otherwise they are stacked.
groups	optional grouping variable
col, border, lty, lwd	
...	extra arguments will be accepted but ignored

Details

A barchart is drawn in the panel. Note that most arguments controlling the display can be supplied to the high-level barchart call directly.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

barchart

---

panel.bwplot *Default Panel Function for bwplot*

Description

This is the default panel function for bwplot.

Usage

```
panel.bwplot(x, y, box.ratio = 1,
             horizontal = TRUE,
             pch, col, cex,
             font, fontfamily, fontface,
             fill, varwidth = FALSE,
             ...,
```

```
              levels.fos,
              stats = boxplot.stats,
              coef = 1.5,
              do.out = TRUE)
```

Arguments

x, y	numeric vector or factor. Boxplots drawn for each unique value of y (x) if horizontal is TRUE (FALSE)
box.ratio	ratio of box height to inter box space
horizontal	logical. If FALSE, the plot is 'transposed' in the sense that the behaviours of x and y are switched. x is now the 'factor'. Interpretation of other arguments change accordingly. See documentation of bwplot for a fuller explanation.
pch, col, cex, font, fontfamily, fontface	graphical parameters controlling the dot. pch="|" is treated specially, by replacing the dot with a line (similar to boxplot)
fill	color to fill the boxplot
varwidth	logical. If TRUE, widths of boxplots are proportional to the number of points used in creating it.
stats	a function, defaulting to boxplot.stats, that accepts a numeric vector and returns a list similar to the return value of boxplot.stats. The function must accept arguments coef and do.out even if they do not use them (a ... argument is good enough). This function is used to determine the box and whisker plot.
coef, do.out	passed to stats
levels.fos	numeric values corresponding to positions of the factor or shingle variable. For internal use.
...	further arguments, ignored.

Details

Creates Box and Whisker plot of x for every level of y (or the reverse if horizontal=FALSE). By default, the actual boxplot statistics are calculated using boxplot.stats. Note that most arguments controlling the display can be supplied to the high-level bwplot call directly. Unlike the traditional analog boxplot, notches are not implemented.

Although the graphical parameters for the dot representing the median can be controlled by optional arguments, many others can not. These parameters are obtained from the relevant settings parameters ("box.rectangle" for the box, "box.umbrella" for the whiskers and "plot.symbol" for the outliers).

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

bwplot, boxplot.stats

Examples

```
bwplot(voice.part ~ height, data = singer,
       xlab = "Height (inches)",
       panel = function(...) {
           panel.grid(v = -1, h = 0)
           panel.bwplot(...)
       },
       par.settings = list(plot.symbol = list(pch = 4)))
```

---

panel.cloud *Default Panel Function for cloud*

Description

These are default panel functions controlling `cloud` and `wireframe` displays.

Usage

```
panel.cloud(x, y, subscripts, z,
            groups = NULL,
            perspective = TRUE,
            distance = if (perspective) 0.2 else 0,
            xlim, ylim, zlim,
            panel.3d.cloud = "panel.3dscatter",
            panel.3d.wireframe = "panel.3dwire",
            screen = list(z = 40, x = -60),
            R.mat = diag(4), aspect = c(1, 1),
            par.box = NULL,
            xlab, ylab, zlab,
            xlab.default, ylab.default, zlab.default,
            scales.3d,
            proportion = 0.6,
            wireframe = FALSE,
            scpos,
            ...,
            at)
panel.wireframe(...)
panel.3dscatter(x, y, z, rot.mat, distance,
                groups, type = "p",
                xlim.scaled, ylim.scaled, zlim.scaled,
                zero.scaled,
                col, col.point, col.line,
                lty, lwd, cex, pch,
                cross, ..., subscripts)
panel.3dwire(x, y, z, rot.mat = diag(4), distance,
             shade = FALSE,
             shade.colors.palette = trellis.par.get("shade.colors")$palette,
             light.source = c(0, 0, 1000),
             xlim.scaled,
             ylim.scaled,
```

```
zlim.scaled,
col = if (shade) "transparent" else "black",
lty = 1, lwd = 1,
alpha,
col.groups = superpose.polygon$col,
polynum = 100,
...,
drape = FALSE,
at,
col.regions = regions$col,
alpha.regions = regions$alpha)
```

Arguments

x, y, z	numeric (or possibly factors) vectors representing the data to be displayed. The interpretation depends on the context. For panel.cloud these are essentially the same as the data passed to the high level plot (except if formula was a matrix, the appropriate x and y vectors are generated). By the time they are passed to panel.3dscatter and panel.3dwire, they have been scaled (to lie inside a bounding box, usually the [-0.5, 0.5] cube). Further, for panel.3dwire, x and y are shorter than z and represent the sorted locations defining a rectangular grid. Also in this case, z may be a matrix if the display is grouped, with each column representing one surface. In panel.cloud (called from wireframe) and panel.3dwire, x, y and z could also be matrices (of the same dimension) when they represent a 3-D surface parametrized on a 2-D grid.
subscripts	index specifying which points to draw. The same x, y and z values (representing the whole data) are passed to panel.cloud for each panel. subscripts specifies the subset of rows to be used for the particular panel.
groups	specification of a grouping variable, passed down from the high level functions.
perspective	logical, whether to plot a perspective view. Setting this to FALSE is equivalent to setting distance to 0
distance	numeric, between 0 and 1, controls amount of perspective. The distance of the viewing point from the origin (in the transformed coordinate system) is 1 / distance. This is described in a little more detail in the documentation for cloud
screen	A list determining the sequence of rotations to be applied to the data before being plotted. The initial position starts with the viewing point along the positive z-axis, and the x and y axes in the usual position. Each component of the list should be named one of "x", "y" or "z" (repititions are allowed), with their values indicating the amount of rotation about that axis in degrees.
R.mat	initial rotation matrix in homogeneous coordinates, to be applied to the data before screen rotates the view further.
par.box	graphical parameters for box, namely, col, lty and lwd. By default obtained from the parameter box.3d
xlim, ylim, zlim	limits for the respective axes. As with other lattice functions, these could each be a numeric 2-vector or a character vector indicating levels of a factor.

`panel.3d.cloud, panel.3d.wireframe`
: functions that draw the data-driven part of the plot (as opposed to the bounding box and scales) in `cloud` and `wireframe`. This function is called after the 'back' of the bounding box is drawn, but before the 'front' is drawn.

  Any user-defined custom display would probably want to change these functions. The intention is to pass as much information to this function as might be useful (not all of which are used by the defaults). In particular, these functions can expect arguments called `xlim`, `ylim`, `zlim` which give the bounding box ranges in the original data scale and `xlim.scaled`, `ylim.scaled`, `zlim.scaled` which give the bounding box ranges in the transformed scale. More arguments can be considered on request.

`aspect`
: aspect as in `cloud`

`xlab, ylab, zlab`
: Labels, have to be lists. Typically the user will not manipulate these, but instead control this via arguments to `cloud` directly.

`xlab.default`
: for internal use

`ylab.default`
: for internal use

`zlab.default`
: for internal use

`scales.3d`
: list defining the scales

`proportion`
: numeric scalar, gives the length of arrows as a proportion of the sides

`scpos`
: A list with three components x, y and z (each a scalar integer), describing which of the 12 sides of the cube the scales should be drawn. The defaults should be OK. Valid values are x: 1, 3, 9, 11; y: 8, 5, 7, 6 and z: 4, 2, 10, 12. (See comments in the source code of `panel.cloud` to see the details of this enumeration.)

`wireframe`
: logical, indicating whether this is a wireframe plot

`drape`
: logical, whether the facets will be colored by height, in a manner similar to `levelplot`. This is ignored if `shade=TRUE`.

`at, col.regions, alpha.regions`
: deals with specification of colors when `drape = TRUE` in `wireframe`. `at` can be a numeric vector, `col.regions` a vector of colors, and `alpha.regions` a numeric scalar controlling transparency. The resulting behaviour is similar to `levelplot`, `at` giving the breakpoints along the z-axis where colors change, and the other two determining the colors of the facets that fall in between.

`rot.mat`
: 4x4 transformation matrix in homogeneous coordinates. This gives the rotation matrix combining the `screen` and `R.mat` arguments to `panel.cloud`

`type`
: character vector, specifying type of cloud plot. Can include one or more of `"p"`, `"l"`, `"h"` or `"b"`. `"p"` and `"l"` mean 'points' and 'lines' respectively, and `"b"` means 'both'. `"h"` stands for 'histogram', and causes a line to be drawn from each point to the X-Y plane (i.e., the plane representing `z = 0`), or the lower (or upper) bounding box face, whichever is closer.

`xlim.scaled, ylim.scaled, zlim.scaled`
: axis limits (after being scaled to the bounding box)

`zero.scaled`
: z-axis location (after being scaled to the bounding box) of the X-Y plane in the original data scale, to which lines will be dropped (if within range) from each point when `type = "h"`

`cross` logical, defaults to `TRUE` if `pch = "+"`. `panel.3dscatter` can represent each point by a 3d 'cross' of sorts (it's much easier to understand looking at an example than from a description). This is different from the usual `pch` argument, and reflects the depth of the points and the orientation of the axes. This argument indicates whether this feature will be used.

This is useful for two reasons. It can be set to `FALSE` to use `"+"` as the plotting character in the regular sense. It can also be used to force this feature in grouped displays.

`shade` logical, indicating whether the surface is to be colored using an illumination model with a single light source

`shade.colors.palette`
a function (or the name of one) that is supposed to calculate the color of a facet when shading is being used. Three pieces of information is available to the function: first, the cosine of the angle between the incident light ray and the normal to the surface (representing foreshortening); second, the cosine of half the angle between the reflected ray and the viewing direction (useful for non-lambertian surfaces); and third, the scaled (average) height of that particular facet with respect to the total plot z-axis limits.

All three numbers should be between 0 and 1. The `shade.colors.palette` function should return a valid color. The default function is obtained from the trellis settings.

`light.source` a 3-vector representing (in cartesian coordinates) the light source. This is relative to the viewing point being (0, 0, 1/distance) (along the positive z-axis), keeping in mind that all observations are bounded within the [-0.5, 0.5] cube

`polynum` quadrilateral faces are drawn in batches of `polynum` at a time. Drawing too few at a time increases the total number of calls to the underlying `grid.polygon` function, which affects speed. Trying to draw too many at once may be unnecessarily memory intensive. This argument controls the trade-off.

`col.groups` colors for different groups

`col, col.point, col.line, lty, lwd, cex, pch, alpha`
graphical parameters

`...` other parameters, passed down when appropriate

Details

These functions together are responsible for the content drawn inside each panel in `cloud` and `wireframe`. `panel.wireframe` is a wrapper to `panel.cloud`, which does the actual work.

`panel.cloud` is responsible for drawing the content that does not depend on the data, namely, the bounding box, the arrows/scales, etc. At some point, depending on whether `wireframe` is TRUE, it calls either `panel.3d.wireframe` or `panel.3d.cloud`, which draws the data-driven part of the plot.

The arguments accepted by these two functions are different, since they have essentially different purposes. For cloud, the data is unstructured, and `x`, `y` and `z` are all passed to the `panel.3d.cloud` function. For wireframe, on the other hand, `x` and `y` are increasing vectors with unique values, defining a rectangular grid. `z` must be a matrix with `length(x) * length(y)` rows, and as many columns as the number of groups.

`panel.3dscatter` is the default `panel.3d.cloud` function. It has a `type` argument similar to `panel.xyplot`, and supports grouped displays. It tries to honour depth ordering, i.e., points and lines closer to the camera are drawn later, overplotting more distant ones. (Of course there is

no absolute ordering for line segments, so an ad hoc ordering is used. There is no hidden point removal.)

`panel.3dwire` is the default `panel.3d.wireframe` function. It calculates polygons corresponding to the facets one by one, but waits till it has collected information about `polynum` facets, and draws them all at once. This avoids the overhead of drawing `grid.polygon` repeatedly, speeding up the rendering considerably. If `shade = TRUE`, these attempt to color the surface as being illuminated from a light source at `light.source`. `palette.shade` is a simple function that provides the deafult shading colors

Multiple surfaces are drawn if `groups` is non-null in the call to `wireframe`, however, the algorithm is not sophisticated enough to render intersecting surfaces correctly.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

`cloud`, `utilities.3d`

---

panel.densityplot *Default Panel Function for densityplot*

Description

This is the default panel function for `densityplot`.

Usage

```
panel.densityplot(x, darg, plot.points = "jitter", ref = FALSE,
                  groups = NULL,
                  jitter.amount, type, ...)
```

Arguments

`x`	data points for which density is to be estimated
`darg`	list of arguments to be passed to the `density` function. Typically, this should be a list with zero or more of the following components : `bw`, `adjust`, `kernel`, `window`, `width`, `give.Rkern`, `n`, `from`, `to`, `cut`, `na.rm` (see `density` for details)
`plot.points`	logical specifying whether or not the data points should be plotted along with the estimated density. Alternatively, a character string specifying how the points should be plotted. Meaningful values are `"rug"`, in which case `panel.rug` is used to plot a 'rug', and `"jitter"`, in which case the points are jittered vertically to better distinguish overlapping points.
`ref`	logical, whether to draw x-axis
`groups`	an optional grouping variable. If present, `panel.superpose` will be used instead to display each subgroup
`jitter.amount`	when `plot.points="jitter"`, the value to use as the `amount` argument to `jitter`.

type	type argument used to plot points, if requested. This is not expected to be useful, it is available mostly to protect a type argument, if specified, from affecting the density curve.
...	extra graphical parameters

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

densityplot, jitter

---

panel.dotplot *Default Panel Function for dotplot*

Description

Default panel function for dotplot.

Usage

```
panel.dotplot(x, y, horizontal = TRUE,
              pch, col, lty, lwd,
              col.line, levels.fos,
              groups = NULL,
              ...)
```

Arguments

x,y	variables to be plotted in the panel. Typically y is the 'factor'
horizontal	logical. If FALSE, the plot is 'transposed' in the sense that the behaviours of x and y are switched. x is now the 'factor'. Interpretation of other arguments change accordingly. See documentation of bwplot for a fuller explanation.
pch, col, lty, lwd, col.line	graphical parameters
levels.fos	locations where reference lines will be drawn
groups	grouping variable (affects graphical parameters)
...	extra parameters, passed to panel.xyplot which is responsible for drawing the foreground points (panel.dotplot only draws the background reference lines).

Details

Creates (possibly grouped) Dotplot of x against y or vice versa

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

dotplot

`panel.functions` *Useful Panel Functions*

Description

These are predefined panel functions available in lattice for use in constructing new panel functions (usually on-the-fly).

Usage

```
panel.abline(a, b = NULL, h = numeric(0), v = numeric(0),
             col, col.line, lty, lwd, type, ...)

panel.curve(expr, from, to, n = 101,
            curve.type = "l",
            col, lty, lwd, type,
            ...)
panel.rug(x = NULL, y = NULL,
          regular = TRUE,
          start = if (regular) 0 else 0.97,
          end = if (regular) 0.03 else 1,
          x.units = rep("npc", 2),
          y.units = rep("npc", 2),
          col, lty, lwd, alpha,
          ...)
panel.linejoin(x, y, fun = mean, horizontal = TRUE,
               lwd, lty, col, col.line, type,
               ...)

panel.fill(col, border, ...)
panel.grid(h=3, v=3, col, col.line, lty, lwd, ...)
panel.lmline(x, y, ...)
panel.loess(x, y, span = 2/3, degree = 1,
            family = c("symmetric", "gaussian"),
            evaluation = 50,
            lwd, lty, col, col.line, type,
            ...)
panel.mathdensity(dmath = dnorm, args = list(mean=0, sd=1),
                  n = 50, col, col.line, lwd, lty, type,
                  ...)
```

Arguments

`x, y`	variables defining the contents of the panel
`a, b`	Coefficients of the line to be added. `a` can be a vector of length 2, representing the coefficients of the line to be added, in which case `b` should be missing. `a` can also be an appropriate 'regression' object, i.e., an object which has a `coef` method that returns a length 2 numeric vector. The corresponding line will be plotted.

h, v	For panel.abline, numerical vectors giving locations respectively of horizontal and vertical lines to be added to the plot, in native coordinates. For panel.grid, number of horizontal and vertical reference lines to be added to the plot; h=-1 and v=-1 make the grids aligned with the axis labels (but this does not always work).
expr	expression as a function of x or a function to plot as a curve
n	the number of points to use for drawing the curve
regular	logical indicating whether the 'rug' is to be drawn on the regular side (left / bottom) or not (right / top)
start, end	endpoints of rug segments, in normalized parent coordinates (between 0 and 1). Defaults depend on value of regular, and cover 3% of the panel width and height
x.units, y.units	character vector, replicated to be of length two. Specifies the (grid) units associated with start and end above. x.units and y.units are for the rug on the x-axis and y-axis respectively (and thus are associated with start and end values on the y and x scales respectively).
from, to	optional lower and upper x-limits of curve. If missing, limits of current panel are used
curve.type	type of curve ("p" for points, etc), passed to llines
col, col.line, lty, lwd, alpha, border	graphical parameters
type	usually ignored. This is present only to make sure an explicitly specified type argument (perhaps meant for another function) doesn't affect the display.
span, degree, family, evaluation	arguments to loess.smooth, for which panel.loess is essentially a wrapper
fun	the function that will be applied to the subset of x(y) determined by the unique values of y(x)
horizontal	logical. If FALSE, the plot is 'transposed' in the sense that the behaviours of x and y are switched. x is now the 'factor'. Interpretation of other arguments change accordingly. See documentation of bwplot for a fuller explanation.
dmath	A vectorized function that produces density values given a numeric vector named x, e.g., dnorm
args	list giving additional arguments to be passed to dmath
...	graphical parameters can be supplied. see function definition for details. Color can usually be specified by col, col.line and col.symbol, the last two overriding the first for lines and points respectively.

Details

panel.abline adds a line of the form y=a+bx or vertical and/or horizontal lines. Graphical parameters are obtained from reference.line for panel.grid, and add.line for the others (can be set using trellis.par.set)

panel.curve adds a curve, similar to what curve does with add = TRUE. Graphical parameters for the line are obtained from the add.line setting.

panel.linejoin treats one of x and y as a factor (according to the value of horizontal), calculates fun applied to the subsets of the other variable determined by each unique value of the factor, and joins them by a line. Can be used in conjunction with panel.xyplot and more

commonly with panel.superpose to produce interaction plots. See xyplot documentation for an example.

panel.mathdensity plots a (usually theoretical) probability density function. This can be useful in conjunction with histogram and densityplot to visually estimate goodness of fit (note, however, that qqmath is more suitable for this).

panel.rug adds a *rug* representation of the (marginal) data to the panel, much like rug.

panel.lmline(x, y) is equivalent to panel.abline(lm(y~x)).

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

loess.smooth, panel.axis, panel.identify identify, trellis.par.get

---

panel.histogram *Default Panel Function for histogram*

Description

This is the default panel function for histogram.

Usage

```
panel.histogram(x,
                breaks,
                equal.widths = TRUE,
                type = "density",
                nint = round(log2(length(x)) + 1),
                alpha, col, border, lty, lwd,
                ...)
```

Arguments

x	The data points for which the histogram is to be drawn
breaks	The breakpoints for the histogram
equal.widths	logical used when breaks==NULL
type	Type of histogram, possible values being "percent", "density" and "count"
nint	Number of bins for the histogram
alpha, col, border, lty, lwd	graphical parameters for bars; defaults are obtained from the plot.polygon settings.
...	other arguments, passed to hist when deemed appropriate

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

histogram

`panel.levelplot` *Default Panel Function for levelplot*

Description

This is the default panel function for `levelplot`.

Usage

```
panel.levelplot(x, y, z,
                subscripts,
                at = pretty(z),
                shrink,
                labels = NULL,
                label.style = c("mixed", "flat", "align"),
                contour = FALSE,
                region = TRUE,
                col = add.line$col,
                lty = add.line$lty,
                lwd = add.line$lwd,
                cex = add.text$cex,
                font = add.text$font,
                fontfamily = add.text$fontfamily,
                fontface = add.text$fontface,
                col.text = add.text$col,
                ...,
                col.regions = regions$col,
                alpha.regions = regions$alpha)
panel.contourplot(...)
```

Arguments

`x, y, z`	variables defining the plot
`subscripts`	integer vector indicating what subset of `x`, `y` and `z` to draw
`at`	numeric vector specifying breakpoints for change in colors
`shrink`	either a numeric vector of length 2 (meant to work as both x and y components), or a list with components x and y which are numeric vectors of length 2. This allows the rectangles to be scaled proportional to the z-value. The specification can be made separately for widths (x) and heights (y). The elements of the length 2 numeric vector gives the minimum and maximum proportion of shrinkage (corresponding to min and max of z).
`labels`	contour labels
`label.style`	controls choice of how label positions are determined.
`contour`	logical, specifying whether contour lines should be drawn
`region`	logical, specifying whether inter-contour regions should be filled with the appropriate color

`col, lty, lwd`
: graphical parameters for contour lines

`cex, col.text, font, fontfamily, fontface`
: graphical parameters for contour labels

`...`
: extra parameters

`col.regions`
: a vector of colors used if `region=TRUE`. Usually a subset of the colors are used, the exact number being one more than the length of `at`. These are chosen to be roughly equally spaced along `col.regions`. In the unusual case when `col.regions` is not longer than `at`, it is repeated to be as long as necessary.

`alpha.regions`
: numeric scalar controlling transparency of facets

Details

The same function is used for both `levelplot` and `contourplot` (which differ only in default values of some arguments). `panel.contourplot` is a simple wrapper to `panel.levelplot`.

When `contour=TRUE`, the `contourLines` function is used to calculate the contour lines.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

`levelplot`, `contourLines`

---

panel.number *Accessing Auxiliary Information During Plotting*

Description

Control over lattice plots are provided through a collection of user specifiable functions that perform various tasks during the plotting. Not all information is available to all functions. The functions documented here attempt to provide a consistent interface to access relevant information from within these user specified functions, namely those specified as the `panel`, `strip` and `axis` functions.

Usage

```
current.row()
current.column()
panel.number()
packet.number()
which.packet()

trellis.currentLayout(which = c("packet", "panel"))
```

Arguments

`which`
: whether return value (a matrix) should contain panel numbers or packet numbers, which are usually, but not necessarily, the same (see below for details).

Value

`trellis.currentLayout` returns a matrix with as many rows and columns as in the layout of panels in the current plot. Entries in the matrix are integer indices indicating which packet (or panel; see below) occupies that position, with 0 indicating the absence of a panel. `current.row` and `current.column` return integer indices specifying which row and column in the layout are currently active. `panel.number` returns an integer counting which panel is being drawn (starting from 1 for the first panel, a.k.a. the panel order). `packet.number` gives the packet number according to the packet order, which is determined by varying the first conditioning variable the fastest, then the second, and so on. `which.packet` returns the combination of levels of the conditioning variables in the form of a numeric vector as long as the number of conditioning variables, with each element an integer indexing the levels of the corresponding variable.

Note

The availability of these functions make redundant some features available in earlier versions of lattice, namely optional arguments called `panel.number` and `packet.number` that were made available to `panel` and `strip`. If you have written such functions, it should be enough to replace instances of `panel.number` and `packet.number` by the corresponding function calls. You should also remove `panel.number` and `packet.number` from the argument list of your function to avoid a warning.

If these accessor functions are not enough for your needs, feel free to contact the maintainer and ask for more.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

Lattice, xyplot

---

panel.pairs *Default Superpanel Function for splom*

Description

This is the default superpanel function for `splom`.

Usage

```
panel.pairs(z,
            panel = "panel.splom",
            lower.panel = panel,
            upper.panel = panel,
            diag.panel = "diag.panel.splom",
            as.matrix = FALSE,
            groups = NULL,
            panel.subscripts,
            subscripts,
            pscales = 5,
            prepanel.limits = function(x) if (is.factor(x)) levels(x) else
```

```
              extend.limits(range(as.numeric(x), finite = TRUE)),

              varname.col, varname.cex, varname.font,
              varname.fontfamily, varname.fontface,
              axis.text.col, axis.text.cex, axis.text.font,
              axis.text.fontfamily, axis.text.fontface,
              axis.line.col, axis.line.lty, axis.line.lwd,
              ...)
diag.panel.splom(x = NULL,
                 varname = NULL, limits, at = NULL, lab = NULL,
                 draw = TRUE,
                 varname.col, varname.cex,
                 varname.lineheight, varname.font,
                 varname.fontfamily, varname.fontface,
                 axis.text.col, axis.text.alpha,
                 axis.text.cex, axis.text.font,
                 axis.text.fontfamily, axis.text.fontface,
                 axis.line.col, axis.line.alpha,
                 axis.line.lty, axis.line.lwd,
                 ...)
```

Arguments

`z`	The data frame used for the plot.
`panel, lower.panel, upper.panel`	The panel function used to display each pair of variables. If specified, `lower.panel` and `upper.panel` are used for panels below and above the diagonal respectively.
`diag.panel`	The panel function used for the diagonals. See arguments to `diag.panel.splom` to know what arguments this function is passed when called.
`as.matrix`	logical. If `TRUE`, the layout of the panels will have origin on the top left instead of bottom left (similar to `pairs`). This is in essence the same functionality as provided by `as.table` for the panel layout
`groups`	Grouping variable, if any
`panel.subscripts`	logical specifying whether the panel function accepts an argument named `subscripts`.
`subscripts`	The indices of the rows of `z` that are to be displayed in this (super)panel.
`pscales`	Controls axis labels, passed down from `splom`. If `pscales` is a single number, it indicates the approximate number of equally-spaced ticks that should appear on each axis. If `pscales` is a list, it should have one component for each column in `z`, each of which itself a list with the following valid components: `at`: a numeric vector specifying tick locations `labels`: character vector labels to go with at `limits`: numeric 2-vector specifying axis limits (should be made more flexible at some point to handle factors) These are specifications on a per-variable basis, and used on all four sides in the diagonal cells used for labelling. Factor variables are labelled with the factor names. Use `pscales=0` to supress the axes entirely.

`prepanel.limits`
: The 'regular' high level lattice plots such as `xyplot` use the `prepanel` function for deciding on axis limits from data. This function serves a similar function, and works on a per-variable basis, by calculating the limits, which can be overridden by the corresponding `limits` component in the `pscales` list.

`x`
: data vector corresponding to that row / column (which will be the same for diagonal 'panels').

`varname`
: (scalar) character string or expression that is to be written centred within the panel

`limits`
: numeric of length 2, or, vector of characters, specifying the scale for that panel (used to calculate tick locations when missing)

`at`
: locations of tick marks

`lab`
: optional labels for tick marks

`draw`
: logical, specifying whether to draw the tick marks and labels. If `FALSE`, only variable names are written

`varname.col, varname.cex, varname.lineheight, varname.font, varname.fontfamily,`
: graphical parameters for the variable name in each diagonal panel

`axis.text.col, axis.text.cex, axis.text.font, axis.text.fontfamily, axis.text.fo`
: graphical parameters for axis tick marks and labels

`...`
: extra arguments passed on to `panel`, `lower.panel`, `upper.panel` and `diag.panel` from `panel.pairs`. Currently ignored by `diag.panel.splom`.

Details

`panel.pairs` is the function that is actually passed in as the `panel` function in a trellis object produced by splom (taking the `panel` function as its argument).

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

`splom`

---

panel.parallel *Default Panel Function for parallel*

Description

This is the default panel function for `parallel`.

Usage

```
panel.parallel(x, y, z, subscripts,
               groups = NULL,
               col, lwd, lty, alpha,
               common.scale = FALSE,
               lower,
               upper,
               ...)
```

Arguments

x, y	dummy variables, ignored.
z	The data frame used for the plot. Each column will be coerced to numeric before being plotted, and an error will be issued if this fails.
subscripts	The indices of the rows of z that are to be displyed in this panel.
groups	An optional grouping variable. If specified, different groups are distinguished by use of different graphical parameters (i.e., rows of z in the same group share parameters).
col, lwd, lty, alpha	graphical parameters (defaults to the settings for superpose.line). If groups is non-null, these parameters used one for each group. Otherwise, they are recycled and used to distinguish between rows of the data frame z.
common.scale	logical, whether a common scale should be used columns of z. Defaults to FALSE, in which case the horizontal range for each column is different (as determined by lower and upper).
lower, upper	numeric vectors replicated to be as long as the number of columns in z. Determines the lower and upper bounds to be used for scaling the corresponding columns of z after coercing them to numeric. Defaults to the minimum and maximum of each column.
...	other arguments (ignored)

Details

difficult to describe. See example for parallel

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

parallel

---

panel.qqmath *Default Panel and Prepanel Function for qqmath*

Description

This is the default panel function for qqmath.

Usage

```
panel.qqmath(x, f.value = NULL,
             distribution = qnorm,
             qtype = 7,
             groups = NULL, ...)
```

Arguments

x	vector (typically numeric, coerced if not) of data values to be used in the panel.
f.value, distribution	Defines how quantiles are calculated. See qqmath for details.
qtype	The type argument to be used in quantile
groups	An optional grouping variable. Within each panel, one Q-Q plot is produced for every level of this grouping variable, differentiated by different graphical parameters.
...	further arguments, often graphical parameters.

Details

Creates a Q-Q plot of the data and the theoretical distribution given by distribution. Note that most of the arguments controlling the display can be supplied directly to the high-level qqmath call.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

qqmath

---

panel.qqmathline *Useful panel function with qqmath*

Description

Useful panel function with qqmath. Draws a line passing through the points (usually) determined by the .25 and .75 quantiles of the sample and the theoretical distribution.

Usage

```
panel.qqmathline(x, y = x,
                 distribution = qnorm,
                 p = c(0.25, 0.75),
                 qtype = 7,
                 groups = NULL,
                 ...)
```

Arguments

x	The original sample, possibly reduced to a fewer number of quantiles, as determined by the f.value argument to qqmath
y	an alias for x for backwards compatibility
distribution	quantile function for reference theoretical distribution.
p	numeric vector of length two, representing probabilities. Corresponding quantile pairs define the line drawn.

qtype	the type of quantile computation used in quantile
groups	optional grouping variable. If non-null, a line will be drawn for each group.
...	other arguments.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

prepanel.qqmathline, qqmath, quantile

---

panel.stripplot *Default Panel Function for stripplot*

Description

This is the default panel function for stripplot. Also see panel.superpose

Usage

```
panel.stripplot(x, y, jitter.data = FALSE,
                factor = 0.5, amount = NULL,
                horizontal = TRUE, groups = NULL,
                ...)
```

Arguments

x,y	coordinates of points to be plotted
jitter.data	whether points should be jittered to avoid overplotting.
factor, amount	amount of jittering, see jitter
horizontal	logical. If FALSE, the plot is 'transposed' in the sense that the behaviours of x and y are switched. x is now the 'factor'. Interpretation of other arguments change accordingly. See documentation of bwplot for a fuller explanation.
groups	optional grouping variable
...	graphical parameters etc can be supplied, passed to panel.xyplot or panel.superpose, depending on whether groups is present

Details

Creates stripplot (one dimensional scatterplot) of x for each level of y

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

stripplot, jitter

`panel.superpose` *Panel Function for Display Marked by groups*

Description

These are panel functions for Trellis displays useful when a grouping variable is specified for use within panels. The `x` (and `y` where appropriate) variables are plotted with different graphical parameters for each distinct value of the grouping variable.

Usage

```
panel.superpose(x, y = NULL, subscripts, groups,
                panel.groups = "panel.xyplot",
                col, col.line, col.symbol,
                pch, cex, fill, font,
                fontface, fontfamily,
                lty, lwd, alpha,
                type = "p",
                ...,
                distribute.type = FALSE)
panel.superpose.2(..., distribute.type = TRUE)
```

Arguments

`x,y`	coordinates of the points to be displayed
`panel.groups`	the panel function to be used for each group of points. Defaults to `panel.xyplot` (behaviour in S). To be able to distinguish between different levels of the originating group inside `panel.groups`, it will be supplied a special argument called `group.number` which will hold the numeric code corresponding to the current level of `groups`. No special care needs to be taken when writing a `panel.groups` function if this feature is not used.
`subscripts`	subscripts giving indices in original data frame
`groups`	a grouping variable. Different graphical parameters will be used to plot the subsets of observations given by each distinct value of `groups`. The default graphical parameters are obtained from `superpose.symbol` and `superpose.line` using `trellis.par.get` wherever appropriate
`type`	usually a character vector specifying what should be drawn for each group, passed on to the `panel.groups` function, which must know what to do with it. By default, this is `panel.xyplot`, whose help page describes the admissible values. The functions `panel.superpose` and `panel.superpose.2` differ only in the default value of `distribute.type`, which controls the way the `type` argument is interpreted. If `distribute.type = FALSE`, then the interpretation is the same as for `panel.xyplot` for each of the unique groups. In other words, if `type` is a vector, all the individual components are honoured concurrently. If `distribute.type = TRUE`, `type` is replicated to be as long as the number of unique values in `groups`, and one component used for the points corresponding to the each different group. Even in this case, it is

possible to request multiple types per group, specifying `type` as a list, each component being the desired `type` vector for the corresponding group.

If `distribute.type = FALSE`, any occurrence of `"g"` in `type` causes a grid to be drawn, and all such occurrences are removed before `type` is passed on to `panel.groups`.

`col, col.line, col.symbol, pch, cex, fill, font, fontface, fontfamily, lty, lwd,`
: graphical parameters, replicated to be as long as the number of groups. These are eventually passed down to `panel.groups`, but as scalars rather than vectors. When `panel.groups` is called for the i-th level of `groups`, the corresponding element of each graphical parameter is passed to it.

`...`
: Extra arguments. Passed down to `panel.superpose` from `panel.superpose.2`, and to `panel.groups` from `panel.superpose`.

`distribute.type`
: logical controlling interpretation of the `type` argument.

Details

`panel.superpose` and `panel.superpose.2` differ essentially in how `type` is interpreted by default. The default behaviour in `panel.superpose` is the opposite of that in S, which is the same as that of `panel.superpose.2`.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩ (`panel.superpose.2` originally contributed by Neil Klepeis)

See Also

Different functions when used as `panel.groups` gives different types of plots, for example `panel.xyplot`, `panel.dotplot` and `panel.linejoin` (This can be used to produce interaction plots).

See `Lattice` for an overview of the package.

`panel.violin` *Panel Function to create Violin Plots*

Description

This is a panel function that can create a violin plot. It is typically used in a high-level call to `bwplot`.

Usage

```
panel.violin(x, y, box.ratio = 1, horizontal = TRUE,
             alpha, border, lty, lwd, col,
             varwidth = FALSE,
             bw, adjust, kernel, window,
             width, n = 50, from, to, cut,
             na.rm, ...)
```

Arguments

x, y	numeric vector or factor. Violin plots are drawn for each unique value of y (x) if horizontal is TRUE (FALSE)
box.ratio	ratio of height of a violin and inter violin space
horizontal	logical. If FALSE, the plot is 'transposed' in the sense that the behaviours of x and y are switched. x is now the 'factor'. See documentation of bwplot for a fuller explanation.
alpha, border, lty, lwd, col	graphical parameters controlling the violin. Defaults are taken from the "plot.polygon" settings.
varwidth	logical. If FALSE, the densities are scaled separately for each group, so that the maximum value of the density reaches the limit of the allocated space for each violin (as determined by box.ratio). If TRUE, densities across violins will have comparable scale.
bw, adjust, kernel, window, width, n, from, to, cut, na.rm	arguments to density, passed on as appropriate
...	arguments passed on to density.

Details

Creates Violin plot of x for every level of y. Note that most arguments controlling the display can be supplied to the high-level (typically bwplot) call directly.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

bwplot, density

Examples

```
bwplot(voice.part ~ height, singer,
       panel = function(..., box.ratio) {
           panel.violin(..., col = "transparent",
                        varwidth = FALSE, box.ratio = box.ratio)
           panel.bwplot(..., fill = NULL, box.ratio = .1)
       } )
```

---

panel.xyplot *Default Panel Function for xyplot*

Description

This is the default panel function for xyplot. Also see panel.superpose. The default panel functions for splom and qq are essentially the same function.

Usage

```
panel.xyplot(x, y, type = "p",
             groups = NULL,
             pch, col, col.line, col.symbol,
             font, fontfamily, fontface,
             lty, cex, fill, lwd,
             horizontal = FALSE, ...)
panel.splom(...)
panel.qq(...)
```

Arguments

x,y	variables to be plotted in the scatterplot
type	character vector consisting of one or more of the following: "p", "l", "h", "b", "o", "s", "S", "r", "a", "g", "smooth". If type has more than one element, an attempt is made to combine the effect of each of the components. The behaviour if any of the first six are included in type is similar to the effect of type in plot (type "b" is actually the same as "o"). "r" adds a regression line (same as panel.lmline, except for default graphical parameters), and "smooth" adds a lowess fit (same as panel.loess). "g" adds a reference grid using panel.grid in the background. "a" has the effect of calling panel.linejoin, which can be useful for creating interaction plots. The effect of several of these specifications depend on the value of horizontal. See example(xyplot) and demo(lattice) for examples.
groups	an optional grouping variable. If present, panel.superpose will be used instead to display each subgroup
col, col.line, col.symbol	default colours are obtained from plot.symbol and plot.line using trellis.par.get.
font, fontface, fontfamily	font used when pch is a character
pch, lty, cex, lwd, fill	other graphical parameters. fill serves the purpose of bg in points for certain values of pch
...	extra arguments, if any, for panel.xyplot. In most cases panel.xyplot ignores these. For types "r" and "smooth", these are passed on to panel.lmline and panel.loess respectively.
horizontal	logical. Controls orientation for certain type's, e.g. one of "h", "s" or "S"

.

Details

Creates scatterplot of x and y, with various modifications possible via the type argument. panel.qq draws a 45 degree line before calling panel.xyplot.

Note that most of the arguments controlling the display can be supplied directly to the high-level (e.g. xyplot) call.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

panel.superpose, xyplot, splom

`prepanel.functions` *Useful Prepanel Function for Lattice*

Description

These are predefined prepanel functions available in Lattice.

Usage

```
prepanel.lmline(x, y, ...)
prepanel.loess(x, y, span, degree, family, evaluation, ...)
prepanel.qqmathline(x, y = x, distribution = qnorm,
                    p = c(0.25, 0.75), qtype = 7,
                    groups, subscripts,
                    ...)
```

Arguments

`x, y`	x and y values, numeric or factor
`distribution`	quantile function for theoretical distribution. This is automatically passed in when this is used as a prepanel function in `qqmath`.
`qtype`	type of `quantile`
`p`	numeric vector of length two, representing probabilities. If used with `aspect="xy"`, the aspect ratio will be chosen to make the line passing through the corresponding quantile pairs as close to 45 degrees as possible.
`span, degree, family, evaluation`	arguments controlling the underlying `loess` smooth
`groups, subscripts`	See `xyplot`. Whenever appropriate, calculations are done separately for each group and then combined.
`...`	other arguments

Value

usually a list with components `xlim`, `ylim`, `dx` and `dy`, the first two being used to calculate panel axes limits, the last two for banking computations. The form of these components are described under `xyplot`. There are also several undocumented prepanel functions that serve as the default for high level functions, e.g., `prepanel.default.xyplot`

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

`trellis.par.get`, `xyplot`, `banking`, `Lattice`. See `loess.smooth` for further options to `prepanel.loess`

print.trellis *Plot and Summarize Trellis Objects*

Description

The print and plot methods produce a graph from a "trellis" object. The print method is necessary for automatic plotting. plot method is essentially an alias, provided for convenience. The summary, dim and dimnames methods

Usage

```
## S3 method for class 'trellis':
print(x, position, split,
      more = FALSE, newpage = TRUE,
      packet.panel = packet.panel.default,
      draw.in = NULL,
      panel.height = lattice.getOption("layout.heights")$panel,
      panel.width = lattice.getOption("layout.widths")$panel,
      save.object = lattice.getOption("save.object"),
      prefix,
      ...)
## S3 method for class 'trellis':
plot(x, ...)

## S3 method for class 'trellis':
summary(object, ...)

## S3 method for class 'trellis':
dim(x)
## S3 method for class 'trellis':
dimnames(x)
```

Arguments

x, object	an object of class "trellis"
position	a vector of 4 numbers, typically c(xmin, ymin, xmax, ymax) that give the lower-left and upper-right corners of a rectangle in which the Trellis plot of x is to be positioned. The coordinate system for this rectangle is [0-1] in both the x and y directions.
split	a vector of 4 integers, c(x,y,nx,ny) , that says to position the current plot at the x,y position in a regular array of nx by ny plots. (Note: this has origin at top left)
more	A logical specifying whether more plots will follow on this page.
newpage	A logical specifying whether the plot should be on a new page. This option is specific to lattice, and is useful for including lattice plots in an arbitrary grid viewport (see the details section).
packet.panel	a function that determines which packet (data subset) is plotted in which panel. Panels are always drawn in an order such that columns vary the fastest, then rows and then pages. This function determines, given the column, row and page and

other relevant information, the packet (if any) which should be used in that panel. By default, the association is determnined by matching panel order with packet order, which is determined by varying the first conditioning variable the fastest, then the second, and so on. This association rule is encoded in the default, namely the function `packet.panel.default`, whose help page details the arguments supplied to whichever function is specified as the `packet.panel` argument.

`draw.in` An optional (grid) viewport (used as the `name` argument in `downViewport`) in which the plot is to be drawn. If specified, the `newpage` argument is ignored. This feature is not well-tested.

`panel.width, panel.height`
lists with 2 components, that should be valid `x` and `units` arguments to `unit()` (the `data` argument cannot be specified currently, but can be considered for addition if needed). The resulting `unit` object will be the width/height of each panel in the Lattice plot. These arguments can be used to explicitly control the dimensions of the panel, rather than letting them expand to maximize available space. Vector widths are allowed, and can specify unequal lengths across rows or columns.

Note that this option should not be used in conjunction with non-default values of the `aspect` argument in the original high level call (no error will be produced, but the resulting behaviour is undefined).

`save.object` logical, specifying whether the object being printed is to be saved. The last object thus saved can be subsequently retrieved. This is an experimental feature that should allow access to a panel's data after the plot is done, making it possible to enhance the plot after the fact. This also allows the user to invoke the `update` method on the current plot, even if it was not assigned to a variable explicitly. For more details, see `trellis.focus`.

`prefix` character string used as a prefix in viewport and grob names, used to distinguish similar viewports if a page contains multiple plots. The default is based on the serial number of the current plot on the current page (which is one more than the number of plots that have been drawn on the page before the current plot). If supplied explicitly, this has to be a valid R symbol name (briefly, it must start with a letter or a period followed by a letter) and must not contain the grid path separator (currently `"::"`).

`...` extra arguments, ignored by the `print` method. All arguments to the `plot` method are passed on to the `print` method.

Details

This is the default print method for objects of class `"trellis"`, produced by calls to functions like `xyplot`, `bwplot` etc. It is usually called automatically when a trellis object is produced. It can also be called explicitly to control plot positioning by means of the arguments `split` and `position`.

When `newpage = FALSE`, the current grid viewport is treated as the plotting area, making it possible to embed a Lattice plot inside an arbitrary grid viewport. The `draw.in` argument provides an alternative mechanism that may be simpler to use.

The print method uses the information in `x` (the object to be printed) to produce a display using the Grid graphics engine. At the heart of the plot is a grid layout, of which the entries of most interest to the user are the ones containing the display panels.

Unlike in older versions of Lattice (and Grid), the grid display tree is retained after the plot is produced, making it possible to access individual viewport locations and make additions to the plot. For more details and a lattice level interface to these viewports, see `trellis.focus`.

Note

Unlike S-PLUS, trying to position a multipage display (using `position` and/or `split`) will mess things up.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

`Lattice`, `unit`, `update.trellis`, `trellis.focus`, `packet.panel.default`

Examples

```
p11 <- histogram( ~ height | voice.part, data = singer, xlab="Height")
p12 <- densityplot( ~ height | voice.part, data = singer, xlab = "Height")
p2 <- histogram( ~ height, data = singer, xlab = "Height")

## simple positioning by split
print(p11, split=c(1,1,1,2), more=TRUE)
print(p2, split=c(1,2,1,2))

## Combining split and position:
print(p11, position = c(0,0,.75,.75), split=c(1,1,1,2), more=TRUE)
print(p12, position = c(0,0,.75,.75), split=c(1,2,1,2), more=TRUE)
print(p2, position = c(.5,.75,1,1), more=FALSE)

## Using seekViewport

## repeat same plot, with different polynomial fits in each panel
xyplot(Armed.Forces ~ Year, longley, index.cond = list(rep(1, 6)),
       layout = c(3, 2),
       panel = function(x, y, ...)
       {
           panel.xyplot(x, y, ...)
           fm <- lm(y ~ poly(x, panel.number()))
           llines(x, predict(fm))
       })

## Not run:
grid::seekViewport(trellis.vpname("panel", 1, 1))
cat("Click somewhere inside the first panel:\n")
ltext(grid::grid.locator(), lab = "linear")
## End(Not run)

grid::seekViewport(trellis.vpname("panel", 1, 1))
grid::grid.text("linear")

grid::seekViewport(trellis.vpname("panel", 2, 1))
grid::grid.text("quadratic")
```

```
grid::seekViewport(trellis.vpname("panel", 3, 1))
grid::grid.text("cubic")

grid::seekViewport(trellis.vpname("panel", 1, 2))
grid::grid.text("degree 4")

grid::seekViewport(trellis.vpname("panel", 2, 2))
grid::grid.text("degree 5")

grid::seekViewport(trellis.vpname("panel", 3, 2))
grid::grid.text("degree 6")
```

---

qq *Quantile-Quantile Plots of Two Samples*

Description

Quantile-Quantile plots for comparing two Distributions

Usage

```
qq(x, data, ...)

## S3 method for class 'formula':
qq(x, data, aspect = "fill",
    panel = panel.qq, prepanel, scales, strip,
    groups, xlab, xlim, ylab, ylim, f.value = NULL,
    drop.unused.levels = lattice.getOption("drop.unused.levels"),
    ...,
    qtype = 7,
    default.scales = list(),
    subscripts,
    subset)
```

Arguments

x	The object on which method dispatch is carried out. For the formula method, a formula of the form `y ~ x | g1 * g2 * ...`, where `x` must be a numeric, and `y` can be a factor, shingle, character or numeric vector, with the restriction that there must be exactly two levels of `y`, which divide the values of `x` into two groups. Quantiles for these groups will be plotted along the two axes.
data	For the `formula` methods, an optional data frame in which variables in the formula (as well as `groups` and `subset`, if any) are to be evaluated.
f.value	optional numeric vector of probabilities, quantiles corresponding to which should be plotted. Can also be a function of a single integer (representing sample size) that returns such a numeric vector. The typical value for this argument is the function `ppoints`, which is also the S-PLUS default. If specified, the probabilities generated by this function is used for the plotted quantiles, using the `quantile` function.

`f.value` defaults to `NULL`, which is equivalent to using `function(n) ppoints(n, a = 1)`. This has the effect of including the minimum and maximum data values in the computed quantiles. This is similar to what happens for `qqplot` but different from the default `qq` behaviour in S-PLUS.

For large data, this argument can be useful in plotting a smaller set of quantiles, which is usually enough to capture the pattern.

`panel` The function that uses the packet (subset of display variables) corresponding to a panel to create a display. Default panel functions are documented separately, and often have arguments that can be used to customize its display in various ways. Such arguments can usually be directly supplied to the high level function.

`qtype` `type` argument for the `quantile`

`aspect, prepanel, scales, strip, groups, xlab, xlim, ylab, ylim, drop.unused.lev` See `xyplot`

`...` Further arguments. See corresponding entry in `xyplot` for non-trivial details.

Details

`qq` produces a Q-Q plot of two samples. The default behaviour of `qq` is different from the corresponding S-PLUS function. See the entry for `f.value` for specifics.

This and all other high level Trellis functions have several arguments in common. These are extensively documented only in the help page for `xyplot`, which should be consulted to learn more detailed usage.

Value

An object of class `"trellis"`. The `update` method can be used to update components of the object and the `print` method (usually called by default) will plot it on an appropriate plotting device.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

`xyplot`, `panel.qq`, `qqmath`, `Lattice`

Examples

```
qq(voice.part ~ height, aspect = 1, data = singer,
   subset = (voice.part == "Bass 2" | voice.part == "Tenor 1"))
```

---

qqmath *Q-Q Plot with Theoretical Distribution*

Description

Quantile-Quantile plot of a sample and a theoretical distribution

Usage

```
qqmath(x, data, ...)

## S3 method for class 'formula':
qqmath(x,
       data,
       allow.multiple = is.null(groups) || outer,
       outer = !is.null(groups),
       distribution = qnorm,
       f.value = NULL,
       auto.key = FALSE,
       aspect = "fill",
       panel = "panel.qqmath",
       prepanel = NULL,
       scales, strip, groups,
       xlab, xlim, ylab, ylim,
       drop.unused.levels = lattice.getOption("drop.unused.levels"),
       ...,
       default.scales = list(),
       subscripts,
       subset)
## S3 method for class 'numeric':
qqmath(x, data, ylab, ...)
```

Arguments

x	The object on which method dispatch is carried out. For the "formula" method, a formula of the form ~ x | g1 * g2 * ..., where x must be a numeric. For the "numeric" method, a numeric vector.
data	For the formula method, an optional data frame in which variables in the formula (as well as groups and subset, if any) are to be evaluated. Usualll ignored with a warning in other methods.
distribution	a quantile function that takes a vector of probabilities as argument and produces the corresponding quantiles. Possible values are qnorm, qunif etc. Distributions with other required arguments need to be passed in as user defined functions.
f.value	optional numeric vector of probabilities, quantiles corresponding to which should be plotted. Can also be a function of a single integer (representing sample size) that returns such a numeric vector. The typical value for this argument is the function ppoints, which is also the S-PLUS default. If specified, the probabilities generated by this function is used for the plotted quantiles, using the quantile function for the sample, and the function specified as the distribution argument for the theoretical distribution. f.value defaults to NULL, which has the effect of using ppoints for the quantiles of the theoretical distribution, but the exact data values for the sample. This is similar to what happens for qqnorm, but different from the S-PLUS default of f.value=ppoints. For large x, this argument can be useful in plotting a smaller set of quantiles, which is usually enough to capture the pattern.

`panel` The panel function to be used. Unlike in older versions, the default panel function does most of the actual computations and has support for grouping. See `panel.qqmath` for details.

`allow.multiple, outer, auto.key, aspect, prepanel, scales, strip, groups, xlab,` See `xyplot`

`...` Further arguments. See corresponding entry in `xyplot` for non-trivial details.

Details

`qqmath` produces a Q-Q plot of the given sample and a theoretical distribution. The default behaviour of `qqmath` is different from the corresponding S-PLUS function, but is similar to `qqnorm`. See the entry for `f.value` for specifics.

The implementation details are also different from S-PLUS. In particular, all the important calculations are done by the panel (and prepanel function) and not `qqmath` itself. In fact, both the arguments `distribution` and `f.value` are passed unchanged to the panel and prepanel function. This allows, among other things, display of grouped Q-Q plots, which are often useful. See the help page for `panel.qqmath` for further details.

This and all other high level Trellis functions have several arguments in common. These are extensively documented only in the help page for `xyplot`, which should be consulted to learn more detailed usage.

Value

An object of class `"trellis"`. The `update` method can be used to update components of the object and the `print` method (usually called by default) will plot it on an appropriate plotting device.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

`xyplot`, `panel.qqmath`, `panel.qqmathline`, `prepanel.qqmathline`, `Lattice`, `quantile`

Examples

```
qqmath(~ rnorm(100), distribution = function(p) qt(p, df = 10))
qqmath(~ height | voice.part, aspect = "xy", data = singer,
       prepanel = prepanel.qqmathline,
       panel = function(x, ...) {
          panel.qqmathline(x, ...)
          panel.qqmath(x, ...)
       })
vp.comb <-
    factor(sapply(strsplit(as.character(singer$voice.part), split = " "),
                  "[", 1),
           levels = c("Bass", "Tenor", "Alto", "Soprano"))
vp.group <-
    factor(sapply(strsplit(as.character(singer$voice.part), split = " "),
                  "[", 2))
qqmath(~ height | vp.comb, data = singer,
       groups = vp.group, auto.key = list(space = "right"),
```

```
        aspect = "xy",
        prepanel = prepanel.qqmathline,
        panel = function(x, ...) {
           panel.qqmathline(x, ...)
           panel.qqmath(x, ...)
        })
```

`rfs` *Residual and Fit Spread Plots*

Description

Plots fitted values and residuals (via qqmath) on a common scale for any object that has methods for fitted values and residuals.

Usage

```
rfs(model, layout=c(2, 1), xlab="f-value", ylab=NULL,
    distribution = qunif,
    panel, prepanel, strip, ...)
```

Arguments

`model`	a fitted model object with methods `fitted.values` and `residuals`. Can be the value returned by `oneway`
`layout`	default layout is c(2,1)
`xlab`	defaults to `"f.value"`
`distribution`	the distribution function to be used for `qqmath`
`ylab, panel, prepanel, strip`	See `xyplot`
`...`	other arguments, passed on to `qqmath`.

Value

An object of class `"trellis"`. The `update` method can be used to update components of the object and the `print` method (usually called by default) will plot it on an appropriate plotting device.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

`oneway`, `qqmath`, `xyplot`, `Lattice`

Examples

```
rfs(oneway(height ~ voice.part, data = singer, spread = 1), aspect = 1)
```

Rows *Extract rows from a list*

Description

Convenience function to extract subset of a list. Usually used in creating keys.

Usage

```
Rows(x, which)
```

Arguments

`x`	list with each member a vector of the same length
`which`	index for members of `x`

Value

A list similar to `x`, with each `x[[i]]` replaced by `x[[i]][which]`

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

`xyplot`, `Lattice`

shingles *shingles*

Description

Functions to handle shingles

Usage

```
shingle(x, intervals=sort(unique(x)))
equal.count(x, ...)
as.shingle(x)
is.shingle(x)

## S3 method for class 'shingle':
plot(x, panel, xlab, ylab, ...)

## S3 method for class 'shingle':
print(x, showValues = TRUE, ...)

## S3 method for class 'shingleLevel':
as.character(x, ...)
```

```
## S3 method for class 'shingleLevel':
print(x, ...)

## S3 method for class 'shingle':
summary(object, showValues = FALSE, ...)

## S3 method for class 'shingle':
x[subset, drop = FALSE]
as.factorOrShingle(x, subset, drop)
```

Arguments

x	numeric variable or R object, shingle in `plot.shingle` and `x[]`. An object (list of intervals) of class "shingleLevel" in `print.shingleLevel`
object	shingle object to be summarized
showValues	logical, whether to print the numeric part. If FALSE, only the intervals are printed
intervals	numeric vector or matrix with 2 columns
subset	logical vector
drop	whether redundant shingle levels are to be dropped
panel, xlab, ylab	standard Trellis arguments (see `xyplot`)
...	other arguments, passed down as appropriate. For example, extra arguments to `equal.count` are passed on to `co.intervals`. graphical parameters can be passed as arguments to the `plot` method.

Details

A shingle is a data structure used in Trellis, and is a generalization of factors to 'continuous' variables. It consists of a numeric vector along with some possibly overlapping intervals. These intervals are the 'levels' of the shingle. The `levels` and `nlevels` functions, usually applicable to factors, also work on shingles. The implementation of shingles is slightly different from S.

There are print methods for shingles, as well as for printing the result of `levels()` applied to a shingle. For use in labelling, the `as.character` method can be used to convert levels of a shingle to character strings.

`equal.count` converts `x` to a shingle using the equal count algorithm. This is essentially a wrapper around `co.intervals`. All arguments are passed to `co.intervals`.

`shingle` creates a shingle using the given `intervals`. If `intervals` is a vector, these are used to form 0 length intervals.

`as.shingle` returns `shingle(x)` if `x` is not a shingle.

`is.shingle` tests whether `x` is a shingle.

`plot.shingle` displays the ranges of shingles via rectangles. `print.shingle` and `summary.shingle` describe the shingle object.

Value

`x$intervals` for `levels.shingle(x)`, logical for `is.shingle`, an object of class `"trellis"` for `plot` (printed by default by `print.trellis`), and an object of class `"shingle"` for the others.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

`xyplot`, `co.intervals`, `Lattice`

Examples

```
z <- equal.count(rnorm(50))
plot(z)
print(z)
print(levels(z))
```

`simpleKey` *Function to generate a simple key*

Description

Simple interface to generate a list appropriate for `draw.key`

Usage

```
simpleKey(text, points = TRUE,
          rectangles = FALSE,
          lines = FALSE,
          col, cex, font,
          fontface, fontfamily,
          lineheight, ...)
```

Arguments

`text`	character or expression vector, to be used as labels for levels of the grouping variable
`points`	logical
`rectangles`	logical
`lines`	logical
`col, cex, font, fontface, fontfamily, lineheight`	Used as top-level components of the list produced, to be used for the text labels. Defaults to the values in `trellis.par.get("add.text")`
`...`	further arguments added to the list, eventually passed to `draw.key`

Details

A lattice plot can include a legend (key) if an appropriate list is specified as the `key` argument to a high level Lattice function such as `xyplot`. This key can be very flexible, but that flexibility comes at the cost of this list being very complicated even in simple situations. The `simpleKey` function is a shortcut, which assumes that the key is being drawn in conjunction with the `groups` argument, and that the default Trellis settings are being used. At most one each of points, rectangles and lines can be drawn.

See also the `auto.key` argument for high level plots.

Value

A list that would work as the `key` argument to `xyplot` etc.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

`draw.key`, `xyplot`, `Lattice`

`singer` *Heights of New York Choral Society singers*

Description

Heights in inches of the singers in the New York Choral Society in 1979. The data are grouped according to voice part. The vocal range for each voice part increases in pitch according to the following order: Bass 2, Bass 1, Tenor 2, Tenor 1, Alto 2, Alto 1, Soprano 2, Soprano 1.

Usage

```
singer
```

Format

A data frame with 235 observations on the following 2 variables.

height Height in inches of the singers.

voice.part (Unordered) factor with levels `"Bass 2"`, `"Bass 1"`, `"Tenor 2"`, `"Tenor 1"`, `"Alto 2"`, `"Alto 1"`, `"Soprano 2"`, `"Soprano 1"`.

Author(s)

Documentation contributed by Kevin Wright.

Source

Chambers, J.M., W. S. Cleveland, B. Kleiner, and P. A. Tukey. (1983). *Graphical Methods for Data Analysis*. Chapman and Hall, New York.

References

Cleveland, William S. (1993). *Visualizing Data*. Hobart Press, Summit, New Jersey.

Examples

```
# Separate histogram for each voice part (Figure 1.2 from Cleveland)
histogram(~ height | voice.part,
          data = singer,
          aspect=1,
          layout = c(2, 4),
          nint=15,
          xlab = "Height (inches)")

# Quantile-Quantile plot (Figure 2.11 from Cleveland)
qqmath(~ height | voice.part,
       data=singer,
       aspect=1,
       layout=c(2,4),
       prepanel = prepanel.qqmathline,
       panel = function(x, ...) {
         panel.grid()
         panel.qqmathline(x, ...)
         panel.qqmath(x, ...)
       },
       xlab = "Unit Normal Quantile",
       ylab="Height (inches)")
```

---

splom *Scatter Plot Matrices*

Description

Draw Conditional Scatter Plot Matrices and Parallel Coordinate Plots

Usage

```
splom(x, data, ...)
parallel(x, data, ...)

## S3 method for class 'formula':
splom(x,
      data,
      auto.key = FALSE,
      aspect = 1,
      between = list(x = 0.5, y = 0.5),
      panel = if (is.null(groups)) "panel.splom" else "panel.superpose",
      prepanel,
      scales,
      strip,
      groups,
      xlab,
      xlim,
      ylab = NULL,
      ylim,
      superpanel = "panel.pairs",
      pscales = 5,
```

```
       varnames,
       drop.unused.levels,
       ...,
       default.scales,
       subset = TRUE)
## S3 method for class 'formula':
parallel(x,
         data,
         aspect = "fill",
         between = list(x = 0.5, y = 0.5),
         panel = "panel.parallel",
         prepanel,
         scales,
         strip,
         groups,
         xlab = NULL,
         xlim,
         ylab = NULL,
         ylim,
         varnames,
         drop.unused.levels,
         ...,
         default.scales,
         subset = TRUE)

## S3 method for class 'data.frame':
splom(x, data = NULL, ...)
## S3 method for class 'matrix':
splom(x, data = NULL, ...)

## S3 method for class 'matrix':
parallel(x, data = NULL, ...)
## S3 method for class 'data.frame':
parallel(x, data = NULL, ...)
```

Arguments

`x`	The object on which method dispatch is carried out. For the `"formula"` method, a formula describing the structure of the plot, which should be of the form `~ x | g1 * g2 * ...`, where `x` is a data frame or matrix. Each of `g1,g2,...` must be either factors or shingles. The conditioning variables `g1, g2, ...` may be omitted. For the `data.frame` methods, a data frame.
`data`	For the `formula` methods, an optional data frame in which variables in the formula (as well as `groups` and `subset`, if any) are to be evaluated.
`aspect`	aspect ratio of each panel (and subpanel), square by default for `splom`.
`between`	to avoid confusion between panels and subpanels, the default is to show the panels of a splom plot with space between them.
`panel`	Usual interpretation for `parallel`, namely the function that creates the display within each panel. For `splom`, the terminology is slightly complicated. The role played by the panel function in most other high-level functions is played here by the

superpanel function, which is responsible for the display for each conditional data subset. panel is simply an argument to the default superpanel function panel.pairs, and is passed on to it unchanged. It is used there to create each pairwise display. See panel.pairs for more useful options.

superpanel — function that sets up the splom display, by default as a scatterplot matrix.

pscales — a numeric value or a list, meant to be a less functional substitute for the scales argument in xyplot etc. This argument is passed to the superpanel function, and is handled by the default superpanel function panel.pairs. The help page for the latter documents this argument in more detail.

varnames — character vector giving the names of the p variables in x. By default, the column names of x.

auto.key, prepanel, scales, strip, groups, xlab, xlim, ylab, ylim, drop.unused.l — See xyplot

... — Further arguments. See corresponding entry in xyplot for non-trivial details.

Details

splom produces Scatter Plot Matrices. The role usually played by panel is taken over by superpanel, which determines how the columns of x are to be arranged for pairwise plots. The only available option currently is panel.pairs.

Many of the finer customizations usually done via arguments to high level function like xyplot are instead done by panel.pairs for splom. These include control of axis limits, tick locations and prepanel calcultions. If you are trying to fine-tune your splom plot, definitely look at the panel.pairs help page. The scales argument is usually not very useful in splom, and trying to change it may have undesired effects.

parallel draws Parallel Coordinate Plots. (Difficult to describe, see example.)

These and all other high level Trellis functions have several arguments in common. These are extensively documented only in the help page for xyplot, which should be consulted to learn more detailed usage.

Value

An object of class "trellis". The update method can be used to update components of the object and the print method (usually called by default) will plot it on an appropriate plotting device.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

xyplot, Lattice, panel.pairs

Examples

```
super.sym <- trellis.par.get("superpose.symbol")
splom(~iris[1:4], groups = Species, data = iris,
      panel = panel.superpose,
      key = list(title = "Three Varieties of Iris",
                 columns = 3,
                 points = list(pch = super.sym$pch[1:3],
```

```
                col = super.sym$col[1:3]),
                text = list(c("Setosa", "Versicolor", "Virginica"))))
splom(~iris[1:3]|Species, data = iris,
      layout=c(2,2), pscales = 0,
      varnames = c("Sepal\nLength", "Sepal\nWidth", "Petal\nLength"),
      page = function(...) {
          ltext(x = seq(.6, .8, len = 4),
                y = seq(.9, .6, len = 4),
                lab = c("Three", "Varieties", "of", "Iris"),
                cex = 2)
      })
parallel(~iris[1:4] | Species, iris)
```

`strip.default` *Default Trellis Strip Function*

Description

`strip.default` is the function that draws the strips by default in Trellis plots. Users can write their own strip functions, but most commonly this involves calling `strip.default` with a slightly different arguments. `strip.custom` provides a convenient way to obtain new strip functions that differ from `strip.default` only in the default values of certain arguments.

Usage

```
strip.default(which.given,
              which.panel,
              var.name,
              factor.levels,
              shingle.intervals,
              strip.names = c(FALSE, TRUE),
              strip.levels = c(TRUE, FALSE),
              sep = " : ",
              style = 1,
              horizontal = TRUE,
              bg = trellis.par.get("strip.background")$col[which.given],
              fg = trellis.par.get("strip.shingle")$col[which.given],
              par.strip.text = trellis.par.get("add.text"))
strip.custom(...)
```

Arguments

`which.given`	integer index specifying which of the conditioning variables this strip corresponds to.
`which.panel`	vector of integers as long as the number of conditioning variables. The contents are indices specifying the current levels of each of the conditioning variables (thus, this would be unique for each distinct packet). This is identical to the return value of `which.packet`, which is a more accurate name.
`var.name`	vector of character strings or expressions as long as the number of conditioning variables. The contents are interpreted as names for the conditioning variables. Whether they are shown on the strip depends on the values of `strip.names` and `style` (see below). By default, the names are shown for shingles, but not for factors.

`factor.levels`
: vector of character strings or expressions giving the levels of the conditioning variable currently being drawn. For more than one conditioning variable, this will vary with `which.given`. Whether these levels are shown on the strip depends on the values of `strip.levels` and `style` (see below). `factor.levels` may be specified for both factors and shingles (despite the name), but by default they are shown only for factors. If shown, the labels may optionally be abbreviated by specicifying suitable components in `par.strip.text` (see `xyplot`)

`shingle.intervals`
: if the current strip corresponds to a shingle, this should be a 2-column matrix giving the levels of the shingle. (of the form that would be produced by **printing** `levels(shingle)`). Otherwise, it should be `NULL`

`strip.names`
: a logical vector of length 2, indicating whether or not the name of the conditioning variable that corresponds to the strip being drawn is to be written on the strip. The two components give the values for factors and shingles respectively.

  This argument is ignored for a factor when `style` is not one of 1 and 3.

`strip.levels`
: a logical vector of length 2, indicating whether or not the level of the conditioning variable that corresponds to the strip being drawn is to be written on the strip. The two components give the values for factors and shingles respectively.

`sep`
: character or expression, serving as a separator if the name and level are both to be shown.

`style`
: integer, with values 1, 2, 3, 4 and 5 currently supported, controlling how the current level of a factor is encoded. Ignored for shingles (actually, when `shingle.intervals` is non-null.

  The best way to find out what effect the value of `style` has is to try them out. Here is a short description: for a style value of 1, the strip is colored in the background color with the strip text (as determined by other arguments) centered on it. A value of 3 is the same, except that a part of the strip is colored in the foreground color, indicating the current level of the factor. For styles 2 and 4, the part corresponding to the current level remains colored in the foreground color, however, for style = 2, the remaining part is not colored at all, whereas for 4, it is colored with the background color. For both these, the names of all the levels of the factor are placed on the strip from left to right. Styles 5 and 6 produce the same effect (they are subtly different in S, this implementation corresponds to 5), they are similar to style 1, except that the strip text is not centered, it is instead positioned according to the current level.

  Note that unlike S-PLUS, the default value of `style` is 1. `strip.names` and `strip.levels` have no effect if `style` is not 1 or 3.

`horizontal`
: logical, specifying whether the labels etc should be horizontal. `horizontal=FALSE` is useful for strips on the left of panels using `strip.left=TRUE`

`par.strip.text`
: list with parameters controlling the text on each strip, with components `col`, `cex`, `font`, etc.

`bg`
: strip background color.

`fg`
: strip foreground color.

`...`
: arguments to be passed on to `strip.default`, overriding whatever value it would have normally assumed

Details

default strip function for trellis functions. Useful mostly because of the `style` argument — non-default styles are often more informative, especially when the names of the levels of the factor `x` are small. Traditional use is as `strip = function(...) strip.default(style=2,...)`, though this can be simplified by the use of `strip.custom`.

Value

`strip.default` is called for its side-effect, which is to draw a strip appropriate for multi-panel Trellis conditioning plots. `strip.custom` returns a function that is similar to `strip.default`, but with different defaults for the arguments specified in the call.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

`xyplot`, `Lattice`

Examples

```
## Traditional use
xyplot(Petal.Length ~ Petal.Width | Species, iris,
       strip = function(..., style) strip.default(..., style = 4))

## equivalent call using strip.custom
xyplot(Petal.Length ~ Petal.Width | Species, iris,
       strip = strip.custom(style = 4))

xyplot(Petal.Length ~ Petal.Width | Species, iris,
       strip = FALSE,
       strip.left = strip.custom(style = 4, horizontal = FALSE))
```

`tmd` *Tukey Mean-Difference Plot*

Description

`tmd` Creates Tukey Mean-Difference Plots from a trellis object returned by `xyplot`, `qq` or `qqmath`. The prepanel and panel functions are used as appropriate. The `formula` method for `tmd` is provided for convenience, and simply calls `tmd` on the object created by calling `xyplot` on that formula.

Usage

```
tmd(object, ...)

## S3 method for class 'trellis':
tmd(object,
    xlab = "mean",
```

```
    ylab = "difference",
    panel,
    prepanel,
    ...)

prepanel.tmd.qqmath(x,
                    f.value = NULL,
                    distribution = qnorm,
                    qtype = 7,
                    groups = NULL,
                    subscripts, ...)
panel.tmd.qqmath(x,
                    f.value = NULL,
                    distribution = qnorm,
                    qtype = 7,
                    groups = NULL,
                    subscripts, ...)
panel.tmd.default(x, y, groups = NULL, ...)
prepanel.tmd.default(x, y, ...)
```

Arguments

`object`	An object of class `"trellis"` returned by `xyplot`, `qq` or `qqmath`.
`xlab`	x label
`ylab`	y label
`panel`	panel function to be used. See details below.
`prepanel`	prepanel function. See details below.
`f.value, distribution, qtype`	see `panel.qqmath`.
`groups, subscripts`	see `xyplot`.
`x, y`	data as passed to panel functions in original call.
`...`	other arguments

Details

The Tukey Mean-difference plot is produced by modifying the (x,y) values of each panel as follows: the new coordinates are given by `x=(x+y)/2` and `y=y-x`, which are then plotted. The default panel function(s) add a reference line at `y=0` as well.

`tmd` acts on the a `"trellis"` object, not on the actual plot this object would have produced. As such, it only uses the arguments supplied to the panel function in the original call, and completely ignores what the original panel function might have done with this data. `tmd` uses these panel arguments to set up its own scales (using its `prepanel` argument) and display (using `panel`). It is thus important to provide suitable prepanel and panel functions to `tmd` depending on the original call.

Such functions currently exist for `xyplot`, `qq` (the ones with `default` in their name) and `qqmath`, as listed in the usage section above. These assume the default displays for the corresponding high-level call. If unspecified, the codeprepanel and `panel` arguments default to suitable choices.

`tmd` uses the `update` method for `"trellis"` objects, which processes all extra arguments supplied to `tmd`.

Value

An object of class "trellis". The update method can be used to update components of the object and the print method (usually called by default) will plot it on an appropriate plotting device.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

qq, qqmath, xyplot, Lattice

Examples

```
tmd(qqmath(~height | voice.part, data = singer))
```

---

trellis.device *Initializing Trellis Displays*

Description

Initialization of a display device with appropriate graphical parameters.

Usage

```
trellis.device(device = getOption("device"),
               color = !(dev.name == "postscript"),
               theme = lattice.getOption("default.theme"),
               new = TRUE,
               retain = FALSE,
               ...)
canonical.theme(name, color)
col.whitebg()
```

Arguments

device	function (or the name of one as a character string) that starts a device. Admissible values depend on the platform and how R was compiled (see Devices), but usually "pdf", "postscript", "png", "jpeg" and at least one of "X11", "windows" and "quartz" will be available.
color	logical, whether the display should be color or black and white. Defaults to FALSE for postscript devices, TRUE otherwise.
theme	list of components that changes the settings of the device opened, or, a function that when called produces such a list. The function name can be supplied as a quoted string. These settings are only used to modify the default settings (determined by other arguments), and need not contain all possible parameters. A possible use of this argument is to change the default settings by specifying lattice.options(default.theme = "col.whitebg"). For back-compatibility, this is initially (when lattice is loaded) set to options(lattice.theme).

If theme is a function, it will not be supplied any arguments, however, it is guaranteed that a device will already be open when it is called, so one may use .Device inside the function to ascertain what device has been opened.

new — logical flag indicating whether a new device should be started. If FALSE, the options for the current device are changed to the defaults determined by the other arguments.

retain — logical. If TRUE and a setting for this device already exists, then that is used instead of the defaults for this device. By default, pre-existing settings are overwritten (and lost).

name — name of the device for which the setting is required, as returned by .Device

... — additional parameters to be passed to the device function, most commonly file for non-screen devices, as well as height, width, etc. See the help file for individual devices for admissible arguments.

Details

Trellis Graphics functions obtain the default values of various graphical parameters (colors, line types, fonts, etc.) from a customizable "settings" list. This functionality is analogous to par for standard R graphics and, together with lattice.options, mostly supplants it (par settings are mostly ignored by Lattice). Unlike par, Trellis settings can be controlled separately for each different device. canonical.theme and col.whitebg produce predefined settings (a.k.a. themes), while trellis.device provides a high level interface to control which "theme" will be in effect when a new device is opened. trellis.device is called automatically when a "trellis" object is plotted, and the defaults can be used to provide sufficient control, so in a properly configured system it is rarely necessary for the user to call trellis.device explicitly.

The canonical.theme function is intended to provide device specific settings (e.g. light colors on a grey background for screen devices, dark colors or black and white for print devices) which were used as defaults prior to R 2.3.0. However, these defaults are not always appropriate, due to the variety of platforms and hardware settings on which R is used, as well as the fact that a plot created on a particular device may be subsequently used in many different ways. For this reason, a "safe" default is used for all devices from R 2.3.0 onwards. The old behaviour can be reinstated by setting canonical.theme as the default theme argument, e.g. by putting options(lattice.theme = "canonical.theme") in a startup script (see the entry for theme above for details).

Value

canonical.theme returns a list of components defining graphical parameter settings for Lattice displays. It is used internally in trellis.device, and can also be used as the theme argument to trellis.par.set, or even as theme in trellis.device to use the defaults for another device.

col.whitebg returns a similar (but smaller) list that is suitable as the theme argument to trellis.device and trellis.par.set. It contains settings values which provide colors suitable for plotting on a white background. Note that the name col.whitebg is somewhat a misnomer, since it actually sets the background to transparent rather than white.

Note

Earlier versions of trellis.device had a bg argument to set the background color, but this is no longer supported. If supplied, the bg argument will be passed on to the device function; however, this will have no effect on the Trellis settings. It is rarely meaningful to change the background alone; if you feel the need to change the background, consider using the theme argument instead.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

`Lattice` for an overview of the `lattice` package.

`Devices` for valid choices of `device` on your platform.

`trellis.par.get` and `trellis.par.set` can be used to query and modify the settings *after* a device has been initialized. The `par.settings` argument to high level functions, described in `xyplot`, can be used to attach transient settings to a `"trellis"` object.

`trellis.object` *A Trellis Plot Object*

Description

This class of objects is returned by high level lattice functions, and is usually plotted by default by its `print` method.

Details

A trellis object, as returned by high level lattice functions like `xyplot`, is a list with the `"class"` attribute set to `"trellis"`. Many of the components of this list are simply the arguments to the high level function that produced the object. Among them are: `as.table`, `layout`, `page`, `panel`, `prepanel`, `main`, `sub`, `par.settings`, `par.strip.text`, `strip`, `skip`, `xlab` and `ylab`. Some other typical components are:

`formula` the Trellis formula used in the call

`index.cond` list with index for each of the conditioning variables

`perm.cond` permutation of the order of the conditioning variables

`aspect.fill` logical, whether `aspect` is `"fill"`

`aspect.ratio` numeric, aspect ratio to be used if `aspect.fill` is `FALSE`

`call` call that generated the object.

`condlevels` list with levels of the conditioning variables

`legend` list describing the legend(s) to be drawn

`panel.args` a list as long as the number of panels, each element being a list itself, containing the arguments in named form to be passed to the panel function in that panel.

`panel.args.common` a list containing the arguments common to all the panel functions in `name=value` form

`x.scales` list describing x-scale, can consist of several other lists, paralleling panel.args, if x-relation is not `"same"`

`y.scales` list describing y-scale, similar to `x.scales`

`x.between` numeric vector of interpanel x-space

`y.between` numeric vector of interpanel y-space

`x.limits` numeric vector of length 2 or list, giving x-axis limits

`y.limits` similar to `x.limits`

`packet.sizes` array recording the number of observations in each packet

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

Lattice, xyplot, print.trellis

---

trellis.par.get *Graphical Parameters for Trellis Displays*

Description

Functions used to query, display and modify graphical parameters for fine control of Trellis displays. Modifications are made to the settings for the currently active device only.

Usage

```
trellis.par.set(name, value, ..., theme, warn = TRUE)
trellis.par.get(name = NULL)
show.settings(x = NULL)
```

Arguments

name	character giving the name of a component. If unspecified, names(trellis.par.get()) returns a list containing all the current settings (this can be used to get the valid values for name)
value	a list giving the desired value of the component. Components that are already defined as part of the current settings but are not mentioned in value will remain unchanged.
theme	a list decribing how to change the settings, similar to what is returned by trellis.par.get(). This is purely for convenience, allowing multiple calls to trellis.par.set to be condensed into one. The name of each component must be a valid name as described above, with the corresponding value a valid value as described above. As in trellis.device, theme can also be a function that produces such a list when called. The function name can be supplied as a quoted string.
...	Multiple settings can be specified in name = value form. Equivalent to calling with theme = list(...)
warn	logical, indicating whether a warning should be issued when trellis.par.get is called when no graphics device is open
x	optional list of components that change the settings (any valid value of theme). These are used to modify the current settings (obtained by trellis.par.get) before they are displayed.

Details

The various graphical parameters (color, line type, background etc) that control the look and feel of Trellis displays are highly customizable. Also, R can produce graphics on a number of devices, and it is expected that a different set of parameters would be more suited to different devices. These parameters are stored internally in a variable named `lattice.theme`, which is a list whose components define settings for particular devices. The components are idenified by the name of the device they represent (as obtained by `.Device`), and are created as and when new devices are opened for the first time using `trellis.device` (or Lattice plots are drawn on a device for the first time in that session).

The initial settings for each device defaults to values appropriate for that device. In practice, this boils down to three distinct settings, one for screen devices like `x11` and `windows`, one for black and white plots (mostly useful for `postscript`) and one for color printers (color `postcript`, `pdf`).

Once a device is open, its settings can be modified. When another instance of the same device is opened later using `trellis.device`, the settings for that device are reset to its defaults, unless otherwise specified in the call to `trellis.device`. But settings for different devices are treated separately, i.e., opening a postscript device will not alter the x11 settings, which will remain in effect whenever an x11 device is active.

The functions `trellis.par.*` are meant to be interfaces to the global settings. They always apply on the settings for the currently ACTIVE device.

`trellis.par.get`, called without any arguments, returns the full list of settings for the active device. With the `name` argument present, it returns that component only. `trellis.par.get` sets the value of the `name` component of the current active device settings to `value`.

`trellis.par.get` is usually used inside trellis functions to get graphical parameters before plotting. Modifications by users via `trellis.par.set` is traditionally done as follows:

```
add.line <- trellis.par.get("add.line")
add.line$col <- "red"
trellis.par.set("add.line", add.line)
```

More convenient (but not S compatible) ways to do this are

```
trellis.par.set(list(add.line = list(col = "red")))
```

and

```
trellis.par.set(add.line = list(col = "red"))
```

The actual list of the components in `trellis.settings` has not been finalized, so I'm not attempting to list them here. The current value can be obtained by `print(trellis.par.get())`. Most names should be self-explanatory.

`show.settings` provides a graphical display summarizing some of the values in the current settings.

Value

`trellis.par.get` returns a list giving parameters for that component. If `name` is missing, it returns the full list.

Note

In some ways, `trellis.par.get` and `trellis.par.set` together are a replacement for the `par` function used in traditional R graphics. In particular, changing `par` settings has little (if any) effect on lattice output. Since lattice plots are implemented using Grid graphics, its parameter

system *does* have an effect unless overridden by a suitable lattice parameter setting. Such parameters can be specified as part of a lattice theme by including them in the `grid.pars` component (see `gpar` for a list of valid parameter names).

One of the uses of `par` is to set `par(ask = TRUE)` making R wait for user input before starting a new graphics page. For Grid graphics, this is done using `grid.prompt`. Lattice has no separate interface for this, and the user must call `grid.prompt` directly. If the grid package is not attached (lattice itself only loads the grid namespace), this may be done using `grid::grid.prompt(TRUE)`.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

`trellis.device`, `Lattice`, `grid.prompt`, `gpar`

Examples

```
show.settings()
```

`update.trellis` *Retrieve and Update Trellis Object*

Description

Update method for objects of class `"trellis"`, and a way to retrieve the last printed trellis object (that was saved).

Usage

```
## S3 method for class 'trellis':
update(object,
       panel,
       aspect,
       as.table,
       between,
       key,
       auto.key,
       legend,
       layout,
       main,
       page,
       par.strip.text,
       prepanel,
       scales,
       skip,
       strip,
       strip.left,
       sub,
       xlab,
```

```
        xlim,
        ylab,
        ylim,
        par.settings,
        index.cond,
        perm.cond,
        ...)

## S3 method for class 'trellis':
t(x)

## S3 method for class 'trellis':
x[i, j, ..., drop = FALSE]

trellis.last.object(warn = TRUE, ...)
```

Arguments

`object, x`	The object to be updated, of class `"trellis"`.
`i, j`	indices to be used. Names are not currently allowed.
`drop`	logical, whether dimensions with only one level are to be dropped. Currently ignored, behaves as if it were `FALSE`.
`warn`	logical, whether to warn when no plot is saved.
`panel, aspect, as.table, between, key, auto.key, legend, layout, main, page, par`	arguments that will be used to update `object`. See details below.

Details

All high level lattice functions such as `xyplot` produce an object of (S3) class `"trellis"`, which is usually displayed by its `print` method. However, the object itself can be manipulated and modified to a large extent using the `update` method, and then re-displayed as needed.

Most arguments to high level functions can also be supplied to the `update` method as well, with some exceptions. Generally speaking, anything that would needs to change the data within each panel is a no-no (this includes the `formula, data, groups, subscripts` and `subset`). Everything else is technically game, though might not be implemented yet. If you find something missing that you wish to have, feel free to make a request.

Not all arguments accepted by a Lattice function are processed by `update`, but the ones listed above should work. The purpose of these arguments are described in the help page for `xyplot`. Any other argument is added to the list of arguments to be passed to the `panel` function. Because of their somewhat special nature, updates to objects produced by `cloud` and `wireframe` do not work very well yet.

The `"["` method is a convenient shortcut for updating `index.cond`. The `t` method is a convenient shortcut for updating `perm.cond` in the special (but frequent) case where there are exactly two conditioning variables, when it has the effect of switching ('transposing') their order.

The print method for `"trellis"` objects optionally saves the object after printing it. If this feature is enabled, `trellis.last.object` can retrieve it. Note that at most one object can be saved at a time. If `trellis.last.object` is called with arguments, these are used to update the retrieved object before returning it.

Value

An object of class `trellis`, by default plotted by `print.trellis`. `trellis.last.object` returns `NULL` is no saved object is available.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

`trellis.object`, `Lattice`, `xyplot`

Examples

```
spots <- by(sunspots, gl(235, 12, lab = 1749:1983), mean)
old.options <- lattice.options(save.object = TRUE)
xyplot(spots ~ 1749:1983, xlab = "", type = "l",
       scales = list(x = list(alternating = 2)),
       main = "Average Yearly Sunspots")
update(trellis.last.object(), aspect = "xy")
## Not run:
trellis.last.object(xlab = "Year")
## End(Not run)
lattice.options(old.options)
```

---

utilities.3d *Utility functions for 3-D plots*

Description

These are (related to) the default panel functions for `cloud` and `wireframe`.

Usage

```
ltransform3dMatrix(screen, R.mat)
ltransform3dto3d(x, R.mat, dist)
```

Arguments

`x`	`x` can be a numeric matrix with 3 rows for `ltransform3dto3d`
`screen`	list, as described in `panel.cloud`
`R.mat`	4x4 transformation matrix in homogeneous coordinates
`dist`	controls transformation to account for perspective viewing

Details

`ltransform3dMatrix` and `ltransform3dto3d` are utility functions to help in computation of projections. These functions are used inside the panel functions for `cloud` and `wireframe`. They may be useful in user-defined panel functions as well.

The first function takes a list of the form of the `screen` argument in `cloud` and `wireframe` and a `R.mat`, a 4x4 transformation matrix in homogeneous coordinates, to return a new 4x4 transformation matrix that is the result of applying `R.mat` followed by the rotations in `screen`. The second function applies a 4x4 transformation matrix in homogeneous coordinates to a 3xn matrix representing points in 3-D space, and optionally does some perspective computations. (There has been no testing with non-trivial transformation matrices, and my knowledge of the homogeneous coordinate system is very limited, so there may be bugs here.)

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

`cloud`, `panel.cloud`

---

xyplot *Common Bivariate Trellis Plots*

Description

These are the most commonly used high level Trellis functions to plot pairs of variables. By far the most common is `xyplot`, designed mainly for two continuous variates (though factors can be supplied as well, in which case they will simply be coerced to numeric), which produces Conditional Scatter plots. The others are useful when one of the variates is a factor or a shingle. Most of these arguments are also applicable to other high level functions in the lattice package, but are only documented here.

Usage

```
xyplot(x, data, ...)
dotplot(x, data, ...)
barchart(x, data, ...)
stripplot(x, data, ...)
bwplot(x, data, ...)

## S3 method for class 'formula':
xyplot(x,
       data,
       allow.multiple = is.null(groups) || outer,
       outer = !is.null(groups),
       auto.key = FALSE,
       aspect = "fill",
       panel = "panel.xyplot",
       prepanel = NULL,
       scales = list(),
```

```
        strip = TRUE,
        groups = NULL,
        xlab,
        xlim,
        ylab,
        ylim,
        drop.unused.levels = lattice.getOption("drop.unused.levels"),
        ...,
        default.scales,
        subscripts = !is.null(groups),
        subset = TRUE)


## S3 method for class 'formula':
dotplot(x,
        data,
        panel = "panel.dotplot",
        ...)

## S3 method for class 'formula':
barchart(x,
         data,
         panel = "panel.barchart",
         box.ratio = 2,
         ...)

## S3 method for class 'formula':
stripplot(x,
          data,
          panel = "panel.stripplot",
          ...)

## S3 method for class 'formula':
bwplot(x,
             data,
             allow.multiple = is.null(groups) || outer,
             outer = FALSE,
             auto.key = FALSE,
             aspect = "fill",
             panel = "panel.bwplot",
             prepanel = NULL,
             scales = list(),
             strip = TRUE,
             groups = NULL,
             xlab,
             xlim,
             ylab,
             ylim,
             box.ratio = 1,
             horizontal = NULL,
             drop.unused.levels = lattice.getOption("drop.unused.levels"),
             ...,
```

```
default.scales,
subscripts = !is.null(groups),
subset = TRUE)
```

Arguments

x	The object on which method dispatch is carried out. For the "formula" methods, a formula describing the form of conditioning plot. The formula is generally of the form y ~ x | g1 * g2 * ..., indicating that plots of y (on the y axis) versus x (on the x axis) should be produced conditional on the variables g1, g2, ... However, the conditioning variables g1,g2,... may be omitted. The formula can also be supplied as y ~ x | g1 + g2 + ... For all of these functions, with the exception of xyplot, a formula of the form ~ x | g1 * g2 * ... is also allowed. In that case, y defaults to names(x) if x is named, and a factor with a single level otherwise. Other usage of the form dotplot(x) is handled by method dispatch as appropriate. The numeric method is equivalent to a call with no left hand side and no conditioning variables in the formula. For barchart and dotplot, non-trivial methods exist for tables and arrays, documented under barchart.table. The conditioning variables g1, g2, ... must be either factors or shingles. Shingles are a way of processing numeric variables for use in conditioning. See documentation of shingle for details. Like factors, they have a "levels" attribute, which is used in producing the conditional plots. Numeric conditioning variables are converted to shingles by the function shingle (however, using equal.count might be more appropriate in many cases) and character vectors are coerced to factors. The formula can involve expressions, e.g. sqrt(), log(). A special case is when the left and/or right sides of the formula (before the conditioning variables) contain a '+' sign, e.g., y1+y2 ~ x | a*b. This formula would be taken to mean that the user wants to plot both y1~x | a*b and y2~x | a*b, but with the y1~x and y2~x superposed in each panel (this is slightly more complicated in barchart). The two parts would be distinguished by different graphical parameters. This is essentially what the groups argument would produce, if y1 and y2 were concatenated to produce a longer vector, with the groups argument being an indicator of which rows come from which variable. In fact, this is exactly what is done internally using the reshape function. This feature cannot be used in conjunction with the groups argument. To interpret y1 + y2 as a sum, one can either set allow.multiple=FALSE or use I(y1+y2). A variation on this feature is when the outer argument is set to TRUE as well as allow.multiple. In that case, the plots are not superposed in each panel, but instead separated into different panels (as if a new conditioning variable had been added). The x and y variables should both be numeric in xyplot, and an attempt is made to coerce them if not. However, if either is a factor, the levels of that factor are used as axis labels. In the other four functions documented here,

exactly one of x and y should be numeric, and the other a factor or shingle. Which of these will happen is determined by the horizontal argument — if horizontal=TRUE, then y will be coerced to be a factor or shingle, otherwise x. The default value of horizontal is FALSE if x is a factor or shingle, TRUE otherwise. (The functionality provided by horizontal=FALSE is not S-compatible.)

Note that this argument used to be called formula in earlier versions (when the high level functions were not generic and the formula method was essentially the only method). This is no longer allowed. It is recommended that this argument not be named in any case, but rather be the first (unnamed) argument.

data — For the formula method, a data frame containing values for any variables in the formula, as well as groups and subset if applicable. If not found in data, or if data is unspecified, the variables are looked for in the environment of the formula. For other methods (where x is not a formula), data is usually ignored, often with a warning.

allow.multiple, outer — logical flags to control what happens with formulas like y1 + y2 ~ x. See the entry for x for details. allow.multiple defaults to TRUE whenever it makes sense, and outer defaults to FALSE except when groups is explicitly specified or grouping doesn't make sense for the default panel function

box.ratio — applicable to bwplot, barchart and stripplot, specifies the ratio of the width of the rectangles to the inter rectangle space.

horizontal — logical, applicable to bwplot, dotplot, barchart and stripplot. Determines which of x and y is to be a factor or shingle (y if TRUE, x otherwise). Defaults to FALSE if x is a factor or shingle, TRUE otherwise. This argument is used to process the arguments to these high level functions, but more importantly, it is passed as an argument to the panel function, which is supposed to use it as appropriate.

A potentially useful component of scales in this case might be abbreviate = TRUE, in which case long labels which would usually overlap will be abbreviated. scales could also contain a minlength argument in this case, which would be passed to the abbreviate function.

panel — Once the subset of rows defined by each unique combination of the levels of the grouping variables are obtained (see details), the corresponding x and y variables (or other variables, as appropriate, in the case of other high level functions) are passed on to be plotted in each panel. The actual plotting is done by the function specified by the panel argument. Each high level function has its own default panel function, which could depend on whether the groups argument was supplied.

The panel function can be a function object or a character string giving the name of a predefined function.

Much of the power of Trellis Graphics comes from the ability to define customized panel functions. A panel function appropriate for the functions described here would usually expect arguments named x and y, which would be provided by the conditioning process. It can also have other arguments. It might be useful to know in this context that all arguments passed to a high level Trellis function (such as xyplot) that are not recognized by it are passed through to the panel function. It is thus generally good practice when defining panel functions to allow a ... argument. Such extra arguments typically control graphical parameters, but other uses are also common. See documentation for individual panel functions for specifics.

Note that unlike in S-PLUS, it is not guaranteed that panel functions will be supplied only numeric vectors for the `x` and `y` arguments; they can be factors as well (but not shingles). Panel functions need to handle this case, which in most cases can be done by simply coercing them to numeric.

Technically speaking, panel functions must be written using Grid graphics functions. However, knowledge of Grid is usually not necessary to construct new custom panel functions, there are several predefined panel functions which can help; for example, `panel.grid`, `panel.loess`, etc. There are also some grid-compatible replacements of commonly used base R graphics functions useful for this purpose. For example, `lines` can be replaced by `llines` (or equivalently, `panel.lines`). Note that base R graphics functions like `lines` will not work in a lattice panel function.

One case where a bit more is required of the panel function is when the `groups` argument is not null. In that case, the panel function should also accept arguments named `groups` and `subscripts` (see below for details). A useful panel function predefined for use in such cases is `panel.superpose`, which can be combined with different `panel.groups` functions determining what is plotted for each group. See the examples section for an interaction plot constructed in this way. Several other panel functions can also handle the `groups` argument, including the default ones for `barchart`, `dotplot` and `stripplot`.

Even when `groups` is not present, the panel function can have `subscripts` as a formal argument. In either case, the `subscripts` argument passed to the panel function are the indices of the `x` and `y` data for that panel in the original `data`, BEFORE taking into account the effect of the `subset` argument. Note that `groups` remains unaffected by any subsetting operations, so `groups[subscripts]` gives the values of `groups` that correspond to the data in that panel. The value of `subscripts` becomes slightly more complicated when `allow.multiple` is in effect. Details can be found in the source code of the function `latticeParseFormula`.

A panel function can have two other optional arguments for convenience, namely `panel.number` and `packet.number`, representing panel order and packet order respectively. Both provide a simple integer index indicating which panel is currently being drawn, but differ in how the count is calculated. `panel.number` is a simple incremental counter that starts with 1 and is incremented each time a panel is drawn. `packet.number` on the other hand indexes the combination of levels of the conditioning variables that is represented by that panel. The two indices coincide unless the order of conditioning variables is permuted and/or the plotting order of levels within one or more conditioning variables is altered (using `perm.cond` and `index.cond` respectively), in which case `packet.number` gives the index corresponding to the 'natural' ordering of that combination of levels of the conditioning variables.

`panel.xyplot` has an argument called `type` which is worth mentioning here because it is quite frequently used (and as mentioned above, can be passed to `xyplot` directly). panel functions for `bwplot` and friends should have an argument called `horizontal` to account for the cases when `x` is the factor or shingle.

`aspect` controls physical aspect ratio of the panels (same for all the panels). It can be specified as a ratio (vertical size/horizontal size) or as a character string. Legitimate values are `"fill"` (the default) which tries to make the panels as big as possible to fill the available space; `"xy"`, which **tries** to compute the aspect based on the 45 degree banking rule (see *Visualizing Data* by William

S. Cleveland for details); and "iso" for isometric scales, where the relation between physical distance on the device and distance in the data scale are forced to be the same for both axes.

If a prepanel function is specified and it returns components dx and dy, these are used for banking calculations. Otherwise, values from the default prepanel function are used. Currently, only the default prepanel function for xyplot can be expected to produce sensible banking calculations. See banking for details on the implementation of banking .

groups — a variable or expression to be evaluated in the data frame specified by data, expected to act as a grouping variable within each panel, typically used to distinguish different groups by varying graphical parameters like color and line type. Formally, if groups is specified, then groups along with subscripts is passed to the panel function, which is expected to handle these arguments. Not all pre-defined panel functions know how to, but for high level functions where grouping is appropriate, the default panel functions are chosen so that they do.

It is very common to use a key (legend) when a grouping variable is specified. See entries for key, auto.key and simpleKey for how to draw a key.

auto.key — A logical (indicating whether a key is to be drawn automatically when a grouping variable is present in the plot), or a list of parameters that would be valid arguments to simpleKey. If auto.key is not FALSE, groups is non-null and there is no key or legend argument specified in the call, a key is created with simpleKey with levels(groups) as the first argument. (Note: this may not work in all high level functions, but it does work for the ones where grouping makes sense with the default panel function)

simpleKey uses the trellis settings to determine the graphical parameters in the key, so this will be meaningful only if the settings are used in the plot as well.

One disadvantage to using key (or even simpleKey) directly is that the graphical parameters used in the key are absolutely determined at the time when the "trellis" object is created. Consequently, if a plot once created is reprinted with different settings, the parameter settings for the original device will be used. However, with auto.key, the key is actually created at printing time, so the key settings will match the device settings.

prepanel — function that takes the same arguments as the panel function and returns a list, possibly containing components named xlim, ylim, dx and dy (and less frequently, xat and yat).

The xlim and ylim components are similar to the high level xlim and ylim arguments (i.e., they are usually a numeric vector of length 2 defining a range of values, or a character vector representing levels of a factor). If the xlim and ylim arguments are not explicitly specified (possibly as components in scales), then the actual limits of the panels are guaranteed to include the limits returned by the prepanel function. This happens globally if the relation component of scales is "same", and on a panel by panel basis otherwise. See xlim to see what forms of the components xlim and ylim are allowed.

The dx and dy components are used for banking computations in case aspect is specified as "xy". See documentation for the function banking for details regarding how this is done.

The return value of the prepanel function need not have all the components named above; in case some are missing, they are replaced by the usual component-wise defaults.

If xlim or ylim is a character vector (which is appropriate when the corresponding variable is a factor), this implicitly indicates that the scale should include the first n integers, where n is the length of xlim or ylim, as the case may be. The elements of the character vector are used as the default labels for these n integers. Thus, to make this information consistent between panels, the xlim or ylim values should represent all the levels of the corresponding factor, even if some are not used within that particular panel.

In such cases, an additional component xat or yat may be returned by the prepanel function, which should be a subset of 1:n, indicating which of the n values (levels) are actually represented in the panel. This is useful when calculating the limits with relation="free" or relation="sliced" in scales.

The prepanel function is responsible for providing a meaningful return value when the x, y (etc.) variables are zero-length vectors. When nothing is appropriate, values of NA should be returned for the xlim and ylim components.

strip — logical flag or function. If FALSE, strips are not drawn. Otherwise, strips are drawn using the strip function, which defaults to strip.default. See documentation of strip.default to see the arguments that are available to the strip function. This description also applies to the strip.left argument (see ... below), which can be used to draw strips on the left of each panel, which can be useful for wide short panels, e.g. in time series plots.

xlab — character string or expression (or a "grob") giving label for the x-axis. Defaults to the expression for x in formula. Can be specified as NULL to omit the label altogether. Finer control is possible, as described in the entry for main, with the additional feature that if the label component is omitted from the list, it is replaced by the default xlab.

ylab — character string or expression (or "grob") giving label for the y-axis. Defaults to the expression for y in formula. Fine control is possible, see entries for main and xlab.

scales — list determining how the x- and y-axes (tick marks and labels) are drawn. The list contains parameters in name=value form, and may also contain two other lists called x and y of the same form (described below). Components of x and y affect the respective axes only, while those in scales affect both. When parameters are specified in both lists, the values in x or y are used. Note that certain high-level functions have defaults that are specific to a particular axis (e.g., bwplot has alternating=FALSE for the y-axis only); these can be overridden only by an entry in the corresponding component of scales.

The possible components are :

relation character string that determines how axis limits are calculated for each panel. Possible values are "same" (default), "free" and "sliced". For relation="same", the same limits, usually large enough to encompass all the data, are used for all the panels. For relation="free", limits for each panel is determined by just the points in that panel. Behavior for relation="sliced" is similar, except that the length (max - min) of the scales are constrained to remain the same across panels.

The determination of what axis limits are suitable for each panel can be controlled by the prepanel function, which can be overridden by xlim, ylim or scales$limits. If relation is not "same", the value of xlim etc is normally ignored, except when it is a list, in which case it is treated

as if its components were the limit values obtained from the prepanel calculations for each panel.

tick.number Suggested number of ticks (ignored for a factor, shingle or character vector, in which case there is no natural rule for leaving out some of the labels. But see `xlim`).

draw logical, defaults to `TRUE`, whether to draw the axis at all.

alternating logical specifying whether axis labels should alternate from one side of the group of panels to the other. For finer control, alternating can be a vector (replicated to be as long as the number of rows or columns per page) consisting of the following numbers

- 0: do not draw tick labels
- 1: bottom/left
- 2: top/right
- 3: both.

`alternating` applies only when `relation="same"`. The default is `TRUE`, or equivalently, `c(1, 2)`

limits same as xlim and ylim.

at location of tick marks along the axis (in native coordinates), or a list as long as the number of panels describing tick locations for each panel.

labels Labels (strings or expressions) to go along with `at`. Can be a list like `at` as well.

cex numeric multiplier to control character sizes for axis labels. Can be a vector of length 2, to control left/bottom and right/top separately.

font, fontface, fontfamily specifies font for axis labels.

tck numeric to control length of tick marks. Can be a vector of length 2, to control left/bottom and right/top separately.

col color of ticks and labels.

rot Angle by which the axis labels are to be rotated. Can be a vector of length 2, to control left/bottom and right/top separately.

abbreviate logical, whether to abbreviate the labels using `abbreviate`. Can be useful for long labels (e.g., in factors), especially on the x-axis.

minlength argument passed to `abbreviate` if `abbreviate=TRUE`.

log whether to use a log scale. Defaults to `FALSE`, other possible values are any number that works as a base for taking logarithm, `TRUE`, equivalent to 10, and `"e"` (for natural logarithm). Note that in this case the values passed to the panel function are already transformed, so all computations done inside the panel function will be affected accordingly. For example, `panel.lmline` will fit a line to the transformed values.

format the `format` to use for POSIXct variables. See `strptime` for description of valid values.

axs character, `"r"` or `"i"`. In the latter case, the axis limits are calculated as the exact data range, instead of being padded on either side. (May not always work as expected.)

Note that much of the function of `scales` is accomplished by `pscales` in `splom`.

`subscripts` logical specifying whether or not a vector named `subscripts` should be passed to the panel function. Defaults to `FALSE`, unless `groups` is specified, or if the panel function accepts an argument named `subscripts`. (One should be careful when defining the panel function on-the-fly.)

`subset` logical or integer indexing vector (can be specified in terms of variables in `data`). Only these rows of `data` will be used for the plot. If `subscripts` is `TRUE`, the subscripts will provide indices to the rows of data before the subsetting is done. Whether levels of factors in the data frame that are unused after the subsetting will be dropped depends on the `drop.unused.levels` argument.

`xlim` Normally a numeric vector of length 2 (possibly a DateTime object) giving minimum and maximum for the x-axis, or, a character vector, expected to denote the levels of `x`. The latter form is interpreted as a range containing c(1, length(xlim)), with the character vector determining labels at tick positions `1:length(xlim)`

`xlim` could also be a list, with as many components as the number of panels (recycled if necessary), with each component as described above. This is meaningful only when `scales$x$relation` is `"free"` or `"sliced"`, in which case these are treated as if they were the corresponding limit components returned by prepanel calculations.

`ylim` similar to `xlim`, applied to the y-axis.

`drop.unused.levels`
logical indicating whether the unused levels of factors will be dropped. Unused levels are usually dropped, but it is sometimes appropriate to suppress dropping to preserve a useful layout. For finer control, this argument could also be list containing components `cond` and `data`, both logical, indicating desired behavior for conditioning variables and data variables respectively. The default is given by `lattice.getOption("drop.unused.levels")` , which is initially set to `TRUE` for both components.

`default.scales`
list giving the default values of `scales` for a particular high level function. This should not be of any interest to the normal user, but may be helpful when defining other functions that act as a wrapper to one of the high level lattice functions.

`...` further arguments, usually not directly processed by the high level functions documented here, but rather passed on to other functions. Such arguments can be broadly categorized into two types: those that affect all high level Trellis functions in a similar manner, and those that are meant for the specific panel function used, which may differ across high level functions.

The first group of arguments are processed by a common, unexported function called `trellis.skeleton`. These arguments affect all high level functions, but are only documented here, except to override the behaviour described here. All other arguments specified in a high level call, specifically those neither described here nor in the help page of the relevant high level function, are passed unchanged to the panel function used. By convention, the default panel function used for any high level function is named as `"panel."` followed by the name of the high level function; for example, the default panel function for `bwplot` is `panel.bwplot`. In practical terms, this means that in addition to the help page of the high level function being used, the user should also consult the help page of the corresponding panel function for arguments that may be specified in the high level call.

The effect of the first group of common arguments are as follows:

`as.table`: logical that controls the order in which panels should be plotted: if `FALSE` (the default), panels are drawn left to right, bottom to top (as in a graph); if `TRUE`, left to right, top to bottom.

between: a list with components x and y (both usually 0 by default), numeric vectors specifying the space between the panels (units are character heights). x and y are repeated to account for all panels in a page and any extra components are ignored. The result is used for all pages in a multi page display. (In other words, it is not possible to use different between values for different pages).

key: A list of arguments that define a legend to be drawn on the plot. This list is used as an argument to the draw.key function, which produces a grid object eventually plotted by the print method for "trellis" objects.
There is also a less flexible but usually sufficient shortcut function simpleKey that can generate such a list, as well as the argument auto.key that can be convenient in the most common situation where legends are used, namely when there is a grouping variable. To use more than one legend, or to have arbitrary legends not constrained by the structure imposed by key, use the legend argument.
The position of the key can be controlled in either of two possible ways. If a component called space is present, the key is positioned outside the plot region, in one of the four sides, determined by the value of space, which can be one of "top", "bottom", "left" and "right". Alternatively, the key can be positioned inside the plot region by specifying components x, y and corner. x and y determine the location of the corner of the key given by corner, which can be one of c(0,0), c(1,0), c(1,1) and c(0,1), which denote the corners of the unit square. x and y must be numbers between 0 and 1, giving coordinates with respect to the whole display area.
The key essentially consists of a number of columns, possibly divided into blocks, each containing some rows. The contents of the key are determined by (possibly repeated) components named "rectangles", "lines", "points" or "text". Each of these must be lists with relevant graphical parameters (see later) controlling their appearance. The key list itself can contain graphical parameters, these would be used if relevant graphical components are omitted from the other components.
The length (number of rows) of each such column (except "text"s) is taken to be the largest of the lengths of the graphical components, including the ones specified outside (see the entry for rep below for details on this). The "text" component has to have a character or expression vector as its first component, and the length of this vector determines the number of rows.
The graphical components that can be included in key (and also in the components named "text", "lines", "points" and "rectangles" as appropriate) are:

- cex=1
- col="black"
- lty=1
- lwd=1
- font=1
- fontface
- fontfamily
- pch=8
- adj=0
- type="l"

- `size=5`
- `angle=0`
- `density=-1`

`angle` and `density` are currently unimplemented. `size` determines the width of columns of rectangles and lines in character widths. `type` is relevant for lines; `"l"` denotes a line, `"p"` denotes a point, and `"b"` and `"o"` both denote both together.

Other possible components of `key` are:

`reverse.rows` logical, defaulting to `FALSE`. If `TRUE`, all components are reversed *after* being replicated (the details of which may depend on the value of `rep`). This is useful in certain situations, e.g. with a grouped `barchart` with `stack = FALSE` with the categorical variable on the vertical axis, where the bars in the plot will usually be ordered from bottom to top, but the corresponding legend will have the levels from top to bottom (unless, of course, `reverse.rows = TRUE`). Note that in this case, unless all columns have the same number or rows, they will no longer be aligned.

`between` numeric vector giving the amount of space (character widths) surrounding each column (split equally on both sides)

`title` string or expression giving a title for the key

`rep` logical, defaults to `TRUE`. By default, it's assumed that all columns in the key (except the `"text"`s) will have the same number of rows, and all components are replicated to be as long as the longest. This can be suppressed by specifying `rep=FALSE`, in which case the length of each column will be determined by components of that column alone.

`cex.title` cex for the title

`lines.title` how many lines the title should occupy (in multiples of itself). Defaults to 2.

`padding.text` how much space (padding) should be used above and below each row containing text, in multiples of the default, which is currently `0.2 * "lines"`. This padding is in addition to the normal height of any row that contains text, which is the minimum amount necessary to contain all the text entries.

`background` background color, defaults to default background

`border` either a color for the border, or a logical. In the latter case, the border color is black if `border` is `TRUE`, and no border is drawn if it is `FALSE` (the default)

`transparent=FALSE` logical, whether key area should have a transparent background

`columns` the number of columns column-blocks the key is to be divided into, which are drawn side by side.

`between.columns` Space between column blocks, in addition to `between`.

`divide` Number of point symbols to divide each line when `type` is `"b"` or `"o"` in `lines`.

`legend:` the legend argument can be useful if one wants to place more than one key. It also allows one to use arbitrary `"grob"`s (grid objects) as legends.

If used, `legend` must be a list, with an arbitrary number of components. Each component must be named one of `"left"`, `"right"`, `"top"`,

"bottom" or "inside". The name "inside" can be repeated, but not the others. This name will be used to determine the location for that component, and is similar to the space component of key. If key (or colorkey for levelplot and wireframe) is specified, their space component must not conflict with the name of any component of legend. Each component of legend must have a component called fun. This can be a "grob", or a function or the name of a function that produces a "grob" when called. If this function expects any arguments, they must be supplied as a list in another component called args. For components named "inside", there can be additional components called x, y and corner, which work in the same way as it does for key.

page: a function of one argument (page number) to be called after drawing each page. The function must be 'grid-compliant', and is called with the whole display area as the default viewport.

main: typically a character string or expression or list describing the main title to be placed on top of each page. Defaults to NULL. Can be a character string or expression, or a list with components label, cex, col and font. The label tag can be omitted if it is the first element of the list. Expressions are treated as specification of LaTeX-like markup as in plotmath.

main can also be an arbitrary "grob" (grid graphical object).

sub: character string or expression (or a "grob") for a subtitle to be placed at the bottom of each page. See entry for main for finer control options.

par.strip.text: list of parameters to control the appearance of strip text. Notable components are col, cex, font and lines. The first three control graphical parameters while the last is a means of altering the height of the strips. This can be useful, for example, if the strip labels (derived from factor levels, say) are double height (i.e., contains "\n"-s) or if the default height seems too small or too large. The lineheight component can control the space between multiple lines. Also, the labels can be abbreviated when shown by specifying abbreviate = TRUE, in which case the components minlength and dot (passed along to the abbreviate function) can be specified to control the details of how this is done.

layout: In general, a Trellis conditioning plot consists of several panels arranged in a rectangular array, possibly spanning multiple pages. layout determines this arrangement.

layout is a numeric vector giving the number of columns, rows and pages in a multi panel display. By default, the number of columns is the number of levels of the first conditioning variable and the number of rows is the number of levels of the second conditioning variable. If there is only one conditioning variable, the default layout vector is c(0,n), where n is the number of levels of the given vector. Any time the first value in the layout vector is 0, the second value is used as the desired number of panels per page and the actual layout is computed from this, taking into account the aspect ratio of the panels and the device dimensions (via par("din")). The number of pages is by default set to as many as is required to plot all the panels. In general, giving a high value of layout[3] is not wasteful because blank pages are never created.

skip: logical vector (default FALSE), replicated to be as long as the number of panels (spanning all pages). For elements that are TRUE, the corresponding panel position is skipped; i.e., nothing is plotted in that position. The panel that was supposed to be drawn there is now drawn in the next available

panel position, and the positions of all the subsequent panels are bumped up accordingly. This is often useful for arranging plots in an informative manner.

strip.left: strip.left can be used to draw strips on the left of each panel, which can be useful for wide short panels, as in time series (or similar) plots. It is a function similar to strip.

xlab.default, ylab.default: fallback default for xlab and ylab when they are not specified. If NULL, the defaults are parsed from the Trellis formula. This is rarely useful for the end-user, but can be helpful when developing new Trellis functions.

xscale.components, yscale.components: functions that determine axis annotation for the x and y axes respectively. See documentation for xscale.components.default, the default values of these arguments, to learn more.

axis: function that draws axis annotation. See documentation for axis.default, the default value of this argument, to learn more.

perm.cond: numeric vector, a permutation of 1:n, where n is the number of conditioning variables. By default, the order in which panels are drawn depends on the order of the conditioning variables specified in the formula. perm.cond can modify this order. If the trellis display is thought of as an n-dimensional array, then during printing, its dimensions are permuted using perm.cond as the perm argument to aperm.

index.cond: While perm.cond permutes the dimensions of the multidimensional array of panels, index.cond can be used to subset (or reorder) margins of that array. index.cond can be a list or a function, with behavior in each case described below.

The panel display order within each conditioning variable depends on the order of their levels. index.cond can be used to choose a 'subset' (in the R sense) of these levels, which is then used as the display order for that variable. If index.cond is a list, it has to be as long as the number of conditioning variables, and the i-th component has to be a valid indexing vector for the integer vector 1:nlevels(g_i) (which can, among other things, repeat some of the levels or drop some altogether). The result of this indexing determines the order of panels within that conditioning variable. To keep the order of a particular variable unchanged, the corresponding component must be set to TRUE.

Note that the components of index.cond are in the order of the conditioning variables in the original call, and is not affected by perm.cond.

Another possibility is to specify index.cond as a function. In this case, this function is called once for each panel, potentially with all arguments that are passed to the panel function for that panel. (More specifically, if this function has a ... argument, then all panel arguments are passed, otherwise, only named arguments that match are passed.) For a single conditioning variable, the levels of that variable are then sorted so that these values are in ascending order. For multiple conditioning variables, the order for each variable is determined by first taking the average over all other conditioning variables.

Although they can be supplied in high level function calls directly, it is more typical to use perm.cond and index.cond to update an existing "trellis" object, thus allowing it to be displayed in a different arrangement without re-calculating the data subsets that go into each panel. In the update method, both can be set to NULL, which reverts these back to their

defaults.

par.settings: a list that could be supplied to trellis.par.set. This enables the user to attach some display settings to the trellis object itself rather than change the settings globally. When the object is plotted, these settings are temporarily in effect for the duration of the plot, after which the settings revert back to whatever they were before.

plot.args: a list of possible arguments to plot.trellis, which will be used by the plot or print methods when drawing the object, unless overridden explicitly. This enables the user to attach such arguments to the trellis object itself.

lattice.options: a list that could be supplied to lattice.options. This enables the user to attach options settings to the trellis object itself rather than change the settings globally. When the object is plotted, these settings are temporarily in effect for the duration of the plot, after which the settings revert back to whatever they were before. Note that most (but not all) "options" controlled by lattice.options affects the object itself and not the plotting of it. This argument affects the plotting stage only, and has no effect on the object being created. This behaviour might be considered a misfeature, and is liable to change in future versions.

Details

All the functions documented here are generic, with the formula method usually doing the actual work. The structure of the plot that is produced is mostly controlled by the formula. For each unique combination of the levels of the conditioning variables g1, g2, ..., a separate panel is produced using the points (x,y) for the subset of the data (also called packet) defined by that combination. The display can be though of as a 3-dimensional array of panels, consisting of one 2-dimensional matrix per page. The dimensions of this array are determined by the layout argument. If there are no conditioning variables, the plot produced consists of a single panel.

The coordinate system used by lattice by default is like a graph, with the origin at the bottom left, with axes increasing to left and up. In particular, panels are by default drawn starting from the bottom left corner, going right and then up; unless as.table = TRUE, in which case panels are drawn from the top left corner, going right and then down. One might wish to set a global preference for a table-like arrangement by changing the default to as.table=TRUE; this can be done by setting lattice.options(default.args = list(as.table = TRUE)). In fact, default values can be set in this manner for the following arguments: as.table, aspect, between, page, main, sub, par.strip.text, layout, skip and strip. Note that these global defaults are sometimes overridden by individual functions.

The order of the panels depends on the order in which the conditioning variables are specified, with g1 varying fastest. Within a conditioning variable, the order depends on the order of the levels (which for factors is usually in alphabetical order). Both of these orders can be modified using the index.cond and perm.cond arguments, possibly using the update (and other related) method(s).

Value

An object of class "trellis". The update method can be used to update components of the object and the print method (usually called by default) will plot it on an appropriate plotting device.

Note

Most of the arguments documented here are also applicable for the other high level functions in the lattice package. These are not described in any detail elsewhere unless relevant, and this should be considered the canonical documentation for such arguments.

Any arguments passed to these functions and not recognized by them will be passed to the panel function. Most predefined panel functions have arguments that customize its output. These arguments are described only in the help pages for these panel functions, but can usually be supplied as arguments to the high level plot.

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

See Also

Lattice for an overview of the package, as well as barchart.table, Lattice, print.trellis, shingle, banking, reshape, panel.xyplot, panel.bwplot, panel.barchart, panel.dotplot, panel.stripplot, panel.superpose, panel.loess, panel.linejoin, strip.default, simpleKey trellis.par.set

Examples

```
## Not run:
## wait for user input before each new page (like 'par(ask = TRUE)')
old.prompt <- grid::grid.prompt(TRUE)
## End(Not run)

require(stats)

## Tonga Trench Earthquakes

Depth <- equal.count(quakes$depth, number=8, overlap=.1)
xyplot(lat ~ long | Depth, data = quakes)
update(trellis.last.object(),
       strip = strip.custom(strip.names = TRUE, strip.levels = TRUE),
       par.strip.text = list(cex = 0.75),
       aspect = "iso")

## Examples with data from `Visualizing Data' (Cleveland)
## (obtained from Bill Cleveland's Homepage :
## http://cm.bell-labs.com/cm/ms/departments/sia/wsc/, also
## available at statlib)

EE <- equal.count(ethanol$E, number=9, overlap=1/4)
## Constructing panel functions on the fly; prepanel
xyplot(NOx ~ C | EE, data = ethanol,
       prepanel = function(x, y) prepanel.loess(x, y, span = 1),
       xlab = "Compression Ratio", ylab = "NOx (micrograms/J)",
       panel = function(x, y) {
           panel.grid(h=-1, v= 2)
           panel.xyplot(x, y)
           panel.loess(x,y, span=1)
       },
       aspect = "xy")
```

```
## with and without banking

plot <- xyplot(sunspot.year ~ 1700:1988, xlab = "", type = "l",
               scales = list(x = list(alternating = 2)),
               main = "Yearly Sunspots")
print(plot, position = c(0, .3, 1, .9), more = TRUE)
print(update(plot, aspect = "xy", main = "", xlab = "Year"),
      position = c(0, 0, 1, .3))



## Multiple variables in formula for grouped displays

xyplot(Sepal.Length + Sepal.Width ~ Petal.Length + Petal.Width | Species,
       data = iris, scales = "free", layout = c(2, 2),
       auto.key = list(x = .6, y = .7, corner = c(0, 0)))

## user defined panel functions

states <- data.frame(state.x77,
                     state.name = dimnames(state.x77)[[1]],
                     state.region = state.region)
xyplot(Murder ~ Population | state.region, data = states,
       groups = state.name,
       panel = function(x, y, subscripts, groups)
       ltext(x = x, y = y, label = groups[subscripts], cex=1,
             fontfamily = "HersheySans"))

barchart(yield ~ variety | site, data = barley,
         groups = year, layout = c(1,6),
         ylab = "Barley Yield (bushels/acre)",
         scales = list(x = list(abbreviate = TRUE,
                        minlength = 5)))
barchart(yield ~ variety | site, data = barley,
         groups = year, layout = c(1,6), stack = TRUE,
         auto.key = list(points = FALSE, rectangles = TRUE, space = "right"),
         ylab = "Barley Yield (bushels/acre)",
         scales = list(x = list(rot = 45)))

bwplot(voice.part ~ height, data=singer, xlab="Height (inches)")
dotplot(variety ~ yield | year * site, data=barley)

dotplot(variety ~ yield | site, data = barley, groups = year,
        key = simpleKey(levels(barley$year), space = "right"),
        xlab = "Barley Yield (bushels/acre) ",
        aspect=0.5, layout = c(1,6), ylab=NULL)

stripplot(voice.part ~ jitter(height), data = singer, aspect = 1,
          jitter = TRUE, xlab = "Height (inches)")
## Interaction Plot

xyplot(decrease ~ treatment, OrchardSprays, groups = rowpos,
       type = "a",
       auto.key =
       list(space = "right", points = FALSE, lines = TRUE))
```

```
## longer version with no x-ticks

## Not run:
bwplot(decrease ~ treatment, OrchardSprays, groups = rowpos,
       panel = "panel.superpose",
       panel.groups = "panel.linejoin",
       xlab = "treatment",
       key = list(lines = Rows(trellis.par.get("superpose.line"),
                  c(1:7, 1)),
                  text = list(lab = as.character(unique(OrchardSprays$rowpos))),
                  columns = 4, title = "Row position"))
## End(Not run)

## Not run:
grid::grid.prompt(old.prompt)
## End(Not run)
```

Chapter 17

The mgcv package

---

mgcv-package *GAMs with GCV smoothness estimation and GAMMs by REML/PQL*

Description

mgcv provides functions for generalized additive modelling and generalized additive mixed modelling. Particular features of the package are facilities for automatic smoothness selection, and the provision of a variety of smooths of more than one variable. User defined smooths are also supported. A Bayesian approach to confidence/credible interval calculation is provided. Lower level routines for generalized ridge regression and penalized linearly constrained least squares are also provided.

Details

mgcv provides generalized additive modelling functions gam, predict.gam and plot.gam, which are very similar in use to the S functions of the same name designed by Trevor Hastie. However the underlying representation and estimation of the models is based on a penalized regression spline approach, with automatic smoothness selection. A number of other functions such as summary.gam and anova.gam are also provided, for extracting information from a fitted gamObject.

Use of gam is much like use of glm, except that within a gam model formula, isotropic smooths of any number of predictors can be specified using s terms, while scale invariant smooths of any number of predictors can be specified using te terms. Estimation is by penalized likelihood or quasi-likelihood maximization, with smoothness selection by GCV or gAIC/ UBRE. See gam, gam.models and gam.selection for some discussion of model specification and selection. For detailed control of fitting see gam.convergence, gam.method and gam.control. For checking and visualization see gam.check, choose.k, vis.gam and plot.gam. While a number of types of smoother are built into the package, it is also extendable with user defined smooths, see p.spline, for example.

A Bayesian approach to smooth modelling is used to derive standard errors on predictions, and hence credible intervals. The Bayesian covariance matrix for the model coefficients is returned in Vp of the gamObject. See predict.gam for examples of how this can be used to obtain credible regions for any quantity derived from the fitted model, either directly, or by direct simulation from the posterior distribution of the model coefficients. Frequentist approximations can be used for hypothesis testing: see anova.gam and summary.gam, but note that the underlying approximations are not always good in this case.

The package also provides a generalized additive mixed modelling function, gamm, based on glmmPQL from the MASS library and lme from the nlme library. gamm is particularly useful for modelling correlated data (i.e. where a simple independence model for the residual variation is inappropriate). In addition, low level routine magic can fit models to data with a known correlation structure.

Some of the underlying GAM fitting methods are available as low level fitting functions: see magic and mgcv. Penalized weighted least squares with linear equality and inequality constraints is provided by pcls.

For a complete list of functions type library(help=mgcv).

Author(s)

Simon Wood <simon.wood@r-project.org>

with contributions and/or help from Kurt Hornik, Mike Lonergan, Henric Nilsson and Brian Ripley.

Maintainer: Simon Wood <simon.wood@r-project.org>

References

Wood, S.N. (2006) *Generalized Additive Models: an introduction with R*, CRC

Examples

```
## see examples for gam and gamm
```

---

anova.gam *Hypothesis tests related to GAM fits*

Description

Performs hypothesis tests relating to one or more fitted gam objects. For a single fitted gam object, Wald tests of the significance of each parametric and smooth term are performed. Otherwise the fitted models are compared using an analysis of deviance table. The tests are usually approximate, unless the models are un-penalized.

Usage

```
## S3 method for class 'gam':
anova(object, ..., dispersion = NULL, test = NULL)
## S3 method for class 'anova.gam':
print(x, digits = max(3, getOption("digits") - 3),...)
```

Arguments

object,...	fitted model objects of class gam as produced by gam().
x	an anova.gam object produced by a single model call to anova.gam().
dispersion	a value for the dispersion parameter: not normally used.
test	what sort of test to perform for a multi-model call. One of "Chisq", "F" or "Cp".
digits	number of digits to use when printing output.

Details

If more than one fitted model is provided than `anova.glm` is used. If only one model is provided then the significance of each model term is assessed using Wald tests: see `summary.gam` for details of the actual computations. In the latter case `print.anova.gam` is used as the printing method.

P-values are usually reliable if the smoothing parameters are known, or the model is unpenalized. If smoothing parameters have been estimated then the p-values are typically somewhat too low under the null. This occurs because the uncertainty associated with the smoothing parameters is neglected in the calculations of the distributions under the null, which tends to lead to underdispersion in these distributions, and in turn to p-value estimates that are too low. (In simulations where the null is correct, I have seen p-values that are as low as half of what they should be.) Note however that tests can have low power if the estimated rank of the test statistic is much higher than the EDF, so that p-values can also be too low in some cases.

If it is important to have p-values that are as accurate as possible, then, at least in the single model case, it is probably advisable to perform tests using unpenalized smooths (i.e. `s(...,fx=TRUE)`) with the basis dimension, `k`, left at what would have been used with penalization. Such tests are not as powerful, of course, but the p-values are more accurate under the null. Whether or not extra accuracy is required will usually depend on whether or not hypothesis testing is a key objective of the analysis.

Value

In the multi-model case `anova.gam` produces output identical to `anova.glm`, which it in fact uses.

In the single model case an object of class `anova.gam` is produced, which is in fact an object returned from `summary.gam`.

`print.anova.gam` simply produces tabulated output.

WARNING

P-values are only approximate, particularly as a result of ignoring smoothing parameter uncertainty.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩ with substantial improvements by Henric Nilsson.

See Also

`gam`, `predict.gam`, `gam.check`, `summary.gam`

Examples

```
library(mgcv)
set.seed(0)
n<-200
sig<-2
x0 <- rep(1:4,50)
x1 <- runif(n, 0, 1)
x2 <- runif(n, 0, 1)
x3 <- runif(n, 0, 1)
y <- 2 * x0
y <- y + exp(2 * x1)
y <- y + 0.2 * x2^11 * (10 * (1 - x2))^6 + 10 * (10 * x2)^3 * (1 - x2)^10
```

```
e <- rnorm(n, 0, sig)
y <- y + e
x0<-as.factor(x0)
b<-gam(y~x0+s(x1)+s(x2)+s(x3))
anova(b)
b1<-gam(y~x0+s(x1)+s(x2))
anova(b,b1,test="F")
```

`choose.k` *Basis dimension choice for smooths*

Description

Choosing the basis dimension, and checking the choice, when using penalized regression smoothers.

Penalized regression smoothers gain computational efficiency by virtue of being defined using a basis of relatively modest size, `k`. When setting up models in the `mgcv` package, using `s` or `te` terms in a model formula, `k` must be chosen.

In practice `k-1` sets the upper limit on the degrees of freedom associated with an `s` smooth (1 degree of freedom is lost to the identifiability constraint on the smooth). For `te` smooths the upper limit of the degrees of freedom is given by the product of the `k` values provided for each marginal smooth less one, for the constraint. However the actual effective degrees of freedom are controlled by the degree of penalization selected during fitting, by GCV, AIC or whatever is specified. The exception to this is if a smooth is specified using the `fx=TRUE` option, in which case it is unpenalized.

So, exact choice of `k` is not generally critical: it should be chosen to be large enough that you are reasonably sure of having enough degrees of freedom to represent the underlying 'truth' reasonably well, but small enough to maintain reasonable computational efficiency. Clearly 'large' and 'small' are dependent on the particular problem being addressed.

As with all model assumptions, it is useful to be able to check the choice of `k` informally. If the effective degrees of freedom for a model term are estimated to be much less than `k-1` then this is unlikely to be very worthwhile, but as the EDF approach `k-1`, checking can be important. A useful general purpose approach goes as follows: (i) fit your model and extract the deviance residuals; (ii) for each smooth term in your model, fit an equivalent, single, smooth to the residuals, using a substantially increased `k` to see if there is pattern in the residuals that could potentially be explained by increasing `k`. Examples are provided below.

More sophisticated approaches based on partial residuals are also possible.

One scenario that can cause confusion is this: a model is fitted with `k=10` for a smooth term, and the EDF for the term is estimated as 7.6, some way below the maximum of 9. The model is then refitted with `k=20` and the EDF increases to 8.7 - what is happening - how come the EDF was not 8.7 the first time around? The explanation is that the function space with `k=20` contains a larger subspace of functions with EDF 8.7 than did the function space with `k=10`: one of the functions in this larger subspace fits the data a little better than did any function in the smaller subspace. These subtleties seldom have much impact on the statistical conclusions to be drawn from a model fit, however.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

Wood, S.N. (2006) Generalized Additive Models: An Introduction with R. CRC.

http://www.maths.bath.ac.uk/~sw283/

Examples

```
## Simulate some data ....
library(mgcv)
set.seed(0)
n<-400;sig<-2
x0 <- runif(n, 0, 1);x1 <- runif(n, 0, 1)
x2 <- runif(n, 0, 1);x3 <- runif(n, 0, 1)
f <- 2 * sin(pi * x0)
f <- f + exp(2 * x1) - 3.75887
f <- f+0.2*x2^11*(10*(1-x2))^6+10*(10*x2)^3*(1-x2)^10-1.396
e <- rnorm(n, 0, sig)
y <- f + e
## fit a GAM with quite low `k'
b<-gam(y~s(x0,k=6)+s(x1,k=6)+s(x2,k=6)+s(x3,k=6))
plot(b,pages=1)

## check for residual pattern, removeable by increasing `k'
## typically `k', below, chould be substantially larger than
## the original, `k' but certainly less than n/2.
## Note use of cheap "cs" shrinkage smoothers, and gamma=1.4
## to reduce chance of overfitting...
rsd <- residuals(b)
gam(rsd~s(x0,k=40,bs="cs"),gamma=1.4) ## fine
gam(rsd~s(x1,k=40,bs="cs"),gamma=1.4) ## fine
gam(rsd~s(x2,k=40,bs="cs"),gamma=1.4) ## original `k' too low
gam(rsd~s(x3,k=40,bs="cs"),gamma=1.4) ## fine

## similar example with multi-dimensional smooth
b1 <- gam(y~s(x0)+s(x1,x2,k=15)+s(x3))
rsd <- residuals(b1)
gam(rsd~s(x0,k=40,bs="cs"),gamma=1.4) ## fine
gam(rsd~s(x1,x2,k=100,bs="ts"),gamma=1.4) ## original `k' too low
gam(rsd~s(x3,k=40,bs="cs"),gamma=1.4) ## fine

## and a `te' example
b2 <- gam(y~s(x0)+te(x1,x2,k=4)+s(x3))
rsd <- residuals(b2)
gam(rsd~s(x0,k=40,bs="cs"),gamma=1.4) ## fine
gam(rsd~te(x1,x2,k=10,bs="cs"),gamma=1.4) ## original `k' too low
gam(rsd~s(x3,k=40,bs="cs"),gamma=1.4) ## fine

## same approach works with other families in the original model
g<-exp(f/4)
y<-rpois(rep(1,n),g)
bp<-gam(y~s(x0,k=6)+s(x1,k=6)+s(x2,k=6)+s(x3,k=6),family=poisson)
rsd <- residuals(bp)
gam(rsd~s(x0,k=40,bs="cs"),gamma=1.4) ## fine
gam(rsd~s(x1,k=40,bs="cs"),gamma=1.4) ## fine
gam(rsd~s(x2,k=40,bs="cs"),gamma=1.4) ## original `k' too low
gam(rsd~s(x3,k=40,bs="cs"),gamma=1.4) ## fine
```

`exclude.too.far` *Exclude prediction grid points too far from data*

Description

Takes two arrays defining the nodes of a grid over a 2D covariate space and two arrays defining the location of data in that space, and returns a logical vector with elements `TRUE` if the corresponding node is too far from data and `FALSE` otherwise. Basically a service routine for `vis.gam` and `plot.gam`.

Usage

```
exclude.too.far(g1,g2,d1,d2,dist)
```

Arguments

`g1`	co-ordinates of grid relative to first axis.
`g2`	co-ordinates of grid relative to second axis.
`d1`	co-ordinates of data relative to first axis.
`d2`	co-ordinates of data relative to second axis.
`dist`	how far away counts as too far. Grid and data are first scaled so that the grid lies exactly in the unit square, and `dist` is a distance within this unit square.

Details

Linear scalings of the axes are first determined so that the grid defined by the nodes in `g1` and `g2` lies exactly in the unit square (i.e. on [0,1] by [0,1]). These scalings are applied to `g1`, `g2`, `d1` and `d2`. The minimum Euclidean distance from each node to a datum is then determined and if it is greater than `dist` the corresponding entry in the returned array is set to `TRUE` (otherwise to `FALSE`). The distance calculations are performed in compiled code for speed without storage overheads.

Value

A logical array with `TRUE` indicating a node in the grid defined by `g1`, `g2` that is 'too far' from any datum.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

http://www.maths.bath.ac.uk/~sw283/

See Also

`vis.gam`

Examples

```
library(mgcv)
x<-rnorm(100);y<-rnorm(100) # some "data"
n<-40 # generate a grid....
mx<-seq(min(x),max(x),length=n)
my<-seq(min(y),max(y),length=n)
gx<-rep(mx,n);gy<-rep(my,rep(n,n))
tf<-exclude.too.far(gx,gy,x,y,0.1)
plot(gx[!tf],gy[!tf],pch=".");points(x,y,col=2)
```

---

extract.lme.cov *Extract the data covariance matrix from an lme object*

Description

This is a service routine for gamm. Extracts the estimated covariance matrix of the data from an lme object, allowing the user control about which levels of random effects to include in this calculation. extract.lme.cov forms the full matrix explicitly: extract.lme.cov2 tries to be more economical than this.

Usage

```
extract.lme.cov(b,data,start.level=1)
extract.lme.cov2(b,data,start.level=1)
```

Arguments

b	A fitted model object returned by a call to lme
data	The data frame/ model frame that was supplied to lme.
start.level	The level of nesting at which to start including random effects in the calculation. This is used to allow smooth terms to be estimated as random effects, but treated like fixed effects for variance calculations.

Details

The random effects, correlation structure and variance structure used for a linear mixed model combine to imply a covariance matrix for the response data being modelled. These routines extracts that covariance matrix. The process is slightly complicated, because different components of the fitted model object are stored in different orders (see function code for details!).

The extract.lme.cov calculation is not optimally efficient, since it forms the full matrix, which may in fact be sparse. extract.lme.cov2 is more efficient. If the covariance matrix is diagonal, then only the leading diagonal is returned; if it can be written as a block diagonal matrix (under some permutation of the original data) then a list of matrices defining the non-zero blocks is returned along with an index indicating which row of the original data each row/column of the block diagonal matrix relates to. The block sizes are defined by the coarsest level of grouping in the random effect structure.

gamm uses extract.lme.cov2.

extract.lme.cov does not currently deal with the situation in which the grouping factors for a correlation structure are finer than those for the random effects. extract.lme.cov2 does deal with this situation.

Value

For `extract.lme.cov` an estimated covariance matrix.

For `extract.lme.cov2` a list containing the estimated covariance matrix and an indexing array. The covariance matrix is stored as the elements on the leading diagonal, a list of the matrices defining a block diagonal matrix, or a full matrix if the previous two options are not possible.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

For `lme` see:

Pinheiro J.C. and Bates, D.M. (2000) Mixed effects Models in S and S-PLUS. Springer

For details of how GAMMs are set up here for estimation using `lme` see:

Wood, S.N. (2006) Low rank scale invariant tensor product smooths for Generalized Additive Mixed Models. Biometrics

`http://www.maths.bath.ac.uk/~sw283/`

See Also

`gamm`, `formXtViX`

Examples

```
library(nlme)
data(Rail)
b <- lme(travel~1,Rail,~1|Rail)
extract.lme.cov(b,Rail)
```

`fix.family.link` *Modify families for use in GAM fitting*

Description

Generalized Additive Model fitting by 'outer' iteration, requires extra derivatives of the variance and link functions to be added to family objects. The functions add what is needed.

Usage

```
fix.family.link(fam)
fix.family.var(fam)
```

Arguments

`fam`	A `family`.

Details

Outer iteration GAM estimation requires derivatives of the GCV/UBRE score, which are obtained by differentiating the P-IRLS GAM fitting iteration w.r.t. the model smoothing parameters. The expressions for the derivatives require the second derivative of the link w.r.t. the mean, and the first derivative of the variance function w.r.t. the mean. These functions add functions evaluating these quantities to a family.

If the family already has functions `dvar` and `d2link` then these functions simply return the family unmodified: this allows non-standard links to be used with `gam` when using outer iteration (performance iteration operates with unmodified families).

The `dvar` function is a function of a mean vector, `mu`, and returns a vector of corresponding first derivatives of the family variance function. The `d2link` function is also a function of a vector of mean values, `mu`: it returns a vector of second derivatives of the link, evaluated at `mu`.

If modifying your own family, note that you can often get away with supplying only a `dvar` function if your family only requires links that occur in one of the standard families.

Value

A family object with extra component functions `dvar` and `d2link`.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

`fixDependence` *Detect linear dependencies of one matrix on another*

Description

Identifies columns of a matrix `X2` which are linearly dependent on columns of a matrix `X1`. Primarily of use in setting up identifiability constraints for nested GAMs.

Usage

```
fixDependence(X1,X2,tol=.Machine$double.eps^.5)
```

Arguments

`X1`	A matrix.
`X2`	A matrix, the columns of which may be partially linearly dependent on the columns of `X1`.
`tol`	The tolernce to use when assessing linear dependence.

Details

The algorithm uses a simple approach based on QR decomposition: see code for details.

Value

An array of the columns of `X2` which are linearly dependent on columns of `X1`. `NULL` if the two matrices are independent.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

Examples

```
n<-20;c1<-4;c2<-7
X1<-matrix(runif(n*c1),n,c1)
X2<-matrix(runif(n*c2),n,c2)
X2[,3]<-X1[,2]+X2[,4]*.1
X2[,5]<-X1[,1]*.2+X1[,2]*.04
fixDependence(X1,X2)
```

---

formula.gam *Extract the formula from a gam object*

Description

Extracts the formula from a fitted gam object.

Usage

```
formula.gam(x,...)
```

Arguments

x	fitted model objects of class gam (see gamObject) as produced by gam().
...	un-used in this case

Details

Returns x$formula. Provided so that anova methods print an appropriate description of the model.

Value

A model formula.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

See Also

gam

formXtViX *Form component of GAMM covariance matrix*

Description

This is a service routine for gamm. Given, V, an estimated covariance matrix obtained using extract.lme.cov2 this routine forms $X^T V^{-1} X$ as efficiently as possible, given the structure of V (usually sparse).

Usage

```
formXtViX(V,X)
```

Arguments

V	A data covariance matrix list returned from extract.lme.cov2
X	A model matrix.

Details

The covariance matrix returned by extract.lme.cov2 may be in a packed and re-ordered format, since it is usually sparse. Hence a special service routine is required to form the required products involving this matrix.

Value

A matrix.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

For lme see:

Pinheiro J.C. and Bates, D.M. (2000) Mixed effects Models in S and S-PLUS. Springer

For details of how GAMMs are set up here for estimation using lme see:

Wood, S.N. (2006) Low rank scale invariant tensor product smooths for Generalized Additive Mixed Models. Biometrics

http://www.maths.bath.ac.uk/~sw283/

See Also

gamm, extract.lme.cov2

full.score *GCV/UBRE score for use within nlm*

Description

Evaluates GCV/UBRE score for a GAM, given smoothing parameters. The routine calls gam.fit to fit the model, and is usually called by nlm to optimize the smoothing parameters.

This is basically a service routine for gam, and is not usually called directly by users. It is only used in this context for GAMs fitted by outer iteration (see gam.outer) when the the outer method is "nlm.fd" (see gam.method).

Usage

```
full.score(sp,G,family,control,gamma,pearson,...)
```

Arguments

sp	The logs of the smoothing parameters
G	a list returned by gam.setup
family	The family object for the GAM.
control	a list returned be gam.control
gamma	the degrees of freedom inflation factor (usually 1).
pearson	whether the GCV/UBRE score should be based on the Pearson statistic or the deviance. The deviance is usually preferable.
...	other arguments, typically for passing on to gam.fit.

Value

The value of the GCV/UBRE score, with attribute "full.gam.object" which is the full object returned by gam.fit.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

gam *Generalized additive models with integrated smoothness estimation*

Description

Fits a generalized additive model (GAM) to data. The degree of smoothness of model terms is estimated as part of fitting; isotropic or scale invariant smooths of any number of variables are available as model terms; confidence/credible intervals are readily available for any quantity predicted using a fitted model; gam is extendable: i.e. users can add smooths.

Smooth terms are represented using penalized regression splines (or similar smoothers) with smoothing parameters selected by GCV/UBRE or by regression splines with fixed degrees of freedom (mixtures of the two are permitted). Multi-dimensional smooths are available using penalized

thin plate regression splines (isotropic) or tensor product splines (when an isotropic smooth is inappropriate). For more on specifying models see gam.models. For more on model selection see gam.selection. For faster fits use the "cr" bases for smooth terms, te smooths for smooths of several variables, and performance iteration for smoothing parameter estimation (see gam.method). For large datasets see warnings.

gam() is not a clone of what S-PLUS provides: the major differences are (i) that by default estimation of the degree of smoothness of model terms is part of model fitting, (ii) a Bayesian approach to variance estimation is employed that makes for easier confidence interval calculation (with good coverage probabilites) and (iii) the facilities for incorporating smooths of more than one variable are different: specifically there are no lo smooths, but instead (a) s terms can have more than one argument, implying an isotropic smooth and (b) te smooths are provided as an effective means for modelling smooth interactions of any number of variables via scale invariant tensor product smooths. If you want a clone of what S-PLUS provides use gam from package gam.

Usage

```
gam(formula,family=gaussian(),data=list(),weights=NULL,subset=NULL,
    na.action,offset=NULL,control=gam.control(),method=gam.method(),
    scale=0,knots=NULL,sp=NULL,min.sp=NULL,H=NULL,gamma=1,
    fit=TRUE,G=NULL,in.out,...)
```

Arguments

formula

A GAM formula (see also gam.models). This is exactly like the formula for a GLM except that smooth terms can be added to the right hand side of the formula (and a formula of the form y ~ . is not allowed). Smooth terms are specified by expressions of the form:
s(var1,var2,...,k=12,fx=FALSE,bs="tp",by=a.var) where var1, var2, etc. are the covariates which the smooth is a function of and k is the dimension of the basis used to represent the smooth term. If k is not specified then k=10*3^(d-1) is used where d is the number of covariates for this term. fx is used to indicate whether or not this term has a fixed number of degrees of freedom (fx=FALSE to select d.f. by GCV/UBRE). bs indicates the basis to use for the smooth: for a full list see s, but note that the default "tp", while it possesses nice optimality properties is slow and memory hungry for very large datasets (but see examples for how to get around this). by can be used to specify a variable by which the smooth should be multiplied. For example gam(y~z+s(x,by=z)) would specify a model $E(y) = f(x)z$ where $f(\cdot)$ is a smooth function (the formula is y~x+s(x,by=z) rather than y~s(x,by=z) because the smooths are always set up to sum to zero over the covariate values). The by option is particularly useful for models in which different functions of the same variable are required for each level of a factor and for 'variable parameter models': see s.

An alternative for specifying smooths of more than one covariate is e.g.:
te(x,z,bs=c("tp","tp"),m=c(2,3),k=c(5,10)) which would specify a tensor product smooth of the two covariates x and z constructed from marginal t.p.r.s. bases of dimension 5 and 10 with marginal penalties of order 2 and 3. Any combination of basis types is possible, as is any number of covariates.

Formulae can involve nested or "overlapping" terms such as
y~s(x)+s(z)+s(x,z) or y~s(x,z)+s(z,v): see gam.side for further details and examples.

`family` This is a family object specifying the distribution and link to use in fitting etc. See `glm` and `family` for more details. The negative binomial families provided by the MASS library can be used, with or without known θ parameter: see `gam.neg.bin` for details.

`data` A data frame containing the model response variable and covariates required by the formula. By default the variables are taken from `environment(formula)`: typically the environment from which `gam` is called.

`weights` prior weights on the data.

`subset` an optional vector specifying a subset of observations to be used in the fitting process.

`na.action` a function which indicates what should happen when the data contain 'NA's. The default is set by the 'na.action' setting of 'options', and is 'na.fail' if that is unset. The "factory-fresh" default is 'na.omit'.

`offset` Can be used to supply a model offset for use in fitting. Note that this offset will always be completely ignored when predicting, unlike an offset included in `formula`: this conforms to the behaviour of `lm` and `glm`.

`control` A list of fit control parameters returned by `gam.control`.

`method` A list controlling the fitting methods used. This can make a big difference to computational speed, and, in some cases, reliability of convergence: see `gam.method` for details.

`scale` If this is zero then GCV is used for all distributions except Poisson and binomial where UBRE is used with scale parameter assumed to be 1. If this is greater than 1 it is assumed to be the scale parameter/variance and UBRE is used: to use the negative binomial in this case θ must be known. If `scale` is negative GCV is always used, which means that the scale parameter will be estimated by GCV and the Pearson estimator, or in the case of the negative binomial θ will be estimated in order to force the GCV/Pearson scale estimate to unity (if this is possible). For binomial models in particular, it is probably worth comparing UBRE and GCV results; for "over-dispersed Poisson" GCV is probably more appropriate than UBRE.

`knots` this is an optional list containing user specified knot values to be used for basis construction. For the `cr` and `cc` bases the user simply supplies the knots to be used, and there must be the same number as the basis dimension, `k`, for the smooth concerned. For the `tp` basis `knots` has two uses. Firstly, for large datasets the calculation of the `tp` basis can be time-consuming. The user can retain most of the advantages of the t.p.r.s. approach by supplying a reduced set of covariate values from which to obtain the basis - typically the number of covariate values used will be substantially smaller than the number of data, and substantially larger than the basis dimension, `k`. The second possibility is to avoid the eigen-decomposition used to find the t.p.r.s. basis altogether and simply use the basis implied by the chosen knots: this will happen if the number of knots supplied matches the basis dimension, `k`. For a given basis dimension the second option is faster, but gives poorer results (and the user must be quite careful in choosing knot locations). Different terms can use different numbers of knots, unless they share a covariate.

`sp` A vector of smoothing parameters for each term can be provided here. Smoothing parameters must be supplied in the order that the smooth terms appear in the model formula. With fit method `"magic"` (see `gam.control` and `magic`) then negative elements indicate that the parameter should be estimated, and

hence a mixture of fixed and estimated parameters is possible. With fit method "mgcv", if sp is supplied then all its elements must be positive. Note that fx=TRUE in a smooth term over-rides what is supplied here effectively setting the smoothing parameter to zero.

min.sp — for fit method "magic" only, lower bounds can be supplied for the smoothing parameters. Note that if this option is used then the smoothing parameters sp, in the returned object, will need to be added to what is supplied here to get the actual smoothing parameters. Lower bounds on the smoothing parameters can sometimes help stabilize otherwise divergent P-IRLS iterations.

H — With fit method "magic" a user supplied fixed quadratic penalty on the parameters of the GAM can be supplied, with this as its coefficient matrix. A common use of this term is to add a ridge penalty to the parameters of the GAM in circumstances in which the model is close to un-identifiable on the scale of the linear predictor, but perfectly well defined on the response scale.

gamma — It is sometimes useful to inflate the model degrees of freedom in the GCV or UBRE score by a constant multiplier. This allows such a multiplier to be supplied if fit method is "magic".

fit — If this argument is TRUE then gam sets up the model and fits it, but if it is FALSE then the model is set up and an object G is returned which is the output from gam.setup plus some extra items required to complete the GAM fitting process.

G — Usually NULL, but may contain the object returned by a previous call to gam with fit=FALSE, in which case all other arguments are ignored except for gamma, in.out, control, method and fit.

in.out — optional list for initializing outer iteration. If supplied then this must contain two elements: sp should be an array of initialization values for all smoothing parameters (there must be a value for all smoothing parameters, whether fixed or to be estimated, but those for fixed s.p.s are not used); scale is the typical scale of the GCV/UBRE function, for passing to the outer optimizer.

... — further arguments for passing on e.g. to gam.fit (such as mustart).

Details

A generalized additive model (GAM) is a generalized linear model (GLM) in which the linear predictor is given by a user specified sum of smooth functions of the covariates plus a conventional parametric component of the linear predictor. A simple example is:

$$\log(E(y_i)) = f_1(x_{1i}) + f_2(x_{2i})$$

where the (independent) response variables $y_i \sim$ Poi, and f_1 and f_2 are smooth functions of covariates x_1 and x_2. The log is an example of a link function.

If absolutely any smooth functions were allowed in model fitting then maximum likelihood estimation of such models would invariably result in complex overfitting estimates of f_1 and f_2. For this reason the models are usually fit by penalized likelihood maximization, in which the model (negative log) likelihood is modified by the addition of a penalty for each smooth function, penalizing its 'wiggliness'. To control the tradeoff between penalizing wiggliness and penalizing badness of fit each penalty is multiplied by an associated smoothing parameter: how to estimate these parameters, and how to practically represent the smooth functions are the main statistical questions introduced by moving from GLMs to GAMs.

The mgcv implementation of gam represents the smooth functions using penalized regression splines, and by default uses basis functions for these splines that are designed to be optimal, given the number basis functions used. The smooth terms can be functions of any number of covariates and the user has some control over how smoothness of the functions is measured.

gam in mgcv solves the smoothing parameter estimation problem by using the Generalized Cross Validation (GCV) criterion

$$nD/(n - DoF)^2$$

or an Un-Biased Risk Estimator (UBRE)criterion

$$D/n + 2sDoF/n - s$$

where D is the deviance, n the number of data, s the scale parameter and DoF the effective degrees of freedom of the model. Notice that UBRE is effectively just AIC rescaled, but is only used when s is known. It is also possible to replace D by the Pearson statistic (see gam.method), but this can lead to over smoothing. Smoothing parameters are chosen to minimize the GCV or UBRE score for the model, and the main computational challenge solved by the mgcv package is to do this efficiently and reliably. Various alternative numerical methods are provided: see gam.method.

Broadly gam works by first constructing basis functions and one or more quadratic penalty coefficient matrices for each smooth term in the model formula, obtaining a model matrix for the strictly parametric part of the model formula, and combining these to obtain a complete model matrix (/design matrix) and a set of penalty matrices for the smooth terms. Some linear identifiability constraints are also obtained at this point. The model is fit using gam.fit, a modification of glm.fit. The GAM penalized likelihood maximization problem is solved by penalized Iteratively Reweighted Least Squares (IRLS) (see e.g. Wood 2000). At each iteration a penalized weighted least squares problem is solved, and the smoothing parameters of that problem are estimated by GCV or UBRE. Eventually both model parameter estimates and smoothing parameter estimates converge. Alternatively the P-IRLS scheme is iterated to convergence for each trial set of smoothing parameters, and GCV or UBRE scores are only evaluated on convergence - optimization is then 'outer' to the P-IRLS loop: in this case extra derivatives have to be carried along with the P-IRLS iteration, to facilitate optimization, and gam.fit2 is used.

Five alternative basis-penalty types are built in for representing model smooths, but alternatives can easily be added (see smooth.construct which uses p-splines to illustrate how to add new smooths). The built in alternatives for univariate smooths terms are: a conventional penalized cubic regression spline basis, parameterized in terms of the function values at the knots; a cyclic cubic spline with a similar parameterization and thin plate regression splines. The cubic spline bases are computationally very efficient, but require 'knot' locations to be chosen (automatically by default). The thin plate regression splines are optimal low rank smooths which do not have knots, but are more computationally costly to set up. Smooths of several variables can be represented using thin plate regression splines, or tensor products of any available basis including user defined bases (tensor product penalties are obtained automatically form the marginal basis penalties). The t.p.r.s. basis is isotropic, so if this is not appropriate tensor product terms should be used. Tensor product smooths have one penalty and smoothing parameter per marginal basis, which means that the relative scaling of covariates is essentially determined automatically by GCV/UBRE. The t.p.r.s. basis and cubic regression spline bases are both available with either conventional 'wiggliness penalties' or penalties augmented with a shrinkage component: the conventional penalties treat some space of functions as 'completely smooth' and do not penalize such functions at all; the penalties with extra shrinkage will zero a term altogether for high enough smoothing parameters: gam.selection has an example of the use of such terms.

For any basis the user specifies the dimension of the basis for each smooth term. The dimension of the basis is one more than the maximum degrees of freedom that the term can have, but usually the term will be fitted by penalized maximum likelihood estimation and the actual degrees of freedom

will be chosen by GCV. However, the user can choose to fix the degrees of freedom of a term, in which case the actual degrees of freedom will be one less than the basis dimension.

Thin plate regression splines are constructed by starting with the basis for a full thin plate spline and then truncating this basis in an optimal manner, to obtain a low rank smoother. Details are given in Wood (2003). One key advantage of the approach is that it avoids the knot placement problems of conventional regression spline modelling, but it also has the advantage that smooths of lower rank are nested within smooths of higher rank, so that it is legitimate to use conventional hypothesis testing methods to compare models based on pure regression splines. The t.p.r.s. basis can become expensive to calculate for large datasets. In this case the user can supply a reduced set of knots to use in basis construction (see knots, in the argument list), or use tensor products of cheaper bases.

In the case of the cubic regression spline basis, knots of the spline are placed evenly throughout the covariate values to which the term refers: For example, if fitting 101 data with an 11 knot spline of `x` then there would be a knot at every 10th (ordered) `x` value. The parameterization used represents the spline in terms of its values at the knots. The values at neighbouring knots are connected by sections of cubic polynomial constrained to be continuous up to and including second derivative at the knots. The resulting curve is a natural cubic spline through the values at the knots (given two extra conditions specifying that the second derivative of the curve should be zero at the two end knots). This parameterization gives the parameters a nice interpretability.

Details of the underlying fitting methods are given in Wood (2000, 2004a, 2006b).

Value

If `fit == FALSE` the function returns a list `G` of items needed to fit a GAM, but doesn't actually fit it.

Otherwise the function returns an object of class `"gam"` as described in `gamObject`.

WARNINGS

If fit method `"mgcv"` is selected, the code does not check for rank deficiency of the model matrix that may result from lack of identifiability between the parametric and smooth components of the model.

You must have more unique combinations of covariates than the model has total parameters. (Total parameters is sum of basis dimensions plus sum of non-spline terms less the number of spline terms).

Automatic smoothing parameter selection is not likely to work well when fitting models to very few response data.

With large datasets (more than a few thousand data) the `"tp"` basis gets very slow to use: use the `knots` argument as discussed above and shown in the examples. Alternatively, for 1-d smooths you can use the `"cr"` basis and for multi-dimensional smooths use `te` smooths.

For data with many zeroes clustered together in the covariate space it is quite easy to set up GAMs which suffer from identifiability problems, particularly when using Poisson or binomial families. The problem is that with e.g. log or logit links, mean value zero corresponds to an infinite range on the linear predictor scale. Some regularization is possible in such cases: see `gam.control` for details.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

Front end design inspired by the S function of the same name based on the work of Hastie and Tibshirani (1990). Underlying methods owe much to the work of Wahba (e.g. 1990) and Gu (e.g. 2002).

References

Key References on this implementation:

Wood, S.N. (2000) Modelling and Smoothing Parameter Estimation with Multiple Quadratic Penalties. J.R.Statist.Soc.B 62(2):413-428

Wood, S.N. (2003) Thin plate regression splines. J.R.Statist.Soc.B 65(1):95-114

Wood, S.N. (2004a) Stable and efficient multiple smoothing parameter estimation for generalized additive models. J. Amer. Statist. Ass. 99:673-686

Wood, S.N. (2004b) On confidence intervals for GAMs based on penalized regression splines. Technical Report 04-12 Department of Statistics, University of Glasgow.

Wood, S.N. (2006a) Low rank scale invariant tensor product smooths for generalized additive mixed models. Biometrics

Wood S.N. (2006b) Generalized Additive Models: An Introduction with R. CRC Press.

Key Reference on GAMs and related models:

Hastie (1993) in Chambers and Hastie (1993) Statistical Models in S. Chapman and Hall.

Hastie and Tibshirani (1990) Generalized Additive Models. Chapman and Hall.

Wahba (1990) Spline Models of Observational Data. SIAM

Background References:

Green and Silverman (1994) Nonparametric Regression and Generalized Linear Models. Chapman and Hall.

Gu and Wahba (1991) Minimizing GCV/GML scores with multiple smoothing parameters via the Newton method. SIAM J. Sci. Statist. Comput. 12:383-398

Gu (2002) Smoothing Spline ANOVA Models, Springer.

O'Sullivan, Yandall and Raynor (1986) Automatic smoothing of regression functions in generalized linear models. J. Am. Statist.Ass. 81:96-103

Wood (2001) mgcv:GAMs and Generalized Ridge Regression for R. R News 1(2):20-25

Wood and Augustin (2002) GAMs with integrated model selection using penalized regression splines and applications to environmental modelling. Ecological Modelling 157:157-177

http://www.maths.bath.ac.uk/~sw283/

See Also

mgcv-package, gamObject, gam.models, s, predict.gam,plot.gam, summary.gam, gam.side, gam.selection,mgcv, gam.control gam.check, gam.neg.bin, magic,vis.gam

Examples

```
library(mgcv)
set.seed(0)
n<-400
sig<-2
x0 <- runif(n, 0, 1)
x1 <- runif(n, 0, 1)
x2 <- runif(n, 0, 1)
x3 <- runif(n, 0, 1)
f0 <- function(x) 2 * sin(pi * x)
f1 <- function(x) exp(2 * x)
```

```
f2 <- function(x) 0.2*x^11*(10*(1-x))^6+10*(10*x)^3*(1-x)^10
f3 <- function(x) 0*x
f <- f0(x0) + f1(x1) + f2(x2)
e <- rnorm(n, 0, sig)
y <- f + e
b<-gam(y~s(x0)+s(x1)+s(x2)+s(x3))
summary(b)
plot(b,pages=1,residuals=TRUE)
# same fit in two parts .....
G<-gam(y~s(x0)+s(x1)+s(x2)+s(x3),fit=FALSE)
b<-gam(G=G)
# an extra ridge penalty (useful with convergence problems) ....
bp<-gam(y~s(x0)+s(x1)+s(x2)+s(x3),H=diag(0.5,37))
print(b);print(bp);rm(bp)
# set the smoothing parameter for the first term, estimate rest ...
bp<-gam(y~s(x0)+s(x1)+s(x2)+s(x3),sp=c(0.01,-1,-1,-1))
plot(bp,pages=1);rm(bp)
# set lower bounds on smoothing parameters ....
bp<-gam(y~s(x0)+s(x1)+s(x2)+s(x3),min.sp=c(0.001,0.01,0,10))
print(b);print(bp);rm(bp)

# now a GAM with 3df regression spline term & 2 penalized terms
b0<-gam(y~s(x0,k=4,fx=TRUE,bs="tp")+s(x1,k=12)+s(x2,k=15))
plot(b0,pages=1)
# now fit a 2-d term to x0,x1
b1<-gam(y~s(x0,x1)+s(x2)+s(x3))
par(mfrow=c(2,2))
plot(b1)
par(mfrow=c(1,1))

# now simulate poisson data
g<-exp(f/4)
y<-rpois(rep(1,n),g)
b2<-gam(y~s(x0)+s(x1)+s(x2)+s(x3),family=poisson)
plot(b2,pages=1)
# repeat fit using performance iteration
gm <- gam.method(gam="perf.magic")
b3<-gam(y~s(x0)+s(x1)+s(x2)+s(x3),family=poisson,method=gm)
plot(b3,pages=1)

# a binary example
g <- (f-5)/3
g <- binomial()$linkinv(g)
y <- rbinom(g,1,g)
lr.fit <- gam(y~s(x0)+s(x1)+s(x2)+s(x3),family=binomial)
## plot model components with truth overlaid in red
op <- par(mfrow=c(2,2))
for (k in 1:4) {
  plot(lr.fit,residuals=TRUE,select=k)
  xx <- sort(eval(parse(text=paste("x",k-1,sep=""))))
  ff <- eval(parse(text=paste("f",k-1,"(xx)",sep="")))
  lines(xx,(ff-mean(ff))/3,col=2)
}
par(op)
anova(lr.fit)
lr.fit1 <- gam(y~s(x0)+s(x1)+s(x2),family=binomial)
lr.fit2 <- gam(y~s(x1)+s(x2),family=binomial)
```

```
AIC(lr.fit,lr.fit1,lr.fit2)

# and a pretty 2-d smoothing example....
test1<-function(x,z,sx=0.3,sz=0.4)
{ (pi**sx*sz)*(1.2*exp(-(x-0.2)^2/sx^2-(z-0.3)^2/sz^2)+
  0.8*exp(-(x-0.7)^2/sx^2-(z-0.8)^2/sz^2))
}
n<-500
old.par<-par(mfrow=c(2,2))
x<-runif(n);z<-runif(n);
xs<-seq(0,1,length=30);zs<-seq(0,1,length=30)
pr<-data.frame(x=rep(xs,30),z=rep(zs,rep(30,30)))
truth<-matrix(test1(pr$x,pr$z),30,30)
contour(xs,zs,truth)
y<-test1(x,z)+rnorm(n)*0.1
b4<-gam(y~s(x,z))
fit1<-matrix(predict.gam(b4,pr,se=FALSE),30,30)
contour(xs,zs,fit1)
persp(xs,zs,truth)
vis.gam(b4)
par(old.par)
# very large dataset example using knots
n<-10000
x<-runif(n);z<-runif(n);
y<-test1(x,z)+rnorm(n)
ind<-sample(1:n,1000,replace=FALSE)
b5<-gam(y~s(x,z,k=50),knots=list(x=x[ind],z=z[ind]))
vis.gam(b5)
# and a pure "knot based" spline of the same data
b6<-gam(y~s(x,z,k=100),knots=list(x= rep((1:10-0.5)/10,10),
        z=rep((1:10-0.5)/10,rep(10,10))))
vis.gam(b6,color="heat")
```

`gam.check` *Some diagnostics for a fitted gam model*

Description

Takes a fitted `gam` object produced by `gam()` and produces some diagnostic information about the fitting procedure and results.

Usage

```
gam.check(b)
```

Arguments

`b`	a fitted `gam` object as produced by `gam()`.

Details

This function plots 4 standard diagnostic plots, and some other convergence diagnostics. Output differs depending on whether the underlying fitting method was `mgcv` or another method (see `gam.method`).

For mgcv based fits , the first plot shows the GCV or UBRE score against model degrees of freedom, given the final estimates of the relative smoothing parameters for the model. This is a slice through the GCV/UBRE score function that passes through the minimum found during fitting. Although not conclusive (except in the single smoothing parameter case), a lack of multiple local minima on this plot is suggestive of a lack of multiple local minima in the GCV/UBRE function and is therefore a good thing. Multiple local minima on this plot indicates that the GCV/UBRE function may have multiple local minima, but in a multiple smoothing parameter case this is not conclusive - multiple local minima on one slice through a function do not necessarily imply that the function has multiple local minima. A 'good' plot here is a smooth curve with only one local minimum (which is therefore its global minimum).

The location of the minimum used for the fitted model is also marked on the first plot. Sometimes this location may be a local minimum that is not the global minimum on the plot. There is a legitimate reason for this to happen, and it does not always indicate problems. Smoothing parameter selection is based on applying GCV/UBRE to the approximating linear model produced by the GLM IRLS fitting method employed in `gam.fit()`. It is sometimes possible for these approximating models to develop 'phantom' minima in their GCV/UBRE scores. These minima usually imply a big change in model parameters, and have the characteristic that the minimia will not be present in the GCV/UBRE score of the approximating model that would result from actually applying this parameter change. In other words, these are spurious minima in regions of parameter space well beyond those for which the weighted least squares problem can be expected to represent the real underlying likelihood well. Such minima can lead to convergence problems. To help ensure convergence even in the presence of phantom minima, `gam.fit` switches to a cautious optimization mode after a user controlled number of iterations of the IRLS algorithm (see gam.control). In the presence of local minima in the GCV/UBRE score, this method selects the minimum that leads to the smallest change in model estimated degrees of freedom. This approach is usually sufficient to deal with phantom minima. Setting `trace` to `TRUE` in `gam.control` will allow you to check exactly what is happening.

If the location of the point indicating the minimum is not on the curve showing the GCV/UBRE function then there are numerical problems with the estimation of the effective degrees of freedom: this usually reflects problems with the relative scaling of covariates that are arguments of a single smooth. In this circumstance reported estimated degrees of freedom can not be trusted, although the fitted model and term estimates are likely to be quite acceptable.

If the fit method is based on `magic` or `gam.fit2` then there is no global search and the problems with phantom local minima are much reduced. The first plot in this case will simply be a normal QQ plot of the standardized residuals.

The other 3 plots are two residual plots and plot of fitted values against original data.

The function also prints out information about the convergence of the GCV minimization algorithm, indicating how many iterations were required to minimise the GCV/UBRE score. A message is printed if the minimization terminated by failing to improve the score with a steepest descent step: otherwise minimization terminated by meeting convergence criteria. The mean absolute gradient or RMS gradient of the GCV/UBRE function at the minimum is given. An indication of whether or not the Hessian of the GCV/UBRE function is positive definite is given. If some smoothing parameters are not well defined (effectively zero, or infinite) then it may not be, although this is not usually a problem. If the fit method is `mgcv`, a message is printed if the second guess smoothing parameters did not improve on the first guess - this is primarily there for the developer. For other fit methods the estimated rank of the model is printed.

The ideal results from this function have a smooth, single minima GCV/UBRE plot (where appropriate), good residual plots, and convergence to small gradients with a positive definite Hessian. However, failure to meet some of these criteria is often acceptable, and the information provided is primarily of use in diagnosing suspected problems. High gradients at convergence are a clear indication of problems, however.

Fuller data can be extracted from `mgcv.conv` part of the `gam` object.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

Wood, S.N. (2000) Modelling and Smoothing Parameter Estimation with Multiple Quadratic Penalties. J.R.Statist.Soc.B 62(2):413-428

Wood, S.N. (2003) Thin plate regression splines. J.R.Statist.Soc.B 65(1):95-114

http://www.maths.bath.ac.uk/~sw283/

See Also

`choose.k`, `gam`, `mgcv`, `magic`

Examples

```
library(mgcv)
set.seed(0)
n<-200
sig<-2
x0 <- runif(n, 0, 1)
x1 <- runif(n, 0, 1)
x2 <- runif(n, 0, 1)
x3 <- runif(n, 0, 1)
y <- 2 * sin(pi * x0)
y <- y + exp(2 * x1) - 3.75887
y <- y+0.2*x2^11*(10*(1-x2))^6+10*(10*x2)^3*(1-x2)^10-1.396
e <- rnorm(n, 0, sig)
y <- y + e
b<-gam(y~s(x0)+s(x1)+s(x2)+s(x3))
plot(b,pages=1)
gam.check(b)
```

---

gam.control *Setting GAM fitting defaults*

Description

This is an internal function of package `mgcv` which allows control of the numerical options for fitting a GAM. Typically users will want to modify the defaults if model fitting fails to converge, or if the warnings are generated which suggest a loss of numerical stability during fitting. To change the default choise of fitting method, see `gam.method`.

Usage

```
gam.control(irls.reg=0.0,epsilon = 1e-06, maxit = 100,globit = 20,
            mgcv.tol=1e-7,mgcv.half=15,nb.theta.mult=10000, trace = FALSE,
            rank.tol=.Machine$double.eps^0.5,absorb.cons=TRUE,
            max.tprs.knots=5000,nlm=list(),optim=list(),newton=list(),
            outerPIsteps=1)
```

Arguments

irls.reg	For most models this should be 0. The iteratively re-weighted least squares method by which GAMs are fitted can fail to converge in some circumstances. For example, data with many zeroes can cause problems in a model with a log link, because a mean of zero corresponds to an infinite range of linear predictor values. Such convergence problems are caused by a fundamental lack of identifiability, but do not show up as lack of identifiability in the penalized linear model problems that have to be solved at each stage of iteration. In such circumstances it is possible to apply a ridge regression penalty to the model to impose identifiability, and irls.reg is the size of the penalty. The penalty can not be used if the underlying fitting method is mgcv (not the default - see gam.method for details).
epsilon	This is used for judging conversion of the GLM IRLS loop in gam.fit or gam.fit2 .
maxit	Maximum number of IRLS iterations to perform using cautious GCV/UBRE optimization, after globit IRLS iterations with normal GCV optimization have been performed. Note that only fitting based on mgcv (not default) makes any distinction between cautious and global optimization.
globit	Maximum number of IRLS iterations to perform with normal GCV/UBRE optimization. If convergence is not achieved after these iterations then a further maxit iterations will be performed using cautious GCV/UBRE optimization.
mgcv.tol	The convergence tolerance parameter to use in GCV/UBRE optimization.
mgcv.half	If a step of the GCV/UBRE optimization method leads to a worse GCV/UBRE score, then the step length is halved. This is the number of halvings to try before giving up.
nb.theta.mult	Controls the limits on theta when negative binomial parameter is to be estimated. Maximum theta is set to the initial value multiplied by nb.theta.mult, while the minimum value is set to the initial value divided by nb.theta.mult.
trace	Set this to TRUE to turn on diagnostic output.
rank.tol	The tolerance used to estimate the rank of the fitting problem, for methods which deal with rank deficient cases (basically all except those based on mgcv).
absorb.cons	If TRUE then the GAM is set up using a parameterization which requires no further constraint. Usually this means that all the smooths are automatically centered (i.e. they sum to zero over the covariate values). If FALSE then the ordinary parameterizations of the smooths are used, which require constraints to be imposed during fitting.
max.tprs.knots	This is the default initial maximum number of knots to allow when constructing a t.p.r.s bases (bs="tp"). The set up cost (and storage) for these smooths scales as the square of the number of initial knots, so if it's too high you can appear to freeze R. Usually one would want to use an alternative smoothing basis (or te terms), or the approach illustrated in the examples in gam, rather than simply increasing this default.
nlm	list of control parameters to pass to nlm if this is used for outer estimation of smoothing parameters. See details.
optim	list of control parameters to pass to optim if this is used for outer estimation of smoothing parameters. See details.

`newton` list of control parameters to pass to default Newton optimizer used for outer estimation of log smoothing parameters. See details.

`outerPIsteps` The number of performance interation steps used to initialize outer iteration. Less than 1 means that only one performance iteration step is taken to get the function scale, but the corresponding smoothing parameter estimates are discarded.

Details

When outer iteration is used for fitting then the control list `nlm` stores control arguments for calls to routine `nlm`. The list has the following named elements: (i) `ndigit` is the number of significant digits in the GCV/UBRE score - by default this is worked out from `epsilon`; (ii) `gradtol` is the tolerance used to judge convergence of the gradient of the GCV/UBRE score to zero - by default set to `100*epsilon`; (iii) `stepmax` is the maximum allowable log smoothing parameter step - defaults to 2; (iv) `steptol` is the minimum allowable step length - defaults to 1e-4; (v) `iterlim` is the maximum number of optimization steps allowed - defaults to 200; (vi) `check.analyticals` indicates whether the built in exact derivative calculations should be checked numerically - defaults to `FALSE`. Any of these which are not supplied and named in the list are set to their default values.

Outer iteration using `newton` is controlled by the list `newton` with the following elements : `conv.tol` (default 1e-6) is the relative convergence tolerance; `maxNstep` is the maximum length allowed for an element of the Newton search direction (default 5); `maxSstep` is the maximum length allowed for an element of the steepest descent direction (only used if Newton fails - default 2); `maxHalf` is the maximum number of step halvings to permit before giving up (default 30); `use.svd` determines whether to use an SVD step or QR step for the part of fitting that determines problem rank, etc. (defaults to `FALSE`).

Outer iteration using `optim` is controlled using list `optim`, which currently has one element: `factr` which takes default value 1e7.

When fitting is been done by calls to routine `mgcv`, `maxit` and `globit` control the maximum iterations of the IRLS algorithm, as follows: the algorithm will first execute up to `globit` steps in which the GCV/UBRE algorithm performs a global search for the best overall smoothing parameter at every iteration. If convergence is not achieved within `globit` iterations, then a further `maxit` steps are taken, in which the overall smoothing parameter estimate is taken as the one locally minimising the GCV/UBRE score and resulting in the lowest EDF change. The difference between the two phases is only significant if the GCV/UBRE function develops more than one minima. The reason for this approach is that the GCV/UBRE score for the IRLS problem can develop 'phantom' minimima for some models: these are minima which are not present in the GCV/UBRE score of the IRLS problem resulting from moving the parameters to the minimum! Such minima can lead to convergence failures, which are usually fixed by the second phase.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

Gu and Wahba (1991) Minimizing GCV/GML scores with multiple smoothing parameters via the Newton method. SIAM J. Sci. Statist. Comput. 12:383-398

Wood, S.N. (2000) Modelling and Smoothing Parameter Estimation with Multiple Quadratic Penalties. J.R.Statist.Soc.B 62(2):413-428

Wood, S.N. (2003) Thin plate regression splines. J.R.Statist.Soc.B 65(1):95-114

Wood, S.N. (2004) Stable and efficient multiple smoothing parameter estimation for generalized additive models. J. Amer. Statist. Ass.

http://www.maths.bath.ac.uk/~sw283/

See Also

gam.method gam, gam.fit, glm.control

---

gam.convergence *GAM convergence and performance issues*

Description

When fitting GAMs there is a tradeoff between speed of fitting and probability of fit convergence. The default fitting options specified by gam.method (as the default for argument method of gam), always opt for certainty of convergence over speed of fit. In the additive modelling contexts this means using fitting routine magic rather than the slightly faster routine mgcv. In the Generalized Additive Model case it means using 'outer' iteration in preference to 'performance iteration': see gam.outer for details.

It is possible for the default 'outer' iteration to fail when finding intial smoothing parameters using a few steps of outer iteration (if you get a convergence failure message from magic when outer iterating, then this is what has happened): lower outerPIsteps in gam.control to fix this.

There are two things that you can do to speed up GAM fitting. (i) Change the method argument to gam so that 'performance iteration' is used in place of the default outer iteration. See the perf.magic option under gam.method, for example. Usually performance iteration converges well and is quick. (ii) For large datasets it may be worth changing the smoothing basis to use bs="cr" (see s for details) for 1-d smooths, and to use te smooths in place of s smooths for smooths of more than one variable. This is because the default thin plate regression spline basis "tp" is costly to set up for large datasets (much over 1000 data, say). Alternatively see the last few examples for gam.

If the GAM estimation process fails to converge when using performance iteration, then switch to outer iteration via the method argument of gam (see gam.method). If it still fails, try increasing the number of IRLS iterations (see gam.control) or perhaps experiment with the convergence tolerance.

If you still have problems, it's worth noting that a GAM is just a (penalized) GLM and the IRLS scheme used to estimate GLMs is not guaranteed to converge. Hence non convergence of a GAM may relate to a lack of stability in the basic IRLS scheme. Therefore it is worth trying to establish whether the IRLS iterations are capable of converging. To do this fit the problematic GAM with all smooth terms specified with fx=TRUE so that the smoothing parameters are all fixed at zero. If this 'largest' model can converge then, then the maintainer would quite like to know about your problem! If it doesn't converge, then its likely that your model is just too flexible for the IRLS process itself. Having tried increasing maxit in gam.control, there are several other possibilities for stabilizing the iteration. It is possible to try (i) setting lower bounds on the smoothing parameters using the min.sp argument of gam: this may or may not change the model being fitted; (ii) reducing the flexibility of the model by reducing the basis dimensions k in the specification of s and te model terms: this obviously changes the model being fitted somewhat; (iii) introduce a small regularization term into the fitting via the irls.reg argument of gam.control: this option obviously changes the nature of the fit somewhat, since parameter estimates are pulled towards zero by doing this.

Usually, a major contributer to fitting difficulties is that the model is a very poor description of the data.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

`gam.fit` *GAM P-IRLS estimation with GCV/UBRE smoothness estimation*

Description

This is an internal function of package `mgcv`. It is a modification of the function `glm.fit`, designed to be called from `gam`. The major modification is that rather than solving a weighted least squares problem at each IRLS step, a weighted, penalized least squares problem is solved at each IRLS step with smoothing parameters associated with each penalty chosen by GCV or UBRE, using routine `mgcv` or `magic`. For further information on usage see code for `gam`. Some regularization of the IRLS weights is also permitted as a way of addressing identifiability related problems (see `gam.control`). Negative binomial parameter estimation is supported.

The basic idea of estimating smoothing parameters at each step of the P-IRLS is due to Gu (1992), and is termed 'performance iteration' or 'performance oriented iteration'.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

Gu (1992) Cross-validating non-Gaussian data. J. Comput. Graph. Statist. 1:169-179

Gu and Wahba (1991) Minimizing GCV/GML scores with multiple smoothing parameters via the Newton method. SIAM J. Sci. Statist. Comput. 12:383-398

Wood, S.N. (2000) Modelling and Smoothing Parameter Estimation with Multiple Quadratic Penalties. J.R.Statist.Soc.B 62(2):413-428

Wood, S.N. (2004) Stable and efficient multiple smoothing parameter estimation for generalized additive models. J. Amer. Statist. Ass. 99:637-686

See Also

`gam.fit2`, `gam`, `mgcv`, `magic`

`gam.fit2` *P-IRLS GAM estimation with GCV & UBRE derivative calculation*

Description

Estimation of GAM smoothing parameters is most stable if optimization of the UBRE or GCV score is outer to the penalized iteratively re-weighted least squares scheme used to estimate the model given smoothing parameters.

These routines estimates a GAM given log smoothing paramaters, and evaluate derivatives of the GCV and UBRE scores of the model with respect to the log smoothing parameters. Calculation of exact derivatives is generally faster than approximating them by finite differencing, as well as generally improving the reliability of GCV/UBRE score minimization.

`gam.fit2` evaluates first derivatives, by accumulating them as P-IRLS progresses. `gam.fit3` uses a more efficient approach in which the P-IRLS is first run to convergence, and only then are the derivatives evaluated by a separate iteration. `gam.fit3` can evaluate second as well as first derivatives.

Not normally called directly, but rather service routines for `gam`.

Usage

```
gam.fit2(x, y, sp, S=list(),rS=list(),off, H=NULL,
         weights = rep(1, nobs), start = NULL, etastart = NULL,
         mustart = NULL, offset = rep(0, nobs), family = gaussian(),
         control = gam.control(), intercept = TRUE,deriv=TRUE,
         gamma=1,scale=1,pearson=FALSE,printWarn=TRUE,...)
gam.fit3(x, y, sp, S=list(),rS=list(),off, H=NULL,
         weights = rep(1, nobs), start = NULL, etastart = NULL,
         mustart = NULL, offset = rep(0, nobs), family = gaussian(),
         control = gam.control(), intercept = TRUE,deriv=2,use.svd=TRUE,
         gamma=1,scale=1,printWarn=TRUE,...)
```

Arguments

`x`	The model matrix for the GAM.
`y`	The response variable.
`sp`	The log smoothing parameters.
`S`	A list of penalty matrices. Typically penalty matrices contain only a smallish square sub-matrix which is non-zero: this is what is actually stored. `off[i]` indicates which parameter is the first one penalized by `S[[i]]`.
`rS`	List of square roots of penalty matrices, each having as few columns as possible, but as many rows as there are parameters.
`off`	`off[i]` indicates which parameter `S[[i]][1,1]` relates to.
`H`	The fixed penalty matrix for the model.
`weights`	prior weights for fitting.
`start`	optional starting parameter guesses.
`etastart`	optional starting values for the linear predictor.
`mustart`	optional starting values for the mean.
`offset`	the model offset
`family`	the family - actually this routine would never be called with `gaussian()`
`control`	control list as returned from `glm.control`
`intercept`	does the model have and intercept, `TRUE` or `FALSE`

deriv	Should derivatives of the GCV and UBRE scores be calculated? TRUE or FALSE for gam.fit2, but 0, 1 or 2, indicating the maximum order of differentiation to apply, for gam.fit3.
use.svd	Should the algorithm use SVD (TRUE) or the cheaper QR (FALSE) as the second matrix decomposition of the final derivative/coefficient evaluation method? Only used by gam.fit3.
gamma	The weight given to each degree of freedom in the GCV and UBRE scores can be varied (usually increased) using this parameter.
scale	The scale parameter - needed for the UBRE score.
pearson	The GCV/UBRE score can be based either on the Pearson statistic or the deviance. The latter is generally to be preferred, as it is less prone to severe undersmoothing. Only used by gam.fit2.
printWarn	Set to FALSE to suppress some warnings. Useful in order to ensure that some warnings are only printed if they apply to the final fitted model, rather than an intermediate used in optimization.
...	Other arguments: ignored.

Details

This routine is basically glm.fit with some modifications to allow (i) for quadratic penalties on the log likelihood; (ii) derivatives of the model coefficients with respect to log smoothing parameters to be obtained (by updating alongside the P-IRLS iteration) and (iii) derivatives of the GAM GCV and UBRE scores to be evaluated at convergence.

In addition the routine applies step halving to any step that increases the penalized deviance substantially.

The most costly parts of the calculation are performed by calls to compiled C code (which in turn calls LAPACK routines) in place of the compiled code that would usually perform least squares estimation on the working model in the IRLS iteration.

Estimation of smoothing parameters by optimizing GCV scores obtained at convergence of the P-IRLS iteration was proposed by O'Sullivan et al. (1986), and is here termed 'outer' iteration.

Note that use of non-standard families with this routine requires modification of the families as described in fix.family.link.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

The routine has been modified from glm.fit in R 2.0.1, written by the R core (see glm.fit for further credits).

References

O 'Sullivan, Yandall & Raynor (1986) Automatic smoothing of regression functions in generalized linear models. J. Amer. Statist. Assoc. 81:96-103.

http://www.maths.bath.ac.uk/~sw283/

See Also

gam.fit, gam, mgcv, magic

gam.method *Setting GAM fitting method*

Description

This is a function of package mgcv which allows selection of the numerical method used to optimize the smoothing parameter estimation criterion for a gam.

It is used to set argument method of gam.

Usage

```
gam.method(am="magic",gam="outer",outer="newton",gcv="deviance",
           family=NULL)
```

Arguments

am	Which method to use for a pure additive model (i.e. identity link, gaussian errors). Either "magic" if the Wood (2004) method (magic) is to be used, or "mgcv" if the faster, but less stable and general Wood (2000) method (mgcv) should be used.
gam	Which method to use in the generalized case (i.e. all case other than gaussian with identity link). "perf.magic" for the performance iteration (see details) with magic as the basic estimation engine. "perf.mgcv" for performance iteration with mgcv as the underlying estimation engine. "perf.outer" for magic based performance iteration followed by outer iteration (see details). "outer" for pure outer iteration.
outer	The optimization approach to use to optimize log smoothing parameters by outer iteration."newton" for modified Newton method backed up by steepest descent, based on exact first and second derivatives. "nlm" to use nlm with exact first derivatives to optimize the smoothness selection criterion. "nlm.fd" to use nlm with finite differenced first derivatives (slower and less reliable). "optim" to use the "L-BFGS-B" quasi-Newton method option of routine optim, with exact first derivatives.
gcv	One of "deviance", "GACV" or "pearson", specifying the flavour of GCV to use with outer iteration. "deviance" simply replaces the residual sum of squares term in a GCV score with the deviance, following Hastie and Tibshirani (1990, section 6.9). "GACV" (only available with outer method "newton") uses a varient of Gu and Xiang's (2001) generalized approximate cross validation, modified to deal with arbitrary link-error combinations. "pearson" uses a suggestion from the literature of replacing the RSS in a GCV score with the Pearson statistic: this is a very poor choise (see Wood, 2006, section 4.5.4). This latter option is not available with outer methods "newton" and "nlm".
family	The routine is called by gam to check the supplied method argument. In this circumstance the family argument is passed, to check that it works with the specified method. Negative binomial families only work with performance iteration, and the method is reset to this if necessary.

Details

The performance iteration was suggested by Gu (and is rather similar to the PQL method in generalized linear mixed modelling). At each step of the P-IRLS (penalized iteratively reweighted least squares) iteration, by which a gam is fitted, the smoothing parameters are estimated by GCV or UBRE applied to the working penalized linear modelling problem. In most cases, this process converges and gives smoothness estimates that perform well. It is usually very fast, since smoothing parameters are estimated alongside other model coefficients in a single P-IRLS process.

The performance iteration has two disadvantages. (i) in the presence of co-linearity or concurvity (a frequent problem when spatial smoothers are included in a model with other covariates) then the process can fail to converge. Suppose we start with some coefficient and smoothing parameter estimates, implying a working penalized linear model: the optimal smoothing parameters and coefficients for this working model may in turn imply a working model for which the original estimates are better than the most recent estimates. This sort of effect can prevent convergence.

Secondly it is often possible to find a set of smoothing parameters that result in a lower GCV or UBRE score, for the final working model, than the final score that results from the performance iterations. This is because the performance iteration is only approximately optimizing this score (since optimization is only performed on the working model). The disadvantage here is not that the model with lower score would perform better (it usually doesn't), but rather that it makes model comparison on the basis of GCV/UBRE score rather difficult.

Both disadvantages of performance iteration are surmountable by using what is basically O'Sullivan's (1986) suggestion. Here the P-IRLS scheme is iterated to convergence for a fixed set of smoothing parameters, with an appropriate GCV/UBRE score evaluated at convergence. This score at convergence is optimized in some way. This is termed `"outer"` optimization, since the optimization is outer to the P-IRLS loop. Outer iteration is slower than performance iteration.

The 'appropriate GCV/UBRE' score in the previous paragraph can be defined in one of two ways either (i) the deviance, or (ii) the Pearson statistic can be used in place of the residual sum of squares in the GCV/UBRE score. (ii) makes the GCV/UBRE score correspond to the score for the working linear model at convergence of the P-IRLS, but in practice tends to result in oversmoothing, particularly with low n binomial data, or low mean counts. Hence the default is to use (i).

Several alternative optimisation methods can be used for outer optimization.Usually the fastest and most reliable approach is to use a modified Newton optimizer with exact first and second derivatives, and this is the default. `nlm` can be used with finite differenced first derivatives. This is not ideal theoretically, since it is possible for the finite difference estimates of derivatives to be very badly in error on rare occasions when the P-IRLS convergence tolerance is close to being matched exactly, so that two components of a finite differenced derivative require different numbers of iterations of P-IRLS in their evaluation. An alternative is provided in which `nlm` uses numerically exact first derivatives, this is faster and less problematic than the other scheme. An further alternative is to use a quasi-Newton scheme with exact derivtives, based on `optim`. In practice this usually seems to be slower than the `nlm` method.

In summary: performance iteration is fast, but can fail to converge. Outer iteration is a little slower, but more reliable. At present only performance iteration is available for negative binomial families.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

Gu and Wahba (1991) Minimizing GCV/GML scores with multiple smoothing parameters via the Newton method. SIAM J. Sci. Statist. Comput. 12:383-398

Wood, S.N. (2000) Modelling and Smoothing Parameter Estimation with Multiple Quadratic Penalties. J.R.Statist.Soc.B 62(2):413-428

Wood, S.N. (2003) Thin plate regression splines. J.R.Statist.Soc.B 65(1):95-114

Wood, S.N. (2004) Stable and efficient multiple smoothing parameter estimation for generalized additive models. J. Amer. Statist. Ass.

http://www.maths.bath.ac.uk/~sw283/

See Also

gam.control gam, gam.fit, glm.control

---

gam.models *Specifying generalized additive models*

Description

This page is intended to provide some more information on how to specify GAMs. Assume that in general we are interested in modelling some response variable y_i with an exponential family distribution (e.g. Normal, Poisson, binomial, etc.) using predictor variables $x_{1i}, x_{2i}, \ldots$, and let $\mu_i \equiv E(y_i)$. A typical GAM might be:

$$g(\mu_i) = \beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + f_1(x_{3i}) + f_2(x_{4i}, x_{4i})$$

where g is a smooth monotonic 'link' function, and f_1 and f_2 are smooth functions. The key idea here is that the dependence of the response on the predictors can be represented as a parametric sub-model plus the sum of some smooth fucntions of one or more of the predictor variables. Thus the model is quite flexible relative to strictly parametric linear or generalized linear models, but still has much more structure than the completely general model that says that the response is just some smooth function of all the covariates.

Note one important point. In order for the model to be identifiable the smooth functions have to be constrained to have zero mean (usually taken over the set of covariate values). Such constraints are always applied by gam.

Specification of the distribution and link function is done using the family argument to gam and works in the same way as for glm. This page hence assumes the default identity link normal error family, since the generalizations are easy.

Starting with the model given above then, the gam formula would be
`y~x1+x2+s(x3)+s(x4,x5).`
This would use the default basis for the smooths (a thin plate regression spline basis for each), with automatic selection of the effective degrees of freedom for both smooths. The dimension of the smoothing basis is given a default value as well (the dimension of the basis sets an upper limit on the maximum possible degrees of freedom for the basis - the limit is typically one less than basis dimension). Full details of how to control smooths are given in s and te, and further discussion of basis dimension choice can be found in choose.k. For the moment suppose that we would like to change the basis of the first smooth to a cubic regression spline basis with a dimension of 20, while fixing the second term at 25 degrees of freedom. The appropriate formula would be:
`y~x1+x2+s(x3,bs="cr",k=20)+s(x4,x5,k=26,fx=T).`

Now consider some more unusual models. Consider a model in which y is a smooth function of x except at a point x^* where the function jumps discontinuously. This model can be written as:

$$E(y_i) = \beta_0 + \beta_1 h(x^*, x_i) + f_1(x_i)$$

where h is a step function jumping from 0 to 1 at x^*. The way to fit this model is to create a variable `h` which is zero for all `x` less than x^* and one otherwise. Then the model formula is:
`y~h+s(x).`

Another situation that occurs quite often is the one in which we would like to find out if the model:

$$E(y_i) = f(x_i, z_i)$$

is really necessary or whether:

$$E(y_i) = f_1(x_i) + f_2(z_i)$$

wouldn't do just as well. One way to do this is to look at the results of fitting:
`y~s(x)+s(z)+s(x,z).`
`gam` automatically generates side conditions to make this model identifiable. You can also estimate 'overlapping' models like:
`y~s(x,z)+s(z,v).`

Sometimes models of the form:

$$E(y_i) = \beta_0 + f(x_i)z_i$$

need to be estimated (where f is a smooth function, as usual.) The appropriate formula is:
`y~z+s(x,by=z)`
- the `by` argument ensures that the smooth function gets multiplied by covariate `z`, but GAM smooths are centred (average value zero), so the `z+` term is needed as well (f is being represented by a constant plus a centred smooth). If we'd wanted:

$$E(y_i) = f(x_i)z_i$$

then the appropriate formula would be:
`y~z+s(x,by=z)-1.`

The `by` mechanism also allows models to be estimated in which the form of a smooth depends on the level of a factor, but to do this the user must generate the dummy variables for each level of the factor. Suppose for example that `fac` is a factor with 3 levels `1`, `2`, `3`, and at each level of this factor ther response depends smoothly on a variable `x` in a manner that is level dependent. Three dummy variables `fac.1`, `fac.2`, `fac.3`, can be generated for the factor (e.g. `fac.1<-as.numeric(fac==1)`). Then the model formula would be:
`y~fac+s(x,by=fac.1)+s(x,by=fac.2)+s(x,by=fac.3).`

In the above examples the smooths of more than one covariate have all employed single penalty thin plate regression splines. These isotropic smooths are not alway appropriate: if variables are not naturally 'well scaled' relative to each other then it is often preferable to use tensor product smooths, with a wiggliness penalty for each covariate of the term. See `te` for examples.

WARNING

There are no identifiability checks made between the smooth and parametric parts of a gam formula, and any such lack of identifiability will cause problems if the underlying fitting routine is `mgcv` (not the default).

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

Examples

```
set.seed(10)
n<-400
sig2<-4
x0 <- runif(n, 0, 1)
x1 <- runif(n, 0, 1)
x2 <- runif(n, 0, 1)
pi <- asin(1) * 2
f1 <- 2 * sin(pi * x2)
f2 <-  exp(2 * x2) - 3.75887
f3 <-  0.2 * x2^11 * (10 * (1 - x2))^6 +
       10 * (10 * x2)^3 * (1 - x2)^10 - 1.396
e <- rnorm(n, 0, sqrt(abs(sig2)))
# A continuous `by' variable example....
y <- f3*x1 + e
b<-gam(y~x1-1+s(x2,by=x1))
plot(b,pages=1)
summary(b)
# A dummy `by' variable example (with a spurious covariate x0)
fac<-as.factor(c(rep(1,100),rep(2,100),rep(3,200)))
fac.1<-as.numeric(fac==1);fac.2<-as.numeric(fac==2);
fac.3<-as.numeric(fac==3)
y<-f1*fac.1+f2*fac.2+f3*fac.3+ e
b<-gam(y~fac+s(x2,by=fac.1)+s(x2,by=fac.2)+s(x2,by=fac.3)+s(x0))
plot(b,pages=1)
summary(b)
```

`gam.neg.bin` *GAMs with the negative binomial distribution*

Description

The `gam` modelling function is designed to be able to use the `negative.binomial` and `neg.bin` families from the MASS library, with or without a known θ parameter. A value for `theta` must always be passed to these families, but if θ is to be estimated then the passed value is treated as a starting value for estimation.

If the `scale` argument passed to `gam` is positive, then it is used as the scale parameter `theta` is treated as a fixed known parameter and any smoothing parameters are chosen by UBRE. If `scale` is not positive then θ is estimated. The method of estimation is to choose $\hat{\theta}$ so that the GCV (Pearson) estimate of the scale parameter is one (since the scale parameter is one for the negative binomial).

θ estimation is nested within the IRLS loop used for GAM fitting. After each call to fit an iteratively weighted additive model to the IRLS pseudodata, the θ estimate is updated. This is done by conditioning on all components of the current GCV/Pearson estimator of the scale parameter except θ and then searching for the $\hat{\theta}$ which equates this conditional estimator to one. The search is a simple bisection search after an initial crude line search to bracket one. The search will terminate at the upper boundary of the search region is a Poisson fit would have yielded an estimated scale parameter <1. Search limits can be set in `gam.control`.

Note that `neg.bin` only allows a log link, while `negative.binomial` also allows `"sqrt"` and `"identity"`. In addition the `negative.binomial` family results in a more informative `gam` summary.

The negative binomial families can not yet be used with 'outer' estimation of smoothing parameters (see `gam.method`).

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

Examples

```
library(MASS) # required for negative binomial families
set.seed(3)
n<-400
x0 <- runif(n, 0, 1)
x1 <- runif(n, 0, 1)
x2 <- runif(n, 0, 1)
x3 <- runif(n, 0, 1)
pi <- asin(1) * 2
f <- 2 * sin(pi * x0)
f <- f + exp(2 * x1) - 3.75887
f <- f + 0.2 * x2^11 * (10 * (1 - x2))^6 + 10 * (10 * x2)^3 * (1 - x2)^10 - 1.396
g<-exp(f/5)
# negative binomial data
y<-rnbinom(g,size=3,mu=g)
# unknown theta ...
b<-gam(y~s(x0)+s(x1)+s(x2)+s(x3),family=negative.binomial(1))
plot(b,pages=1)
print(b)
b<-gam(y~s(x0)+s(x1)+s(x2)+s(x3),family=neg.bin(1)) # unknown theta
plot(b,pages=1)
print(b)
# known theta example ...
b<-gam(y~s(x0)+s(x1)+s(x2)+s(x3),family=negative.binomial(3),scale=1)
plot(b,pages=1)
print(b)
# Now use "sqrt" link available in negative.binomial (but not neg.bin)
set.seed(1)
f<-f-min(f);g<-f^2
y<-rnbinom(g,size=3,mu=g)
b<-gam(y~s(x0)+s(x1)+s(x2)+s(x3),family=negative.binomial(1,link="sqrt"))
plot(b,pages=1)
print(b)
```

`gam.outer` *Minimize GCV or UBRE score of a GAM using 'outer' iteration*

Description

Estimation of GAM smoothing parameters is most stable if optimization of the UBRE or GCV score is outer to the penalized iteratively re-weighted least squares scheme used to estimate the model given smoothing parameters.

This routine optimizes a GCV or UBRE score in this way. Basically the GCV or UBRE score is evaluated for each trial set of smoothing parameters by estimating the GAM for those smoothing parameters. The score is minimized w.r.t. the parameters numerically, using `optim` or `nlm`. Exact derivatives of the score can be used by fitting with `gam.fit2`, which improves efficiency and reliability relative to relying solely on finite difference derivatives.

Note that there is a choise between basing GCV/UBRE scores on the deviance or the Pearson statistic: see `gam.method`.

Not normally called directly, but rather a service routine for gam.

Usage

```
gam.outer(lsp,fscale,family,control,method,gamma,G,...)
```

Arguments

lsp	The log smoothing parameters.
fscale	Typical scale of the GCV or UBRE score.
family	the model family.
control	control argument to pass to gam.fit if pure finite differencing is being used.
method	method list reurned from gam.method. This defines the optimization method to use.
gamma	The degree of freedom inflation factor for the GCV/UBRE score.
G	List produced by gam.setup, containing most of what's needed to actually fit GAM.
...	other arguments, typically for passing on to gam.fit2 (ultimately).

Details

Estimation of smoothing parameters by optimizing GCV scores obtained at convergence of the P-IRLS iteration was proposed by O'Sullivan et al. (1986), and is here termed 'outer' iteration.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

O 'Sullivan, Yandall & Raynor (1986) Automatic smoothing of regression functions in generalized linear models. J. Amer. Statist. Assoc. 81:96-103.

http://www.maths.bath.ac.uk/~sw283/

See Also

gam.fit2, gam, mgcv, magic

---

gam.selection *Generalized Additive Model Selection*

Description

This page is intended to provide some more information on how to select GAMs. Given a model structure specified by a gam model formula, `gam()` attempts to find the appropriate smoothness for each applicable model term using Generalized Cross Validation (GCV) or an Un-Biased Risk Estimator (UBRE), the latter being used in cases in which the scale parameter is assumed known. GCV and UBRE are covered in Craven and Wahba (1979) and Wahba (1990), see `gam.method` for more detail about the numerical optimization approaches available.

Automatic smoothness selection is unlikely to be successful with few data, particularly with multiple terms to be selected. In addition GCV and UBRE score can occasionally display local minima that can trap the minimisation algorithms. GCV/UBRE scores become constant with changing smoothing parameters at very low or very high smoothng parameters, and on occasion these 'flat' regions can be separated from regions of lower score by a small 'lip'. This seems to be the most common form of local minimum, but is usually avoidable by avoiding extreme smoothing parameters as starting values in optimization, and by avoiding big jumps in smoothing parameters while optimizing. Never the less, if you are suspicious of smoothing parameter estimates, try changing fit method (see `gam.method`) and see if the estimates change, or try changing some or all of the smoothing parameters 'manually' (argument `sp` of `gam`).

In general the most logically consistent method to use for deciding which terms to include in the model is to compare GCV/UBRE scores for models with and without the term. More generally the score for the model with a smooth term can be compared to the score for the model with the smooth term replaced by appropriate parametric terms. Candidates for removal can be identified by reference to the approximate p-values provided by `summary.gam`. Candidates for replacement by parametric terms are smooth terms with estimated degrees of freedom close to their minimum possible.

One appealing approach to model selection is via shrinkage. Smooth classes `cs.smooth` and `tprs.smooth` (specified by `"cs"` and `"ts"` respectively) have smoothness penalties which include a small shrinkage component, so that for large enough smoothing parameters the smooth becomes identically zero. This allows automatic smoothing parameter selection methods to effectively remove the term from the model altogether. The shrinkage component of the penalty is set at a level that usually makes negligable contribution to the penalization of the model, only becoming effective when the term is effectively 'completely smooth' according to the conventional penalty.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

Craven and Wahba (1979) Smoothing Noisy Data with Spline Functions. Numer. Math. 31:377-403

Venables and Ripley (1999) Modern Applied Statistics with S-PLUS

Wahba (1990) Spline Models of Observational Data. SIAM.

Wood, S.N. (2000) Modelling and Smoothing Parameter Estimation with Multiple Quadratic Penalties. J.R.Statist.Soc.B 62(2):413-428

Wood, S.N. (2003) Thin plate regression splines. J.R.Statist.Soc.B 65(1):95-114

http://www.maths.bath.ac.uk/~sw283/

Examples

```
## an example of GCV based model selection
library(mgcv)
```

```
set.seed(0)
n<-400;sig<-2
x0 <- runif(n, 0, 1);x1 <- runif(n, 0, 1)
x2 <- runif(n, 0, 1);x3 <- runif(n, 0, 1)
x4 <- runif(n, 0, 1);x5 <- runif(n, 0, 1)
f <- 2 * sin(pi * x0)
f <- f + exp(2 * x1) - 3.75887
f <- f+0.2*x2^11*(10*(1-x2))^6+10*(10*x2)^3*(1-x2)^10-1.396
e <- rnorm(n, 0, sig)
y <- f + e
## Note the increased gamma parameter below to favour
## slightly smoother models (See Kim & Gu, 2004, JRSSB)...
b<-gam(y~s(x0,bs="ts")+s(x1,bs="ts")+s(x2,bs="ts")+
   s(x3,bs="ts")+s(x4,bs="ts")+s(x5,bs="ts"),gamma=1.4)
summary(b)
plot(b,pages=1)
```

`gam.setup` *Generalized additive model set up*

Description

This is an internal function of package `mgcv`. It is called by `gam` to obtain the design matrix and penalty matrices for a GAM set up using penalized regression splines. This is done by calling a mixture of R routines and compiled C code. For further information on usuage see code for `gam`.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

Gu and Wahba (1991) Minimizing GCV/GML scores with multiple smoothing parameters via the Newton method. SIAM J. Sci. Statist. Comput. 12:383-398

Wood, S.N. (2000) Modelling and Smoothing Parameter Estimation with Multiple Quadratic Penalties. J.R.Statist.Soc.B 62(2):413-428

Wood, S.N. (2003) Thin plate regression splines. J.R.Statist.Soc.B 65(1):95-114

http://www.maths.bath.ac.uk/~sw283/

See Also

`gam`, `gamm`, `magic`, `mgcv`

gam.side *Identifiability side conditions for a GAM*

Description

GAM formulae with repeated variables only correspond to identifiable models given some side conditions. This routine works out appropriate side conditions, based on zeroing redundant parameters. It is called from `gam.setup` and is not intended to be called by users.

The method identifies nested and repeated variables by their names, but evaluates what constraints to impose, numerically. Constraints are always applied to smooths of more variables in preference to smooths of fewer variables. The numerical approach allows appropriate constraints to be applied to models constructed using any smooths, including user defined smooths.

Usage

```
gam.side(sm,tol=.Machine$double.eps^.5)
```

Arguments

`sm`	A list of smooth objects as returned by `smooth.construct`.
`tol`	The tolernce to use when assessing linear dependence of smooths.

Details

Models such as `y~s(x)+s(z)+s(x,z)` can be estimated by `gam`, but require identifiability constraints to be applied, to make them identifiable. This routine does this, effectively setting redundant parameters to zero. When the redundancy is between smooths of lower and higher numbers of variables, the constraint is always applied to the smooth of the higher number of variables.

Dependent smooths are identified symbolically, but which constraints are needed to ensure identifiability of these smooths is determined numerically, using `fixDependence`. This makes the routine rather general, and not dependent on any particular basis.

Value

A list of smooths, with model matrices and penalty matrices adjusted to automatically impose the required constraints. Any smooth that has been modified will have an attribute `"del.index"`, listing the columns of its model matrix that were deleted. This index is used in the creation of prediction matrices for the term.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

Examples

```
set.seed(0)
n<-400
sig2<-4
x0 <- runif(n, 0, 1)
x1 <- runif(n, 0, 1)
x2 <- runif(n, 0, 1)
```

```
pi <- asin(1) * 2
f <- 2 * sin(pi * x0)
f <- f + exp(2 * x1) - 3.75887
f <- f + 0.2 * x2^11 * (10 * (1 - x2))^6 + 10 * (10 * x2)^3 * (1 - x2)^10 - 1.396
e <- rnorm(n, 0, sqrt(abs(sig2)))
y <- f + e
b<-gam(y~s(x0)+s(x1)+s(x0,x1)+s(x2))
plot(b,pages=1)
test1<-function(x,z,sx=0.3,sz=0.4)
{ (pi**sx*sz)*(1.2*exp(-(x-0.2)^2/sx^2-(z-0.3)^2/sz^2)+
  0.8*exp(-(x-0.7)^2/sx^2-(z-0.8)^2/sz^2))
}
n<-500
old.par<-par(mfrow=c(2,2))
x<-runif(n);z<-runif(n);
y<-test1(x,z)+rnorm(n)*0.1
## a fully nested tensor product example
b<-gam(y~s(x,bs="cr",k=6)+s(z,bs="cr",k=6)+te(x,z,k=6))
plot(b)
par(old.par)
rm(list=c("f","x0","x1","x2","x","z","y","b","test1","n","sig2","pi","e"))
```

`gam2objective` *Objective functions for GAM smoothing parameter estimation*

Description

Estimation of GAM smoothing parameters is most stable if optimization of the UBRE or GCV score is outer to the penalized iteratively re-weighted least squares scheme used to estimate the model given smoothing parameters. These functions evaluate the GCV/UBRE score of a GAM model, given smoothing parameters, in a manner sutable for use by `optim` or `nlm`. Not normally called directly, but rather service routines for `gam.outer`.

Usage

```
gam2objective(lsp,args,printWarn=FALSE,...)
gam2derivative(lsp,args,...)
gam3objective(lsp,args,...)
```

Arguments

`lsp`	The log smoothing parameters.
`args`	List of arguments required to call `gam.fit2`.
`printWarn`	Should `gam.fit2` print some warnings? Used to suppress warnings that are only of interest for the final fitted model, until that model is reached.
`...`	Other arguments for passing to `gam.fit2`.

Details

gam2objective and gam2derivative are functions suitable for calling by optim, to evaluate the GCV/UBRE score and it's derivatives w.r.t. log smoothing parameters.

gam3objective is an equaivalent to gam2objective, suitable for optimization by nlm - derivatives of the GCV/UBRE function are calculated and returned as attributes.

The basic idea of optimizing smoothing parameters 'outer' to the P-IRLS loop was first proposed in O'Sullivan et al. (1986).

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

O 'Sullivan, Yandall & Raynor (1986) Automatic smoothing of regression functions in generalized linear models. J. Amer. Statist. Assoc. 81:96-103.

http://www.maths.bath.ac.uk/~sw283/

See Also

gam.fit2, gam, mgcv, magic

---

gamm *Generalized Additive Mixed Models*

Description

Fits the specified generalized additive mixed model (GAMM) to data, by a call to lme in the normal errors identity link case, or by a call to glmmPQL from the MASS library otherwise. In the latter case estimates are only approximately MLEs. The routine is typically slower than gam, and not quite as numerically robust.

Smooths are specified as in a call to gam as part of the fixed effects model formula, but the wiggly components of the smooth are treated as random effects. The random effects structures and correlation structures availabale for lme are used to specify other random effects and correlations.

It is assumed that the random effects and correlation structures are employed primarily to model residual correlation in the data and that the prime interest is in inference about the terms in the fixed effects model formula including the smooths. For this reason the routine calculates a posterior covariance matrix for the coefficients of all the terms in the fixed effects formula, including the smooths.

To use this function effectively it helps to be quite familiar with the use of gam and lme.

Usage

```
gamm(formula,random=NULL,correlation=NULL,family=gaussian(),
data=list(),weights=NULL,subset=NULL,na.action,knots=NULL,
control=nlme::lmeControl(niterEM=0,optimMethod="L-BFGS-B"),
niterPQL=20,verbosePQL=TRUE,method="ML",...)
```

Arguments

formula	A GAM formula (see also gam.models). This is exactly like the formula for a glm except that smooth terms can be added to the right hand side of the formula (and a formula of the form y ~ . is not allowed). Smooth terms are specified by expressions of the form: s(var1,var2,...,k=12,fx=FALSE,bs="tp",by=a.var) where var1, var2, etc. are the covariates which the smooth is a function of and k is the dimension of the basis used to represent the smooth term. If k is not specified then k=10*3^(d-1) is used where d is the number of covariates for this term. fx is used to indicate whether or not this term has a fixed muber of degrees of freedom (fx=FALSE to select d.f. by GCV/UBRE). bs indicates the basis to use: see s for full details, but note that the default "tp" can be slow with large data sets. Tensor product smooths are specified using te terms.
random	The (optional) random effects structure as specified in a call to lme: only the list form is allowed, to facilitate manipulation of the random effects structure within gamm in order to deal with smooth terms. See example below.
correlation	An optional corStruct object (see corClasses) as used to define correlation structures in lme. Any grouping factors in the formula for this object are assumed to be nested within any random effect grouping factors, without the need to make this explicit in the formula (this is slightly different to the behaviour of lme). See examples below.
family	A family as used in a call to glm or gam. The default gaussian with identity link causes gamm to fit by a direct call to lme procided there is no offset term, otherwise glmmPQL from the MASS library is used.
data	A data frame containing the model response variable and covariates required by the formula. By default the variables are taken from environment(formula), typically the environment from which gamm is called.
weights	In the generalized case, weights with the same meaning as glm weights. An lme type weights argument may only be used in the identity link gaussian case, with no offset (see documentation for lme for details of how to use such an argument).
subset	an optional vector specifying a subset of observations to be used in the fitting process.
na.action	a function which indicates what should happen when the data contain 'NA's. The default is set by the 'na.action' setting of 'options', and is 'na.fail' if that is unset. The "factory-fresh" default is 'na.omit'.
knots	this is an optional list containing user specified knot values to be used for basis construction. For the cr basis the user simply supplies the knots to be used, and there must be the same number as the basis dimension, k, for the smooth concerned. For the tp basis knots has two uses. Firstly, for large datasets the calculation of the tp basis can be time-consuming. The user can retain most of the advantages of the t.p.r.s. approach by supplying a reduced set of covariate values from which to obtain the basis - typically the number of covariate values used will be substantially smaller than the number of data, and substantially larger than the basis dimension, k. The second possibility is to avoid the eigen-decomposition used to find the t.p.r.s. basis altogether and simply use the basis implied by the chosen knots: this will happen if the number of knots supplied matches the basis dimension, k. For a given basis dimension the second option is faster, but gives poorer results (and the user must be quite careful in choosing

knot locations). Different terms can use different numbers of knots, unless they share a covariate.

`control` A list of fit control parameters for `lme` returned by `lmeControl`. Note the default setting for the number of EM iterations used by `lme`: smooths are set up using custom `pdMat` classes, which are currently not supported by the EM iteration code. Only increase this number if you want to perturb the starting values used in model fitting (usually to worse values!). The `optimMethod` option is only used if your version of R does not have the `nlminb` optimizer function.

`niterPQL` Maximum number of PQL iterations (if any).

`verbosePQL` Should PQL report its progress as it goes along?

`method` Which of `"ML"` or `"REML"` to use in the Gaussian additive mixed model case when `lme` is called directly. Ignored in the generalized case (or if the model has an offset), in which case `glmmPQL` is used.

`...` further arguments for passing on e.g. to `lme`

Details

The Bayesian model of spline smoothing introduced by Wahba (1983) and Silverman (1985) opens up the possibility of estimating the degree of smoothness of terms in a generalized additive model as variances of the wiggly components of the smooth terms treated as random effects. Several authors have recognised this (see Wang 1998; Ruppert, Wand and Carroll, 2003) and in the normal errors, identity link case estimation can be performed using general linear mixed effects modelling software such as `lme`. In the generalized case only approximate inference is so far available, for example using the Penalized Quasi-Likelihood approach of Breslow and Clayton (1993) as implemented in `glmmPQL` by Venables and Ripley (2002). One advantage of this approach is that it allows correlated errors to be dealt with via random effects or the correlation structures available in the `nlme` library.

Some brief details of how GAMs are represented as mixed models and estimated using `lme` or `glmmPQL` in `gamm` can be found in Wood (2004a,b). In addition `gamm` obtains a posterior covariance matrix for the parameters of all the fixed effects and the smooth terms. The approach is similar to that described in (Lin & Zhang, 1999) - the covariance matrix of the data (or pseudodata in the generalized case) implied by the weights, correlation and random effects structure is obtained, based on the estimates of the parameters of these terms and this is used to obtain the posterior covariance matrix of the fixed and smooth effects.

The bases used to represent smooth terms are the same as those used in `gam`.

In the event of `lme` convergence failures, consider modifying `option(mgcv.vc.logrange)`: reducing it helps to remove indefiniteness in the likelihood, if that is the problem, but too large a reduction can force over or undersmoothing. See `notExp2` for more information on this option. Failing that, you can try increasing the `niterEM` option in `control`: this will perturb the starting values used in fitting, but usually to values with lower likelihood! Note that this version of `gamm` works best with R 2.2.0 or above and `nlme`, 3.1-62 and above, since these use an improved optimizer.

Value

Returns a list with two items:

`gam` an object of class `gam`, less information relating to GCV/UBRE model selection. At present this contains enough information to use `predict`, `summary` and `print` methods and `vis.gam`, but not to use e.g. the `anova` method function to compare models.

lme — the fitted model object returned by lme or glmmPQL. Note that the model formulae and grouping structures may appear to be rather bizarre, because of the manner in which the GAMM is split up and the calls to lme and glmmPQL are constructed.

WARNINGS

This version of gamm works best with nlme 3.1-62 and above and R 2.2.0 and above. This is because it is designed to work with the optimizers used from these versions onwards, rather than the earlier default optimizers.

The routine will be very slow and memory intensive if correlation structures are used for the very large groups of data. e.g. attempting to run the spatial example in the examples section with many 1000's of data is definitely not recommended: often the correlations should only apply within clusters that can be defined by a grouping factor, and provided these clusters do not get too huge then fitting is usually possible.

Models must contain at least one random effect: either a smooth with non-zero smoothing parameter, or a random effect specified in argument random.

Models like s(z)+s(x)+s(x,z) are not currently supported.

gamm is not as numerically stable as gam: an lme call will occasionally fail. See details section for suggestions.

gamm is usually much slower than gam, and on some platforms you may need to increase the memory available to R in order to use it with large data sets (see mem.limits).

Note that the weights returned in the fitted GAM object are dummy, and not those used by the PQL iteration: this makes partial residual plots look odd.

Note that the gam object part of the returned object is not complete in the sense of having all the elements defined in gamObject and does not inherit from glm: hence e.g. multi-model anova calls will not work.

The parameterization used for the smoothing parameters in gamm, bounds them above and below by an effective infinity and effective zero. See notExp2 for details of how to change this.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

Breslow, N. E. and Clayton, D. G. (1993) Approximate inference in generalized linear mixed models. Journal of the American Statistical Association 88, 9-25.

Lin, X and Zhang, D. (1999) Inference in generalized additive mixed models by using smoothing splines. JRSSB. 55(2):381-400

Pinheiro J.C. and Bates, D.M. (2000) Mixed effects Models in S and S-PLUS. Springer

Ruppert, D., Wand, M.P. and Carroll, R.J. (2003) Semiparametric Regression. Cambridge

Silverman, B.W. (1985) Some aspects of the spline smoothing approach to nonparametric regression. JRSSB 47:1-52

Venables, W. N. and Ripley, B. D. (2002) Modern Applied Statistics with S. Fourth edition. Springer.

Wahba, G. (1983) Bayesian confidence intervals for the cross validated smoothing spline. JRSSB 45:133-150

Wood, S.N. (2004a) Stable and efficient multiple smoothing parameter estimation for generalized additive models. Journal of the American Statistical Association. 99:673-686

Wood, S.N. (2003) Thin plate regression splines. J.R.Statist.Soc.B 65(1):95-114

Wood, S.N. (2006) Low rank scale invariant tensor product smooths for Generalized Additive Mixed Models. Biometrics.

Wang, Y. (1998) Mixed effects smoothing spline analysis of variance. J.R. Statist. Soc. B 60, 159-174

http://www.maths.bath.ac.uk/~sw283/

See Also

magic for an alternative for correlated data, te, s, predict.gam, plot.gam, summary.gam, gam.neg.bin, vis.gam,pdTens,gamm.setup

Examples

```
library(mgcv)
## simple examples using gamm as alternative to gam
set.seed(0)
n <- 400
sig <- 2
x0 <- runif(n, 0, 1)
x1 <- runif(n, 0, 1)
x2 <- runif(n, 0, 1)
x3 <- runif(n, 0, 1)
f <- 2 * sin(pi * x0)
f <- f + exp(2 * x1) - 3.75887
f <- f+0.2*x2^11*(10*(1-x2))^6+10*(10*x2)^3*(1-x2)^10-1.396
e <- rnorm(n, 0, sig)
y <- f + e
b <- gamm(y~s(x0)+s(x1)+s(x2)+s(x3))
plot(b$gam,pages=1)
summary(b$lme) # details of underlying lme fit
summary(b$gam) # gam style summary of fitted model
anova(b$gam)

b <- gamm(y~te(x0,x1)+s(x2)+s(x3))
op <- par(mfrow=c(2,2))
plot(b$gam)
par(op)

## Add a factor to the linear predictor, to be modelled as random
fac <- rep(1:4,n/4)
f <- f + fac*3
fac<-as.factor(fac)

g<-exp(f/5)
y<-rpois(rep(1,n),g)
b2<-gamm(y~s(x0)+s(x1)+s(x2)+s(x3),family=poisson,random=list(fac=~1))
plot(b2$gam,pages=1)

## now an example with autocorrelated errors....
x <- 0:(n-1)/(n-1)
f <- 0.2*x^11*(10*(1-x))^6+10*(10*x)^3*(1-x)^10-1.396
e <- rnorm(n,0,sig)
```

```
for (i in 2:n) e[i] <- 0.6*e[i-1] + e[i]
y <- f + e
op <- par(mfrow=c(2,2))
b <- gamm(y~s(x,k=20),correlation=corAR1())
plot(b$gam);lines(x,f-mean(f),col=2)
b <- gamm(y~s(x,k=20))
plot(b$gam);lines(x,f-mean(f),col=2)
b <- gam(y~s(x,k=20))
plot(b);lines(x,f-mean(f),col=2)

## more complicated autocorrelation example - AR errors
## only within groups defined by `fac'
e <- rnorm(n,0,sig)
for (i in 2:n) e[i] <- 0.6*e[i-1]*(fac[i-1]==fac[i]) + e[i]
y <- f + e
b <- gamm(y~s(x,k=20),correlation=corAR1(form=~1|fac))
plot(b$gam);lines(x,f-mean(f),col=2)
par(op)

## more complex situation with nested random effects and within
## group correlation

set.seed(0)
n.g <- 10
n<-n.g*10*4
sig <- 2
## simulate smooth part
x0 <- runif(n, 0, 1)
x1 <- runif(n, 0, 1)
x2 <- runif(n, 0, 1)
x3 <- runif(n, 0, 1)
f <- 2 * sin(pi * x0)
f <- f + exp(2 * x1) - 3.75887
f <- f+0.2*x2^11*(10*(1-x2))^6+10*(10*x2)^3*(1-x2)^10-1.396
## simulate nested random effects....
fa <- as.factor(rep(1:10,rep(4*n.g,10)))
ra <- rep(rnorm(10),rep(4*n.g,10))
fb <- as.factor(rep(rep(1:4,rep(n.g,4)),10))
rb <- rep(rnorm(4),rep(n.g,4))
for (i in 1:9) rb <- c(rb,rep(rnorm(4),rep(n.g,4)))
## simulate auto-correlated errors within groups
e<-array(0,0)
for (i in 1:40) {
eg <- rnorm(n.g, 0, sig)
for (j in 2:n.g) eg[j] <- eg[j-1]*0.6+ eg[j]
e<-c(e,eg)
}
y <- f + ra + rb + e
dat<-data.frame(y=y,x0=x0,x1=x1,x2=x2,x3=x3,fa=fa,fb=fb)
## fit model ....
b <- gamm(y~s(x0,bs="cr")+s(x1,bs="cr")+s(x2,bs="cr")+
  s(x3,bs="cr"),data=dat,random=list(fa=~1,fb=~1),
  correlation=corAR1())
plot(b$gam,pages=1)

## and a "spatial" example
library(nlme);set.seed(1)
```

```
test1<-function(x,z,sx=0.3,sz=0.4)
{ (pi**sx*sz)*(1.2*exp(-(x-0.2)^2/sx^2-(z-0.3)^2/sz^2)+
  0.8*exp(-(x-0.7)^2/sx^2-(z-0.8)^2/sz^2))
}
n<-200
old.par<-par(mfrow=c(2,2))
x<-runif(n);z<-runif(n);
xs<-seq(0,1,length=30);zs<-seq(0,1,length=30)
pr<-data.frame(x=rep(xs,30),z=rep(zs,rep(30,30)))
truth <- matrix(test1(pr$x,pr$z),30,30)
contour(xs,zs,truth)  # true function
f <- test1(x,z)  # true expectation of response
## Now simulate correlated errors...
cstr <- corGaus(.1,form = ~x+z)
cstr <- Initialize(cstr,data.frame(x=x,z=z))
V <- corMatrix(cstr) # correlation matrix for data
Cv <- chol(V)
e <- t(Cv) %*% rnorm(n)*0.05 # correlated errors
## next add correlated simulated errors to expected values
y <- f + e ## ... to produce response
b<- gamm(y~s(x,z,k=50),correlation=corGaus(.1,form=~x+z))
plot(b$gam) # gamm fit accounting for correlation
# overfits when correlation ignored.....
b1 <- gamm(y~s(x,z,k=50));plot(b1$gam)
b2 <- gam(y~s(x,z,k=50));plot(b2)
par(old.par)
```

`gamm.setup` *Generalized additive mixed model set up*

Description

This is an internal function of package `mgcv`. It is called by `gamm` to set up a generalized additive mixed model in a form suitable for fitting by calls to `glmmPQL` from the `MASS` library or `lme` from the `nlme` library. The main task is the representation of the smooth terms as random effects.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

Wood, S.N. (2004) Stable and efficient multiple smoothing parameter estimation for generalized additive models. Journal of the American Statistical Association. 99:673-686

Wood, S.N. (2006) Low rank scale invariant tensor product smooths for Generalized Additive Mixed Models. Biometrics

http://www.maths.bath.ac.uk/~sw283/

See Also

`gamm`

gamObject *Fitted gam object*

Description

A fitted GAM object returned by function gam and of class "gam" inheriting from classes "glm" and "lm". Method functions anova, logLik, influence, plot, predict, print, residuals and summary exist for this class.

All compulsory elements of "glm" and "lm" objects are present, but the fitting method for a GAM is different to a linear model or GLM, so that the elements relating to the QR decomposition of the model matrix are absent.

Value

A gam object has the following elements:

aic	AIC of the fitted model: bear in mind that the degrees of freedom used to calculate this are the effective degrees of freedom of the model, and the likelihood is evaluated at the maximum of the penalized likelihood in most cases, not at the MLE.
assign	Array whose elements indicate which model term (listed in pterms) each parameter relates to: applies only to non-smooth terms.
boundary	did parameters end up at boundary of parameter space?
call	the matched call (allows update to be used with gam objects, for example).
coefficients	the coefficients of the fitted model. Parametric coefficients are first, followed by coefficients for each spline term in turn.
control	the gam control list used in the fit.
converged	indicates whether or not the iterative fitting method converged.
data	the original supplied data argument (for class "glm" compatibility).
deviance	model deviance (not penalized deviance).
df.null	null degrees of freedom.
df.residual	effective residual degrees of freedom of the model.
edf	estimated degrees of freedom for each model parameter. Penalization means that many of these are less than 1.
family	family object specifying distribution and link used.
fit.method	Character string describing the multiple GCV/UBRE smoothing parameter estimation method used.
fitted.values	fitted model predictions of expected value for each datum.
formula	the model formula.
full.formula	the model formula with each smooth term fully expanded and with option arguments given explicitly (i.e. not with reference to other variables) - useful for later prediction from the model.
gcv.ubre	The minimized GCV or UBRE score.
hat	array of elements from the leading diagonal of the 'hat' (or 'influence') matrix. Same length as response data vector.

iter	number of iterations of P-IRLS taken to get convergence.
linear.predictors	fitted model prediction of link function of expected value for each datum.
method	One of "GCV" or "UBRE", depending on the fitting criterion used.
mgcv.conv	A list of convergence diagnostics relating to the "mgcv" or "magic" parts of smoothing parameter estimation - this will not be very meaningful for pure "outer" estimation of smoothing parameters. mgcv.conv differs for method "magic" and "mgcv". Here is the "mgcv" version:
score	corresponding to edf, an array of GCV or UBRE scores for the model given the final estimated relative smoothing parameters.
g	the gradient of the GCV/UBRE score w.r.t. the relative smoothing parameters at termination.
h	the second derivatives corresponding to g above - i.e. the leading diagonal of the Hessian.
e	the eigen-values of the Hessian. All non-negative indicates a positive definite Hessian.
iter	the number of iterations taken.
in.ok	TRUE if the second smoothing parameter guess improved the GCV/UBRE score.
step.fail	TRUE if the algorithm terminated by failing to improve the GCV/UBRE score rather than by 'converging'. Not necessarily a problem, but check the above derivative information quite carefully.
	In the case of "magic" the items are:
full.rank	The apparent rank of the problem given the model matrix and constraints.
rank	The numerical rank of the problem.
fully.converged	TRUE is multiple GCV/UBRE converged by meeting convergence criteria. FALSE if method stopped with a steepest descent step failure.
hess.pos.def	Was the hessian of the GCV/UBRE score positive definite at smoothing parameter estimation convergence?
iter	How many iterations were required to find the smoothing parameters?
score.calls	and how many times did the GCV/UBRE score have to be evaluated?
rms.grad	root mean square of the gradient of the GCV/UBRE score at convergence.
min.edf	Minimum possible degrees of freedom for whole model.
model	model frame containing all variables needed in original model fit.
na.action	The na.action used in fitting.
nsdf	number of parametric, non-smooth, model terms including the intercept.
null.deviance	deviance for single parameter model.
offset	model offset.
outer.info	If 'outer' iteration has been used to fit the model (see gam.method) then this is present and contains whatever was returned by the optimization routine used (currently nlm or optim).
prior.weights	prior weights on observations.
pterms	terms object for strictly parametric part of model.

rank	apparent rank of fitted model.
residuals	the working residuals for the fitted model.
sig2	estimated or supplied variance/scale parameter.
smooth	list of smooth objects, containing the basis information for each term in the model formula in the order in which they appear. These smooth objects are what gets returned by the smooth.construct objects.
sp	smoothing parameter for each smooth.
terms	terms object of model model frame.
Vp	estimated covariance matrix for the parameters. This is a Bayesian posterior covariance matrix that results from adopting a particular Bayesian model of the smoothing process. Paricularly useful for creating credible/confidence intervals.
Ve	frequentist estimated covariance matrix for the parameter estimators. Particularly useful for testing whether terms are zero. Not so useful for CI's as smooths are usually biased.
weights	final weights used in IRLS iteration.
y	response data.

WARNINGS

This model object is different to that described in Chambers and Hastie (1993) in order to allow smoothing parameter estimation etc.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

Key References on this implementation:

Wood, S.N. (2000) Modelling and Smoothing Parameter Estimation with Multiple Quadratic Penalties. J.R.Statist.Soc.B 62(2):413-428

Wood, S.N. (2003) Thin plate regression splines. J.R.Statist.Soc.B 65(1):95-114

Wood, S.N. (in press) Stable and efficient multiple smoothing parameter estimation for generalized additive models. J. Amer. Statist. Ass.

Wood, S.N. (2004) On confidence intervals for GAMs based on penalized regression splines. Technical Report 04-12 Department of Statistics, University of Glasgow.

Wood, S.N. (2006) Low rank scale invariant tensor product smooths for generalized additive mixed models. Biometrics.

Key Reference on GAMs and related models:

Hastie (1993) in Chambers and Hastie (1993) Statistical Models in S. Chapman and Hall.

Hastie and Tibshirani (1990) Generalized Additive Models. Chapman and Hall.

Wahba (1990) Spline Models of Observational Data. SIAM

See Also

gam

get.var *Get named variable or evaluate expression from list or data.frame*

Description

This routine takes a text string and a data frame or list. It first sees if the string is the name of a variable in the data frame/ list. If it is then the value of this variable is returned. Otherwise the routine tries to evaluate the expression within the data.frame/list (but nowhere else) and if successful returns the result. If neither step works then NULL is returned. The routine is useful for processing gam formulae.

Usage

```
get.var(txt,data)
```

Arguments

txt	a text string which is either the name of a variable in data or when parsed is an expression that can be evaluated in data. It can also be neither in which case the function returns NULL.
data	A data frame or list.

Value

The evaluated variable, or NULL

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

http://www.maths.bath.ac.uk/~sw283/

See Also

gam

Examples

```
y <- 1:4;dat<-data.frame(x=5:10)
get.var("x",dat)
get.var("y",dat)
get.var("x==6",dat)
```

influence.gam *Extract the diagonal of the influence/hat matrix for a GAM*

Description

Extracts the leading diagonal of the influence matrix (hat matrix) of a fitted gam object.

Usage

```
## S3 method for class 'gam':
influence(model,...)
```

Arguments

model	fitted model objects of class gam as produced by gam().
...	un-used in this case

Details

Simply extracts hat array from fitted model. (More may follow!)

Value

An array (see above).

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

See Also

gam

initial.sp *Starting values for multiple smoothing parameter estimation*

Description

Finds initial smoothing parameter guesses for multiple smoothing parameter estimation. The idea is to find values such that the estimated degrees of freedom per penalized parameter should be well away from 0 and 1 for each penalized parameter, thus ensuring that the values are in a region of parameter space where the smoothing parameter estimation criterion is varying substantially with smoothing parameter value.

Usage

```
initial.sp(X,S,off,expensive=FALSE)
```

Arguments

X	is the model matrix.
S	is a list of of penalty matrices. S[[i]] is the ith penalty matrix, but note that it is not stored as a full matrix, but rather as the smallest square matrix including all the non-zero elements of the penalty matrix. Element 1,1 of S[[i]] occupies element off[i], off[i] of the ith penalty matrix. Each S[[i]] must be positive semi-definite.
off	is an array indicating the first parameter in the parameter vector that is penalized by the penalty involving S[[i]].
expensive	if TRUE then the overall amount of smoothing is adjusted so that the average degrees of freedom per penalized parameter is exactly 0.5: this is numerically costly.

Details

Basically uses a crude approximation to the estimated degrees of freedom per model coefficient, to try and find smoothing parameters which bound these e.d.f.'s away from 0 and 1.

Usually only called by magic and gam.

Value

An array of initial smoothing parameter estimates.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

See Also

magic, gam.outer, gam,

---

interpret.gam *Interpret a GAM formula*

Description

This is an internal function of package mgcv. It is a service routine for gam which splits off the strictly parametric part of the model formula, returning it as a formula, and interprets the smooth parts of the model formula.

Not normally called directly.

Usage

```
interpret.gam(gf)
```

Arguments

gf	A GAM formula as supplied to gam or gamm.

Value

An object of class `split.gam.formula` with the following items:

`pf`	A model formula for the strictly parametric part of the model.
`pfok`	TRUE if there is a `pf` formula.
`smooth.spec`	A list of class `xx.smooth.spec` objects where `xx` depends on the basis specified for the term. (These can be passed to smooth constructor method functions to actually set up penalties and bases.)
`full.formula`	An expanded version of the model formula in which the options are fully expanded, and the options do not depend on variables which might not be available later.
`fake.formula`	A formula suitable for use in evaluating a model frame.
`response`	Name of the response variable.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

Wood, S.N. (2000) Modelling and Smoothing Parameter Estimation with Multiple Quadratic Penalties. J.R.Statist.Soc.B 62(2):413-428

Wood, S.N. (2003) Thin plate regression splines. J.R.Statist.Soc.B 65(1):95-114

http://www.maths.bath.ac.uk/~sw283/

See Also

`gam` `gamm`

`logLik.gam` *Extract the log likelihood for a fitted GAM*

Description

Function to extract the log-likelihood for a fitted `gam` model (note that the models are usually fitted by penalized likelihood maximization).

Usage

```
logLik.gam(object,...)
```

Arguments

`object`	fitted model objects of class `gam` as produced by `gam()`.
`...`	un-used in this case

Details

Modification of `logLik.glm` which corrects the degrees of freedom for use with `gam` objects.

The function is provided so that `AIC` functions correctly with `gam` objects, and uses the appropriate degrees of freedom (accounting for penalization). Note, when using `AIC` for penalized models, that the degrees of freedom are the effective degrees of freedom and not the number of parameters, and the model maximizes the penalized likelihood, not the actual likelihood! This seems to be reasonably well founded for the known scale parameter case (see Hastie and Tibshirani, 1990, section 6.8.3), and in fact in this case the default smoothing parameter estimation criterion is effectively this modified AIC.

Value

Standard `logLik` object: see `logLik`.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩ based directly on `logLik.glm`

References

Hastie and Tibshirani, 1990, Generalized Additive Models.

See Also

`AIC`

---

magic *Stable Multiple Smoothing Parameter Estimation by GCV or UBRE, with optional fixed penalty*

Description

Function to efficiently estimate smoothing parameters in generalized ridge regression problems with multiple (quadratic) penalties, by GCV or UBRE. The function uses Newton's method in multi-dimensions, backed up by steepest descent to iteratively adjust the smoothing parameters for each penalty (one penalty may have a smoothing parameter fixed at unity).

For maximal numerical stability the method is based on orthogonal decomposition methods, and attempts to deal with numerical rank deficiency gracefully using a truncated singular value decomposition approach.

Usage

```
magic(y,X,sp,S,off,rank=NULL,H=NULL,C=NULL,w=NULL,gamma=1,
      scale=1,gcv=TRUE,ridge.parameter=NULL,
      control=list(maxit=50,tol=1e-6,step.half=25,
      rank.tol=.Machine$double.eps^0.5),extra.rss=0,n.score=length(y))
```

Arguments

y	is the response data vector.
X	is the model matrix.
sp	is the array of smoothing parameters multiplying the penalty matrices stored in S. Any that are negative are autoinitialized, otherwise they are taken as supplying starting values. A supplied starting value will be reset to a default starting value if the gradient of the GCV/UBRE score is too small at the supplied value.
S	is a list of of penalty matrices. S[[i]] is the ith penalty matrix, but note that it is not stored as a full matrix, but rather as the smallest square matrix including all the non-zero elements of the penalty matrix. Element 1,1 of S[[i]] occupies element off[i], off[i] of the ith penalty matrix. Each S[[i]] must be positive semi-definite.
off	is an array indicating the first parameter in the parameter vector that is penalized by the penalty involving S[[i]].
rank	is an array specifying the ranks of the penalties. This is useful, but not essential, for forming square roots of the penalty matrices.
H	is the optional offset penalty - i.e. a penalty with a smoothing parameter fixed at 1. This is useful for allowing regularization of the estimation process, fixed smoothing penalties etc.
C	is the optional matrix specifying any linear equality constraints on the fitting problem. If $\mathbf{b}$ is the parameter vector then the parameters are forced to satisfy $\mathbf{Cb} = \mathbf{0}$.
w	the regression weights. If this is a matrix then it is taken as being the square root of the inverse of the covariance matrix of y, specifically $\mathbf{V}_y^{-1} = \mathbf{w}'\mathbf{w}$. If w is an array then it is taken as the diagonal of this matrix, or simply the weight for each element of y. See below for an example using this.
gamma	is an inflation factor for the model degrees of freedom in the GCV or UBRE score.
scale	is the scale parameter for use with UBRE.
gcv	should be set to TRUE if GCV is to be used, FALSE for UBRE.
ridge.parameter	It is sometimes useful to apply a ridge penalty to the fitting problem, penalizing the parameters in the constrained space directly. Setting this parameter to a value greater than zero will cause such a penalty to be used, with the magnitude given by the parameter value.
control	is a list of iteration control constants with the following elements:
maxit	The maximum number of iterations of the magic algorithm to allow.
tol	The tolerance to use in judging convergence.
step.half	If a trial step fails then the method tries halving it up to a maximum of step.half times.
rank.tol	is a constant used to test for numerical rank deficiency of the problem. Basically any singular value less than rank_tol multiplied by the largest singular value of the problem is set to zero.
extra.rss	is a constant to be added to the residual sum of squares (squared norm) term in the calculation of the GCV, UBRE and scale parameter estimate. In conjuction with n.score, this is useful for certain methods for dealing with very large data sets.

`n.score` number to use as the number of data in GCV/UBRE score calculation: usually the actual number of data, but there are methods for dealing with very large datasets that change this.

Details

The method is a computationally efficient means of applying GCV or UBRE (often approximately AIC) to the problem of smoothing parameter selection in generalized ridge regression problems of the form:

$$minimise \; \|\mathbf{W}(\mathbf{X}\mathbf{b}-\mathbf{y})\|^2 + \mathbf{b}'\mathbf{H}\mathbf{b} + \sum_{i=1}^{m} \theta_i \mathbf{b}'\mathbf{S}_i\mathbf{b}$$

possibly subject to constraints $\mathbf{C}\mathbf{b} = \mathbf{0}$. $\mathbf{X}$ is a design matrix, $\mathbf{b}$ a parameter vector, $\mathbf{y}$ a data vector, $\mathbf{W}$ a weight matrix, $\mathbf{S}_i$ a positive semi-definite matrix of coefficients defining the ith penalty with associated smoothing parameter θ_i, $\mathbf{H}$ is the positive semi-definite offset penalty matrix and $\mathbf{C}$ a matrix of coefficients defining any linear equality constraints on the problem. $\mathbf{X}$ need not be of full column rank.

The θ_i are chosen to minimize either the GCV score:

$$V_g = \frac{n\|\mathbf{W}(\mathbf{y}-\mathbf{A}\mathbf{y})\|^2}{[tr(\mathbf{I}-\gamma\mathbf{A})]^2}$$

or the UBRE score:

$$V_u = \|\mathbf{W}(\mathbf{y}-\mathbf{A}\mathbf{y})\|^2/n - 2\phi tr(\mathbf{I}-\gamma\mathbf{A})/n + \phi$$

where γ is `gamma` the inflation factor for degrees of freedom (usually set to 1) and ϕ is `scale`, the scale parameter. $\mathbf{A}$ is the hat matrix (influence matrix) for the fitting problem (i.e the matrix mapping data to fitted values). Dependence of the scores on the smoothing parameters is through $\mathbf{A}$.

The method operates by Newton or steepest descent updates of the logs of the θ_i. A key aspect of the method is stable and economical calculation of the first and second derivatives of the scores w.r.t. the log smoothing parameters. Because the GCV/UBRE scores are flat w.r.t. very large or very small θ_i, it's important to get good starting parameters, and to be careful not to step into a flat region of the smoothing parameter space. For this reason the algorithm rescales any Newton step that would result in a $log(\theta_i)$ change of more than 5. Newton steps are only used if the Hessian of the GCV/UBRE is postive definite, otherwise steepest descent is used. Similarly steepest descent is used if the Newton step has to be contracted too far (indicating that the quadratic model underlying Newton is poor). All initial steepest descent steps are scaled so that their largest component is 1. However a step is calculated, it is never expanded if it is successful (to avoid flat portions of the objective), but steps are successively halved if they do not decrease the GCV/UBRE score, until they do, or the direction is deemed to have failed. (Given the smoothing parameters the optimal $\mathbf{b}$ parameters are easily found.)

The method is coded in `C` with matrix factorizations performed using LINPACK and LAPACK routines.

Value

The function returns a list with the following items:

`b` The best fit parameters given the estimated smoothing parameters.

`scale` the estimated (GCV) or supplied (UBRE) scale parameter.

score	the minimized GCV or UBRE score.
sp	an array of the estimated smoothing parameters.
rV	a factored form of the parameter covariance matrix. The (Bayesian) covariance matrix of the parametes b is given by rV%*%t(rV)*scale.
gcv.info	is a list of information about the performance of the method with the following elements:
full.rank	The apparent rank of the problem: number of parameters less number of equality constraints.
rank	The estimated actual rank of the problem (at the final iteration of the method).
fully.converged	is TRUE if the method converged by satisfying the convergence criteria, and FALSE if it coverged by failing to decrease the score along the search direction.
hess.pos.def	is TRUE if the hessian of the UBRE or GCV score was positive definite at convergence.
iter	is the number of Newton/Steepest descent iterations taken.
score.calls	is the number of times that the GCV/UBRE score had to be evaluated.
rms.grad	is the root mean square of the gradient of the UBRE/GCV score w.r.t. the smoothing parameters.

Note that some further useful quantities can be obtained using magic.post.proc.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

Wood, S.N. (2004) Stable and efficient multiple smoothing parameter estimation for generalized additive models. J. Amer. Statist. Ass. 99:673-686

http://www.maths.bath.ac.uk/~sw283/

See Also

magic.post.proc, mgcv, gam,

Examples

```
## Use `magic' for a standard additive model fit ...
   library(mgcv)
   set.seed(1);n <- 400;sig2 <- 4
   x0 <- runif(n, 0, 1);x1 <- runif(n, 0, 1)
   x2 <- runif(n, 0, 1);x3 <- runif(n, 0, 1)
   f <- 2 * sin(pi * x0)
   f <- f + exp(2 * x1) - 3.75887
   f <- f+0.2*x2^11*(10*(1-x2))^6+10*(10*x2)^3*(1-x2)^10
   e <- rnorm(n, 0, sqrt(sig2))
   y <- f + e
## set up additive model
   G <- gam(y~s(x0)+s(x1)+s(x2)+s(x3),fit=FALSE)
## fit using magic
   mgfit <- magic(G$y,G$X,G$sp,G$S,G$off,G$rank,C=G$C)
## and fit using gam as consistency check
```

```
   b <- gam(G=G)
   mgfit$sp;b$sp  # compare smoothing parameter estimates
   edf <- magic.post.proc(G$X,mgfit,G$w)$edf  # extract e.d.f. per parameter
## get termwise e.d.f.s
   twedf <- 0;for (i in 1:4) twedf[i] <- sum(edf[((i-1)*10+1):(i*10)])
   twedf;b$edf  # compare

## Now a correlated data example ...
    library(nlme)
## simulate truth
    set.seed(1);n<-400;sig<-2
    x <- 0:(n-1)/(n-1)
    f <- 0.2*x^11*(10*(1-x))^6+10*(10*x)^3*(1-x)^10-1.396
## produce scaled covariance matrix for AR1 errors...
    V <- corMatrix(Initialize(corAR1(.6),data.frame(x=x)))
    Cv <- chol(V)  # t(Cv)
## Simulate AR1 errors ...
    e <- t(Cv)%*%rnorm(n,0,sig) # so cov(e) = V * sig^2
## Observe truth + AR1 errors
    y <- f + e
## GAM ignoring correlation
    par(mfrow=c(1,2))
    b <- gam(y~s(x,k=20))
    plot(b);lines(x,f-mean(f),col=2);title("Ignoring correlation")
## Fit smooth, taking account of *known* correlation...
    w <- solve(t(Cv)) # V^{-1} = w'w
    ## Use `gam' to set up model for fitting...
    G <- gam(y~s(x,k=20),fit=FALSE)
    ## fit using magic, with weight *matrix*
    mgfit <- magic(G$y,G$X,G$sp,G$S,G$off,G$rank,C=G$C,w=w)
## Modify previous gam object using new fit, for plotting...
    mg.stuff <- magic.post.proc(G$X,mgfit,w)
    b$edf <- mg.stuff$edf;b$Vp <- mg.stuff$Vb
    b$coefficients <- mgfit$b
    plot(b);lines(x,f-mean(f),col=2);title("Known correlation")
```

`magic.post.proc` *Auxilliary information from magic fit*

Description

Obtains Bayesian parameter covariance matrix, frequentist parameter estimator covariance matrix, estimated degrees of freedom for each parameter and leading diagonal of influence/hat matrix, for a penalized regression estimated by `magic`.

Usage

```
magic.post.proc(X,object,w)
```

Arguments

`X`	is the model matrix.
`object`	is the list returned by `magic` after fitting the model with model matrix `X`.

`w` is the weight vector used in fitting, or the weight matrix used in fitting (i.e. supplied to `magic`, if one was.) `t(w)%*%w` should typically give the inverse of the covariance matrix of the response data supplied to `magic`.

Details

`object` contains `rV` ($\mathbf{V}$, say), and `scale` (ϕ, say) which can be used to obtain the require quantities as follows. The Bayesian covariance matrix of the parameters is $\mathbf{V}\mathbf{V}'\phi$. The vector of estimated degrees of freedom for each parameter is the leading diagonal of $\mathbf{V}\mathbf{V}'\mathbf{X}'\mathbf{W}'\mathbf{W}\mathbf{X}$ where $\mathbf{W}$ is either the weight matrix `w` or the matrix `diag(w)`. The hat/influence matrix is given by $\mathbf{W}\mathbf{X}\mathbf{V}\mathbf{V}'\mathbf{X}'\mathbf{W}'$.

The frequentist parameter estimator covariance matrix is $\mathbf{V}\mathbf{V}'\mathbf{X}'\mathbf{W}'\mathbf{W}\mathbf{X}\mathbf{V}\mathbf{V}'\phi$: it is useful for testing terms for equality to zero.

Value

A list with three items:

`Vb`	the Bayesian covariance matrix of the model parameters.
`Ve`	the frequentist covariance matrix for the parameter estimators.
`hat`	the leading diagonal of the hat (influence) matrix.
`edf`	the array giving the estimated degrees of freedom associated with each parameter.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

See Also

`magic`

---

mgcv *Multiple Smoothing Parameter Estimation by GCV or UBRE*

Description

Function to efficiently estimate smoothing parameters in Generalized Ridge Regression Problem with multiple (quadratic) penalties, by GCV or UBRE. The function uses Newton's method in multi-dimensions, backed up by steepest descent to iteratively adjust a set of relative smoothing parameters for each penalty. To ensure that the overall level of smoothing is optimal, and to guard against trapping by local minima, a highly efficient global minimisation with respect to one overall smoothing parameter is also made at each iteration.

For a listing of all routines in the `mgcv` package type:
`library(help="mgcv")`

Usage

```
mgcv(y,X,sp,S,off,C=NULL,w=rep(1,length(y)),H=NULL,
     scale=1,gcv=TRUE,control=mgcv.control())
```

Arguments

`y`	The response data vector.
`X`	The design matrix for the problem, note that `ncol(X)` must give the number of model parameters, while `nrow(X)` should give the number of data.
`sp`	An array of smoothing parameters. If `control$fixed==TRUE` then these are taken as being the smoothing parameters. Otherwise any positive values are assumed to be initial estimates and negative values to signal auto-initialization.
`S`	A list of penalty matrices. Only the smallest square block containing all non-zero matrix elements is actually stored, and `off[i]` indicates the element of the parameter vector that `S[[i]][1,1]` relates to.
`off`	Offset values indicating where in the overall parameter a particular stored penalty starts operating. For example if `p` is the model parameter vector and `k=nrow(S[[i]])-1`, then the ith penalty is given by `t(p[off[i]:(off[i]+k)])%*%S[[i]]%*%p[off[i]:(off[i]+k)]`.
`C`	Matrix containing any linear equality constraints on the problem (i.e. $\mathbf{C}$ in $\mathbf{Cp} = \mathbf{0}$).
`w`	A vector of weights for the data (often proportional to the reciprocal of the standard deviation of `y`).
`H`	A single fixed penalty matrix to be used in place of the multiple penalty matrices in `S`. `mgcv` cannot mix fixed and estimated penalties.
`scale`	This is the known scale parameter/error variance to use with UBRE. Note that it is assumed that the variance of y_i is given by σ^2/w_i.
`gcv`	If `gcv` is TRUE then smoothing parameters are estimated by GCV, otherwise UBRE is used.
`control`	A list of control options returned by `mgcv.control`.

Details

This is documentation for the code implementing the method described in section 4 of Wood (2000) . The method is a computationally efficient means of applying GCV to the problem of smoothing parameter selection in generalized ridge regression problems of the form:

$$minimise\ \|\mathbf{W}(\mathbf{Xp}-\mathbf{y})\|^2\rho + \sum_{i=1}^{m}\lambda_i \mathbf{p}'\mathbf{S}_i\mathbf{p}$$

possibly subject to constraints $\mathbf{Cp} = \mathbf{0}$. $\mathbf{X}$ is a design matrix, $\mathbf{p}$ a parameter vector, $\mathbf{y}$ a data vector, $\mathbf{W}$ a diagonal weight matrix, $\mathbf{S}_i$ a positive semi-definite matrix of coefficients defining the ith penalty and $\mathbf{C}$ a matrix of coefficients defining any linear equality constraints on the problem. The smoothing parameters are the λ_i but there is an overall smoothing parameter ρ as well. Note that $\mathbf{X}$ must be of full column rank, at least when projected into the null space of any equality constraints.

The method operates by alternating very efficient direct searches for ρ with Newton or steepest descent updates of the logs of the λ_i. Because the GCV/UBRE scores are flat w.r.t. very large or very small λ_i, it's important to get good starting parameters, and to be careful not to step into a flat region of the smoothing parameter space. For this reason the algorithm rescales any Newton step that would result in a $log(\lambda_i)$ change of more than 5. Newton steps are only used if the Hessian of the GCV/UBRE is postive definite, otherwise steepest descent is used. Similarly steepest descent is used if the Newton step has to be contracted too far (indicating that the quadratic model underlying Newton is poor). All initial steepest descent steps are scaled so that their largest component is

1. However a step is calculated, it is never expanded if it is successful (to avoid flat portions of the objective), but steps are successively halved if they do not decrease the GCV/UBRE score, until they do, or the direction is deemed to have failed. `M$conv` provides some convergence diagnostics.

The method is coded in `C` and is intended to be portable. It should be noted that seriously ill conditioned problems (i.e. with close to column rank deficiency in the design matrix) may cause problems, especially if weights vary wildly between observations.

Value

An object is returned with the following elements:

`b` The best fit parameters given the estimated smoothing parameters.

`scale` The estimated or supplied scale parameter/error variance.

`score` The UBRE or GCV score.

`sp` The estimated (or supplied) smoothing parameters (λ_i/ρ)

`Vb` Estimated covariance matrix of model parameters.

`hat` diagonal of the hat/influence matrix.

`edf` array of estimated degrees of freedom for each parameter.

`info` A list of convergence diagnostics, with the following elements:

- edf Array of whole model estimated degrees of freedom.
- score Array of ubre/gcv scores at the edfs for the final set of relative smoothing parameters.
- g the gradient of the GCV/UBRE score w.r.t. the smoothing parameters at termination.
- h the second derivatives corresponding to `g` above - i.e. the leading diagonal of the Hessian.
- e the eigenvalues of the Hessian. These should all be non-negative!
- iter the number of iterations taken.
- in.ok `TRUE` if the second smoothing parameter guess improved the GCV/UBRE score. (Please report examples where this is `FALSE`)
- step.fail `TRUE` if the algorithm terminated by failing to improve the GCV/UBRE score rather than by "converging". Not necessarily a problem, but check the above derivative information quite carefully.

WARNING

The method may not behave well with near column rank deficient $\mathbf{X}$ especially in contexts where the weights vary wildly.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

Gu and Wahba (1991) Minimizing GCV/GML scores with multiple smoothing parameters via the Newton method. SIAM J. Sci. Statist. Comput. 12:383-398

Wood, S.N. (2000) Modelling and Smoothing Parameter Estimation with Multiple Quadratic Penalties. J.R.Statist.Soc.B 62(2):413-428

http://www.maths.bath.ac.uk/~sw283/

See Also

gam, magic

Examples

```
library(help="mgcv") # listing of all routines

set.seed(1);n<-400;sig2<-4
x0 <- runif(n, 0, 1);x1 <- runif(n, 0, 1)
x2 <- runif(n, 0, 1);x3 <- runif(n, 0, 1)
f <- 2 * sin(pi * x0)
f <- f + exp(2 * x1) - 3.75887
f <- f+0.2*x2^11*(10*(1-x2))^6+10*(10*x2)^3*(1-x2)^10-1.396
e <- rnorm(n, 0, sqrt(sig2))
y <- f + e
# set up additive model
G<-gam(y~s(x0)+s(x1)+s(x2)+s(x3),fit=FALSE)
# fit using mgcv
mgfit<-mgcv(G$y,G$X,G$sp,G$S,G$off,C=G$C)
```

---

mgcv.control *Setting mgcv defaults*

Description

This is an internal function of package mgcv which allows control of the numerical options for fitting a generalized ridge regression problem using routine mgcv.

Usage

```
mgcv.control(conv.tol=1e-7,max.half=20,target.edf=NULL,min.edf=-1)
```

Arguments

conv.tol	The convergence tolerance.
max.half	successive step halvings are employed if the Newton method and then the steepest descent backup fail to improve the UBRE/GCV score. This is how many to use before giving up.
target.edf	If this is non-null it indicates that cautious optimization should be used, which opts for the local minimum closest to the target model edf if there are multiple local minima in the GCV/UBRE score.
min.edf	Lower bound on the model edf. Useful for avoiding numerical problems at high smoothing parameter values. Negative for none.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

Gu and Wahba (1991) Minimizing GCV/GML scores with multiple smoothing parameters via the Newton method. SIAM J. Sci. Statist. Comput. 12:383-398

Wood, S.N. (2000) Modelling and Smoothing Parameter Estimation with Multiple Quadratic Penalties. J.R.Statist.Soc.B 62(2):413-428

http://www.maths.bath.ac.uk/~sw283/

See Also

mgcv

`mono.con` *Monotonicity constraints for a cubic regression spline*

Description

Finds linear constraints sufficient for monotonicity (and optionally upper and/or lower boundedness) of a cubic regression spline. The basis representation assumed is that given by the `gam`, `"cr"` basis: that is the spline has a set of knots, which have fixed x values, but the y values of which constitute the parameters of the spline.

Usage

```
mono.con(x,up=TRUE,lower=NA,upper=NA)
```

Arguments

`x`	The array of knot locations.
`up`	If `TRUE` then the constraints imply increase, if `FALSE` then decrease.
`lower`	This specifies the lower bound on the spline unless it is `NA` in which case no lower bound is imposed.
`upper`	This specifies the upper bound on the spline unless it is `NA` in which case no upper bound is imposed.

Details

Consider the natural cubic spline passing through the points: $\{x_i, p_i : i = 1 \ldots n\}$. Then it is possible to find a relatively small set of linear constraints on $\mathbf{p}$ sufficient to ensure monotonicity (and bounds if required): $\mathbf{Ap} \geq \mathbf{b}$. Details are given in Wood (1994). This function returns a list containing `A` and `b`.

Value

The function returns a list containing constraint matrix `A` and constraint vector `b`.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

Gill, P.E., Murray, W. and Wright, M.H. (1981) Practical Optimization. Academic Press, London.

Wood, S.N. (1994) Monotonic smoothing splines fitted by cross validation SIAM Journal on Scientific Computing 15(5):1126-1133

http://www.maths.bath.ac.uk/~sw283/

See Also

mgcv pcls

Examples

```
## see ?pcls
```

---

mroot *Smallest square root of matrix*

Description

Find a square root of a positive semi-definite matrix, having as few columns as possible. Uses either pivoted choleski decomposition or singular value decomposition to do this.

Usage

```
mroot(A,rank=NULL,method="chol")
```

Arguments

A	The positive semi-definite matrix, a square root of which is to be found.
rank	if the rank of the matrix A is known then it should be supplied.
method	"chol" to use pivoted choloeski decompositon, which is fast but tends to overestimate rank. "svd" to use singular value decomposition, which is slow, but is the most accurate way to estimate rank.

Details

The routine uses an LAPACK SVD routine, or the LINPACK pivoted Choleski routine. It is primarily of use for turning penalized regression problems into ordinary regression problems.

Value

A matrix, $\mathbf{B}$ with as many columns as the rank of $\mathbf{A}$, and such that $\mathbf{A} = \mathbf{B}\mathbf{B}'$.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

Examples

```
set.seed(0)
a <- matrix(runif(24),6,4)
A <- a%*%t(a) ## A is +ve semi-definite, rank 4
B <- mroot(A) ## default pivoted choleski method
tol <- 100*.Machine$double.eps
chol.err <- max(abs(A-B%*%t(B)));chol.err
if (chol.err>tol) warning("mroot (chol) suspect")
B <- mroot(A,method="svd") ## svd method
svd.err <- max(abs(A-B%*%t(B)));svd.err
if (svd.err>tol) warning("mroot (svd) suspect")
```

---

new.name *Obtain a name for a new variable that is not already in use*

Description

gamm works by transforming a GAMM into something that can be estimated by lme, but this involves creating new variables, the names of which should not clash with the names of other variables on which the model depends. This simple service routine checks a suggested name against a list of those in use, and if neccesary modifies it so that there is no clash.

Usage

```
new.name(proposed,old.names)
```

Arguments

proposed	a suggested name
old.names	An array of names that must not be duplicated

Value

A name that is not in old.names.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

http://www.maths.bath.ac.uk/~sw283/

See Also

gamm

Examples

```
old <- c("a","tuba","is","tubby")
new.name("tubby",old)
```

notExp *Functions for better-than-log positive parameterization*

Description

It is common practice in statistical optimization to use log-parameterizations when a parameter ought to be positive. i.e. if an optimization parameter `a` should be non-negative then we use `a=exp(b)` and optimize with respect to the unconstrained parameter `b`. This often works well, but it does imply a rather limited working range for `b`: using 8 byte doubles, for example, if `b`'s magnitude gets much above 700 then `a` overflows or underflows. This can cause problems for numerical optimization methods.

`notExp` is a monotonic function for mapping the real line into the positive real line with much less extreme underflow and overflow behaviour than `exp`. It is a piece-wise function, but is continuous to second derivative: see the source code for the exact definition, and the example below to see what it looks like.

`notLog` is the inverse function of `notExp`.

The major use of these functions was originally to provide more robust `pdMat` classes for `lme` for use by `gamm`. Currently the `notExp2` and `notLog2` functions are used in their place, as a result of changes to the nlme optimization routines.

Usage

```
notExp(x)

notLog(x)
```

Arguments

`x`	Argument array of real numbers (`notExp`) or positive real numbers (`notLog`).

Value

An array of function values evaluated at the supplied argument values.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

http://www.maths.bath.ac.uk/~sw283/

See Also

`pdTens`, `pdIdnot`, `gamm`

Examples

```
## Illustrate the notExp function:
## less steep than exp, but still monotonic.
x <- -100:100/10
op <- par(mfrow=c(2,2))
plot(x,notExp(x),type="l")
lines(x,exp(x),col=2)
plot(x,log(notExp(x)),type="l")
lines(x,log(exp(x)),col=2) # redundancy intended
x <- x/4
plot(x,notExp(x),type="l")
lines(x,exp(x),col=2)
plot(x,log(notExp(x)),type="l")
lines(x,log(exp(x)),col=2) # redundancy intended
par(op)
range(notLog(notExp(x))-x) # show that inverse works!
```

---

notExp2 *Alternative to log parameterization for variance components*

Description

notLog2 and notExp2 are alternatives to log and exp or notLog and notExp for re-parameterization of variance parameters. They are used by the pdTens and pdIdnot classes which in turn implement smooths for gamm.

The functions are typically used to ensure that smoothing parameters are positive, but the notExp2 is not monotonic: rather it cycles between 'effective zero' and 'effective infinity' as its argument changes. The notLog2 is the inverse function of the notExp2 only over an interval centered on zero.

Parameterizations using these functions ensure that estimated smoothing parameters remain positive, but also help to ensure that the likelihood is never indefinite: once a working parameter pushes a smoothing parameter below 'effetive zero' or above 'effective infinity' the cyclic nature of the notExp2 causes the likelihood to decrease, where otherwise it might simply have flattened.

This parameterization is really just a numerical trick, in order to get lme to fit gamm models, without failing due to indefiniteness. Note in particular that asymptoticresults on the likelihood/REML criterion are not invalidated by the trick, unless parameter estimates end up close to the effective zero or effective infinity: but if this is the case then the asymptotics would also have been invalid for a conventional monotonic parameterization.

This reparameterization was made necessary by some modifications to the underlying optimization method in lme introduced in nlme 3.1-62. It is possible that future releases will return to the notExp parameterization.

Note that you can reset 'effective zero' and 'effective infinity': see below.

Usage

```
notExp2(x,d=.Options$mgcv.vc.logrange,b=1/d)

notLog2(x,d=.Options$mgcv.vc.logrange,b=1/d)
```

Arguments

x	Argument array of real numbers (notExp) or positive real numbers (notLog).
d	the range of notExp2 runs from exp(-d) to exp(d). To change the range used by gamm reset mgcv.vc.logrange using options.
b	determines the period of the cycle of notExp2.

Value

An array of function values evaluated at the supplied argument values.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

http://www.maths.bath.ac.uk/~sw283/

See Also

pdTens, pdIdnot, gamm

Examples

```
## Illustrate the notExp2 function:
x <- seq(-50,50,length=1000)
op <- par(mfrow=c(2,2))
plot(x,notExp2(x),type="l")
lines(x,exp(x),col=2)
plot(x,log(notExp2(x)),type="l")
lines(x,log(exp(x)),col=2) # redundancy intended
x <- x/4
plot(x,notExp2(x),type="l")
lines(x,exp(x),col=2)
plot(x,log(notExp2(x)),type="l")
lines(x,log(exp(x)),col=2) # redundancy intended
par(op)
```

---

null.space.dimension

The basis of the space of un-penalized functions for a TPRS

Description

The thin plate spline penalties give zero penalty to some functions. The space of these functions is spanned by a set of polynomial terms. null.space.dimension finds the dimension of this space, M, given the number of covariates that the smoother is a function of, d, and the order of the smoothing penalty, m. If m does not satisfy $2m > d$ then the smallest possible dimension for the null space is found given d and the requirement that the smooth should be visually smooth.

Usage

```
null.space.dimension(d,m)
```

Arguments

d	is a positive integer - the number of variables of which the t.p.s. is a function.
m	a non-negative integer giving the order of the penalty functional, or signalling that the default order should be used.

Details

Thin plate splines are only visually smooth if the order of the wiggliness penalty, m, satisfies $2m > d + 1$. If $2m < d + 1$ then this routine finds the smallest m giving visual smoothness for the given d, otherwise the supplied m is used. The null space dimension is given by:

$M = (m + d + 1)!/(d!(m - d)!)$

which is the value returned.

Value

An integer (array), the null space dimension M.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

Wood, S.N. (2003) Thin plate regression splines. J.R.Statist.Soc.B 65(1):95-114

http://www.maths.bath.ac.uk/~sw283/

See Also

gam

Examples

```
null.space.dimension(2,0)
```

---

pcls *Penalized Constrained Least Squares Fitting*

Description

Solves least squares problems with quadratic penalties subject to linear equality and inequality constraints using quadratic programming.

Usage

```
pcls(M)
```

Arguments

`M` is the single list argument to `pcls`. It should have the following elements:

- y The response data vector.
- w A vector of weights for the data (often proportional to the reciprocal of the variance).
- X The design matrix for the problem, note that `ncol(M$X)` must give the number of model parameters, while `nrow(M$X)` should give the number of data.
- C Matrix containing any linear equality constraints on the problem (e.g. $\mathbf{C}$ in $\mathbf{Cp} = \mathbf{c}$). If you have no equality constraints initialize this to a zero by zero matrix. Note that there is no need to supply the vector $\mathbf{c}$, it is defined implicitly by the initial parameter estimates $\mathbf{p}$.
- S A list of penalty matrices. `S[[i]]` is the smallest contiguous matrix including all the non-zero elements of the ith penalty matrix. The first parameter it penalizes is given by `off[i]+1` (starting counting at 1).
- off Offset values locating the elements of `M$S` in the correct location within each penalty coefficient matrix. (Zero offset implies starting in first location)
- sp An array of smoothing parameter estimates.
- p An array of feasible initial parameter estimates - these must satisfy the constraints, but should avoid satisfying the inequality constraints as equality constraints.
- Ain Matrix for the inequality constraints $\mathbf{A}_{in}\mathbf{p} > \mathbf{b}_{in}$.
- bin vector in the inequality constraints.

Details

This solves the problem:

$$minimise \, \|\mathbf{W}^{1/2}(\mathbf{Xp} - \mathbf{y})\|^2 + \sum_{i=1}^{m} \lambda_i \mathbf{p}'\mathbf{S}_i\mathbf{p}$$

subject to constraints $\mathbf{Cp} = \mathbf{c}$ and $\mathbf{A}_{in}\mathbf{p} > \mathbf{b}_{in}$, w.r.t. $\mathbf{p}$ given the smoothing parameters λ_i. $\mathbf{X}$ is a design matrix, $\mathbf{p}$ a parameter vector, $\mathbf{y}$ a data vector, $\mathbf{W}$ a diagonal weight matrix, $\mathbf{S}_i$ a positive semi-definite matrix of coefficients defining the ith penalty and $\mathbf{C}$ a matrix of coefficients defining the linear equality constraints on the problem. The smoothing parameters are the λ_i. Note that $\mathbf{X}$ must be of full column rank, at least when projected into the null space of any equality constraints. $\mathbf{A}_{in}$ is a matrix of coefficients defining the inequality constraints, while $\mathbf{b}_{in}$ is a vector involved in defining the inequality constraints.

Quadratic programming is used to perform the solution. The method used is designed for maximum stability with least squares problems: i.e. $\mathbf{X}'\mathbf{X}$ is not formed explicitly. See Gill et al. 1981.

Value

The function returns an array containing the estimated parameter vector.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

Gill, P.E., Murray, W. and Wright, M.H. (1981) Practical Optimization. Academic Press, London.

Wood, S.N. (1994) Monotonic smoothing splines fitted by cross validation SIAM Journal on Scientific Computing 15(5):1126-1133

http://www.maths.bath.ac.uk/~sw283/

See Also

mgcv mono.con

Examples

```
# first an un-penalized example - fit E(y)=a+bx subject to a>0
set.seed(0)
n<-100
x<-runif(n);y<-x-0.2+rnorm(n)*0.1
M<-list(X=matrix(0,n,2),p=c(0.1,0.5),off=array(0,0),S=list(),
Ain=matrix(0,1,2),bin=0,C=matrix(0,0,0),sp=array(0,0),y=y,w=y*0+1)
M$X[,1]<-1;M$X[,2]<-x;M$Ain[1,]<-c(1,0)
pcls(M)->M$p
plot(x,y);abline(M$p,col=2);abline(coef(lm(y~x)),col=3)

# Penalized example: monotonic penalized regression spline .....

# Generate data from a monotonic truth.
x<-runif(100)*4-1;x<-sort(x);
f<-exp(4*x)/(1+exp(4*x));y<-f+rnorm(100)*0.1;plot(x,y)
dat<-data.frame(x=x,y=y)
# Show regular spline fit (and save fitted object)
f.ug<-gam(y~s(x,k=10,bs="cr"));lines(x,fitted(f.ug))
# Create Design matrix, constraints etc. for monotonic spline....
sm<-smoothCon(s(x,k=10,bs="cr"),dat,knots=NULL)
F<-mono.con(sm$xp);   # get constraints
G<-list(X=sm$X,C=matrix(0,0,0),sp=f.ug$sp,p=sm$xp,y=y,w=y*0+1)
G$Ain<-F$A;G$bin<-F$b;G$S<-sm$S;G$off<-0

p<-pcls(G);  # fit spline (using s.p. from unconstrained fit)

fv<-Predict.matrix(sm,data.frame(x=x))%*%p
lines(x,fv,col=2)

# now a tprs example of the same thing....

f.ug<-gam(y~s(x,k=10));lines(x,fitted(f.ug))
# Create Design matrix, constriants etc. for monotonic spline....
sm<-smoothCon(s(x,k=10,bs="tp"),dat,knots=NULL)
xc<-0:39/39 # points on [0,1]
nc<-length(xc)  # number of constraints
xc<-xc*4-1  # points at which to impose constraints
A0<-Predict.matrix(sm,data.frame(x=xc))
# ... A0
A1<-Predict.matrix(sm,data.frame(x=xc+1e-6))
A<-(A1-A0)/1e-6
# ... approx. constraint matrix (A%*%p is -ve spline gradient at points xc)
G<-list(X=sm$X,C=matrix(0,0,0),sp=f.ug$sp,y=y,w=y*0+1,S=sm$S,off=0)
```

```
G$Ain<-A;    # constraint matrix
G$bin<-rep(0,nc);  # constraint vector
G$p<-rep(0,10);G$p[10]<-0.1
# ... monotonic start params, got by setting coefs of polynomial part
p<-pcls(G);  # fit spline (using s.p. from unconstrained fit)

fv2<-Predict.matrix(sm,data.frame(x=x))%*%p
lines(x,fv2,col=3)
```

`pdIdnot` *Overflow proof pdMat class for multiples of the identity matrix*

Description

This set of functions is a modification of the `pdMat` class `pdIdent` from library `nlme`. The modification is to replace the log parameterization used in `pdMat` with a `notLog2` parameterization, since the latter avoids indefiniteness in the likelihood and associated convergence problems: the parameters also relate to variances rather than standard deviations, for consistency with the `pdTens` class. The functions are particularly useful for working with Generalized Additive Mixed Models where variance parameters/smoothing parameters can be very large or very small, so that overflow or underflow can be a problem.

These functions would not normally be called directly, although unlike the `pdTens` class it is easy to do so.

Usage

```
pdIdnot(value = numeric(0), form = NULL,
        nam = NULL, data = sys.frame(sys.parent()))
```

Arguments

`value`	Initialization values for parameters. Not normally used.
`form`	A one sided formula specifying the random effects structure.
`nam`	a names argument, not normally used with this class.
`data`	data frame in which to evaluate formula.

Details

Note that while the `pdFactor` and `pdMatrix` functions return the inverse of the scaled random effect covariance matrix or its factor, the `pdConstruct` function is initialised with estimates of the scaled covariance matrix itself.

Value

A class `pdIdnot` object, or related quantities. See the `nlme` documentation for further details.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

Pinheiro J.C. and Bates, D.M. (2000) Mixed effects Models in S and S-PLUS. Springer

The nlme source code.

http://www.maths.bath.ac.uk/~sw283/

See Also

te, pdTens, notLog2, gamm

Examples

```
# see gamm
```

---

pdTens *Functions implementing a pdMat class for tensor product smooths*

Description

This set of functions implements an nlme library pdMat class to allow tensor product smooths to be estimated by lme as called by gamm. Tensor product smooths have a penalty matrix made up of a weighted sum of penalty matrices, where the weights are the smoothing parameters. In the mixed model formulation the penalty matrix is the inverse of the covariance matrix for the random effects of a term, and the smoothing parameters (times a half) are variance parameters to be estimated. It's not possible to transform the problem to make the required random effects covariance matrix look like one of the standard pdMat classes: hence the need for the pdTens class. A notLog2 parameterization ensures that the parameters are positive.

These functions (pdTens, pdConstruct.pdTens, pdFactor.pdTens, pdMatrix.pdTens, coef.pdTens and summary.pdTens) would not normally be called directly.

Usage

```
pdTens(value = numeric(0), form = NULL,
       nam = NULL, data = sys.frame(sys.parent()))
```

Arguments

value	Initialization values for parameters. Not normally used.
form	A one sided formula specifying the random effects structure. The formula should have an attribute S which is a list of the penalty matrices the weighted sum of which gives the inverse of the covariance matrix for these random effects.
nam	a names argument, not normally used with this class.
data	data frame in which to evaluate formula.

Details

If using this class directly note that it is worthwhile scaling the S matrices to be of 'moderate size', for example by dividing each matrix by its largest singular value: this avoids problems with lme defaults (ref{smooth.construct.tensor.smooth.spec} does this automatically).

This appears to be the minimum set of functions required to implement a new pdMat class.

Note that while the pdFactor and pdMatrix functions return the inverse of the scaled random effect covariance matrix or its factor, the pdConstruct function is sometimes initialised with estimates of the scaled covariance matrix, and sometimes intialized with its inverse.

Value

A class pdTens object, or its coefficients or the matrix it represents or the factor of that matrix. pdFactor returns the factor as a vector (packed column-wise) (pdMatrix always returns a matrix).

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

Pinheiro J.C. and Bates, D.M. (2000) Mixed effects Models in S and S-PLUS. Springer

The nlme source code.

http://www.maths.bath.ac.uk/~sw283/

See Also

te gamm

Examples

```
# see gamm
```

---

place.knots *Automatically place a set of knots evenly through covariate values*

Description

Given a univariate array of covariate values, places a set of knots for a regression spline evenly through the covariate values.

Usage

```
place.knots(x,nk)
```

Arguments

x	array of covariate values (need not be sorted).
nk	integer indicating the required number of knots.

Details

Places knots evenly throughout a set of covariates. For example, if you had 11 covariate values and wanted 6 knots then a knot would be placed at the first (sorted) covariate value and every second (sorted) value thereafter. With less convenient numbers of data and knots the knots are placed within intervals between data in order to achieve even coverage, where even means having approximately the same number of data between each pair of knots.

Value

An array of knot locations.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

http://www.maths.bath.ac.uk/~sw283/

See Also

smooth.construct.cc.smooth.spec

Examples

```
x<-runif(30)
place.knots(x,7)
rm(x)
```

---

plot.gam *Default GAM plotting*

Description

Takes a fitted gam object produced by gam() and plots the component smooth functions that make it up, on the scale of the linear predictor. Optionally produces term plots for parametric model components as well.

Usage

```
## S3 method for class 'gam':
plot(x,residuals=FALSE,rug=TRUE,se=TRUE,pages=0,select=NULL,scale=-1,
         n=100,n2=40,pers=FALSE,theta=30,phi=30,jit=FALSE,xlab=NULL,
         ylab=NULL,main=NULL,ylim=NULL,xlim=NULL,too.far=0.1,
         all.terms=FALSE,shade=FALSE,shade.col="gray80",
         shift=0,trans=I,...)
```

Arguments

x	a fitted gam object as produced by gam().
residuals	If TRUE then partial residuals are added to plots of 1-D smooths. If FALSE then no residuals are added. If this is an array of the correct length then it is used as the array of residuals to be used for producing partial residuals. If TRUE then the residuals are the working residuals from the IRLS iteration weighted by the IRLS weights. Partial residuals for a smooth term are the residuals that would be obtained by dropping the term concerned from the model, while leaving all other estimates fixed (i.e. the estimates for the term plus the residuals).
rug	when TRUE (default) then the covariate to which the plot applies is displayed as a rug plot at the foot of each plot of a 1-d smooth, and the locations of the covariates are plotted as points on the contour plot representing a 2-d smooth.
se	when TRUE (default) upper and lower lines are added to the 1-d plots at 2 standard errors above and below the estimate of the smooth being plotted while for 2-d plots, surfaces at +1 and -1 standard errors are contoured and overlayed on the contour plot for the estimate. If a positive number is supplied then this number is multiplied by the standard errors when calculating standard error curves or surfaces. See also shade, below.
pages	(default 0) the number of pages over which to spread the output. For example, if pages=1 then all terms will be plotted on one page with the layout performed automatically. Set to 0 to have the routine leave all graphics settings as they are.
select	Allows the plot for a single model term to be selected for printing. e.g. if you just want the plot for the second smooth term set select=2.
scale	set to -1 (default) to have the same y-axis scale for each plot, and to 0 for a different y axis for each plot. Ignored if ylim supplied.
n	number of points used for each 1-d plot - for a nice smooth plot this needs to be several times the estimated degrees of freedom for the smooth. Default value 100.
n2	Square root of number of points used to grid estimates of 2-d functions for contouring.
pers	Set to TRUE if you want perspective plots for 2-d terms.
theta	One of the perspective plot angles.
phi	The other perspective plot angle.
jit	Set to TRUE if you want rug plots for 1-d terms to be jittered.
xlab	If supplied then this will be used as the x label for all plots.
ylab	If supplied then this will be used as the y label for all plots.
main	Used as title (or z axis label) for plots if supplied.
ylim	If supplied then this pair of numbers are used as the y limits for each plot.
xlim	If supplied then this pair of numbers are used as the x limits for each plot.
too.far	If greater than 0 then this is used to determine when a location is too far from data to be plotted when plotting 2-D smooths. This is useful since smooths tend to go wild away from data. The data are scaled into the unit square before deciding what to exclude, and too.far is a distance within the unit square.
all.terms	if set to TRUE then the partial effects of parametric model components are also plotted, via a call to termplot. Only terms of order 1 can be plotted in this way.

shade	Set to TRUE to produce shaded regions as confidence bands for smooths (not avaliable for parametric terms, which are plotted using termplot).
shade.col	define the color used for shading confidence bands.
shift	constant to add to each smooth (on the scale of the linear predictor) before plotting. Can be useful for some diagnostics, or with trans.
trans	function to apply to each smooth (after any shift), before plotting. shift and trans are occasionally useful as a means for getting plots on the response scale, when the model consists only of a single smooth.
...	other graphics parameters to pass on to plotting commands.

Details

Produces default plot showing the smooth components of a fitted GAM, and optionally parametric terms as well, when these can be handled by termplot.

For plots of 1-d smooths, the x axis of each plot is labelled with the covariate name, while the y axis is labelled s(cov,edf) where cov is the covariate name, and edf the estimated (or user defined for regression splines) degrees of freedom of the smooth.

Contour plots are produced for 2-d smooths with the x-axes labelled with the first covariate name and the y axis with the second covariate name. The main title of the plot is something like s(var1,var2,edf), indicating the variables of which the term is a function, and the estimated degrees of freedom for the term. When se=TRUE, estimator variability is shown by overlaying contour plots at plus and minus 1 s.e. relative to the main estimate. If se is a positive number then contour plots are at plus or minus se multiplied by the s.e. Contour levels are chosen to try and ensure reasonable separation of the contours of the different plots, but this is not always easy to achieve. Note that these plots can not be modified to the same extent as the other plot.

Smooths of more than 2 variables are not currently dealt with, but simply generate a warning, but see vis.gam.

Fine control of plots for parametric terms can be obtained by calling termplot directly, taking care to use its terms argument.

Value

The function simply generates plots.

WARNING

Note that the behaviour of this function is not identical to plot.gam() in S-PLUS.

Plots of 2-D smooths with standard error contours shown can not easily be customized.

The function can not deal with smooths of more than 2 variables!

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

Henric Nilsson ⟨henric.nilsson@statisticon.se⟩ donated the code for the shade option.

The design is inspired by the S function of the same name described in Chambers and Hastie (1993) (but is not a clone).

References

Chambers and Hastie (1993) Statistical Models in S. Chapman & Hall.

Gu and Wahba (1991) Minimizing GCV/GML scores with multiple smoothing parameters via the Newton method. SIAM J. Sci. Statist. Comput. 12:383-398

Wood, S.N. (2000) Modelling and Smoothing Parameter Estimation with Multiple Quadratic Penalties. J.R.Statist.Soc.B 62(2):413-428

Wood, S.N. (2003) Thin plate regression splines. J.R.Statist.Soc.B 65(1):95-114

Wood, S.N. (2004) Stable and efficient multiple smoothing parameter estimation for generalized additive models. J. Amer. Statist. Ass. 99:637-686

http://www.maths.bath.ac.uk/~sw283/

See Also

gam, predict.gam, vis.gam

Examples

```
library(mgcv)
set.seed(0)
## fake some data...

f1 <- function(x) {exp(2 * x)}
f2 <- function(x) {
  0.2*x^11*(10*(1-x))^6+10*(10*x)^3*(1-x)^10
}
f3 <- function(x) {x*0}

n<-200
sig2<-4
x0 <- rep(1:4,50)
x1 <- runif(n, 0, 1)
x2 <- runif(n, 0, 1)
x3 <- runif(n, 0, 1)
e <- rnorm(n, 0, sqrt(sig2))
y <- 2*x0 + f1(x1) + f2(x2) + f3(x3) + e
x0 <- factor(x0)

## fit and plot...
b<-gam(y~x0+s(x1)+s(x2)+s(x3))
plot(b,pages=1,residuals=TRUE,all.terms=TRUE,shade=TRUE,shade.col=2)

## just parametric term alone...
termplot(b,terms="x0",se=TRUE)

## more use of color...
op <- par(mfrow=c(2,2),bg="blue")
x <- 0:1000/1000
for (i in 1:3) {
  plot(b,select=i,rug=FALSE,col="green",
    col.axis="white",col.lab="white",all.terms=TRUE)
  for (j in 1:2) axis(j,col="white",labels=FALSE)
  box(col="white")
  eval(parse(text=paste("fx <- f",i,"(x)",sep="")))
  fx <- fx-mean(fx)
```

```
  lines(x,fx,col=2) ## overlay `truth' in red
}
par(op)

## example with 2-d plots...
b1<-gam(y~x0+s(x1,x2)+s(x3))
op<-par(mfrow=c(2,2))
plot(b1,all.terms=TRUE)
par(op)
```

`predict.gam` *Prediction from fitted GAM model*

Description

Takes a fitted `gam` object produced by `gam()` and produces predictions given a new set of values for the model covariates or the original values used for the model fit. Predictions can be accompanied by standard errors, based on the posterior distribution of the model coefficients. The routine can optionally return the matrix by which the model coefficients must be pre-multiplied in order to yield the values of the linear predictor at the supplied covariate values: this is useful for obtaining credible regions for quantities derived from the model, and for lookup table prediction outside `R` (see example code below).

Usage

```
## S3 method for class 'gam':
predict(object,newdata,type="link",se.fit=FALSE,terms=NULL,
        block.size=1000,newdata.guaranteed=FALSE,na.action=na.pass,...)
```

Arguments

`object`	a fitted `gam` object as produced by `gam()`.
`newdata`	A data frame containing the values of the model covariates at which predictions are required. If this is not provided then predictions corresponding to the original data are returned. If `newdata` is provided then it should contain all the variables needed for prediction: a warning is generated if not.
`type`	When this has the value `"link"` (default) the linear predictor (possibly with associated standard errors) is returned. When `type="terms"` each component of the linear predictor is returned seperately (possibly with standard errors): this includes parametric model components, followed by each smooth component, but excludes any offset and any intercept. When `type="response"` predictions on the scale of the response are returned (possibly with approximate standard errors). When `type="lpmatrix"` then a matrix is returned which yields the values of the linear predictor (minus any offset) when postmultiplied by the parameter vector (in this case `se.fit` is ignored). The latter option is most useful for getting variance estimates for quantities derived from the model: for example integrated quantities, or derivatives of smooths. A linear predictor matrix can also be used to implement approximate prediction outside `R` (see example code, below).
`se.fit`	when this is TRUE (not default) standard error estimates are returned for each prediction.

terms — if type=="terms" then only results for the terms given in this array will be returned.

block.size — maximum number of predictions to process per call to underlying code: larger is quicker, but more memory intensive. Set to < 1 to use total number of predictions as this.

newdata.guaranteed — Set to TRUE to turn off all checking of newdata except for sanity of factor levels: this can speed things up for large prediction tasks, but newdata must be complete, with no NA values for predictors required in the model.

na.action — what to do about NA values in newdata. With the default na.pass, any row of newdata containing NA values for required predictors, gives rise to NA predictions (even if the term concerned has no NA predictors). na.exclude or na.omit result in the dropping of newdata rows, if they contain any NA values for required predictors. If newdata is missing then NA handling is determined from object$na.action.

... — other arguments.

Details

The standard errors produced by predict.gam are based on the Bayesian posterior covariance matrix of the parameters Vp in the fitted gam object.

To facilitate plotting with termplot, if object possesses an attribute "para.only" and type=="terms" then only parametric terms of order 1 are returned (i.e. those that termplot can handle).

Note that, in common with other prediction functions, any offset supplied to gam as an argument is always ignored when predicting, unlike offsets specified in the gam model formula.

See the examples for how to use the lpmatrix for obtaining credible regions for quantities derived from the model.

Value

If type=="lpmatrix" then a matrix is returned which will give a vector of linear predictor values (minus any offest) at the supplied covariate values, when applied to the model coefficient vector. Otherwise, if se.fit is TRUE then a 2 item list is returned with items (both arrays) fit and se.fit containing predictions and associated standard error estimates, otherwise an array of predictions is returned. The dimensions of the returned arrays depends on whether type is "terms" or not: if it is then the array is 2 dimensional with each term in the linear predictor separate, otherwise the array is 1 dimensional and contains the linear predictor/predicted values (or corresponding s.e.s). The linear predictor returned termwise will not include the offset or the intercept.

newdata can be a data frame, list or model.frame: if it's a model frame then all variables must be supplied.

WARNING

Note that the behaviour of this function is not identical to predict.gam() in Splus.

type=="terms" does not exactly match what predict.lm does for parametric model components.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

The design is inspired by the S function of the same name described in Chambers and Hastie (1993) (but is not a clone).

References

Chambers and Hastie (1993) Statistical Models in S. Chapman & Hall.

Gu and Wahba (1991) Minimizing GCV/GML scores with multiple smoothing parameters via the Newton method. SIAM J. Sci. Statist. Comput. 12:383-398

Wood, S.N. (2000) Modelling and Smoothing Parameter Estimation with Multiple Quadratic Penalties. J.R.Statist.Soc.B 62(2):413-428

Wood, S.N. (2003) Thin plate regression splines. J.R.Statist.Soc.B 65(1):95-114

Wood, S.N. (2004) Stable and efficient multiple smoothing parameter estimation for generalized additive models. J. Amer. Statist. Ass. 99:637-686

http://www.maths.bath.ac.uk/~sw283/

See Also

gam, gamm, plot.gam

Examples

```
library(mgcv)
n<-200
sig <- 2
x0 <- runif(n, 0, 1)
x1 <- runif(n, 0, 1)
x2 <- runif(n, 0, 1)
x3 <- runif(n, 0, 1)
y <- 2 * sin(pi * x0)
y <- y + exp(2 * x1)
y <- y + 0.2 * x2^11 * (10 * (1 - x2))^6 + 10 * (10 * x2)^3 * (1 - x2)^10
y <- y + x3
e <- rnorm(n, 0, sig)
y <- y + e
b<-gam(y~s(x0)+s(I(x1^2))+s(x2)+offset(x3))
rm(y,x0,x1,x2,x3)
newd <- data.frame(x0=(0:30)/30,x1=(0:30)/30,x2=(0:30)/30,x3=(0:30)/30)
pred <- predict.gam(b,newd)

## now get variance of sum of predictions using lpmatrix

Xp <- predict(b,newd,type="lpmatrix")

## Xp %*% coef(b) yields vector of predictions

a <- rep(1,31)
Xs <- t(a) %*% Xp ## Xs %*% coef(b) gives sum of predictions
var.sum <- Xs %*% b$Vp %*% t(Xs)

## Now get the variance of non-linear function of predictions
## by simulation from posterior distribution of the params
```

```
library(MASS)
br<-mvrnorm(1000,coef(b),b$Vp) ## 1000 replicate param. vectors
res <- rep(0,1000)
for (i in 1:1000)
{ pr <- Xp %*% br[i,] ## replicate predictions
  res[i] <- sum(log(abs(pr))) ## example non-linear function
}
mean(res);var(res)

## loop is replace-able by following ....

res <- colSums(log(abs(Xp %*% t(br))))

## The following shows how to use use an "lpmatrix" as a lookup
## table for approximate prediction. The idea is to create
## approximate prediction matrix rows by appropriate linear
## interpolation of an existing prediction matrix. The additivity
## of a GAM makes this possible.
## There is no reason to ever do this in R, but the following
## code provides a useful template for predicting from a fitted
## gam *outside* R: all that is needed is the coefficient vector
## and the prediction matrix. Use larger `Xp'/ smaller `dx' and/or
## higher order interpolation for higher accuracy.

xn <- c(.341,.122,.476,.981) ## want prediction at these values
x0 <- 1          ## intercept column
dx <- 1/30       ## covariate spacing in `newd'
for (j in 0:2) { ## loop through smooth terms
  cols <- 1+j*9 +1:9      ## relevant cols of Xp
  i <- floor(xn[j+1]*30)  ## find relevant rows of Xp
  w1 <- (xn[j+1]-i*dx)/dx ## interpolation weights
  ## find approx. predict matrix row portion, by interpolation
  x0 <- c(x0,Xp[i+2,cols]*w1 + Xp[i+1,cols]*(1-w1))
}
dim(x0)<-c(1,28)
fv <- x0%*%coef(b) + xn[4];fv    ## evaluate and add offset
se <- sqrt(x0%*%b$Vp%*%t(x0));se ## get standard error
## compare to normal prediction
predict(b,newdata=data.frame(x0=xn[1],x1=xn[2],
        x2=xn[3],x3=xn[4]),se=TRUE)
```

`Predict.matrix` *Prediction methods for smooth terms in a GAM*

Description

Takes `smooth` objects produced by `smooth.construct` methods and obtains the matrix mapping the parameters associated with such a smooth to the predicted values of the smooth at a set of new covariate values.

In practice this method is often called via the wrapper function `PredictMat`.

Usage

```
Predict.matrix(object,data)
```

Arguments

`object`	is a smooth object produced by a `smooth.construct` method function. The object contains all the information required to specify the basis for a term of its class, and this information is used by the appropriate `Predict.matrix` function to produce a prediction matrix for new covariate values. Further details are given in `smooth.construct`.
`data`	A data frame containing the values of the (named) covariates at which the smooth term is to be evaluated.

Details

Smooth terms in a GAM formula are turned into smooth specification objects of class `xx.smooth.spec` during processing of the formula. Each of these objects is converted to a smooth object using an appropriate `smooth.construct` function. The `Predict.matrix` functions are used to obtain the matrix that will map the parameters associated with a smooth term to the predicted values for the term at new covariate values.

Note that new smooth classes can be added by writing a new `smooth.construct` method function and a corresponding `Predict.matrix` method function: see the example code provided for `smooth.construct` for details.

Value

A matrix which will map the parameters associated with the smooth to the vector of values of the smooth evaluated at the covariate values given in `object`.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

Wood, S.N. (2000) Modelling and Smoothing Parameter Estimation with Multiple Quadratic Penalties. J.R.Statist.Soc.B 62(2):413-428

Wood, S.N. (2003) Thin plate regression splines. J.R.Statist.Soc.B 65(1):95-114

Wood, S.N. (2004) Stable and efficient multiple smoothing parameter estimation for generalized additive models. J. Amer. Statist. Ass. 99:637-686

http://www.maths.bath.ac.uk/~sw283/

See Also

`gam`,`gamm`, `smooth.construct`, `PredictMat`

Examples

```
# See smooth.construct examples
```

print.gam *Generalized Additive Model default print statement*

Description

This is the default print statement for a GAM object. If you need a list of everything that is part of a gam object see gam, or use `names()`. The family (including link), model formula, and estimated degrees of freedom for each model term (plus total) are printed, as well as the minimized GCV or UBRE score, depending on which was used.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

Wood, S.N. (2006) Generalized Additive Models: An Introduction with R. CRC/ Chapmand and Hall, Boca Raton, Florida.

http://www.maths.bath.ac.uk/~sw283/

See Also

gam,mgcv

residuals.gam *Generalized Additive Model residuals*

Description

Returns residuals for a fitted `gam` model object. Pearson, deviance, working and response residuals are available.

Usage

```
## S3 method for class 'gam':
residuals(object, type = c("deviance", "pearson","scaled.pearson", "working", "r
```

Arguments

`object`	a `gam` fitted model object.
`type`	the type of residuals wanted.
`...`	other arguments.

Details

Response residuals are the raw residuals (data minus fitted values). Scaled Pearson residuals are raw residuals divided by the standard deviation of the data according to the model mean variance relationship and estimated scale parameter. Pearson residuals are the same, but multiplied by the square root of the scale parameter (so they are independent of the scale parameter): $((y-\mu)/\sqrt{V(\mu)}$, where y is data μ is model fitted value and V is model mean-variance relationship.). Both are provided since not all texts agree on the definition of Pearson residuals. Deviance residuals simply return the deviance residuals defined by the model family. Working residuals are the residuals returned from model fitting at convergence.

There is a special function for `gam` objects because of a bug in the calculation of Pearson residuals in some earlier versions of `residual.glm`.

Value

An array of residuals.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

See Also

`gam`

---

s *Defining smooths in GAM formulae*

Description

Function used in definition of smooth terms within `gam` model formulae. The function does not evaluate a (spline) smooth - it exists purely to help set up a model using spline based smooths.

Usage

```
s(..., k=-1,fx=FALSE,bs="tp",m=0,by=NA)
```

Arguments

`...`	a list of variables that are the covariates that this smooth is a function of.
`k`	the dimension of the basis used to represent the smooth term. The default depends on the number of variables that the smooth is a function of. `k` should not be less than the dimension of the null space of the penalty for the term (see `null.space.dimension`), but will be reset if it is. See `choose.k` for further information.
`fx`	indicates whether the term is a fixed d.f. regression spline (`TRUE`) or a penalized regression spline (`FALSE`).

bs — this can be "cr" for a cubic regression spline, "cs" for a cubic regression spline with shrinkage, "cc" for a cyclic (periodic) spline, "tp" for a thin plate regression spline, "ts" for a thin plate regression spline with shrinkage or a user defined character string for other user defined smooth classes. Of the built in alternatives, only thin plate regression splines can be used for multidimensional smooths. Note that the "cr" and "cc" bases are faster to set up than the "tp" basis, particularly on large data sets (although the knots argument to gam can be used to get round this).

m — The order of the penalty for this t.p.r.s. term (e.g. 2 for normal cubic spline penalty with 2nd derivatives). O signals autoinitialization, which sets the order to the lowest value satisfying 2m>d+1, where d is the number of covariates: this choise ensures visual smoothness. In addition, m must satisfy the technical restriction 2m>d, otherwise it will be autoinitialized.

by — specifies a covariate by which the whole smooth term is to be multiplied. This is particularly useful for creating models in which a smooth interacts with a factor: in this case the by variable would usually be the dummy variable coding one level of the factor. See the examples below. This is the means by which 'variable coefficient models' can be specified.

Details

The function does not evaluate the variable arguments. To use this function to specify use of your own smooths, note the relationships between the inputs and the output object and see the example in smooth.construct.

Value

A class xx.smooth.spec object, where xx is a basis identifying code given by the bs argument of s. These smooth.spec objects define smooths and are turned into bases and penalties by smooth.construct method functions.

The returned object contains the following items:

term — An array of text strings giving the names of the covariates that the term is a function of.

bs.dim — The dimension of the basis used to represent the smooth.

fixed — TRUE if the term is to be treated as a pure regression spline (with fixed degrees of freedom); FALSE if it is to be treated as a penalized regression spline

dim — The dimension of the smoother - i.e. the number of covariates that it is a function of.

p.order — The order of the t.p.r.s. penalty, or 0 for auto-selection of the penalty order.

by — is the name of any by variable as text ("NA" for none).

full.call — Text for pasting into a string to be converted to a gam formula, which has the values of function options given explicitly - this is useful for constructing a fully expanded gam formula which can be used without needing access to any variables that may have been used to define k, fx, bs or m in the original call. i.e. this is text which when parsed and evaluated generates a call to s() with all the options spelled out explicitly.

label — A suitable text label for this smooth term.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

Wood, S.N. (2000) Modelling and Smoothing Parameter Estimation with Multiple Quadratic Penalties. J.R.Statist.Soc.B 62(2):413-428

Wood, S.N. (2003) Thin plate regression splines. J.R.Statist.Soc.B 65(1):95-114

http://www.maths.bath.ac.uk/~sw283/

See Also

te, gam, gamm

Examples

```
# example utilising `by' variables
library(mgcv)
set.seed(0)
n<-200;sig2<-4
x1 <- runif(n, 0, 1);x2 <- runif(n, 0, 1);x3 <- runif(n, 0, 1)
fac<-c(rep(1,n/2),rep(2,n/2)) # create factor
fac.1<-rep(0,n)+(fac==1);fac.2<-1-fac.1 # and dummy variables
fac<-as.factor(fac)
f1 <-  exp(2 * x1) - 3.75887
f2 <-  0.2 * x1^11 * (10 * (1 - x1))^6 + 10 * (10 * x1)^3 * (1 - x1)^10
f<-f1*fac.1+f2*fac.2+x2
e <- rnorm(n, 0, sqrt(abs(sig2)))
y <- f + e
# NOTE: smooths will be centered, so need to include fac in model....
b<-gam(y~fac+s(x1,by=fac.1)+s(x1,by=fac.2)+x2)
plot(b,pages=1)
```

---

smooth.construct *Constructor functions for smooth terms in a GAM*

Description

Smooth terms in a GAM formula are turned into smooth specification objects of class `xx.smooth.spec` during processing of the formula. Each of these objects is converted to a smooth object using an appropriate `smooth.construct` function. New smooth classes can be added by writing a new `smooth.construct` method function and a corresponding `Predict.matrix` method function (see example code below).

In practice, `smooth.construct` is usually called via the wrapper function `smoothCon`.

Usage

```
smooth.construct(object,data,knots)
```

Arguments

`object` is a smooth specification object, generated by an `s` or `te` term in a GAM formula. Objects generated by `s` terms have class `xx.smooth.spec` where `xx` is given by the `bs` argument of `s` (this convention allows the user to add their own smoothers). If `object` is not class `tensor.smooth.spec` it will have the following elements:

- `term` The names of the covariates for this smooth, in an array.
- `bs.dim` Argument `k` of the `s` term generating the object. This is the dimension of the basis used to represent the term (or, arguably, 1 greater than the basis dimension for `cc` terms).
- `fixed` `TRUE` if the term is to be unpenalized, otherwise `FALSE`.
- `dim` the number covariates of which this smooth is a function.
- `p.order` the order of the smoothness penalty or 0 for autoselection of this. This is argument `m` of the `s` term that generated `object`.
- `by` the name of any `by` variable to multiply this term as supplied as an argument to `s`. `"NA"` if there is no such term.
- `full.call` The full version of the `s` term, with all defaults expanded explicitly.
- `label` A suitable label for use with this term.
- `null.space.dim` The dimension of the null space of the wiggliness penalty.

If `object` is of class `tensor.smooth.spec` then it was generated by a `te` term in the GAM formula, and specifies a smooth of several variables with a basis generated as a tensor product of lower dimensional bases. In this case the object will be different and will have the following elements:

- `margin` is a list of smooth specification objects of the type listed above, defining the bases which have their tensor product formed in order to construct this term.
- `term` is the array of names of the covariates that are arguments of the smooth.
- `by` is the name of any `by` variable, or `"NA"`.
- `fx` is an array, the elements of which indicate whether (`TRUE`) any of the margins in the tensor product should be unpenalized.
- `full.call` The full version of the `s` term, with all defaults expanded explicitly.
- `label` A suitable label for use with this term.
- `dim` is the number of covariates of which this smooth is a function.
- `mp` `TRUE` if multiple penalties are to be used.
- `np` `TRUE` if 1-D marginal smooths are to be re-parameterized in terms of function values.

`data` a data frame in which the covariates and any `by` variable can be found.

`knots` an optional data frame specifying knot locations for each covariate. If it is null then the knot locations are generated automatically.

Details

The returned objects for the built in smooth classes have the following extra elements. `cr.smooth` objects (generated using `bs="cr"`) have an additional array `xp` giving the knot locations used to generate the basis.

`cyclic.smooth` objects (generated using `bs="cc"`) have an array `xp` of knot locations and a matrix `BD` used to define the basis (BD transforms function values at the knots to second derivatives at the knots).

`tprs.smooth` objects require several items to be stored in order to define the basis. These are:

shift A record of the shift applied to each covariate in order to center it around zero and avoid any co-linearity problems that might otehrwise occur in the penalty null space basis of the term.

Xu A matrix of the unique covariate combinations for this smooth (the basis is constructed by first stripping out duplicate locations).

UZ The matrix mapping the t.p.r.s. parameters back to the parameters of a full thin plate spline.

null.space.dimension The dimension of the space of functions that have zero wiggliness according to the wiggliness penalty for this term.

Again, these extra elements would be found in the elements of the `margin` list of `tensor.smooth` class object.

The classes `cs.smooth` and `ts.smooth` are modifications of the classes `cr.smooth` and `tprs.smooth` respsectively: they differ only in having an extra shrinkage component added to their penalties, so that automatic model selection can effectively remove such terms from a model altogether.

Value

The input argument `object`, assigned a new class to indicate what type of smooth it is and with at least the following items added:

`X`	The model matrix from this term.
`C`	The matrix defining any constraints on the term - usually a one row matrix giving the column sums of the model matrix, which defines the constraint that each term should sum to zero over the covariate values.
`S`	A list of positive semi-definite penalty matrices that apply to this term. The list will be empty if the term is to be left un-penalized.
`rank`	an array giving the ranks of the penalties.
`df`	the degrees of freedom associated with this term (at least when unpenalized).

Usually the returned object will also include extra information required to define the basis, and used by `Predict.matrix` methods to make predictions using the basis. See the `Details` section for the information included for the built in smooth classes.

`tensor.smooth` returned objects will additionally have each element of the `margin` list updated in the same way. `tensor.smooths` also have a list, `XP`, containing re-parameterization matrices for any 1-D marginal terms re-parameterized in terms of function values. This list will have `NULL` entries for marginal smooths that are not re-parameterized, and is only long enough to reach the last re-parameterized marginal in the list.

WARNING

User defined smooth objects should avoid having attributes names `"qrc"` or `"nCons"` as these are used internally to provide constraint free parameterizations.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

Wood, S.N. (2000) Modelling and Smoothing Parameter Estimation with Multiple Quadratic Penalties. J.R.Statist.Soc.B 62(2):413-428

Wood, S.N. (2003) Thin plate regression splines. J.R.Statist.Soc.B 65(1):95-114

Wood, S.N. (2004a) Stable and efficient multiple smoothing parameter estimation for generalized additive models. J. Amer. Statist. Ass. 99:673-686

The p-spline code given in the example is based on:

Eilers, P.H.C. and B.D. Marx (1996) Flexible Smoothing with B-splines and Penalties. Statistical Science, 11(2):89-121

http://www.maths.bath.ac.uk/~sw283/

See Also

get.var, gamm, gam, Predict.matrix, smoothCon, PredictMat

Examples

```
# adding "p-spline" classes and methods

smooth.construct.ps.smooth.spec<-function(object,data,knots)
# a p-spline constructor method function
{ require(splines)
  if (length(object$p.order)==1) m<-rep(object$p.order,2)
  else m<-object$p.order  # m[1] - basis order, m[2] - penalty order
  nk<-object$bs.dim-m[1]  # number of interior knots
  if (nk<=0) stop("basis dimension too small for b-spline order")
  x <- get.var(object$term,data)  # find the data
  xl<-min(x);xu<-max(x);xr<-xu-xl # data limits and range
  xl<-xl-xr*0.001;xu<-xu+xr*0.001;dx<-(xu-xl)/(nk-1)
  if (!is.null(knots)) k <- get.var(object$term,knots)
  else k<-NULL
  if (is.null(k))
  k<-seq(min(x)-dx*(m[1]+1),max(x)+dx*(m[1]+1),length=nk+2*m[1]+2)
  if (length(k)!=nk+2*m[1]+2)
  stop(paste("there should be ",nk+2*m[1]+2," supplied knots"))
  object$X<-spline.des(k,x,m[1]+2,x*0)$design # get model matrix
  if (!object$fixed)
  { S<-diag(object$bs.dim);if (m[2]) for (i in 1:m[2]) S<-diff(S)
    object$S<-list(t(S)%*%S)  # get penalty
    object$S[[1]] <- (object$S[[1]]+t(object$S[[1]]))/2 # exact symmetry
  }
  object$rank<-object$bs.dim-m[2]  # penalty rank
  object$null.space.dim <- m[2]  # dimension of unpenalized space
  object$knots<-k;object$m<-m      # store p-spline specific info.
  object$C<-matrix(colSums(object$X),1,object$bs.dim) #constraint
  object$df<-ncol(object$X)-1      # maximum DoF
  if (object$by!="NA")  # deal with "by" variable
  { by <- get.var(object$by,data) # find by variable
    if (is.null(by)) stop("Can't find by variable")
    object$X<-as.numeric(by)*object$X # form diag(by)%*%X
  }
  class(object)<-"pspline.smooth"  # Give object a class
  object
}
```

```
Predict.matrix.pspline.smooth<-function(object,data)
# prediction method function for the p.spline smooth class
{ require(splines)
  x <- get.var(object$term,data)
  X <- spline.des(object$knots,x,object$m[1]+2,x*0)$design
  if (object$by != "NA") { ## handle any by variables
        by <- get.var(object$by, data)
        if (is.null(by))
            stop("Can't find by variable")
        X <- as.numeric(by) * X
  }
  X
}

# an example, using the new class....
set.seed(0);n<-400;
x0 <- runif(n, 0, 1);x1 <- runif(n, 0, 1)
x2 <- runif(n, 0, 1);x3 <- runif(n, 0, 1)
f <- 2 * sin(pi * x0)
f <- f + exp(2 * x1) - 3.75887
f <- f+0.2*x2^11*(10*(1-x2))^6+10*(10*x2)^3*(1-x2)^10-1.396
e <- rnorm(n)*2
y <- f + e
b<-gam(y~s(x0,bs="ps",m=2)+s(x1,bs="ps",m=c(1,3))+
         s(x2,bs="ps",m=2)+s(x3,bs="ps",m=2))
plot(b,pages=1)
# another example using tensor products of the new class

test1<-function(x,z,sx=0.3,sz=0.4)
{ (pi**sx*sz)*(1.2*exp(-(x-0.2)^2/sx^2-(z-0.3)^2/sz^2)+
  0.8*exp(-(x-0.7)^2/sx^2-(z-0.8)^2/sz^2))
}
n<-400
x<-runif(n);z<-runif(n);
f <- test1(x,z)
y <- f + rnorm(n)*0.1
b <- gam(y~te(x,z,bs=c("ps","ps"),m=c(2,2)))
vis.gam(b)
```

`smoothCon` *Prediction/Construction wrapper functions for GAM smooth terms*

Description

Wrapper functions for construction of and prediction from smooth terms in a GAM. The purpose of the wrappers is to allow user-transparant re-parameterization of smooth terms, in order to allow identifiability constraints to be absorbed into the parameterization of each term, if required.

Usage

```
smoothCon(object,data,knots,absorb.cons=FALSE,scale.penalty=TRUE)
PredictMat(object,data)
```

Arguments

object	is a smooth specification object or a smooth object.
data	A data frame containing the values of the (named) covariates at which the smooth term is to be evaluated.
knots	An optional data frame supplying any knot locations to be supplied for basis construction.
absorb.cons	Set to TRUE in order to have identifiability constraints absorbed into the basis.
scale.penalty	should the penalty coefficient matrix be scaled to have approximately the same 'size' as the inner product of the terms model matrix with itself? This can improve the performance of gamm fitting.

Details

These wrapper functions exist to allow smooths specified using smooth.construct and Predict.matrix method functions to be re-parameterized so that identifiability constraints are no longer required in fitting. This is done in a user transparent manner, but is typically of no importance in use of GAMs.

The parameterization used by gam can be controlled via gam.control.

Value

From smoothCon a smooth object returned by the appropriate smooth.construct method function. If constraints are to be absorbed then the object will have attributes "qrc" and "nCons", the qr decomposition of the constraint matrix (returned by qr) and the number of constraints, respectively: these are used in the re-parameterization.

For predictMat a matrix which will map the parameters associated with the smooth to the vector of values of the smooth evaluated at the covariate values given in object.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

http://www.maths.bath.ac.uk/~sw283/

See Also

gam.control, smooth.construct, Predict.matrix

step.gam *Alternatives to step.gam*

Description

There is no `step.gam` in package `mgcv`. The `mgcv` default for model selection is to use MSE/KL-distance criteria such as GCV or UBRE/AIC. Since the smoothness estimation part of model selection is done in this way it is logically most consistent to perform model selection on the basis of such criteria: i.e. to decide which terms to include or omit by looking at changes in GCV/UBRE score.

To facilitate fully automatic model selection the package includes 2 classes of smoothers (`"cs"` and `"ts"`: see `s`) which can be penalized to zero for sufficiently high smoothing parameter estimates: use of such smooths provides an effective alternative to step-wise model selection. The example below shows an example of the application of this approach, where selection is a fully integrated part of model estimation.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

Examples

```
## an example of GCV based model selection as
## an alternative to stepwise selection
library(mgcv)
set.seed(0)
n<-400;sig<-2
x0 <- runif(n, 0, 1);x1 <- runif(n, 0, 1)
x2 <- runif(n, 0, 1);x3 <- runif(n, 0, 1)
x4 <- runif(n, 0, 1);x5 <- runif(n, 0, 1)
f <- 2 * sin(pi * x0)
f <- f + exp(2 * x1) - 3.75887
f <- f+0.2*x2^11*(10*(1-x2))^6+10*(10*x2)^3*(1-x2)^10-1.396
e <- rnorm(n, 0, sig)
y <- f + e
## Note the increased gamma parameter below to favour
## slightly smoother models...
b<-gam(y~s(x0,bs="ts")+s(x1,bs="ts")+s(x2,bs="ts")+
   s(x3,bs="ts")+s(x4,bs="ts")+s(x5,bs="ts"),gamma=1.4)
summary(b)
plot(b,pages=1)
```

summary.gam *Summary for a GAM fit*

Description

Takes a fitted `gam` object produced by `gam()` and produces various useful summaries from it.

Usage

```
## S3 method for class 'gam':
summary(object, dispersion=NULL, freq=TRUE, ...)

## S3 method for class 'summary.gam':
print(x,digits = max(3, getOption("digits") - 3),
                  signif.stars = getOption("show.signif.stars"),...)
```

Arguments

object	a fitted gam object as produced by gam().
x	a summary.gam object produced by summary.gam().
dispersion	A known dispersion parameter. NULL to use estimate or default (e.g. 1 for Poisson).
freq	By default p-values for individual terms are calculated using the frequentist estimated covariance matrix of the parameter estimators. If this is set to FALSE then the Bayesian covariance matrix of the parameters is used instead. See details.
digits	controls number of digits printed in output.
signif.stars	Should significance stars be printed alongside output.
...	other arguments.

Details

Model degrees of freedom are taken as the trace of the influence (or hat) matrix $\mathbf{A}$ for the model fit. Residual degrees of freedom are taken as number of data minus model degrees of freedom. Let $\mathbf{P}_i$ be the matrix giving the parameters of the ith smooth when applied to the data (or pseudodata in the generalized case) and let $\mathbf{X}$ be the design matrix of the model. Then $tr(\mathbf{X}\mathbf{P}_i)$ is the edf for the ith term. Clearly this definition causes the edf's to add up properly!

print.summary.gam tries to print various bits of summary information useful for term selection in a pretty way.

If freq=FALSE then the Bayesian parameter covariance matrix, object$Vp, is used to calculate test statistics for terms, and the degrees of freedom for reference distributions is taken as the estimated degrees of freedom for the term concerned. This is not easy to justify theoretically, and the resulting 'Bayesian p-values' are difficult to interpret and often have much worse frequentist performance than the default p-values.

Value

summary.gam produces a list of summary information for a fitted gam object.

p.coeff	is an array of estimates of the strictly parametric model coefficients.
p.t	is an array of the p.coeff's divided by their standard errors.
p.pv	is an array of p-values for the null hypothesis that the corresponding parameter is zero. Calculated with reference to the t distribution with the estimated residual degrees of freedom for the model fit if the dispersion parameter has been estimated, and the standard normal if not.
m	The number of smooth terms in the model.

`chi.sq` An array of test statistics for assessing the significance of model smooth terms. If $\mathbf{p}_i$ is the parameter vector for the ith smooth term, and this term has estimated covariance matrix $\mathbf{V}_i$ then the statistic is $\mathbf{p}_i'\mathbf{V}_i^{k-}\mathbf{p}_i$, where $\mathbf{V}_i^{k-}$ is the rank k pseudo-inverse of $\mathbf{V_i}$, and k is estimated rank of $\mathbf{V_i}$.

`s.pv` An array of approximate p-values for the null hypotheses that each smooth term is zero. Be warned, these are only approximate. In the case of known dispersion parameter, they are obtained by comparing the chi.sq statistic given above to the chi-squared distribution with k degrees of freedom, where k is the estimated rank of $\mathbf{V_i}$. If the dispersion parameter is unknown (in which case it will have been estimated) the statistic is compared to an F distribution with k upper d.f. and lower d.f. given by the residual degrees of freedom for the model . Typically the p-values will be somewhat too low, because they are conditional on the smoothing parameters, which are usually uncertain, but note that the statistic can also have low power if the rank, k, is too high relative to the EDF of the term!

`se` array of standard error estimates for all parameter estimates.

`r.sq` The adjusted r-squared for the model. Defined as the proportion of variance explained, where original variance and residual variance are both estimated using unbiased estimators. This quantity can be negative if your model is worse than a one parameter constant model, and can be higher for the smaller of two nested models! Note that proportion null deviance explained is probably more appropriate for non-normal errors.

`dev.expl` The proportion of the null deviance explained by the model.

`edf` array of estimated degrees of freedom for the model terms.

`residual.df` estimated residual degrees of freedom.

`n` number of data.

`gcv` minimized GCV score for the model, if GCV used.

`ubre` minimized UBRE score for the model, if UBRE used.

`scale` estimated (or given) scale parameter.

`family` the family used.

`formula` the original GAM formula.

`dispersion` the scale parameter.

`pTerms.df` the degrees of freedom associated with each parameteric term (excluding the constant).

`pTerms.chi.sq` a Wald statistic for testing the null hypothesis that the each parametric term is zero.

`pTerms.pv` p-values associated with the tests that each term is zero. For penalized fits these are approximate, being conditional on the smoothing parameters. The reference distribution is an appropriate chi-squared when the scale parameter is known, and is based on an F when it is not.

`cov.unscaled` The estimated covariance matrix of the parameters (or estimators if `freq=TRUE`), divided by scale parameter.

`cov.scaled` The estimated covariance matrix of the parameters (estimators if `freq=TRUE`).

`p.table` significance table for parameters

`s.table` significance table for smooths

`p.Terms` significance table for parametric model terms

WARNING

The supplied p-values will often be underestimates if smoothing parameters have been estimated as part of model fitting.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩ with substantial improvements by Henric Nilsson.

References

Gu and Wahba (1991) Minimizing GCV/GML scores with multiple smoothing parameters via the Newton method. SIAM J. Sci. Statist. Comput. 12:383-398

Wood, S.N. (2000) Modelling and Smoothing Parameter Estimation with Multiple Quadratic Penalties. J.R.Statist.Soc.B 62(2):413-428

Wood, S.N. (2003) Thin plate regression splines. J.R.Statist.Soc.B 65(1):95-114

Wood, S.N. (2004a) Stable and efficient multiple smoothing parameter estimation for generalized additive models. J. Amer. Statist. Ass. 99:673-686

Wood, S.N. (2004b) On confidence intervals for GAMs based on penalized regression splines. Technical Report 04-12 Department of Statistics, University of Glasgow.

http://www.maths.bath.ac.uk/~sw283/

See Also

gam, predict.gam, gam.check, anova.gam

Examples

```
library(mgcv)
set.seed(0)
n<-200
sig2<-4
x0 <- runif(n, 0, 1)
x1 <- runif(n, 0, 1)
x2 <- runif(n, 0, 1)
x3 <- runif(n, 0, 1)
pi <- asin(1) * 2
y <- 2 * sin(pi * x0)
y <- y + exp(2 * x1) - 3.75887
y <- y + 0.2 * x2^11 * (10 * (1 - x2))^6 + 10 * (10 * x2)^3 * (1 - x2)^10
e <- rnorm(n, 0, sqrt(abs(sig2)))
y <- y + e
b<-gam(y~s(x0)+s(x1)+s(x2)+s(x3))
plot(b,pages=1)
summary(b)
# now check the p-values by using a pure regression spline.....
b.d<-round(summary(b)$edf)+1 ## get edf per smooth
b.d<-pmax(b.d,3) # can't have basis dimension less than 3!
bc<-gam(y~s(x0,k=b.d[1],fx=TRUE)+s(x1,k=b.d[2],fx=TRUE)+
        s(x2,k=b.d[3],fx=TRUE)+s(x3,k=b.d[4],fx=TRUE))
plot(bc,pages=1)
summary(bc)
# p-value check - increase k to make this useful!
n<-200;p<-0;k<-20
```

```
for (i in 1:k)
{ b<-gam(y~s(x,z),data=data.frame(y=rnorm(n),x=runif(n),z=runif(n)))
  p[i]<-summary(b)$s.p[1]
}
plot(((1:k)-0.5)/k,sort(p))
abline(0,1,col=2)
```

`te` *Define tensor product smooths in GAM formulae*

Description

Function used in definition of tensor product smooth terms within `gam` model formulae. The function does not evaluate a smooth - it exists purely to help set up a model using tensor product based smooths.

Usage

```
te(..., k=NA,bs="cr",m=0,d=NA,by=NA,fx=FALSE,mp=TRUE,np=TRUE)
```

Arguments

`...`	a list of variables that are the covariates that this smooth is a function of.
`k`	the dimension(s) of the bases used to represent the smooth term. If not supplied then set to `5^d`. If supplied as a single number then this basis dimension is used for each basis. If supplied as an array then the elements are the dimensions of the component (marginal) bases of the tensor product. See `choose.k` for further information.
`bs`	array (or single character string) specifying the type for each marginal basis. `"cr"` for cubic regression spline; `"cs"` for cubic regression spline with shrinkage; `"cc"` for periodic/cyclic cubic regression spline; `"tp"` for thin plate regression spline; `"ts"` for t.p.r.s. with extra shrinkage. User defined bases can also be used here (see `smooth.construct` for an example). If only one identifier is given then this is used for all bases.
`m`	The order of the penalty for each t.p.r.s. term (e.g. 2 for normal cubic spline penalty with 2nd derivatives). If a single number is given then it is used for all terms. `0` autoinitializes. `m` is ignored for the `"cr"` and `"cc"` bases.
`d`	array of marginal basis dimensions. For example if you want a smooth for 3 covariates made up of a tensor product of a 2 dimensional t.p.r.s. basis and a 1-dimensional basis, then set `d=c(2,1)`.
`by`	specifies a covariate by which the whole smooth term is to be multiplied. This is particularly useful for creating models in which a smooth interacts with a factor: in this case the `by` variable would usually be the dummy variable coding one level of the factor. See the examples below.
`fx`	indicates whether the term is a fixed d.f. regression spline (`TRUE`) or a penalized regression spline (`FALSE`).
`mp`	`TRUE` to use multiple penalties for the smooth. `FALSE` to use only a single penalty: single penalties are not recommended - they tend to allow only rather wiggly models.

np — TRUE to use the 'normal parameterization' for a tensor product smooth. This parameterization represents any 1-d marginal smooths using a parameterization where the parameters are function values at 'knots' spread evenly through the data. The parameterization makes the penalties easily interpretable, however it can reduce numerical stability.

Details

Smooths of several covariates can be constructed from tensor products of the bases used to represent smooths of one (or sometimes more) of the covariates. To do this 'marginal' bases are produced with associated model matrices and penalty matrices, and these are then combined in the manner described in tensor.prod.model.matrix and tensor.prod.penalties, to produce a single model matrix for the smooth, but multiple penalties (one for each marginal basis). The basis dimension of the whole smooth is the product of the basis dimensions of the marginal smooths.

An option for operating with a single penalty (The Kronecker product of the marginal penalties) is provided, but it is rarely of practical use: the penalty is typically so rank deficient that even the smoothest resulting model will have rather high estimated degrees of freedom.

Tensor product smooths are especially useful for representing functions of covariates measured in different units, although they are typically not quite as nicely behaved as t.p.r.s. smooths for well scaled covariates.

Note also that GAMs constructed from lower rank tensor product smooths are nested within GAMs constructed from higher rank tensor product smooths if the same marginal bases are used in both cases (the marginal smooths themselves are just special cases of tensor product smooths.)

The 'normal parameterization' (np=TRUE) re-parameterizes the marginal smooths of a tensor product smooth so that the parameters are function values at a set of points spread evenly through the range of values of the covariate of the smooth. This means that the penalty of the tensor product associated with any particular covariate direction can be interpreted as the penalty of the appropriate marginal smooth applied in that direction and averaged over the smooth. Currently this is only done for marginals of a single variable. This parameterization can reduce numerical stability for when used with marginal smooths other than "cc", "cr" and "cs": if this causes problems, set np=FALSE.

The function does not evaluate the variable arguments.

Value

A class tensor.smooth.spec object defining a tensor product smooth to be turned into a basis and penalties by the smooth.construct.tensor.smooth.spec function.

The returned object contains the following items:

margin — A list of smooth.spec objects of the type returned by s, defining the basis from which the tensor product smooth is constructed.

term — An array of text strings giving the names of the covariates that the term is a function of.

by — is the name of any by variable as text ("NA" for none).

fx — logical array with element for each penalty of the term (tensor product smooths have multiple penalties). TRUE if the penalty is to be ignored, FALSE, otherwise.

full.call — Text for pasting into a string to be converted to a gam formula, which has the values of function options given explicitly - this is useful for constructing a fully expanded gam formula which can be used without needing access to any

	variables that may have been used to define k, fx, bs or m in the original call. i.e. this is text which when parsed and evaluated generates a call to `s()` with all the options spelled out explicitly.
`label`	A suitable text label for this smooth term.
`dim`	The dimension of the smoother - i.e. the number of covariates that it is a function of.
`mp`	`TRUE` is multiple penalties are to be used (default).
`np`	`TRUE` to re-parameterize 1-D marginal smooths in terms of function values (defualt).

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

Wood, S.N. (2006) Low rank scale invariant tensor product smooths for Generalized Additive Mixed Models. Biometrics

http://www.maths.bath.ac.uk/~sw283/

See Also

s,gam,gamm

Examples

```
# following shows how tensor pruduct deals nicely with
# badly scaled covariates (range of x 5% of range of z )
test1<-function(x,z,sx=0.3,sz=0.4)
{ x<-x*20
  (pi**sx*sz)*(1.2*exp(-(x-0.2)^2/sx^2-(z-0.3)^2/sz^2)+
  0.8*exp(-(x-0.7)^2/sx^2-(z-0.8)^2/sz^2))
}
n<-500
old.par<-par(mfrow=c(2,2))
x<-runif(n)/20;z<-runif(n);
xs<-seq(0,1,length=30)/20;zs<-seq(0,1,length=30)
pr<-data.frame(x=rep(xs,30),z=rep(zs,rep(30,30)))
truth<-matrix(test1(pr$x,pr$z),30,30)
f <- test1(x,z)
y <- f + rnorm(n)*0.2
b1<-gam(y~s(x,z))
persp(xs,zs,truth);title("truth")
vis.gam(b1);title("t.p.r.s")
b2<-gam(y~te(x,z))
vis.gam(b2);title("tensor product")
b3<-gam(y~te(x,z,bs=c("tp","tp")))
vis.gam(b3);title("tensor product")
par(old.par)

test2<-function(u,v,w,sv=0.3,sw=0.4)
{ ((pi**sv*sw)*(1.2*exp(-(v-0.2)^2/sv^2-(w-0.3)^2/sw^2)+
  0.8*exp(-(v-0.7)^2/sv^2-(w-0.8)^2/sw^2)))*(u-0.5)^2*20
}
```

```
n <- 500
v <- runif(n);w<-runif(n);u<-runif(n)
f <- test2(u,v,w)
y <- f + rnorm(n)*0.2
# tensor product of a 2-d thin plate regression spline and 1-d cr spline
b <- gam(y~te(v,w,u,k=c(30,5),d=c(2,1),bs=c("tp","cr")))
op <- par(mfrow=c(2,2))
vis.gam(b,cond=list(u=0),color="heat",zlim=c(-0.2,3.5))
vis.gam(b,cond=list(u=.33),color="heat",zlim=c(-0.2,3.5))
vis.gam(b,cond=list(u=.67),color="heat",zlim=c(-0.2,3.5))
vis.gam(b,cond=list(u=1),color="heat",zlim=c(-0.2,3.5))
par(op)
```

`tensor.prod.model.matrix`

Utility functions for constructing tensor product smooths

Description

Produce model matrices or penalty matrices for a tensor product smooth from the model matrices or penalty matrices for the marginal bases of the smooth.

Usage

```
tensor.prod.model.matrix(X)
tensor.prod.penalties(S)
```

Arguments

`X`	a list of model matrices for the marginal bases of a smooth
`S`	a list of penalties for the marginal bases of a smooth.

Details

If `X[[1]]`, `X[[2]]` ... `X[[m]]` are the model matrices of the marginal bases of a tensor product smooth then the ith row of the model matrix for the whole tensor product smooth is given by `X[[1]][i,]%x%X[[2]][i,]%x% ... X[[m]][i,]`, where `%x%` is the Kronecker product. Of course the routine operates column-wise, not row-wise!

If `S[[1]]`, `S[[2]]` ... `S[[m]]` are the penalty matrices for the marginal bases, and `I[[1]]`, `I[[2]]` ... `I[[m]]` are corresponding identity matrices, each of the same dimension as its corresponding penalty, then the tensor product smooth has m associate penalties of the form:

`S[[1]]%x%I[[2]]%x% ... I[[m]],`

`I[[1]]%x%S[[2]]%x% ... I[[m]]`

...

`I[[1]]%x%I[[2]]%x% ... S[[m]].`

Of course it's important that the model matrices and penalty matrices are presented in the same order when constructing tensor product smooths.

Value

Either a single model matrix for a tensor product smooth, or a list of penalty terms for a tensor product smooth.

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

References

http://www.maths.bath.ac.uk/~sw283/

See Also

te, smooth.construct.tensor.smooth.spec

Examples

```
X <- list(matrix(1:4,2,2),matrix(5:10,2,3))
tensor.prod.model.matrix(X)

S<-list(matrix(c(2,1,1,2),2,2),matrix(c(2,1,0,1,2,1,0,1,2),3,3))
tensor.prod.penalties(S)
```

---

uniquecombs *find the unique rows in a matrix*

Description

This routine returns a matrix containing all the unique rows of the matrix supplied as its argument. That is, all the duplicate rows are stripped out. Note that the ordering of the rows on exit is not the same as on entry.

Usage

```
uniquecombs(x)
```

Arguments

x	is an R matrix

Details

Models with more parameters than unique combinations of covariates are not identifiable. This routine provides a means of evaluating the number of unique combinations of coavariates in a model. The routine calls compiled C code.

Value

A matrix consisting of the unique rows of x (in arbitrary order).

Author(s)

Simon N. Wood ⟨simon.wood@r-project.org⟩

Examples

```
X<-matrix(c(1,2,3,1,2,3,4,5,6,1,3,2,4,5,6,1,1,1),6,3,byrow=TRUE)
print(X)
uniquecombs(X)
```

---

vcov.gam *Extract parameter (estimator) covariance matrix from GAM fit*

Description

Extracts the Bayesian posterior covariance matrix of the parameters or frequentist covariance matrix of the parameter estimators from a fitted gam object.

Usage

```
## S3 method for class 'gam':
vcov(object, freq = TRUE, dispersion = NULL, ...)
```

Arguments

object	fitted model object of class gam as produced by gam().
freq	TRUE to return the frequentist covariance matrix of the parameter estimators, FALSE to return the Bayesian posterior covariance matrix of the parameters.
dispersion	a value for the dispersion parameter: not normally used.
...	other arguments, currently ignored.

Details

Basically, just extracts object$Ve or object$Vp from a gamObject.

Value

A matrix corresponding to the estimated frequentist covariance matrix of the model parameter estimators/coefficients, or the estimated posterior covariance matrix of the parameters, depending on the argument freq.

Author(s)

Henric Nilsson. Maintained by Simon N. Wood ⟨simon.wood@r-project.org⟩

See Also

gam

Examples

```
n <- 100
x <- runif(n)
y <- sin(x*2*pi) + rnorm(n)*.2
mod <- gam(y~s(x,bs="cc",k=10),knots=list(x=seq(0,1,length=10)))
diag(vcov(mod))
```

---

vis.gam *Visualization of GAM objects*

Description

Produces perspective or contour plot views of gam model predictions, fixing all but the values in view to the values supplied in cond.

Usage

```
vis.gam(x,view=NULL,cond=list(),n.grid=30,too.far=0,col=NA,
        color="heat",contour.col=NULL,se=-1,type="link",
        plot.type="persp",zlim=NULL,nCol=50,...)
```

Arguments

x	a gam object, produced by gam()
view	an array containing the names of the two main effect terms to be displayed on the x and y dimensions of the plot. If omitted the first two suitable terms will be used. Names must be names from names(x$model). i.e. terms are used, rather than variables.
cond	a named list of the values to use for the other predictor terms (not in view). Terms omitted from this list will have their values set to their mean for continuous variables, or first level for factors. Names must correspond to names(x$model).
n.grid	The number of grid nodes in each direction used for calculating the plotted surface.
too.far	plot grid nodes that are too far from the points defined by the variables given in view can be excluded from the plot. too.far determines what is too far. The grid is scaled into the unit square along with the view variables and then grid nodes more than too.far from the predictor variables are excluded.
col	The colours for the facets of the plot. If this is NA then if se>0 the facets are transparent, otherwise the colour scheme specified in color is used. If col is not NA then it is used as the facet colour.
color	the colour scheme to use for plots when se<=0. One of "topo", "heat", "cm", "terrain", "gray" or "bw". Schemes "gray" and "bw" also modify the colors used when se>0.
contour.col	sets the colour of contours when using plot.type="contour". Default scheme used if NULL.
se	if less than or equal to zero then only the predicted surface is plotted, but if greater than zero, then 3 surfaces are plotted, one at the predicted values minus se standard errors, one at the predicted values and one at the predicted values plus se standard errors.

type	"link" to plot on linear predictor scale and "response" to plot on the response scale.
plot.type	one of "contour" or "persp".
zlim	a two item array giving the lower and upper limits for the z-axis scale. NULL to choose automatically.
nCol	The number of colors to use in color schemes.
...	other options to pass on to persp, image or contour. In particular ticktype="detailed" will add proper axes labelling to the plots.

Details

The x and y limits are determined by the ranges of the terms named in view. If se<=0 then a single (height colour coded, by default) surface is produced, otherwise three (by default see-through) meshes are produced at mean and +/- se standard errors. Parts of the x-y plane too far from data can be excluded by setting too.far

All options to the underlying graphics functions can be reset by passing them as extra arguments ...: such supplied values will always over-ride the defualt values used by vis.gam.

Value

Simply produces a plot.

Author(s)

Simon Wood ⟨simon.wood@r-project.org⟩

Based on an original idea and design by Mike Lonergan.

See Also

persp and gam.

Examples

```
library(mgcv)
set.seed(0)
n<-200;sig2<-4
x0 <- runif(n, 0, 1);x1 <- runif(n, 0, 1)
x2 <- runif(n, 0, 1)
y<-x0^2+x1*x2 +runif(n,-0.3,0.3)
g<-gam(y~s(x0,x1,x2))
old.par<-par(mfrow=c(2,2))
# display the prediction surface in x0, x1 ....
vis.gam(g,ticktype="detailed",color="heat",theta=-35)
vis.gam(g,se=2,theta=-35) # with twice standard error surfaces
vis.gam(g, view=c("x1","x2"),cond=list(x0=0.75)) # different view
vis.gam(g, view=c("x1","x2"),cond=list(x0=.75),theta=210,phi=40,
        too.far=.07)
# contour examples....
vis.gam(g, view=c("x1","x2"),plot.type="contour",color="heat")
vis.gam(g, view=c("x1","x2"),plot.type="contour",color="terrain")
vis.gam(g, view=c("x1","x2"),plot.type="contour",color="topo")
vis.gam(g, view=c("x1","x2"),plot.type="contour",color="cm")
```

```
# ..... areas where there is no data are not plotted
par(old.par)

# Examples with factor and "by" variables

fac<-rep(1:4,20)
x<-runif(80)
y<-fac+2*x^2+rnorm(80)*0.1
fac<-factor(fac)
b<-gam(y~fac+s(x))

vis.gam(b,theta=-35,color="heat") # factor example

z<-rnorm(80)*0.4
y<-as.numeric(fac)+3*x^2*z+rnorm(80)*0.1
b<-gam(y~fac+s(x,by=z))

vis.gam(b,theta=-35,color="heat",cond=list(z=1)) # by variable example

vis.gam(b,view=c("z","x"),theta= 35) # plot against by variable
```

Chapter 18

The `nlme` package

`ACF` *Autocorrelation Function*

Description

This function is generic; method functions can be written to handle specific classes of objects. Classes which already have methods for this function include: `gls` and `lme`.

Usage

```
ACF(object, maxLag, ...)
```

Arguments

`object`	any object from which an autocorrelation function can be obtained. Generally an object resulting from a model fit, from which residuals can be extracted.
`maxLag`	maximum lag for which the autocorrelation should be calculated.
`...`	some methods for this generic require additional arguments.

Value

will depend on the method function used; see the appropriate documentation.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨Bates@stat.wisc.edu⟩

References

Box, G.E.P., Jenkins, G.M., and Reinsel G.C. (1994) "Time Series Analysis: Forecasting and Control", 3rd Edition, Holden-Day.

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

`ACF.gls`, `ACF.lme`, `plot.ACF`

Examples

```
## see the method function documentation
```

ACF.gls *Autocorrelation Function for gls Residuals*

Description

This method function calculates the empirical autocorrelation function for the residuals from a gls fit. If a grouping variable is specified in form, the autocorrelation values are calculated using pairs of residuals within the same group; otherwise all possible residual pairs are used. The autocorrelation function is useful for investigating serial correlation models for equally spaced data.

Usage

```
## S3 method for class 'gls':
ACF(object, maxLag, resType, form, na.action, ...)
```

Arguments

object	an object inheriting from class gls, representing a generalized least squares fitted model.
maxLag	an optional integer giving the maximum lag for which the autocorrelation should be calculated. Defaults to maximum lag in the residuals.
resType	an optional character string specifying the type of residuals to be used. If "response", the "raw" residuals (observed - fitted) are used; else, if "pearson", the standardized residuals (raw residuals divided by the corresponding standard errors) are used; else, if "normalized", the normalized residuals (standardized residuals pre-multiplied by the inverse square-root factor of the estimated error correlation matrix) are used. Partial matching of arguments is used, so only the first character needs to be provided. Defaults to "pearson".
form	an optional one sided formula of the form ~ t, or ~ t | g, specifying a time covariate t and, optionally, a grouping factor g. The time covariate must be integer valued. When a grouping factor is present in form, the autocorrelations are calculated using residual pairs within the same group. Defaults to ~ 1, which corresponds to using the order of the observations in the data as a covariate, and no groups.
na.action	a function that indicates what should happen when the data contain NAs. The default action (na.fail) causes ACF.gls to print an error message and terminate if there are any incomplete observations.
...	some methods for this generic require additional arguments.

Value

a data frame with columns lag and ACF representing, respectively, the lag between residuals within a pair and the corresponding empirical autocorrelation. The returned value inherits from class ACF.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Box, G.E.P., Jenkins, G.M., and Reinsel G.C. (1994) "Time Series Analysis: Forecasting and Control", 3rd Edition, Holden-Day.

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

`ACF.gls`, `plot.ACF`

Examples

```
fm1 <- gls(follicles ~ sin(2*pi*Time) + cos(2*pi*Time), Ovary)
ACF(fm1, form = ~ 1 | Mare)

# Pinheiro and Bates, p. 255-257
fm1Dial.gls <- gls(rate ~
  (pressure+I(pressure^2)+I(pressure^3)+I(pressure^4))*QB,
                   Dialyzer)

fm2Dial.gls <- update(fm1Dial.gls,
                  weights = varPower(form = ~ pressure))

ACF(fm2Dial.gls, form = ~ 1 | Subject)
```

ACF.lme *Autocorrelation Function for lme Residuals*

Description

This method function calculates the empirical autocorrelation function for the within-group residuals from an `lme` fit. The autocorrelation values are calculated using pairs of residuals within the innermost group level. The autocorrelation function is useful for investigating serial correlation models for equally spaced data.

Usage

```
## S3 method for class 'lme':
ACF(object, maxLag, resType, ...)
```

Arguments

`object`	an object inheriting from class `lme`, representing a fitted linear mixed-effects model.
`maxLag`	an optional integer giving the maximum lag for which the autocorrelation should be calculated. Defaults to maximum lag in the within-group residuals.

`resType` an optional character string specifying the type of residuals to be used. If `"response"`, the "raw" residuals (observed - fitted) are used; else, if `"pearson"`, the standardized residuals (raw residuals divided by the corresponding standard errors) are used; else, if `"normalized"`, the normalized residuals (standardized residuals pre-multiplied by the inverse square-root factor of the estimated error correlation matrix) are used. Partial matching of arguments is used, so only the first character needs to be provided. Defaults to `"pearson"`.

`...` some methods for this generic require additional arguments – not used.

Value

a data frame with columns `lag` and `ACF` representing, respectively, the lag between residuals within a pair and the corresponding empirical autocorrelation. The returned value inherits from class `ACF`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Box, G.E.P., Jenkins, G.M., and Reinsel G.C. (1994) "Time Series Analysis: Forecasting and Control", 3rd Edition, Holden-Day.

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

`ACF.gls`, `plot.ACF`

Examples

```
fm1 <- lme(follicles ~ sin(2*pi*Time) + cos(2*pi*Time),
           Ovary, random = ~ sin(2*pi*Time) | Mare)
ACF(fm1, maxLag = 11)

# Pinheiro and Bates, p240-241
fm1Over.lme <- lme(follicles  ~ sin(2*pi*Time) +
           cos(2*pi*Time), data=Ovary,
     random=pdDiag(~sin(2*pi*Time)) )
(ACF.fm1Over <- ACF(fm1Over.lme, maxLag=10))
plot(ACF.fm1Over, alpha=0.01)
```

`Alfalfa` *Split-Plot Experiment on Varieties of Alfalfa*

Description

The `Alfalfa` data frame has 72 rows and 4 columns.

Format

This data frame contains the following columns:

Variety a factor with levels `Cossack`, `Ladak`, and `Ranger`

Date a factor with levels `None S1 S20 O7`

Block a factor with levels `1 2 3 4 5 6`

Yield a numeric vector

Details

These data are described in Snedecor and Cochran (1980) as an example of a split-plot design. The treatment structure used in the experiment was a $3x4$ full factorial, with three varieties of alfalfa and four dates of third cutting in 1943. The experimental units were arranged into six blocks, each subdivided into four plots. The varieties of alfalfa (*Cossac*, *Ladak*, and *Ranger*) were assigned randomly to the blocks and the dates of third cutting (*None*, *S1*—September 1, *S20*—September 20, and *O7*—October 7) were randomly assigned to the plots. All four dates were used on each block.

Source

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York. (Appendix A.1)

Snedecor, G. W. and Cochran, W. G. (1980), *Statistical Methods (7th ed)*, Iowa State University Press, Ames, IA

---

allCoef *Extract Coefficients from a Set of Objects*

Description

The extractor function is applied to each object in `...`, with the result being converted to a vector. A `map` attribute is included to indicate which pieces of the returned vector correspond to the original objects in `dots`.

Usage

```
allCoef(..., extract)
```

Arguments

`...`	objects to which `extract` will be applied. Generally these will be model components, such as `corStruct` and `varFunc` objects.
`extract`	an optional extractor function. Defaults to `coef`.

Value

a vector with all elements, generally coefficients, obtained by applying `extract` to the objects in `....`

Author(s)

Jose' Pinheiro and Douglas Bates

See Also

lmeStruct,nlmeStruct

Examples

```
cs1 <- corAR1(0.1)
vf1 <- varPower(0.5)
allCoef(cs1, vf1)
```

`anova.gls` *Compare Likelihoods of Fitted Objects*

Description

When only one fitted model object is present, a data frame with the sums of squares, numerator degrees of freedom, F-values, and P-values for Wald tests for the terms in the model (when `Terms` and `L` are `NULL`), a combination of model terms (when `Terms` in not `NULL`), or linear combinations of the model coefficients (when `L` is not `NULL`). Otherwise, when multiple fitted objects are being compared, a data frame with the degrees of freedom, the (restricted) log-likelihood, the Akaike Information Criterion (AIC), and the Bayesian Information Criterion (BIC) of each object is returned. If `test=TRUE`, whenever two consecutive objects have different number of degrees of freedom, a likelihood ratio statistic, with the associated p-value is included in the returned data frame.

Usage

```
## S3 method for class 'gls':
anova(object, ..., test, type, adjustSigma, Terms, L, verbose)
```

Arguments

`object`	a fitted model object inheriting from class `gls`, representing a generalized least squares fit.
`...`	other optional fitted model objects inheriting from classes `gls`, `gnls`, `lm`, `lme`, `lmList`, `nlme`, `nlsList`, or `nls`.
`test`	an optional logical value controlling whether likelihood ratio tests should be used to compare the fitted models represented by `object` and the objects in `...`. Defaults to `TRUE`.
`type`	an optional character string specifying the type of sum of squares to be used in F-tests for the terms in the model. If `"sequential"`, the sequential sum of squares obtained by including the terms in the order they appear in the model is used; else, if `"marginal"`, the marginal sum of squares obtained by deleting a term from the model at a time is used. This argument is only used when a single fitted object is passed to the function. Partial matching of arguments is used, so only the first character needs to be provided. Defaults to `"sequential"`.
`adjustSigma`	an optional logical value. If `TRUE` and the estimation method used to obtain `object` was maximum likelihood, the residual standard error is multiplied by $\sqrt{n_{obs}/(n_{obs} - n_{par})}$, converting it to a REML-like estimate. This argument is only used when a single fitted object is passed to the function. Default is `TRUE`.

Terms	an optional integer or character vector specifying which terms in the model should be jointly tested to be zero using a Wald F-test. If given as a character vector, its elements must correspond to term names; else, if given as an integer vector, its elements must correspond to the order in which terms are included in the model. This argument is only used when a single fitted object is passed to the function. Default is NULL.
L	an optional numeric vector or array specifying linear combinations of the coefficients in the model that should be tested to be zero. If given as an array, its rows define the linear combinations to be tested. If names are assigned to the vector elements (array columns), they must correspond to coefficients names and will be used to map the linear combination(s) to the coefficients; else, if no names are available, the vector elements (array columns) are assumed in the same order as the coefficients appear in the model. This argument is only used when a single fitted object is passed to the function. Default is NULL.
verbose	an optional logical value. If TRUE, the calling sequences for each fitted model object are printed with the rest of the output, being omitted if verbose = FALSE. Defaults to FALSE.

Value

a data frame inheriting from class anova.lme.

Note

Likelihood comparisons are not meaningful for objects fit using restricted maximum likelihood and with different fixed effects.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York.

See Also

gls, gnls, lme, logLik.gls, AIC, BIC, print.anova.lme

Examples

```
# AR(1) errors within each Mare
fm1 <- gls(follicles ~ sin(2*pi*Time) + cos(2*pi*Time), Ovary,
           correlation = corAR1(form = ~ 1 | Mare))
anova(fm1)
# variance changes with a power of the absolute fitted values?
fm2 <- update(fm1, weights = varPower())
anova(fm1, fm2)

# Pinheiro and Bates, p. 251-252
fm1Orth.gls <- gls(distance ~ Sex * I(age - 11), Orthodont,
                correlation = corSymm(form = ~ 1 | Subject),
                weights = varIdent(form = ~ 1 | age))
```

```
fm2Orth.gls <- update(fm1Orth.gls,
                corr = corCompSymm(form = ~ 1 | Subject))
anova(fm1Orth.gls, fm2Orth.gls)

# Pinheiro and Bates, pp. 215-215, 255-260
#p. 215
fm1Dial.lme <-
  lme(rate ~(pressure + I(pressure^2) + I(pressure^3) + I(pressure^4))*QB,
      Dialyzer, ~ pressure + I(pressure^2))
# p. 216
fm2Dial.lme <- update(fm1Dial.lme,
                  weights = varPower(form = ~ pressure))
# p. 255
fm1Dial.gls <- gls(rate ~ (pressure +
     I(pressure^2) + I(pressure^3) + I(pressure^4))*QB,
        Dialyzer)
fm2Dial.gls <- update(fm1Dial.gls,
                 weights = varPower(form = ~ pressure))
anova(fm1Dial.gls, fm2Dial.gls)
fm3Dial.gls <- update(fm2Dial.gls,
                    corr = corAR1(0.771, form = ~ 1 | Subject))
anova(fm2Dial.gls, fm3Dial.gls)
# anova.gls to compare a gls and an lme fit
anova(fm3Dial.gls, fm2Dial.lme, test = FALSE)

# Pinheiro and Bates, pp. 261-266
fm1Wheat2 <- gls(yield ~ variety - 1, Wheat2)
fm3Wheat2 <- update(fm1Wheat2,
      corr = corRatio(c(12.5, 0.2),
        form = ~ latitude + longitude, nugget = TRUE))
# Test a specific contrast
anova(fm3Wheat2, L = c(-1, 0, 1))
```

`anova.lme` *Compare Likelihoods of Fitted Objects*

Description

When only one fitted model object is present, a data frame with the sums of squares, numerator degrees of freedom, denominator degrees of freedom, F-values, and P-values for Wald tests for the terms in the model (when `Terms` and `L` are `NULL`), a combination of model terms (when `Terms` in not `NULL`), or linear combinations of the model coefficients (when `L` is not `NULL`). Otherwise, when multiple fitted objects are being compared, a data frame with the degrees of freedom, the (restricted) log-likelihood, the Akaike Information Criterion (AIC), and the Bayesian Information Criterion (BIC) of each object is returned. If `test=TRUE`, whenever two consecutive objects have different number of degrees of freedom, a likelihood ratio statistic, with the associated p-value is included in the returned data frame.

Usage

```
## S3 method for class 'lme':
anova(object, ..., test, type, adjustSigma, Terms, L, verbose)
## S3 method for class 'anova.lme':
print(x, verbose, ...)
```

Arguments

object	a fitted model object inheriting from class lme, representing a mixed-effects model.
...	other optional fitted model objects inheriting from classes gls, gnls, lm, lme, lmList, nlme, nlsList, or nls.
test	an optional logical value controlling whether likelihood ratio tests should be used to compare the fitted models represented by object and the objects in .... Defaults to TRUE.
type	an optional character string specifying the type of sum of squares to be used in F-tests for the terms in the model. If "sequential", the sequential sum of squares obtained by including the terms in the order they appear in the model is used; else, if "marginal", the marginal sum of squares obtained by deleting a term from the model at a time is used. This argument is only used when a single fitted object is passed to the function. Partial matching of arguments is used, so only the first character needs to be provided. Defaults to "sequential".
adjustSigma	an optional logical value. If TRUE and the estimation method used to obtain object was maximum likelihood, the residual standard error is multiplied by $\sqrt{n_{obs}/(n_{obs} - n_{par})}$, converting it to a REML-like estimate. This argument is only used when a single fitted object is passed to the function. Default is TRUE.
Terms	an optional integer or character vector specifying which terms in the model should be jointly tested to be zero using a Wald F-test. If given as a character vector, its elements must correspond to term names; else, if given as an integer vector, its elements must correspond to the order in which terms are included in the model. This argument is only used when a single fitted object is passed to the function. Default is NULL.
L	an optional numeric vector or array specifying linear combinations of the coefficients in the model that should be tested to be zero. If given as an array, its rows define the linear combinations to be tested. If names are assigned to the vector elements (array columns), they must correspond to coefficients names and will be used to map the linear combination(s) to the coefficients; else, if no names are available, the vector elements (array columns) are assumed in the same order as the coefficients appear in the model. This argument is only used when a single fitted object is passed to the function. Default is NULL.
x	an object inheriting from class anova.lme
verbose	an optional logical value. If TRUE, the calling sequences for each fitted model object are printed with the rest of the output, being omitted if verbose = FALSE. Defaults to FALSE.

Value

a data frame inheriting from class anova.lme.

Note

Likelihood comparisons are not meaningful for objects fit using restricted maximum likelihood and with different fixed effects.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

gls, gnls, nlme, lme, AIC, BIC, print.anova.lme, logLik.lme,

Examples

```
fm1 <- lme(distance ~ age, Orthodont, random = ~ age | Subject)
anova(fm1)
fm2 <- update(fm1, random = pdDiag(~age))
anova(fm1, fm2)

# Pinheiro and Bates, pp. 251-254
fm1Orth.gls <- gls(distance ~ Sex * I(age - 11), Orthodont,
            correlation = corSymm(form = ~ 1 | Subject),
            weights = varIdent(form = ~ 1 | age))
fm2Orth.gls <- update(fm1Orth.gls,
                corr = corCompSymm(form = ~ 1 | Subject))
# anova.gls
anova(fm1Orth.gls, fm2Orth.gls)
fm3Orth.gls <- update(fm2Orth.gls, weights = NULL)
# anova.gls
anova(fm2Orth.gls, fm3Orth.gls)
fm4Orth.gls <- update(fm3Orth.gls,
                 weights = varIdent(form = ~ 1 | Sex))
# anova.gls
anova(fm3Orth.gls, fm4Orth.gls)
# not in book but needed for the following command
fm3Orth.lme <-
    lme(distance~Sex*I(age-11), data = Orthodont,
        random = ~ I(age-11) | Subject,
        weights = varIdent(form = ~ 1 | Sex))
# anova.lme to compare an "lme" object with a "gls" object
anova(fm3Orth.lme, fm4Orth.gls, test = FALSE)

# Pinheiro and Bates, pp. 222-225
options(contrasts = c("contr.treatment", "contr.poly"))
fm1BW.lme <- lme(weight ~ Time * Diet, BodyWeight,
                  random = ~ Time)
fm2BW.lme <- update(fm1BW.lme, weights = varPower())
# Test a specific contrast
anova(fm2BW.lme, L = c("Time:Diet2" = 1, "Time:Diet3" = -1))

fm1Theo.lis <- nlsList(
     conc ~ SSfol(Dose, Time, lKe, lKa, lCl), data=Theoph)
fm1Theo.lis

# Pinheiro and Bates, pp. 352-365
fm1Theo.lis <- nlsList(
     conc ~ SSfol(Dose, Time, lKe, lKa, lCl), data=Theoph)
fm1Theo.nlme <- nlme(fm1Theo.lis)
fm2Theo.nlme <- update(fm1Theo.nlme,
   random=pdDiag(lKe+lKa+lCl~1) )
fm3Theo.nlme <- update(fm2Theo.nlme,
```

```
    random=pdDiag(lKa+lCl~1) )

# anova comparing 3 models
anova(fm1Theo.nlme, fm3Theo.nlme, fm2Theo.nlme)
```

`as.matrix.corStruct` *Matrix of a corStruct Object*

Description

This method function extracts the correlation matrix, or list of correlation matrices, associated with `object`.

Usage

```
## S3 method for class 'corStruct':
as.matrix(x)
```

Arguments

`x`	an object inheriting from class `corStruct`, representing a correlation structure.

Value

If the correlation structure includes a grouping factor, the returned value will be a list with components given by the correlation matrices for each group. Otherwise, the returned value will be a matrix representing the correlation structure associated with `object`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York.

See Also

`corClasses`, `corMatrix`

Examples

```
cst1 <- corAR1(form = ~1|Subject)
cst1 <- Initialize(cst1, data = Orthodont)
as.matrix(cst1)
```

as.matrix.pdMat *Matrix of a pdMat Object*

Description

This method function extracts the positive-definite matrix represented by `x`.

Usage

```
## S3 method for class 'pdMat':
as.matrix(x)
```

Arguments

`x`	an object inheriting from class `pdMat`, representing a positive-definite matrix.

Value

a matrix corresponding to the positive-definite matrix represented by `x`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York.

See Also

`pdMat`, `corMatrix`

Examples

```
as.matrix(pdSymm(diag(4)))
```

as.matrix.reStruct *Matrices of an reStruct Object*

Description

This method function extracts the positive-definite matrices corresponding to the `pdMat` elements of `object`.

Usage

```
## S3 method for class 'reStruct':
as.matrix(x)
```

Arguments

x	an object inheriting from class reStruct, representing a random effects structure and consisting of a list of pdMat objects.

Value

a list with components given by the positive-definite matrices corresponding to the elements of object.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York.

See Also

as.matrix.pdMat, reStruct, pdMat

Examples

```
rs1 <- reStruct(pdSymm(diag(3), ~age+Sex, data = Orthodont))
as.matrix(rs1)
```

---

asOneFormula *Combine Formulas of a Set of Objects*

Description

The names of all variables used in the formulas extracted from the objects defined in ... are converted into a single linear formula, with the variables names separated by +.

Usage

```
asOneFormula(..., omit)
```

Arguments

...	objects, or lists of objects, from which a formula can be extracted.
omit	an optional character vector with the names of variables to be omitted from the returned formula. Defaults to c(".", "pi").

Value

a one-sided linear formula with all variables named in the formulas extracted from the objects in ..., except the ones listed in omit.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`formula`, `all.vars`

Examples

```
asOneFormula(y ~ x + z | g, list(~ w, ~ t * sin(2 * pi)))
```

`Assay` *Bioassay on Cell Culture Plate*

Description

The `Assay` data frame has 60 rows and 4 columns.

Format

This data frame contains the following columns:

Block an ordered factor with levels `2` < `1` identifying the block where the wells are measured.

sample a factor with levels `a` to `f` identifying the sample corresponding to the well.

dilut a factor with levels `1` to `5` indicating the dilution applied to the well

logDens a numeric vector of the log-optical density

Details

These data, courtesy of Rich Wolfe and David Lansky from Searle, Inc., come from a bioassay run on a 96-well cell culture plate. The assay is performed using a split-block design. The 8 rows on the plate are labeled A–H from top to bottom and the 12 columns on the plate are labeled 1–12 from left to right. Only the central 60 wells of the plate are used for the bioassay (the intersection of rows B–G and columns 2–11). There are two blocks in the design: Block 1 contains columns 2–6 and Block 2 contains columns 7–11. Within each block, six samples are assigned randomly to rows and five (serial) dilutions are assigned randomly to columns. The response variable is the logarithm of the optical density. The cells are treated with a compound that they metabolize to produce the stain. Only live cells can make the stain, so the optical density is a measure of the number of cells that are alive and healthy.

Source

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York. (Appendix A.2)

`asTable` *Convert groupedData to a matrix*

Description

Create a tabular representation of the response in a balanced groupedData object.

Usage

```
asTable(object)
```

Arguments

`object`	A balanced `groupedData` object

Details

A balanced groupedData object can be represented as a matrix or table of response values corresponding to the values of a primary covariate for each level of a grouping factor. This function creates such a matrix representation of the data in `object`.

Value

A matrix. The data in the matrix are the values of the response. The columns correspond to the distinct values of the primary covariate and are labelled as such. The rows correspond to the distinct levels of the grouping factor and are labelled as such.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York.

See Also

`groupedData`, `isBalanced`, `balancedGrouped`

Examples

```
asTable(Orthodont)

# Pinheiro and Bates, p. 109
ergoStool.mat <- asTable(ergoStool)
```

augPred *Augmented Predictions*

Description

Predicted values are obtained at the specified values of primary. If object has a grouping structure (i.e. getGroups(object) is not NULL), predicted values are obtained for each group. If level has more than one element, predictions are obtained for each level of the max(level) grouping factor. If other covariates besides primary are used in the prediction model, their average (numeric covariates) or most frequent value (categorical covariates) are used to obtain the predicted values. The original observations are also included in the returned object.

Usage

```
augPred(object, primary, minimum, maximum, length.out, ...)
```

Arguments

object	a fitted model object from which predictions can be extracted, using a predict method.
primary	an optional one-sided formula specifying the primary covariate to be used to generate the augmented predictions. By default, if a covariate can be extracted from the data used to generate object (using getCovariate), it will be used as primary.
minimum	an optional lower limit for the primary covariate. Defaults to min(primary).
maximum	an optional upper limit for the primary covariate. Defaults to max(primary).
length.out	an optional integer with the number of primary covariate values at which to evaluate the predictions. Defaults to 51.
...	some methods for the generic may require additional arguments.

Value

a data frame with four columns representing, respectively, the values of the primary covariate, the groups (if object does not have a grouping structure, all elements will be 1), the predicted or observed values, and the type of value in the third column: original for the observed values and predicted (single or no grouping factor) or predict.groupVar (multiple levels of grouping), with groupVar replaced by the actual grouping variable name (fixed is used for population predictions). The returned object inherits from class augPred.

Note

This function is generic; method functions can be written to handle specific classes of objects. Classes which already have methods for this function include: gls, lme, and lmList.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York.

See Also

`plot.augPred`, `getGroups`, `predict`

Examples

```
fm1 <- lme(Orthodont, random = ~1)
augPred(fm1, length.out = 2, level = c(0,1))
```

`balancedGrouped` *Create a groupedData object from a matrix*

Description

Create a `groupedData` object from a data matrix. This function can be used only with balanced data. The opposite conversion, from a `groupedData` object to a `matrix`, is done with `asTable`.

Usage

```
balancedGrouped(form, data, labels=NULL, units=NULL)
```

Arguments

`form`	A formula of the form `y ~ x	g` giving the name of the response, the primary covariate, and the grouping factor.
`data`	A matrix or data frame containing the values of the response grouped according to the levels of the grouping factor (rows) and the distinct levels of the primary covariate (columns). The `dimnames` of the matrix are used to construct the levels of the grouping factor and the primary covariate.
`labels`	an optional list of character strings giving labels for the response and the primary covariate. The label for the primary covariate is named `x` and that for the response is named `y`. Either label can be omitted.
`units`	an optional list of character strings giving the units for the response and the primary covariate. The units string for the primary covariate is named `x` and that for the response is named `y`. Either units string can be omitted.

Value

A balanced `groupedData` object.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York.

See Also

groupedData, isBalanced, asTable

Examples

```
OrthoMat <- asTable( Orthodont )
Orth2 <- balancedGrouped(distance ~ age | Subject, data = OrthoMat,
    labels = list(x = "Age",
                  y = "Distance from pituitary to pterygomaxillary fissure"),
    units = list(x = "(yr)", y = "(mm)"))
Orth2[ 1:10, ]        ## check the first few entries

# Pinheiro and Bates, p. 109
ergoStool.mat <- asTable(ergoStool)
balancedGrouped(effort~Type|Subject,
                data=ergoStool.mat)
```

---

bdf *Language scores*

Description

The bdf data frame has 2287 rows and 25 columns of language scores from grade 8 pupils in elementary schools in The Netherlands.

Usage

```
data(bdf)
```

Format

schoolNR a factor denoting the school.

pupilNR a factor denoting the pupil.

IQ.verb a numeric vector of verbal IQ scores

IQ.perf a numeric vector of IQ scores.

sex Sex of the student.

Minority a factor indicating if the student is a member of a minority group.

repeatgr an ordered factor indicating if one or more grades have been repeated.

aritPRET a numeric vector

classNR a numeric vector

aritPOST a numeric vector

langPRET a numeric vector

langPOST a numeric vector

ses a numeric vector of socioeconomic status indicators.

denomina a factor indicating of the school is a public school, a Protestant private school, a Catholic private school, or a non-denominational private school.

schoolSES a numeric vector

satiprin a numeric vector

natitest a factor with levels `0` and `1`

meetings a numeric vector

currmeet a numeric vector

mixedgra a factor indicating if the class is a mixed-grade class.

percmino a numeric vector

aritdiff a numeric vector

homework a numeric vector

classsiz a numeric vector

groupsiz a numeric vector

Source

`http://stat.gamma.rug.nl/snijders/multilevel.htm`

References

Snijders, Tom and Bosker, Roel (1999), *Multilevel Analysis: An Introduction to Basic and Advanced Multilevel Modeling*, Sage.

Examples

```
summary(bdf)
```

`BIC` *Bayesian Information Criterion*

Description

This generic function calculates the Bayesian information criterion, also known as Schwarz's Bayesian criterion (SBC), for one or several fitted model objects for which a log-likelihood value can be obtained, according to the formula -2log-likelihood$+n_{par}\log(n_{obs})$, where n_{par} represents the number of parameters and n_{obs} the number of observations in the fitted model.

Usage

```
BIC(object, ...)
```

Arguments

`object`	a fitted model object, for which there exists a `logLik` method to extract the corresponding log-likelihood, or an object inheriting from class `logLik`.
`...`	optional fitted model objects.

Value

if just one object is provided, returns a numeric value with the corresponding BIC; if more than one object are provided, returns a `data.frame` with rows corresponding to the objects and columns representing the number of parameters in the model (`df`) and the BIC.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Schwarz, G. (1978) "Estimating the Dimension of a Model", Annals of Statistics, 6, 461-464.

See Also

`logLik`, `AIC`, `BIC.logLik`

Examples

```
fm1 <- lm(distance ~ age, data = Orthodont) # no random effects
BIC(fm1)
fm2 <- lme(distance ~ age, data = Orthodont) # random is ~age
BIC(fm1, fm2)
```

BIC.logLik *BIC of a logLik Object*

Description

This function calculates the Bayesian information criterion, also known as Schwarz's Bayesian criterion (SBC) for an object inheriting from class `logLik`, according to the formula -2log-likelihood $+ n_{par}\log(n_{obs})$, where n_{par} represents the number of parameters and n_{obs} the number of observations in the fitted model. When comparing fitted objects, the smaller the BIC, the better the fit.

Usage

```
## S3 method for class 'logLik':
BIC(object, ...)
```

Arguments

`object`	an object inheriting from class `logLik`, usually resulting from applying a `logLik` method to a fitted model object.
`...`	some methods for this generic use optional arguments. None are used in this method.

Value

a numeric value with the corresponding BIC.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Schwarz, G. (1978) "Estimating the Dimension of a Model", Annals of Statistics, 6, 461-464.

See Also

BIC, logLik, AIC.

Examples

```
fm1 <- lm(distance ~ age, data = Orthodont)
BIC(logLik(fm1))
```

BodyWeight *Rat weight over time for different diets*

Description

The BodyWeight data frame has 176 rows and 4 columns.

Format

This data frame contains the following columns:

weight a numeric vector giving the body weight of the rat (grams).

Time a numeric vector giving the time at which the measurement is made (days).

Rat an ordered factor with levels 2 < 3 < 4 < 1 < 8 < 5 < 6 < 7 < 11 < 9 < 10 < 12 < 13 < 15 < 14 < 16 identifying the rat whose weight is measured.

Diet a factor with levels 1 to 3 indicating the diet that the rat receives.

Details

Hand and Crowder (1996) describe data on the body weights of rats measured over 64 days. These data also appear in Table 2.4 of Crowder and Hand (1990). The body weights of the rats (in grams) are measured on day 1 and every seven days thereafter until day 64, with an extra measurement on day 44. The experiment started several weeks before "day 1." There are three groups of rats, each on a different diet.

Source

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York. (Appendix A.3)

Crowder, M. and Hand, D. (1990), *Analysis of Repeated Measures*, Chapman and Hall, London.

Hand, D. and Crowder, M. (1996), *Practical Longitudinal Data Analysis*, Chapman and Hall, London.

Cefamandole *Pharmacokinetics of Cefamandole*

Description

The Cefamandole data frame has 84 rows and 3 columns.

Format

This data frame contains the following columns:

Subject a factor giving the subject from which the sample was drawn.

Time a numeric vector giving the time at which the sample was drawn (minutes post-injection).

conc a numeric vector giving the observed plasma concentration of cefamandole (mcg/ml).

Details

Davidian and Giltinan (1995, 1.1, p. 2) describe data obtained during a pilot study to investigate the pharmacokinetics of the drug cefamandole. Plasma concentrations of the drug were measured on six healthy volunteers at 14 time points following an intraveneous dose of 15 mg/kg body weight of cefamandole.

Source

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York. (Appendix A.4)

Davidian, M. and Giltinan, D. M. (1995), *Nonlinear Models for Repeated Measurement Data*, Chapman and Hall, London.

Examples

```
plot(Cefamandole)
fm1 <- nlsList(SSbiexp, data = Cefamandole)
summary(fm1)
```

Coef *Assign Values to Coefficients*

Description

This function is generic; method functions can be written to handle specific classes of objects. Classes which already have methods for this function include all pdMat, corStruct, and varFunc classes, reStruct, and modelStruct.

Usage

```
coef(object, ...) <-  value
```

Arguments

object	any object representing a fitted model, or, by default, any object with a coef component.
...	some methods for this generic function may require additional arguments.
value	a value to be assigned to the coefficients associated with object.

Value

will depend on the method function; see the appropriate documentation.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

coef

Examples

```
## see the method function documentation
```

---

coef.corStruct *Coefficients of a corStruct Object*

Description

This method function extracts the coefficients associated with the correlation structure represented by object.

Usage

```
## S3 method for class 'corStruct':
coef(object, unconstrained, ...)
## S3 method for class 'corStruct':
coef(object, ...) <- value
```

Arguments

object	an object inheriting from class corStruct, representing a correlation structure.
unconstrained	a logical value. If TRUE the coefficients are returned in unconstrained form (the same used in the optimization algorithm). If FALSE the coefficients are returned in "natural", possibly constrained, form. Defaults to TRUE.
value	a vector with the replacement values for the coefficients associated with object. It must be a vector with the same length of coef{object} and must be given in unconstrained form.
...	some methods for this generic require additional arguments. None are used in this method.

Value

a vector with the coefficients corresponding to `object`.

SIDE EFFECTS

On the left side of an assignment, sets the values of the coefficients of `object` to `value`. `Object` must be initialized (using `Initialize`) before new values can be assigned to its coefficients.

Author(s)

Jose Pinheiro and Douglas Bates

References

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York.

See Also

`corAR1`, `corARMA`, `corCAR1`, `corCompSymm`, `corExp`, `corGaus`, `corLin`, `corRatio`, `corSpatial`, `corSpher`, `corSymm`,`Initialize`

Examples

```
cst1 <- corARMA(p = 1, q = 1)
coef(cst1)
```

---

coef.gnls *Extract gnls Coefficients*

Description

The estimated coefficients for the nonlinear model represented by `object` are extracted.

Usage

```
## S3 method for class 'gnls':
coef(object, ...)
```

Arguments

`object`	an object inheriting from class `gnls`, representing a generalized nonlinear least squares fitted model.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

a vector with the estimated coefficients for the nonlinear model represented by `object`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

gnls

Examples

```
fm1 <- gnls(weight ~ SSlogis(Time, Asym, xmid, scal), Soybean,
            weights = varPower())
coef(fm1)
```

---

coef.lme *Extract lme Coefficients*

Description

The estimated coefficients at level i are obtained by adding together the fixed effects estimates and the corresponding random effects estimates at grouping levels less or equal to i. The resulting estimates are returned as a data frame, with rows corresponding to groups and columns to coefficients. Optionally, the returned data frame may be augmented with covariates summarized over groups.

Usage

```
## S3 method for class 'lme':
coef(object, augFrame, level, data, which, FUN,
       omitGroupingFactor, subset, ...)
```

Arguments

object	an object inheriting from class lme, representing a fitted linear mixed-effects model.
augFrame	an optional logical value. If TRUE, the returned data frame is augmented with variables defined in data; else, if FALSE, only the coefficients are returned. Defaults to FALSE.
level	an optional positive integer giving the level of grouping to be used in extracting the coefficients from an object with multiple nested grouping levels. Defaults to the highest or innermost level of grouping.
data	an optional data frame with the variables to be used for augmenting the returned data frame when augFrame = TRUE. Defaults to the data frame used to fit object.
which	an optional positive integer or character vector specifying which columns of data should be used in the augmentation of the returned data frame. Defaults to all columns in data.
FUN	an optional summary function or a list of summary functions to be applied to group-varying variables, when collapsing data by groups. Group-invariant variables are always summarized by the unique value that they assume within that group. If FUN is a single function it will be applied to each non-invariant variable by group to produce the summary for that variable. If FUN is a list of

functions, the names in the list should designate classes of variables in the frame such as `ordered`, `factor`, or `numeric`. The indicated function will be applied to any group-varying variables of that class. The default functions to be used are `mean` for numeric factors, and `Mode` for both `factor` and `ordered`. The `Mode` function, defined internally in `gsummary`, returns the modal or most popular value of the variable. It is different from the `mode` function that returns the S-language mode of the variable.

`omitGroupingFactor`
: an optional logical value. When `TRUE` the grouping factor itself will be omitted from the group-wise summary of `data` but the levels of the grouping factor will continue to be used as the row names for the returned data frame. Defaults to `FALSE`.

`subset`
: an optional expression specifying a subset

`...`
: some methods for this generic require additional arguments. None are used in this method.

Value

a data frame inheriting from class `coef.lme` with the estimated coefficients at level `level` and, optionally, other covariates summarized over groups. The returned object also inherits from classes `ranef.lme` and `data.frame`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York, esp. pp. 455-457.

See Also

`lme`, `ranef.lme`, `plot.ranef.lme`, `gsummary`

Examples

```
fm1 <- lme(distance ~ age, Orthodont, random = ~ age | Subject)
coef(fm1)
coef(fm1, augFrame = TRUE)
```

---

coef.lmList *Extract lmList Coefficients*

Description

The coefficients of each `lm` object in the `object` list are extracted and organized into a data frame, with rows corresponding to the `lm` components and columns corresponding to the coefficients. Optionally, the returned data frame may be augmented with covariates summarized over the groups associated with the `lm` components.

Usage

```
## S3 method for class 'lmList':
coef(object, augFrame, data, which, FUN,
   omitGroupingFactor, ...)
```

Arguments

object	an object inheriting from class lmList, representing a list of lm objects with a common model.
augFrame	an optional logical value. If TRUE, the returned data frame is augmented with variables defined in the data frame used to produce object; else, if FALSE, only the coefficients are returned. Defaults to FALSE.
data	an optional data frame with the variables to be used for augmenting the returned data frame when augFrame = TRUE. Defaults to the data frame used to fit object.
which	an optional positive integer or character vector specifying which columns of the data frame used to produce object should be used in the augmentation of the returned data frame. Defaults to all variables in the data.
FUN	an optional summary function or a list of summary functions to be applied to group-varying variables, when collapsing the data by groups. Group-invariant variables are always summarized by the unique value that they assume within that group. If FUN is a single function it will be applied to each non-invariant variable by group to produce the summary for that variable. If FUN is a list of functions, the names in the list should designate classes of variables in the frame such as ordered, factor, or numeric. The indicated function will be applied to any group-varying variables of that class. The default functions to be used are mean for numeric factors, and Mode for both factor and ordered. The Mode function, defined internally in gsummary, returns the modal or most popular value of the variable. It is different from the mode function that returns the S-language mode of the variable.
omitGroupingFactor	an optional logical value. When TRUE the grouping factor itself will be omitted from the group-wise summary of data but the levels of the grouping factor will continue to be used as the row names for the returned data frame. Defaults to FALSE.
...	some methods for this generic require additional arguments. None are used in this method.

Value

a data frame inheriting from class coef.lmList with the estimated coefficients for each lm component of object and, optionally, other covariates summarized over the groups corresponding to the lm components. The returned object also inherits from classes ranef.lmList and data.frame.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York, esp. pp. 457-458.

See Also

lmList, fixed.effects.lmList, ranef.lmList, plot.ranef.lmList, gsummary

Examples

```
fm1 <- lmList(distance ~ age|Subject, data = Orthodont)
coef(fm1)
coef(fm1, augFrame = TRUE)
```

---

coef.modelStruct *Extract modelStruct Object Coefficients*

Description

This method function extracts the coefficients associated with each component of the modelStruct list.

Usage

```
## S3 method for class 'modelStruct':
coef(object, unconstrained, ...)
## S3 method for class 'modelStruct':
coef(object, ...) <- value
```

Arguments

object	an object inheriting from class modelStruct, representing a list of model components, such as corStruct and varFunc objects.
unconstrained	a logical value. If TRUE the coefficients are returned in unconstrained form (the same used in the optimization algorithm). If FALSE the coefficients are returned in "natural", possibly constrained, form. Defaults to TRUE.
value	a vector with the replacement values for the coefficients associated with object. It must be a vector with the same length of coef{object} and must be given in unconstrained form.
...	some methods for this generic require additional arguments. None are used in this method.

Value

a vector with all coefficients corresponding to the components of object.

SIDE EFFECTS

On the left side of an assignment, sets the values of the coefficients of object to value. Object must be initialized (using Initialize) before new values can be assigned to its coefficients.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

Initialize

Examples

```
lms1 <- lmeStruct(reStruct = reStruct(pdDiag(diag(2), ~age)),
   corStruct = corAR1(0.3))
coef(lms1)
```

---

coef.pdMat *pdMat Object Coefficients*

Description

This method function extracts the coefficients associated with the positive-definite matrix represented by `object`.

Usage

```
## S3 method for class 'pdMat':
coef(object, unconstrained, ...)
## S3 method for class 'pdMat':
coef(object, ...) <- value
```

Arguments

`object`	an object inheriting from class `pdMat`, representing a positive-definite matrix.
`unconstrained`	a logical value. If `TRUE` the coefficients are returned in unconstrained form (the same used in the optimization algorithm). If `FALSE` the upper triangular elements of the positive-definite matrix represented by `object` are returned. Defaults to `TRUE`.
`value`	a vector with the replacement values for the coefficients associated with `object`. It must be a vector with the same length of `coef{object}` and must be given in unconstrained form.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

a vector with the coefficients corresponding to `object`.

SIDE EFFECTS

On the left side of an assignment, sets the values of the coefficients of `object` to `value`.

Author(s)

Jose Pinheiro and Douglas Bates

References

Pinheiro, J.C. and Bates., D.M. (1996) "Unconstrained Parametrizations for Variance-Covariance Matrices", Statistics and Computing, 6, 289-296.

See Also

pdMat

Examples

```
coef(pdSymm(diag(3)))
```

---

coef.reStruct *reStruct Object Coefficients*

Description

This method function extracts the coefficients associated with the positive-definite matrix represented by `object`.

Usage

```
## S3 method for class 'reStruct':
coef(object, unconstrained, ...)
## S3 method for class 'reStruct':
coef(object, ...) <- value
```

Arguments

`object`	an object inheriting from class `reStruct`, representing a random effects structure and consisting of a list of `pdMat` objects.
`unconstrained`	a logical value. If `TRUE` the coefficients are returned in unconstrained form (the same used in the optimization algorithm). If `FALSE` the coefficients are returned in "natural", possibly constrained, form. Defaults to `TRUE`.
`value`	a vector with the replacement values for the coefficients associated with `object`. It must be a vector with the same length of `coef(object)` and must be given in unconstrained form.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

a vector with the coefficients corresponding to `object`.

SIDE EFFECTS

On the left side of an assignment, sets the values of the coefficients of `object` to `value`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`coef.pdMat`, `reStruct`, `pdMat`

Examples

```
rs1 <- reStruct(list(A = pdSymm(diag(1:3), form = ~Score),
  B = pdDiag(2 * diag(4), form = ~Educ)))
coef(rs1)
```

`coef.varFunc` *varFunc Object Coefficients*

Description

This method function extracts the coefficients associated with the variance function structure represented by `object`.

Usage

```
## S3 method for class 'varFunc':
coef(object, unconstrained, allCoef, ...)
## S3 method for class 'varIdent':
coef(object, ...) <- value
```

Arguments

`object`	an object inheriting from class `varFunc` representing a variance function structure.
`unconstrained`	a logical value. If `TRUE` the coefficients are returned in unconstrained form (the same used in the optimization algorithm). If `FALSE` the coefficients are returned in "natural", generally constrained form. Defaults to `TRUE`.
`allCoef`	a logical value. If `FALSE` only the coefficients which may vary during the optimization are returned. If `TRUE` all coefficients are returned. Defaults to `FALSE`.
`value`	a vector with the replacement values for the coefficients associated with `object`. It must be have the same length of `coef{object}` and must be given in unconstrained form. `Object` must be initialized before new values can be assigned to its coefficients.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

a vector with the coefficients corresponding to `object`.

SIDE EFFECTS

On the left side of an assignment, sets the values of the coefficients of `object` to `value`.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

`varFunc`

Examples

```
vf1 <- varPower(1)
coef(vf1)
coef(vf1) <- 2
```

---

collapse *Collapse According to Groups*

Description

This function is generic; method functions can be written to handle specific classes of objects. Currently, only a `groupedData` method is available.

Usage

```
collapse(object, ...)
```

Arguments

`object`	an object to be collapsed, usually a data frame.
`...`	some methods for the generic may require additional arguments.

Value

will depend on the method function used; see the appropriate documentation.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`collapse.groupedData`

Examples

```
## see the method function documentation
```

collapse.groupedData

Collapse a groupedData Object

Description

If `object` has a single grouping factor, it is returned unchanged. Else, it is summarized by the values of the `displayLevel` grouping factor (or the combination of its values and the values of the covariate indicated in `preserve`, if any is present). The collapsed data is used to produce a new `groupedData` object, with grouping factor given by the `displayLevel` factor.

Usage

```
## S3 method for class 'groupedData':
collapse(object, collapseLevel, displayLevel,
       outer, inner, preserve, FUN, subset, ...)
```

Arguments

`object`	an object inheriting from class `groupedData`, generally with multiple grouping factors.
`collapseLevel`	an optional positive integer or character string indicating the grouping level to use when collapsing the data. Level values increase from outermost to innermost grouping. Default is the highest or innermost level of grouping.
`displayLevel`	an optional positive integer or character string indicating the grouping level to use as the grouping factor for the collapsed data. Default is `collapseLevel`.
`outer`	an optional logical value or one-sided formula, indicating covariates that are outer to the `displayLevel` grouping factor. If equal to `TRUE`, the `displayLevel` element `attr(object, "outer")` is used to indicate the outer covariates. An outer covariate is invariant within the sets of rows defined by the grouping factor. Ordering of the groups is done in such a way as to preserve adjacency of groups with the same value of the outer variables. Defaults to `NULL`, meaning that no outer covariates are to be used.
`inner`	an optional logical value or one-sided formula, indicating a covariate that is inner to the `displayLevel` grouping factor. If equal to `TRUE`, `attr(object, "outer")` is used to indicate the inner covariate. An inner covariate can change within the sets of rows defined by the grouping factor. Defaults to `NULL`, meaning that no inner covariate is present.
`preserve`	an optional one-sided formula indicating a covariate whose levels should be preserved when collapsing the data according to the `collapseLevel` grouping factor. The collapsing factor is obtained by pasting together the levels of the `collapseLevel` grouping factor and the values of the covariate to be preserved. Default is `NULL`, meaning that no covariates need to be preserved.
`FUN`	an optional summary function or a list of summary functions to be used for collapsing the data. The function or functions are applied only to variables in `object` that vary within the groups defined by `collapseLevel`. Invariant variables are always summarized by group using the unique value that they assume within that group. If `FUN` is a single function it will be applied to each

non-invariant variable by group to produce the summary for that variable. If FUN is a list of functions, the names in the list should designate classes of variables in the data such as ordered, factor, or numeric. The indicated function will be applied to any non-invariant variables of that class. The default functions to be used are mean for numeric factors, and Mode for both factor and ordered. The Mode function, defined internally in gsummary, returns the modal or most popular value of the variable. It is different from the mode function that returns the S-language mode of the variable.

subset — an optional named list. Names can be either positive integers representing grouping levels, or names of grouping factors. Each element in the list is a vector indicating the levels of the corresponding grouping factor to be preserved in the collapsed data. Default is NULL, meaning that all levels are used.

... — some methods for this generic require additional arguments. None are used in this method.

Value

a groupedData object with a single grouping factor given by the displayLevel grouping factor, resulting from collapsing object over the levels of the collapseLevel grouping factor.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C. and Bates, D.M. (1997) "Future Directions in Mixed-Effects Software: Design of NLME 3.0" available at http://franz.stat.wisc.edu/pub/NLME.

See Also

groupedData, plot.nmGroupedData

Examples

```
# collapsing by Dog
collapse(Pixel, collapse = 1)  # same as collapse(Pixel, collapse = "Dog")
```

---

compareFits *Compare Fitted Objects*

Description

The columns in object1 and object2 are put together in matrices which allow direct comparison of the individual elements for each object. Missing columns in either object are replaced by NAs.

Usage

```
compareFits(object1, object2, which)
```

Arguments

object1,object2	data frames, or matrices, with the same row names, but possibly different column names. These will usually correspond to coefficients from fitted objects with a grouping structure (e.g. lme and lmList objects).
which	an optional integer or character vector indicating which columns in object1 and object2 are to be used in the returned object. Defaults to all columns.

Value

a three-dimensional array, with the third dimension given by the number of unique column names in either object1 or object2. To each column name there corresponds a matrix with as many rows as the rows in object1 and two columns, corresponding to object1 and object2. The returned object inherits from class compareFits.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

plot.compareFits, pairs.compareFits, comparePred, coef, random.effects

Examples

```
fm1 <- lmList(Orthodont)
fm2 <- lme(fm1)
compareFits(coef(fm1), coef(fm2))
```

---

comparePred *Compare Predictions*

Description

Predicted values are obtained at the specified values of primary for each object. If either object1 or object2 have a grouping structure (i.e. getGroups(object) is not NULL), predicted values are obtained for each group. When both objects determine groups, the group levels must be the same. If other covariates besides primary are used in the prediction model, their group-wise averages (numeric covariates) or most frequent values (categorical covariates) are used to obtain the predicted values. The original observations are also included in the returned object.

Usage

```
comparePred(object1, object2, primary, minimum, maximum,
    length.out, level, ...)
```

Arguments

object1,object2	fitted model objects, from which predictions can be extracted using the predict method.
primary	an optional one-sided formula specifying the primary covariate to be used to generate the augmented predictions. By default, if a covariate can be extracted from the data used to generate the objects (using getCovariate), it will be used as primary.
minimum	an optional lower limit for the primary covariate. Defaults to min(primary), after primary is evaluated in the data used in fitting object1.
maximum	an optional upper limit for the primary covariate. Defaults to max(primary), after primary is evaluated in the data used in fitting object1.
length.out	an optional integer with the number of primary covariate values at which to evaluate the predictions. Defaults to 51.
level	an optional integer specifying the desired prediction level. Levels increase from outermost to innermost grouping, with level 0 representing the population (fixed effects) predictions. Only one level can be specified. Defaults to the innermost level.
...	some methods for the generic may require additional arguments.

Value

a data frame with four columns representing, respectively, the values of the primary covariate, the groups (if object does not have a grouping structure, all elements will be 1), the predicted or observed values, and the type of value in the third column: the objects' names are used to classify the predicted values and original is used for the observed values. The returned object inherits from classes comparePred and augPred.

Note

This function is generic; method functions can be written to handle specific classes of objects. Classes which already have methods for this function include: gls, lme, and lmList.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

augPred, getGroups

Examples

```
fm1 <- lme(distance ~ age * Sex, data = Orthodont, random = ~ age)
fm2 <- update(fm1, distance ~ age)
comparePred(fm1, fm2, length.out = 2)
```

corAR1 *AR(1) Correlation Structure*

Description

This function is a constructor for the corAR1 class, representing an autocorrelation structure of order 1. Objects created using this constructor must later be initialized using the appropriate Initialize method.

Usage

```
corAR1(value, form, fixed)
```

Arguments

value	the value of the lag 1 autocorrelation, which must be between -1 and 1. Defaults to 0 (no autocorrelation).
form	a one sided formula of the form ~ t, or ~ t | g, specifying a time covariate t and, optionally, a grouping factor g. A covariate for this correlation structure must be integer valued. When a grouping factor is present in form, the correlation structure is assumed to apply only to observations within the same grouping level; observations with different grouping levels are assumed to be uncorrelated. Defaults to ~ 1, which corresponds to using the order of the observations in the data as a covariate, and no groups.
fixed	an optional logical value indicating whether the coefficients should be allowed to vary in the optimization, or kept fixed at their initial value. Defaults to FALSE, in which case the coefficients are allowed to vary.

Value

an object of class corAR1, representing an autocorrelation structure of order 1.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Box, G.E.P., Jenkins, G.M., and Reinsel G.C. (1994) "Time Series Analysis: Forecasting and Control", 3rd Edition, Holden-Day.

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer, esp. pp. 235, 397.

See Also

ACF.lme, corARMA, corClasses, Dim.corSpatial, Initialize.corStruct, summary.corStruct

Examples

```
## covariate is observation order and grouping factor is Mare
cs1 <- corAR1(0.2, form = ~ 1 | Mare)

# Pinheiro and Bates, p. 236
cs1AR1 <- corAR1(0.8, form = ~ 1 | Subject)
cs1AR1. <- Initialize(cs1AR1, data = Orthodont)
corMatrix(cs1AR1.)

# Pinheiro and Bates, p. 240
fm1Ovar.lme <- lme(follicles ~ sin(2*pi*Time) + cos(2*pi*Time),
                   data = Ovary, random = pdDiag(~sin(2*pi*Time)))
fm2Ovar.lme <- update(fm1Ovar.lme, correlation = corAR1())

# Pinheiro and Bates, pp. 255-258:  use in gls
fm1Dial.gls <-
  gls(rate ~(pressure + I(pressure^2) + I(pressure^3) + I(pressure^4))*QB,
      Dialyzer)
fm2Dial.gls <- update(fm1Dial.gls,
                 weights = varPower(form = ~ pressure))
fm3Dial.gls <- update(fm2Dial.gls,
                    corr = corAR1(0.771, form = ~ 1 | Subject))

# Pinheiro and Bates use in nlme:
# from p. 240 needed on p. 396
fm1Ovar.lme <- lme(follicles ~ sin(2*pi*Time) + cos(2*pi*Time),
                   data = Ovary, random = pdDiag(~sin(2*pi*Time)))
fm5Ovar.lme <- update(fm1Ovar.lme,
                corr = corARMA(p = 1, q = 1))
# p. 396
fm1Ovar.nlme <- nlme(follicles~
     A+B*sin(2*pi*w*Time)+C*cos(2*pi*w*Time),
   data=Ovary, fixed=A+B+C+w~1,
   random=pdDiag(A+B+w~1),
   start=c(fixef(fm5Ovar.lme), 1) )
# p. 397
fm2Ovar.nlme <- update(fm1Ovar.nlme,
         corr=corAR1(0.311) )
```

`corARMA` *ARMA(p,q) Correlation Structure*

Description

This function is a constructor for the `corARMA` class, representing an autocorrelation-moving average correlation structure of order (p, q). Objects created using this constructor must later be initialized using the appropriate `Initialize` method.

Usage

```
corARMA(value, form, p, q, fixed)
```

Arguments

value	a vector with the values of the autoregressive and moving average parameters, which must have length p + q and all elements between -1 and 1. Defaults to a vector of zeros, corresponding to uncorrelated observations.
form	a one sided formula of the form ~ t, or ~ t | g, specifying a time covariate t and, optionally, a grouping factor g. A covariate for this correlation structure must be integer valued. When a grouping factor is present in form, the correlation structure is assumed to apply only to observations within the same grouping level; observations with different grouping levels are assumed to be uncorrelated. Defaults to ~ 1, which corresponds to using the order of the observations in the data as a covariate, and no groups.
p, q	non-negative integers specifying respectively the autoregressive order and the moving average order of the ARMA structure. Both default to 0.
fixed	an optional logical value indicating whether the coefficients should be allowed to vary in the optimization, or kept fixed at their initial value. Defaults to FALSE, in which case the coefficients are allowed to vary.

Value

an object of class corARMA, representing an autocorrelation-moving average correlation structure.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Box, G.E.P., Jenkins, G.M., and Reinsel G.C. (1994) "Time Series Analysis: Forecasting and Control", 3rd Edition, Holden-Day.

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer, esp. pp. 236, 397.

See Also

corAR1, corClasses Initialize.corStruct, summary.corStruct

Examples

```
## ARMA(1,2) structure, with observation order as a covariate and
## Mare as grouping factor
cs1 <- corARMA(c(0.2, 0.3, -0.1), form = ~ 1 | Mare, p = 1, q = 2)

# Pinheiro and Bates, p. 237
cs1ARMA <- corARMA(0.4, form = ~ 1 | Subject, q = 1)
cs1ARMA <- Initialize(cs1ARMA, data = Orthodont)
corMatrix(cs1ARMA)

cs2ARMA <- corARMA(c(0.8, 0.4), form = ~ 1 | Subject, p=1, q=1)
cs2ARMA <- Initialize(cs2ARMA, data = Orthodont)
corMatrix(cs2ARMA)

# Pinheiro and Bates use in nlme:
# from p. 240 needed on p. 396
```

```
fm1Ovar.lme <- lme(follicles ~ sin(2*pi*Time) + cos(2*pi*Time),
                   data = Ovary, random = pdDiag(~sin(2*pi*Time)))
fm5Ovar.lme <- update(fm1Ovar.lme,
                corr = corARMA(p = 1, q = 1))
# p. 396
fm1Ovar.nlme <- nlme(follicles~
     A+B*sin(2*pi*w*Time)+C*cos(2*pi*w*Time),
   data=Ovary, fixed=A+B+C+w~1,
   random=pdDiag(A+B+w~1),
   start=c(fixef(fm5Ovar.lme), 1) )
# p. 397
fm3Ovar.nlme <- update(fm1Ovar.nlme,
         corr=corARMA(p=0, q=2) )
```

`corCAR1` *Continuous AR(1) Correlation Structure*

Description

This function is a constructor for the `corCAR1` class, representing an autocorrelation structure of order 1, with a continuous time covariate. Objects created using this constructor must be later initialized using the appropriate `Initialize` method.

Usage

```
corCAR1(value, form, fixed)
```

Arguments

`value`	the correlation between two observations one unit of time apart. Must be between 0 and 1. Defaults to 0.2.
`form`	a one sided formula of the form `~ t`, or `~ t	g`, specifying a time covariate `t` and, optionally, a grouping factor `g`. Covariates for this correlation structure need not be integer valued. When a grouping factor is present in `form`, the correlation structure is assumed to apply only to observations within the same grouping level; observations with different grouping levels are assumed to be uncorrelated. Defaults to `~ 1`, which corresponds to using the order of the observations in the data as a covariate, and no groups.
`fixed`	an optional logical value indicating whether the coefficients should be allowed to vary in the optimization, or kept fixed at their initial value. Defaults to `FALSE`, in which case the coefficients are allowed to vary.

Value

an object of class `corCAR1`, representing an autocorrelation structure of order 1, with a continuous time covariate.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Box, G.E.P., Jenkins, G.M., and Reinsel G.C. (1994) "Time Series Analysis: Forecasting and Control", 3rd Edition, Holden-Day.

Jones, R.H. (1993) "Longitudinal Data with Serial Correlation: A State-space Approach", Chapman and Hall.

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer, esp. pp. 236, 243.

See Also

corClasses, Initialize.corStruct, summary.corStruct

Examples

```
## covariate is Time and grouping factor is Mare
cs1 <- corCAR1(0.2, form = ~ Time | Mare)

# Pinheiro and Bates, pp. 240, 243
fm1Ovar.lme <- lme(follicles ~
           sin(2*pi*Time) + cos(2*pi*Time),
   data = Ovary, random = pdDiag(~sin(2*pi*Time)))
fm4Ovar.lme <- update(fm1Ovar.lme,
          correlation = corCAR1(form = ~Time))
```

---

corClasses *Correlation Structure Classes*

Description

Standard classes of correlation structures (corStruct) available in the nlme library.

Value

Available standard classes:

corAR1	autoregressive process of order 1.
corARMA	autoregressive moving average process, with arbitrary orders for the autoregressive and moving average components.
corCAR1	continuous autoregressive process (AR(1) process for a continuous time covariate).
corCompSymm	compound symmetry structure corresponding to a constant correlation.
corExp	exponential spatial correlation.
corGaus	Gaussian spatial correlation.
corLin	linear spatial correlation.
corRatio	Rational quadratics spatial correlation.
corSpher	spherical spatial correlation.
corSymm	general correlation matrix, with no additional structure.

Note

Users may define their own corStruct classes by specifying a constructor function and, at a minimum, methods for the functions corMatrix and coef. For examples of these functions, see the methods for classes corSymm and corAR1.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

corAR1, corARMA, corCAR1, corCompSymm, corExp, corGaus, corLin, corRatio, corSpher, corSymm, summary.corStruct

---

corCompSymm *Compound Symmetry Correlation Structure*

Description

This function is a constructor for the corCompSymm class, representing a compound symmetry structure corresponding to uniform correlation. Objects created using this constructor must later be initialized using the appropriate Initialize method.

Usage

```
corCompSymm(value, form, fixed)
```

Arguments

value	the correlation between any two correlated observations. Defaults to 0.
form	a one sided formula of the form ~ t, or ~ t | g, specifying a time covariate t and, optionally, a grouping factor g. When a grouping factor is present in form, the correlation structure is assumed to apply only to observations within the same grouping level; observations with different grouping levels are assumed to be uncorrelated. Defaults to ~ 1, which corresponds to using the order of the observations in the data as a covariate, and no groups.
fixed	an optional logical value indicating whether the coefficients should be allowed to vary in the optimization, or kept fixed at their initial value. Defaults to FALSE, in which case the coefficients are allowed to vary.

Value

an object of class corCompSymm, representing a compound symmetry correlation structure.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Milliken, G. A. and Johnson, D. E. (1992) "Analysis of Messy Data, Volume I: Designed Experiments", Van Nostrand Reinhold.

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer, esp. pp. 233-234.

See Also

corClasses, Initialize.corStruct, summary.corStruct

Examples

```
## covariate is observation order and grouping factor is Subject
cs1 <- corCompSymm(0.5, form = ~ 1 | Subject)

# Pinheiro and Bates, pp. 222-225
fm1BW.lme <- lme(weight ~ Time * Diet, BodyWeight,
                 random = ~ Time)
# p. 223
fm2BW.lme <- update(fm1BW.lme, weights = varPower())
# p. 225
cs1CompSymm <- corCompSymm(value = 0.3, form = ~ 1 | Subject)
cs2CompSymm <- corCompSymm(value = 0.3, form = ~ age | Subject)
cs1CompSymm <- Initialize(cs1CompSymm, data = Orthodont)
corMatrix(cs1CompSymm)
```

---

corExp *Exponential Correlation Structure*

Description

This function is a constructor for the corExp class, representing an exponential spatial correlation structure. Letting d denote the range and n denote the nugget effect, the correlation between two observations a distance r apart is $\exp(-r/d)$ when no nugget effect is present and $(1-n)\exp(-r/d)$ when a nugget effect is assumed. Objects created using this constructor must later be initialized using the appropriate Initialize method.

Usage

```
corExp(value, form, nugget, metric, fixed)
```

Arguments

value	an optional vector with the parameter values in constrained form. If nugget is FALSE, value can have only one element, corresponding to the "range" of the exponential correlation structure, which must be greater than zero. If nugget is TRUE, meaning that a nugget effect is present, value can contain one or two elements, the first being the "range" and the second the "nugget effect" (one minus the correlation between two observations taken arbitrarily close together); the first must be greater than zero and the second must be between zero and one.

Defaults to `numeric(0)`, which results in a range of 90% of the minimum distance and a nugget effect of 0.1 being assigned to the parameters when `object` is initialized.

`form` a one sided formula of the form `~ S1+...+Sp`, or `~ S1+...+Sp | g`, specifying spatial covariates `S1` through `Sp` and, optionally, a grouping factor `g`. When a grouping factor is present in `form`, the correlation structure is assumed to apply only to observations within the same grouping level; observations with different grouping levels are assumed to be uncorrelated. Defaults to `~ 1`, which corresponds to using the order of the observations in the data as a covariate, and no groups.

`nugget` an optional logical value indicating whether a nugget effect is present. Defaults to `FALSE`.

`metric` an optional character string specifying the distance metric to be used. The currently available options are `"euclidean"` for the root sum-of-squares of distances; `"maximum"` for the maximum difference; and `"manhattan"` for the sum of the absolute differences. Partial matching of arguments is used, so only the first three characters need to be provided. Defaults to `"euclidean"`.

`fixed` an optional logical value indicating whether the coefficients should be allowed to vary in the optimization, or kept fixed at their initial value. Defaults to `FALSE`, in which case the coefficients are allowed to vary.

Value

an object of class `corExp`, also inheriting from class `corSpatial`, representing an exponential spatial correlation structure.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Cressie, N.A.C. (1993), "Statistics for Spatial Data", J. Wiley & Sons. Venables, W.N. and Ripley, B.D. (1997) "Modern Applied Statistics with S-plus", 2nd Edition, Springer-Verlag.

Littel, Milliken, Stroup, and Wolfinger (1996) "SAS Systems for Mixed Models", SAS Institute.

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer, esp. p. 238.

See Also

`corClasses`, `Initialize.corStruct`, `summary.corStruct`, `dist`

Examples

```
sp1 <- corExp(form = ~ x + y + z)

# Pinheiro and Bates, p. 238
spatDat <- data.frame(x = (0:4)/4, y = (0:4)/4)

cs1Exp <- corExp(1, form = ~ x + y)
cs1Exp <- Initialize(cs1Exp, spatDat)
corMatrix(cs1Exp)
```

```
cs2Exp <- corExp(1, form = ~ x + y, metric = "man")
cs2Exp <- Initialize(cs2Exp, spatDat)
corMatrix(cs2Exp)

cs3Exp <- corExp(c(1, 0.2), form = ~ x + y,
                 nugget = TRUE)
cs3Exp <- Initialize(cs3Exp, spatDat)
corMatrix(cs3Exp)

# example lme(..., corExp ...)
# Pinheiro and Bates, pp. 222-247
# p. 222
options(contrasts = c("contr.treatment", "contr.poly"))
fm1BW.lme <- lme(weight ~ Time * Diet, BodyWeight,
                 random = ~ Time)
# p. 223
fm2BW.lme <- update(fm1BW.lme, weights = varPower())
# p. 246
fm3BW.lme <- update(fm2BW.lme,
           correlation = corExp(form = ~ Time))
# p. 247
fm4BW.lme <-
      update(fm3BW.lme, correlation = corExp(form =  ~ Time,
                        nugget = TRUE))
anova(fm3BW.lme, fm4BW.lme)
```

`corFactor` *Factor of a Correlation Matrix*

Description

This function is generic; method functions can be written to handle specific classes of objects. Classes which already have methods for this function include all `corStruct` classes.

Usage

```
corFactor(object, ...)
```

Arguments

`object`	an object from which a correlation matrix can be extracted.
`...`	some methods for this generic function require additional arguments.

Value

will depend on the method function used; see the appropriate documentation.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

corFactor.corStruct, recalc.corStruct

Examples

```
## see the method function documentation
```

---

corFactor.corStruct

Factor of a corStruct Object Matrix

Description

This method function extracts a transpose inverse square-root factor, or a series of transpose inverse square-root factors, of the correlation matrix, or list of correlation matrices, represented by `object`. Letting Σ denote a correlation matrix, a square-root factor of Σ is any square matrix L such that $\Sigma = L'L$. This method extracts L^{-t}.

Usage

```
## S3 method for class 'corStruct':
corFactor(object, ...)
```

Arguments

`object`	an object inheriting from class `corStruct` representing a correlation structure, which must have been initialized (using `Initialize`).
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

If the correlation structure does not include a grouping factor, the returned value will be a vector with a transpose inverse square-root factor of the correlation matrix associated with `object` stacked column-wise. If the correlation structure includes a grouping factor, the returned value will be a vector with transpose inverse square-root factors of the correlation matrices for each group, stacked by group and stacked column-wise within each group.

Note

This method function is used intensively in optimization algorithms and its value is returned as a vector for efficiency reasons. The `corMatrix` method function can be used to obtain transpose inverse square-root factors in matrix form.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

corFactor, corMatrix.corStruct, recalc.corStruct, Initialize.corStruct

Examples

```
cs1 <- corAR1(form = ~1 | Subject)
cs1 <- Initialize(cs1, data = Orthodont)
corFactor(cs1)
```

`corGaus` *Gaussian Correlation Structure*

Description

This function is a constructor for the `corGaus` class, representing a Gaussian spatial correlation structure. Letting d denote the range and n denote the nugget effect, the correlation between two observations a distance r apart is $\exp(-(r/d)^2)$ when no nugget effect is present and $(1-n)\exp(-(r/d)^2)$ when a nugget effect is assumed. Objects created using this constructor must later be initialized using the appropriate ‘ `Initialize` method.

Usage

```
corGaus(value, form, nugget, metric, fixed)
```

Arguments

`value`	an optional vector with the parameter values in constrained form. If `nugget` is `FALSE`, `value` can have only one element, corresponding to the "range" of the Gaussian correlation structure, which must be greater than zero. If `nugget` is `TRUE`, meaning that a nugget effect is present, `value` can contain one or two elements, the first being the "range" and the second the "nugget effect" (one minus the correlation between two observations taken arbitrarily close together); the first must be greater than zero and the second must be between zero and one. Defaults to `numeric(0)`, which results in a range of 90% of the minimum distance and a nugget effect of 0.1 being assigned to the parameters when `object` is initialized.
`form`	a one sided formula of the form `~ S1+...+Sp`, or `~ S1+...+Sp | g`, specifying spatial covariates `S1` through `Sp` and, optionally, a grouping factor `g`. When a grouping factor is present in `form`, the correlation structure is assumed to apply only to observations within the same grouping level; observations with different grouping levels are assumed to be uncorrelated. Defaults to `~ 1`, which corresponds to using the order of the observations in the data as a covariate, and no groups.
`nugget`	an optional logical value indicating whether a nugget effect is present. Defaults to `FALSE`.
`metric`	an optional character string specifying the distance metric to be used. The currently available options are `"euclidean"` for the root sum-of-squares of distances; `"maximum"` for the maximum difference; and `"manhattan"` for the sum of the absolute differences. Partial matching of arguments is used, so only the first three characters need to be provided. Defaults to `"euclidean"`.
`fixed`	an optional logical value indicating whether the coefficients should be allowed to vary in the optimization, or kept fixed at their initial value. Defaults to `FALSE`, in which case the coefficients are allowed to vary.

Value

an object of class `corGaus`, also inheriting from class `corSpatial`, representing a Gaussian spatial correlation structure.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Cressie, N.A.C. (1993), "Statistics for Spatial Data", J. Wiley & Sons.

Venables, W.N. and Ripley, B.D. (1997) "Modern Applied Statistics with S-plus", 2nd Edition, Springer-Verlag.

Littel, Milliken, Stroup, and Wolfinger (1996) "SAS Systems for Mixed Models", SAS Institute.

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

`Initialize.corStruct`, `summary.corStruct`, `dist`

Examples

```
sp1 <- corGaus(form = ~ x + y + z)

# example lme(..., corGaus ...)
# Pinheiro and Bates, pp. 222-249
fm1BW.lme <- lme(weight ~ Time * Diet, BodyWeight,
                 random = ~ Time)
# p. 223
fm2BW.lme <- update(fm1BW.lme, weights = varPower())
# p 246
fm3BW.lme <- update(fm2BW.lme,
           correlation = corExp(form = ~ Time))
# p. 249
fm8BW.lme <- update(fm3BW.lme, correlation = corGaus(form = ~ Time))
```

---

corLin *Linear Correlation Structure*

Description

This function is a constructor for the `corLin` class, representing a linear spatial correlation structure. Letting d denote the range and n denote the nugget effect, the correlation between two observations a distance $r < d$ apart is $1 - (r/d)$ when no nugget effect is present and $(1 - n)(1 - (r/d))$ when a nugget effect is assumed. If $r \geq d$ the correlation is zero. Objects created using this constructor must later be initialized using the appropriate `Initialize` method.

Usage

```
corLin(value, form, nugget, metric, fixed)
```

Arguments

value	an optional vector with the parameter values in constrained form. If nugget is FALSE, value can have only one element, corresponding to the "range" of the linear correlation structure, which must be greater than zero. If nugget is TRUE, meaning that a nugget effect is present, value can contain one or two elements, the first being the "range" and the second the "nugget effect" (one minus the correlation between two observations taken arbitrarily close together); the first must be greater than zero and the second must be between zero and one. Defaults to numeric(0), which results in a range of 90% of the minimum distance and a nugget effect of 0.1 being assigned to the parameters when object is initialized.
form	a one sided formula of the form ~ S1+...+Sp, or ~ S1+...+Sp | g, specifying spatial covariates S1 through Sp and, optionally, a grouping factor g. When a grouping factor is present in form, the correlation structure is assumed to apply only to observations within the same grouping level; observations with different grouping levels are assumed to be uncorrelated. Defaults to ~ 1, which corresponds to using the order of the observations in the data as a covariate, and no groups.
nugget	an optional logical value indicating whether a nugget effect is present. Defaults to FALSE.
metric	an optional character string specifying the distance metric to be used. The currently available options are "euclidean" for the root sum-of-squares of distances; "maximum" for the maximum difference; and "manhattan" for the sum of the absolute differences. Partial matching of arguments is used, so only the first three characters need to be provided. Defaults to "euclidean".
fixed	an optional logical value indicating whether the coefficients should be allowed to vary in the optimization, or kept fixed at their initial value. Defaults to FALSE, in which case the coefficients are allowed to vary.

Value

an object of class corLin, also inheriting from class corSpatial, representing a linear spatial correlation structure.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Cressie, N.A.C. (1993), "Statistics for Spatial Data", J. Wiley & Sons.

Venables, W.N. and Ripley, B.D. (1997) "Modern Applied Statistics with S-plus", 2nd Edition, Springer-Verlag.

Littel, Milliken, Stroup, and Wolfinger (1996) "SAS Systems for Mixed Models", SAS Institute.

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

Initialize.corStruct, summary.corStruct, dist

Examples

```
sp1 <- corLin(form = ~ x + y)

# example lme(..., corLin ...)
# Pinheiro and Bates, pp. 222-249
fm1BW.lme <- lme(weight ~ Time * Diet, BodyWeight,
                 random = ~ Time)
# p. 223
fm2BW.lme <- update(fm1BW.lme, weights = varPower())
# p 246
fm3BW.lme <- update(fm2BW.lme,
           correlation = corExp(form = ~ Time))
# p. 249
fm7BW.lme <- update(fm3BW.lme, correlation = corLin(form = ~ Time))
```

---

corMatrix *Extract Correlation Matrix*

Description

This function is generic; method functions can be written to handle specific classes of objects. Classes which already have methods for this function include all `corStruct` classes.

Usage

```
corMatrix(object, ...)
```

Arguments

`object`	an object for which a correlation matrix can be extracted.
`...`	some methods for this generic function require additional arguments.

Value

will depend on the method function used; see the appropriate documentation.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`corMatrix.corStruct`, `corMatrix.pdMat`

Examples

```
## see the method function documentation
```

corMatrix.corStruct
Matrix of a corStruct Object

Description

This method function extracts the correlation matrix (or its transpose inverse square-root factor), or list of correlation matrices (or their transpose inverse square-root factors) corresponding to `covariate` and `object`. Letting Σ denote a correlation matrix, a square-root factor of Σ is any square matrix L such that $\Sigma = L'L$. When `corr = FALSE`, this method extracts L^{-t}.

Usage

```
## S3 method for class 'corStruct':
corMatrix(object, covariate, corr, ...)
```

Arguments

`object`	an object inheriting from class `corStruct` representing a correlation structure.
`covariate`	an optional covariate vector (matrix), or list of covariate vectors (matrices), at which values the correlation matrix, or list of correlation matrices, are to be evaluated. Defaults to `getCovariate(object)`.
`corr`	a logical value. If `TRUE` the function returns the correlation matrix, or list of correlation matrices, represented by `object`. If `FALSE` the function returns a transpose inverse square-root of the correlation matrix, or a list of transpose inverse square-root factors of the correlation matrices.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

If `covariate` is a vector (matrix), the returned value will be an array with the corresponding correlation matrix (or its transpose inverse square-root factor). If the `covariate` is a list of vectors (matrices), the returned value will be a list with the correlation matrices (or their transpose inverse square-root factors) corresponding to each component of `covariate`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

`corFactor.corStruct`, `Initialize.corStruct`

Examples

```
cs1 <- corAR1(0.3)
corMatrix(cs1, covariate = 1:4)
corMatrix(cs1, covariate = 1:4, corr = FALSE)

# Pinheiro and Bates, p. 225
cs1CompSymm <- corCompSymm(value = 0.3, form = ~ 1 | Subject)
cs1CompSymm <- Initialize(cs1CompSymm, data = Orthodont)
corMatrix(cs1CompSymm)

# Pinheiro and Bates, p. 226
cs1Symm <- corSymm(value = c(0.2, 0.1, -0.1, 0, 0.2, 0),
                   form = ~ 1 | Subject)
cs1Symm <- Initialize(cs1Symm, data = Orthodont)
corMatrix(cs1Symm)

# Pinheiro and Bates, p. 236
cs1AR1 <- corAR1(0.8, form = ~ 1 | Subject)
cs1AR1 <- Initialize(cs1AR1, data = Orthodont)
corMatrix(cs1AR1)

# Pinheiro and Bates, p. 237
cs1ARMA <- corARMA(0.4, form = ~ 1 | Subject, q = 1)
cs1ARMA <- Initialize(cs1ARMA, data = Orthodont)
corMatrix(cs1ARMA)

# Pinheiro and Bates, p. 238
spatDat <- data.frame(x = (0:4)/4, y = (0:4)/4)
cs1Exp <- corExp(1, form = ~ x + y)
cs1Exp <- Initialize(cs1Exp, spatDat)
corMatrix(cs1Exp)
```

---

corMatrix.pdMat *Extract Correlation Matrix from a pdMat Object*

Description

The correlation matrix corresponding to the positive-definite matrix represented by `object` is obtained.

Usage

```
## S3 method for class 'pdMat':
corMatrix(object, ...)
```

Arguments

`object`	an object inheriting from class `pdMat`, representing a positive definite matrix.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

the correlation matrix corresponding to the positive-definite matrix represented by `object`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

as.matrix.pdMat, pdMatrix

Examples

```
pd1 <- pdSymm(diag(1:4))
corMatrix(pd1)
```

---

corMatrix.reStruct *Extract Correlation Matrix from Components of an reStruct Object*

Description

This method function extracts the correlation matrices corresponding to the pdMat elements of object.

Usage

```
## S3 method for class 'reStruct':
corMatrix(object, ...)
```

Arguments

object	an object inheriting from class reStruct, representing a random effects structure and consisting of a list of pdMat objects.
...	some methods for this generic require additional arguments. None are used in this method.

Value

a list with components given by the correlation matrices corresponding to the elements of object.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

as.matrix.reStruct, corMatrix, reStruct, pdMat

Examples

```
rs1 <- reStruct(pdSymm(diag(3), ~age+Sex, data = Orthodont))
corMatrix(rs1)
```

corNatural *General correlation in natural parameterization*

Description

This function is a constructor for the corNatural class, representing a general correlation structure in the "natural" parameterization, which is described under pdNatural. Objects created using this constructor must later be initialized using the appropriate Initialize method.

Usage

```
corNatural(value, form, fixed)
```

Arguments

value	an optional vector with the parameter values. Default is numeric(0), which results in a vector of zeros of appropriate dimension being assigned to the parameters when object is initialized (corresponding to an identity correlation structure).
form	a one sided formula of the form ~ t, or ~ t | g, specifying a time covariate t and, optionally, a grouping factor g. A covariate for this correlation structure must be integer valued. When a grouping factor is present in form, the correlation structure is assumed to apply only to observations within the same grouping level; observations with different grouping levels are assumed to be uncorrelated. Defaults to ~ 1, which corresponds to using the order of the observations in the data as a covariate, and no groups.
fixed	an optional logical value indicating whether the coefficients should be allowed to vary in the optimization, or kept fixed at their initial value. Defaults to FALSE, in which case the coefficients are allowed to vary.

Value

an object of class corNatural representing a general correlation structure.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

Initialize.corNatural, pdNatural, summary.corNatural

Examples

```
## covariate is observation order and grouping factor is Subject
cs1 <- corNatural(form = ~ 1 | Subject)
```

corRatio *Rational Quadratic Correlation Structure*

Description

This function is a constructor for the corRatio class, representing a rational quadratic spatial correlation structure. Letting d denote the range and n denote the nugget effect, the correlation between two observations a distance r apart is $(r/d)^2/(1+(r/d)^2)$ when no nugget effect is present and $(1-n)(r/d)^2/(1+(r/d)^2)$ when a nugget effect is assumed. Objects created using this constructor need to be later initialized using the appropriate Initialize method.

Usage

```
corRatio(value, form, nugget, metric, fixed)
```

Arguments

value	an optional vector with the parameter values in constrained form. If nugget is FALSE, value can have only one element, corresponding to the "range" of the rational quadratic correlation structure, which must be greater than zero. If nugget is TRUE, meaning that a nugget effect is present, value can contain one or two elements, the first being the "range" and the second the "nugget effect" (one minus the correlation between two observations taken arbitrarily close together); the first must be greater than zero and the second must be between zero and one. Defaults to numeric(0), which results in a range of 90% of the minimum distance and a nugget effect of 0.1 being assigned to the parameters when object is initialized.
form	a one sided formula of the form ~ S1+...+Sp, or ~ S1+...+Sp | g, specifying spatial covariates S1 through Sp and, optionally, a grouping factor g. When a grouping factor is present in form, the correlation structure is assumed to apply only to observations within the same grouping level; observations with different grouping levels are assumed to be uncorrelated. Defaults to ~ 1, which corresponds to using the order of the observations in the data as a covariate, and no groups.
nugget	an optional logical value indicating whether a nugget effect is present. Defaults to FALSE.
metric	an optional character string specifying the distance metric to be used. The currently available options are "euclidean" for the root sum-of-squares of distances; "maximum" for the maximum difference; and "manhattan" for the sum of the absolute differences. Partial matching of arguments is used, so only the first three characters need to be provided. Defaults to "euclidean".
fixed	an optional logical value indicating whether the coefficients should be allowed to vary in the optimization, or kept fixed at their initial value. Defaults to FALSE, in which case the coefficients are allowed to vary.

Value

an object of class corRatio, also inheriting from class corSpatial, representing a rational quadratic spatial correlation structure.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Cressie, N.A.C. (1993), "Statistics for Spatial Data", J. Wiley & Sons.

Venables, W.N. and Ripley, B.D. (1997) "Modern Applied Statistics with S-plus", 2nd Edition, Springer-Verlag.

Littel, Milliken, Stroup, and Wolfinger (1996) "SAS Systems for Mixed Models", SAS Institute.

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

`Initialize.corStruct`, `summary.corStruct`, `dist`

Examples

```
sp1 <- corRatio(form = ~ x + y + z)

# example lme(..., corRatio ...)
# Pinheiro and Bates, pp. 222-249
fm1BW.lme <- lme(weight ~ Time * Diet, BodyWeight,
                 random = ~ Time)
# p. 223
fm2BW.lme <- update(fm1BW.lme, weights = varPower())
# p 246
fm3BW.lme <- update(fm2BW.lme,
          correlation = corExp(form = ~ Time))
# p. 249
fm5BW.lme <- update(fm3BW.lme, correlation =
                 corRatio(form = ~ Time))

# example gls(..., corRatio ...)
# Pinheiro and Bates, pp. 261, 263
fm1Wheat2 <- gls(yield ~ variety - 1, Wheat2)
# p. 263
fm3Wheat2 <- update(fm1Wheat2, corr =
    corRatio(c(12.5, 0.2),
       form = ~ latitude + longitude,
             nugget = TRUE))
```

`corSpatial` *Spatial Correlation Structure*

Description

This function is a constructor for the `corSpatial` class, representing a spatial correlation structure. This class is "virtual", having four "real" classes, corresponding to specific spatial correlation structures, associated with it: `corExp`, `corGaus`, `corLin`, `corRatio`, and `corSpher`. The returned object will inherit from one of these "real" classes, determined by the `type` argument, and from the "virtual" `corSpatial` class. Objects created using this constructor must later be initialized using the appropriate `Initialize` method.

Usage

```
corSpatial(value, form, nugget, type, metric, fixed)
```

Arguments

value	an optional vector with the parameter values in constrained form. If nugget is FALSE, value can have only one element, corresponding to the "range" of the spatial correlation structure, which must be greater than zero. If nugget is TRUE, meaning that a nugget effect is present, value can contain one or two elements, the first being the "range" and the second the "nugget effect" (one minus the correlation between two observations taken arbitrarily close together); the first must be greater than zero and the second must be between zero and one. Defaults to numeric(0), which results in a range of 90% of the minimum distance and a nugget effect of 0.1 being assigned to the parameters when object is initialized.
form	a one sided formula of the form ~ S1+...+Sp, or ~ S1+...+Sp | g, specifying spatial covariates S1 through Sp and, optionally, a grouping factor g. When a grouping factor is present in form, the correlation structure is assumed to apply only to observations within the same grouping level; observations with different grouping levels are assumed to be uncorrelated. Defaults to ~ 1, which corresponds to using the order of the observations in the data as a covariate, and no groups.
nugget	an optional logical value indicating whether a nugget effect is present. Defaults to FALSE.
type	an optional character string specifying the desired type of correlation structure. Available types include "spherical", "exponential", "gaussian", "linear", and "rational". See the documentation on the functions corSpher, corExp, corGaus, corLin, and corRatio for a description of these correlation structures. Partial matching of arguments is used, so only the first character needs to be provided.Defaults to "spherical".
metric	an optional character string specifying the distance metric to be used. The currently available options are "euclidean" for the root sum-of-squares of distances; "maximum" for the maximum difference; and "manhattan" for the sum of the absolute differences. Partial matching of arguments is used, so only the first three characters need to be provided. Defaults to "euclidean".
fixed	an optional logical value indicating whether the coefficients should be allowed to vary in the optimization, or kept fixed at their initial value. Defaults to FALSE, in which case the coefficients are allowed to vary.

Value

an object of class determined by the type argument and also inheriting from class corSpatial, representing a spatial correlation structure.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Cressie, N.A.C. (1993), "Statistics for Spatial Data", J. Wiley & Sons.

Venables, W.N. and Ripley, B.D. (1997) "Modern Applied Statistics with S-plus", 2nd Edition, Springer-Verlag.

Littel, Milliken, Stroup, and Wolfinger (1996) "SAS Systems for Mixed Models", SAS Institute.

See Also

corExp, corGaus, corLin, corRatio, corSpher, Initialize.corStruct, summary.corStruct, dist

Examples

```
sp1 <- corSpatial(form = ~ x + y + z, type = "g", metric = "man")
```

---

corSpher *Spherical Correlation Structure*

Description

This function is a constructor for the corSpher class, representing a spherical spatial correlation structure. Letting d denote the range and n denote the nugget effect, the correlation between two observations a distance $r < d$ apart is $1-1.5(r/d)+0.5(r/d)^3$ when no nugget effect is present and $(1-n)(1-1.5(r/d)+0.5(r/d)^3)$ when a nugget effect is assumed. If $r \geq d$ the correlation is zero. Objects created using this constructor must later be initialized using the appropriate Initialize method.

Usage

```
corSpher(value, form, nugget, metric, fixed)
```

Arguments

value	an optional vector with the parameter values in constrained form. If nugget is FALSE, value can have only one element, corresponding to the "range" of the spherical correlation structure, which must be greater than zero. If nugget is TRUE, meaning that a nugget effect is present, value can contain one or two elements, the first being the "range" and the second the "nugget effect" (one minus the correlation between two observations taken arbitrarily close together); the first must be greater than zero and the second must be between zero and one. Defaults to numeric(0), which results in a range of 90% of the minimum distance and a nugget effect of 0.1 being assigned to the parameters when object is initialized.
form	a one sided formula of the form ~ S1+...+Sp, or ~ S1+...+Sp | g, specifying spatial covariates S1 through Sp and, optionally, a grouping factor g. When a grouping factor is present in form, the correlation structure is assumed to apply only to observations within the same grouping level; observations with different grouping levels are assumed to be uncorrelated. Defaults to ~ 1, which corresponds to using the order of the observations in the data as a covariate, and no groups.

nugget	an optional logical value indicating whether a nugget effect is present. Defaults to FALSE.
metric	an optional character string specifying the distance metric to be used. The currently available options are "euclidean" for the root sum-of-squares of distances; "maximum" for the maximum difference; and "manhattan" for the sum of the absolute differences. Partial matching of arguments is used, so only the first three characters need to be provided. Defaults to "euclidean".
fixed	an optional logical value indicating whether the coefficients should be allowed to vary in the optimization, or kept fixed at their initial value. Defaults to FALSE, in which case the coefficients are allowed to vary.

Value

an object of class corSpher, also inheriting from class corSpatial, representing a spherical spatial correlation structure.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Cressie, N.A.C. (1993), "Statistics for Spatial Data", J. Wiley & Sons.

Venables, W.N. and Ripley, B.D. (1997) "Modern Applied Statistics with S-plus", 2nd Edition, Springer-Verlag.

Littel, Milliken, Stroup, and Wolfinger (1996) "SAS Systems for Mixed Models", SAS Institute.

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

Initialize.corStruct, summary.corStruct, dist

Examples

```
sp1 <- corSpher(form = ~ x + y)

# example lme(..., corSpher ...)
# Pinheiro and Bates, pp. 222-249
fm1BW.lme <- lme(weight ~ Time * Diet, BodyWeight,
                 random = ~ Time)
# p. 223
fm2BW.lme <- update(fm1BW.lme, weights = varPower())
# p 246
fm3BW.lme <- update(fm2BW.lme,
           correlation = corExp(form = ~ Time))
# p. 249
fm6BW.lme <- update(fm3BW.lme,
          correlation = corSpher(form = ~ Time))

# example gls(..., corSpher ...)
# Pinheiro and Bates, pp. 261, 263
fm1Wheat2 <- gls(yield ~ variety - 1, Wheat2)
# p. 262
fm2Wheat2 <- update(fm1Wheat2, corr =
```

```
corSpher(c(28, 0.2),
  form = ~ latitude + longitude, nugget = TRUE))
```

`corSymm` *General Correlation Structure*

Description

This function is a constructor for the `corSymm` class, representing a general correlation structure. The internal representation of this structure, in terms of unconstrained parameters, uses the spherical parametrization defined in Pinheiro and Bates (1996). Objects created using this constructor must later be initialized using the appropriate `Initialize` method.

Usage

```
corSymm(value, form, fixed)
```

Arguments

`value`	an optional vector with the parameter values. Default is `numeric(0)`, which results in a vector of zeros of appropriate dimension being assigned to the parameters when `object` is initialized (corresponding to an identity correlation structure).
`form`	a one sided formula of the form `~ t`, or `~ t	g`, specifying a time covariate `t` and, optionally, a grouping factor `g`. A covariate for this correlation structure must be integer valued. When a grouping factor is present in `form`, the correlation structure is assumed to apply only to observations within the same grouping level; observations with different grouping levels are assumed to be uncorrelated. Defaults to `~ 1`, which corresponds to using the order of the observations in the data as a covariate, and no groups.
`fixed`	an optional logical value indicating whether the coefficients should be allowed to vary in the optimization, or kept fixed at their initial value. Defaults to `FALSE`, in which case the coefficients are allowed to vary.

Value

an object of class `corSymm` representing a general correlation structure.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C. and Bates., D.M. (1996) "Unconstrained Parametrizations for Variance-Covariance Matrices", Statistics and Computing, 6, 289-296.

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

`Initialize.corSymm`, `summary.corSymm`

Examples

```
## covariate is observation order and grouping factor is Subject
cs1 <- corSymm(form = ~ 1 | Subject)

# Pinheiro and Bates, p. 225
cs1CompSymm <- corCompSymm(value = 0.3, form = ~ 1 | Subject)
cs1CompSymm <- Initialize(cs1CompSymm, data = Orthodont)
corMatrix(cs1CompSymm)

# Pinheiro and Bates, p. 226
cs1Symm <- corSymm(value =
        c(0.2, 0.1, -0.1, 0, 0.2, 0),
                   form = ~ 1 | Subject)
cs1Symm <- Initialize(cs1Symm, data = Orthodont)
corMatrix(cs1Symm)

# example gls(..., corSpher ...)
# Pinheiro and Bates, pp. 261, 263
fm1Wheat2 <- gls(yield ~ variety - 1, Wheat2)
# p. 262
fm2Wheat2 <- update(fm1Wheat2, corr =
   corSpher(c(28, 0.2),
     form = ~ latitude + longitude, nugget = TRUE))

# example gls(..., corSymm ... )
# Pinheiro and Bates, p. 251
fm1Orth.gls <- gls(distance ~ Sex * I(age - 11), Orthodont,
                   correlation = corSymm(form = ~ 1 | Subject),
                   weights = varIdent(form = ~ 1 | age))
```

`Covariate` *Assign Covariate Values*

Description

This function is generic; method functions can be written to handle specific classes of objects. Classes which already have methods for this function include all `varFunc` classes.

Usage

```
covariate(object) <- value
```

Arguments

`object`	any object with a `covariate` component.
`value`	a value to be assigned to the covariate associated with `object`.

Value

will depend on the method function; see the appropriate documentation.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`getCovariate`

Examples

```
## see the method function documentation
```

`Covariate.varFunc` *Assign varFunc Covariate*

Description

The covariate(s) used in the calculation of the weights of the variance function represented by `object` is (are) replaced by `value`. If `object` has been initialized, `value` must have the same dimensions as `getCovariate(object)`.

Usage

```
## S3 method for class 'varFunc':
covariate(object) <- value
```

Arguments

`object`	an object inheriting from class `varFunc`, representing a variance function structure.
`value`	a value to be assigned to the covariate associated with `object`.

Value

a `varFunc` object similar to `object`, but with its `covariate` attribute replaced by `value`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`getCovariate.varFunc`

Examples

```
vf1 <- varPower(1.1, form = ~age)
covariate(vf1) <- Orthodont[["age"]]
```

Dialyzer *High-Flux Hemodialyzer*

Description

The Dialyzer data frame has 140 rows and 5 columns.

Format

This data frame contains the following columns:

Subject an ordered factor with levels 10 < 8 < 2 < 6 < 3 < 5 < 9 < 7 < 1 < 4 < 17 < 20 < 11 < 12 < 16 < 13 < 14 < 18 < 15 < 19 giving the unique identifier for each subject

QB a factor with levels 200 and 300 giving the bovine blood flow rate (dL/min).

pressure a numeric vector giving the transmembrane pressure (dmHg).

rate the hemodialyzer ultrafiltration rate (mL/hr).

index index of observation within subject—1 through 7.

Details

Vonesh and Carter (1992) describe data measured on high-flux hemodialyzers to assess their *in vivo* ultrafiltration characteristics. The ultrafiltration rates (in mL/hr) of 20 high-flux dialyzers were measured at seven different transmembrane pressures (in dmHg). The *in vitro* evaluation of the dialyzers used bovine blood at flow rates of either 200 dl/min or 300 dl/min. The data, are also analyzed in Littell, Milliken, Stroup, and Wolfinger (1996).

Source

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York. (Appendix A.6)

Vonesh, E. F. and Carter, R. L. (1992), Mixed-effects nonlinear regression for unbalanced repeated measures, *Biometrics*, **48**, 1-18.

Littell, R. C., Milliken, G. A., Stroup, W. W. and Wolfinger, R. D. (1996), *SAS System for Mixed Models*, SAS Institute, Cary, NC.

Dim *Extract Dimensions from an Object*

Description

This function is generic; method functions can be written to handle specific classes of objects. Classes which already have methods for this function include: corSpatial, corStruct, pdCompSymm, pdDiag, pdIdent, pdMat, and pdSymm.

Usage

```
Dim(object, ...)
```

Arguments

object	any object for which dimensions can be extracted.
...	some methods for this generic function require additional arguments.

Value

will depend on the method function used; see the appropriate documentation.

Note

If dim allowed more than one argument, there would be no need for this generic function.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

Dim.pdMat, Dim.corStruct

Examples

```
## see the method function documentation
```

Dim.corSpatial *Dimensions of a corSpatial Object*

Description

if groups is missing, it returns the Dim attribute of object; otherwise, calculates the dimensions associated with the grouping factor.

Usage

```
## S3 method for class 'corSpatial':
Dim(object, groups, ...)
```

Arguments

object	an object inheriting from class corSpatial, representing a spatial correlation structure.
groups	an optional factor defining the grouping of the observations; observations within a group are correlated and observations in different groups are uncorrelated.
...	further arguments to be passed to or from methods.

Value

a list with components:

`N`	length of `groups`
`M`	number of groups
`spClass`	an integer representing the spatial correlation class; 0 = user defined class, 1 = `corSpher`, 2 = `corExp`, 3 = `corGaus`, 4 = `corLin`
`sumLenSq`	sum of the squares of the number of observations per group
`len`	an integer vector with the number of observations per group
`start`	an integer vector with the starting position for the distance vectors in each group, beginning from zero

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`Dim`, `Dim.corStruct`

Examples

```
Dim(corGaus(), getGroups(Orthodont))

cs1ARMA <- corARMA(0.4, form = ~ 1 | Subject, q = 1)
cs1ARMA <- Initialize(cs1ARMA, data = Orthodont)
Dim(cs1ARMA)
```

`Dim.corStruct` *Dimensions of a corStruct Object*

Description

if `groups` is missing, it returns the `Dim` attribute of `object`; otherwise, calculates the dimensions associated with the grouping factor.

Usage

```
## S3 method for class 'corStruct':
Dim(object, groups, ...)
```

Arguments

`object`	an object inheriting from class `corStruct`, representing a correlation structure.
`groups`	an optional factor defining the grouping of the observations; observations within a group are correlated and observations in different groups are uncorrelated.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

a list with components:

N	length of groups
M	number of groups
maxLen	maximum number of observations in a group
sumLenSq	sum of the squares of the number of observations per group
len	an integer vector with the number of observations per group
start	an integer vector with the starting position for the observations in each group, beginning from zero

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

Dim, Dim.corSpatial

Examples

```
Dim(corAR1(), getGroups(Orthodont))
```

Dim.pdMat *Dimensions of a pdMat Object*

Description

This method function returns the dimensions of the matrix represented by object.

Usage

```
## S3 method for class 'pdMat':
Dim(object, ...)
```

Arguments

object	an object inheriting from class pdMat, representing a positive-definite matrix.
...	some methods for this generic require additional arguments. None are used in this method.

Value

an integer vector with the number of rows and columns of the matrix represented by object.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`Dim`

Examples

```
Dim(pdSymm(diag(3)))
```

`Earthquake` *Earthquake Intensity*

Description

The `Earthquake` data frame has 182 rows and 5 columns.

Format

This data frame contains the following columns:

Quake an ordered factor with levels `20` < `16` < `14` < `10` < `3` < `8` < `23` < `22` < `6` < `13` < `7` < `21` < `18` < `15` < `4` < `12` < `19` < `5` < `9` < `1` < `2` < `17` < `11` indicating the earthquake on which the measurements were made.

Richter a numeric vector giving the intensity of the earthquake on the Richter scale.

distance the distance from the seismological measuring station to the epicenter of the earthquake (km).

soil a factor with levels `0` and `1` giving the soil condition at the measuring station, either soil or rock.

accel maximum horizontal acceleration observed (g).

Details

Measurements recorded at available seismometer locations for 23 large earthquakes in western North America between 1940 and 1980. They were originally given in Joyner and Boore (1981); are mentioned in Brillinger (1987); and are analyzed in Davidian and Giltinan (1995).

Source

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York. (Appendix A.8)

Davidian, M. and Giltinan, D. M. (1995), *Nonlinear Models for Repeated Measurement Data*, Chapman and Hall, London.

Joyner and Boor (1981), Peak horizontal acceleration and velocity from strong-motion records including records from the 1979 Imperial Valley, California, earthquake, *Bulletin of the Seismological Society of America*, **71**, 2011-2038.

Brillinger, D. (1987), Comment on a paper by C. R. Rao, *Statistical Science*, **2**, 448-450.

ergoStool *Ergometrics experiment with stool types*

Description

The `ergoStool` data frame has 36 rows and 3 columns.

Format

This data frame contains the following columns:

effort a numeric vector giving the effort (Borg scale) required to arise from a stool.

Type a factor with levels `T1`, `T2`, `T3`, and `T4` giving the stool type.

Subject an ordered factor giving a unique identifier for the subject in the experiment.

Details

Devore (2000) cites data from an article in *Ergometrics* (1993, pp. 519-535) on "The Effects of a Pneumatic Stool and a One-Legged Stool on Lower Limb Joint Load and Muscular Activity."

Source

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York. (Appendix A.9)

Devore, J. L. (2000), *Probability and Statistics for Engineering and the Sciences (5th ed)*, Duxbury, Boston, MA.

Examples

```
fm1 <-
   lme(effort ~ Type, data = ergoStool, random = ~ 1 | Subject)
anova( fm1 )
```

Fatigue *Cracks caused by metal fatigue*

Description

The `Fatigue` data frame has 262 rows and 3 columns.

Format

This data frame contains the following columns:

Path an ordered factor with levels `1` < `2` < `3` < `4` < `5` < `6` < `7` < `8` < `9` < `10` < `11` < `12` < `13` < `14` < `15` < `16` < `17` < `18` < `19` < `20` < `21` giving the test path (or test unit) number. The order is in terms of increasing failure time or decreasing terminal crack length.

cycles number of test cycles at which the measurement is made (millions of cycles).

relLength relative crack length (dimensionless).

Details

These data are given in Lu and Meeker (1993) where they state "We obtained the data in Table 1 visually from figure 4.5.2 on page 242 of Bogdanoff and Kozin (1985)." The data represent the growth of cracks in metal for 21 test units. An initial notch of length 0.90 inches was made on each unit which then was subjected to several thousand test cycles. After every 10,000 test cycles the crack length was measured. Testing was stopped if the crack length exceeded 1.60 inches, defined as a failure, or at 120,000 cycles.

Source

Lu, C. Joseph , and Meeker, William Q. (1993), Using degradation measures to estimate a time-to-failure distribution, *Technometrics*, **35**, 161-174

`fdHess` *Finite difference Hessian*

Description

Evaluate an approximate Hessian and gradient of a scalar function using finite differences.

Usage

```
fdHess(pars, fun, ..., .relStep=(.Machine$double.eps)^(1/3), minAbsPar=0)
```

Arguments

`pars`	the numeric values of the parameters at which to evaluate the function `fun` and its derivatives.
`fun`	a function depending on the parameters `pars` that returns a numeric scalar.
`...`	Optional additional arguments to `fun`
`.relStep`	The relative step size to use in the finite differences. It defaults to the cube root of `.Machine$double.eps`
`minAbsPar`	The minimum magnitude of a parameter value that is considered non-zero. It defaults to zero meaning that any non-zero value will be considered different from zero.

Details

This function uses a second-order response surface design known as a Koschal design to determine the parameter values at which the function is evaluated.

Value

A list with components

`mean`	the value of function `fun` evaluated at the parameter values `pars`
`gradient`	an approximate gradient
`Hessian`	a matrix whose upper triangle containst an approximate Hessian.

Author(s)

Jose Pinheiro ⟨jcp@research.bell-labs.com⟩, Douglas Bates ⟨bates@stat.wisc.edu⟩

Examples

```
fdHess(c(12.3, 2.34), function(x) x[1]*(1-exp(-0.4*x[2])))
```

---

fitted.glsStruct *Calculate glsStruct Fitted Values*

Description

The fitted values for the linear model represented by `object` are extracted.

Usage

```
## S3 method for class 'glsStruct':
fitted(object, glsFit, ...)
```

Arguments

`object`	an object inheriting from class `glsStruct`, representing a list of linear model components, such as `corStruct` and `varFunc` objects.
`glsFit`	an optional list with components `logLik` (log-likelihood), `beta` (coefficients), `sigma` (standard deviation for error term), `varBeta` (coefficients' covariance matrix), `fitted` (fitted values), and `residuals` (residuals). Defaults to `attr(object, "glsFit")`.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

a vector with the fitted values for the linear model represented by `object`.

Note

This method function is generally only used inside `gls` and `fitted.gls`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`gls`, `residuals.glsStruct`

fitted.gnlsStruct *Calculate gnlsStruct Fitted Values*

Description

The fitted values for the nonlinear model represented by `object` are extracted.

Usage

```
## S3 method for class 'gnlsStruct':
fitted(object, ...)
```

Arguments

`object`	an object inheriting from class `gnlsStruct`, representing a list of model components, such as `corStruct` and `varFunc` objects, and attributes specifying the underlying nonlinear model and the response variable.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

a vector with the fitted values for the nonlinear model represented by `object`.

Note

This method function is generally only used inside `gnls` and `fitted.gnls`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`gnls`, `residuals.gnlsStruct`

fitted.lme *Extract lme Fitted Values*

Description

The fitted values at level i are obtained by adding together the population fitted values (based only on the fixed effects estimates) and the estimated contributions of the random effects to the fitted values at grouping levels less or equal to i. The resulting values estimate the best linear unbiased predictions (BLUPs) at level i.

Usage

```
## S3 method for class 'lme':
fitted(object, level, asList, ...)
```

Arguments

object	an object inheriting from class lme, representing a fitted linear mixed-effects model.
level	an optional integer vector giving the level(s) of grouping to be used in extracting the fitted values from object. Level values increase from outermost to innermost grouping, with level zero corresponding to the population fitted values. Defaults to the highest or innermost level of grouping.
asList	an optional logical value. If TRUE and a single value is given in level, the returned object is a list with the fitted values split by groups; else the returned value is either a vector or a data frame, according to the length of level. Defaults to FALSE.
...	some methods for this generic require additional arguments. None are used in this method.

Value

If a single level of grouping is specified in level, the returned value is either a list with the fitted values split by groups (asList = TRUE) or a vector with the fitted values (asList = FALSE); else, when multiple grouping levels are specified in level, the returned object is a data frame with columns given by the fitted values at different levels and the grouping factors.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Bates, D.M. and Pinheiro, J.C. (1998) "Computational methods for multilevel models" available in PostScript or PDF formats at http://nlme.stat.wisc.edu/pub/NLME/

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer, esp. pp. 235, 397.

See Also

lme, residuals.lme

Examples

```
fm1 <- lme(distance ~ age + Sex, data = Orthodont, random = ~ 1)
fitted(fm1, level = 0:1)
```

---

fitted.lmeStruct *Calculate lmeStruct Fitted Values*

Description

The fitted values at level i are obtained by adding together the population fitted values (based only on the fixed effects estimates) and the estimated contributions of the random effects to the fitted values at grouping levels less or equal to i. The resulting values estimate the best linear unbiased predictions (BLUPs) at level i.

Usage

```
## S3 method for class 'lmeStruct':
fitted(object, level, conLin, lmeFit, ...)
```

Arguments

object	an object inheriting from class lmeStruct, representing a list of linear mixed-effects model components, such as reStruct, corStruct, and varFunc objects.
level	an optional integer vector giving the level(s) of grouping to be used in extracting the fitted values from object. Level values increase from outermost to innermost grouping, with level zero corresponding to the population fitted values. Defaults to the highest or innermost level of grouping.
conLin	an optional condensed linear model object, consisting of a list with components "Xy", corresponding to a regression matrix (X) combined with a response vector (y), and "logLik", corresponding to the log-likelihood of the underlying lme model. Defaults to attr(object, "conLin").
lmeFit	an optional list with components beta and b containing respectively the fixed effects estimates and the random effects estimates to be used to calculate the fitted values. Defaults to attr(object, "lmeFit").
...	some methods for this generic accept other optional arguments.

Value

if a single level of grouping is specified in level, the returned value is a vector with the fitted values at the desired level; else, when multiple grouping levels are specified in level, the returned object is a matrix with columns given by the fitted values at different levels.

Note

This method function is generally only used inside lme and fitted.lme.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Bates, D.M. and Pinheiro, J.C. (1998) "Computational methods for multilevel models" available in PostScript or PDF formats at http://franz.stat.wisc.edu/pub/NLME/

See Also

lme, fitted.lme, residuals.lmeStruct

`fitted.lmList` *Extract lmList Fitted Values*

Description

The fitted values are extracted from each `lm` component of `object` and arranged into a list with as many components as `object`, or combined into a single vector.

Usage

```
## S3 method for class 'lmList':
fitted(object, subset, asList, ...)
```

Arguments

`object`	an object inheriting from class `lmList`, representing a list of `lm` objects with a common model.
`subset`	an optional character or integer vector naming the `lm` components of `object` from which the fitted values are to be extracted. Default is `NULL`, in which case all components are used.
`asList`	an optional logical value. If `TRUE`, the returned object is a list with the fitted values split by groups; else the returned value is a vector. Defaults to `FALSE`.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

a list with components given by the fitted values of each `lm` component of `object`, or a vector with the fitted values for all `lm` components of `object`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`lmList`, `residuals.lmList`

Examples

```
fm1 <- lmList(distance ~ age | Subject, Orthodont)
fitted(fm1)
```

`fitted.nlmeStruct` *Calculate nlmeStruct Fitted Values*

Description

The fitted values at level i are obtained by adding together the contributions from the estimated fixed effects and the estimated random effects at levels less or equal to i and evaluating the model function at the resulting estimated parameters. The resulting values estimate the predictions at level i.

Usage

```
## S3 method for class 'nlmeStruct':
fitted(object, level, conLin, ...)
```

Arguments

`object`	an object inheriting from class `nlmeStruct`, representing a list of mixed-effects model components, such as `reStruct`, `corStruct`, and `varFunc` objects, plus attributes specifying the underlying nonlinear model and the response variable.
`level`	an optional integer vector giving the level(s) of grouping to be used in extracting the fitted values from `object`. Level values increase from outermost to innermost grouping, with level zero corresponding to the population fitted values. Defaults to the highest or innermost level of grouping.
`conLin`	an optional condensed linear model object, consisting of a list with components `"Xy"`, corresponding to a regression matrix (`X`) combined with a response vector (`y`), and `"logLik"`, corresponding to the log-likelihood of the underlying nlme model. Defaults to `attr(object, "conLin")`.
`...`	additional arguments that could be given to this method. None are used.

Value

if a single level of grouping is specified in `level`, the returned value is a vector with the fitted values at the desired level; else, when multiple grouping levels are specified in `level`, the returned object is a matrix with columns given by the fitted values at different levels.

Note

This method function is generally only used inside `nlme` and `fitted.nlme`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Bates, D.M. and Pinheiro, J.C. (1998) "Computational methods for multilevel models" available in PostScript or PDF formats at http://nlme.stat.wisc.edu/pub/NLME/

See Also

nlme, residuals.nlmeStruct

`fixed.effects` *Extract Fixed Effects*

Description

This function is generic; method functions can be written to handle specific classes of objects. Classes which already have methods for this function include `lmList` and `lme`.

Usage

```
fixed.effects(object, ...)
fixef(object, ...)
```

Arguments

`object`	any fitted model object from which fixed effects estimates can be extracted.
`...`	some methods for this generic function require additional arguments.

Value

will depend on the method function used; see the appropriate documentation.

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

fixef.lmList

Examples

```
## see the method function documentation
```

fixef.lmList *Extract lmList Fixed Effects*

Description

The average of the coefficients corresponding to the `lm` components of `object` is calculated.

Usage

```
## S3 method for class 'lmList':
fixef(object, ...)
```

Arguments

`object`	an object inheriting from class `lmList`, representing a list of `lm` objects with a common model.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

a vector with the average of the individual `lm` coefficients in `object`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`lmList`, `random.effects.lmList`

Examples

```
fm1 <- lmList(distance ~ age | Subject, Orthodont)
fixed.effects(fm1)
```

formula.pdBlocked *Extract pdBlocked Formula*

Description

The `formula` attributes of the `pdMat` elements of `x` are extracted and returned as a list, in case `asList=TRUE`, or converted to a single one-sided formula when `asList=FALSE`. If the `pdMat` elements do not have a `formula` attribute, a `NULL` value is returned.

Usage

```
## S3 method for class 'pdBlocked':
formula(x, asList, ...)
```

Arguments

x	an object inheriting from class pdBlocked, representing a positive definite block diagonal matrix.
asList	an optional logical value. If TRUE, a list with the formulas for the individual block diagonal elements of x is returned; else, if FALSE, a one-sided formula combining all individual formulas is returned. Defaults to FALSE.
...	some methods for this generic require additional arguments. None are used in this method.

Value

a list of one-sided formulas, or a single one-sided formula, or NULL.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

pdBlocked, pdMat

Examples

```
pd1 <- pdBlocked(list(~ age, ~ Sex - 1))
formula(pd1)
formula(pd1, asList = TRUE)
```

---

formula.pdMat *Extract pdMat Formula*

Description

This method function extracts the formula associated with a pdMat object, in which the column and row names are specified.

Usage

```
## S3 method for class 'pdMat':
formula(x, asList, ...)
```

Arguments

x	an object inheriting from class pdMat, representing a positive definite matrix.
asList	logical. Should the asList argument be applied to each of the components? Never used.
...	some methods for this generic require additional arguments. None are used in this method.

Value

if x has a formula attribute, its value is returned, else NULL is returned.

Note

Because factors may be present in `formula(x)`, the `pdMat` object needs to have access to a data frame where the variables named in the formula can be evaluated, before it can resolve its row and column names from the formula.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`pdMat`

Examples

```
pd1 <- pdSymm(~Sex*age)
formula(pd1)
```

`formula.reStruct` *Extract reStruct Object Formula*

Description

This method function extracts a formula from each of the components of `x`, returning a list of formulas.

Usage

```
## S3 method for class 'reStruct':
formula(x, asList, ...)
```

Arguments

`x`	an object inheriting from class `reStruct`, representing a random effects structure and consisting of a list of `pdMat` objects.
`asList`	logical. Should the asList argument be applied to each of the components?
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

a list with the formulas of each component of `x`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`formula`

Examples

```
rs1 <- reStruct(list(A = pdDiag(diag(2), ~age), B = ~1))
formula(rs1)
```

`gapply` *Apply a Function by Groups*

Description

Applies the function to the distinct sets of rows of the data frame defined by `groups`.

Usage

```
gapply(object, which, FUN, form, level, groups, ...)
```

Arguments

`object`	an object to which the function will be applied - usually a `groupedData` object or a `data.frame`. Must inherit from class `data.frame`.
`which`	an optional character or positive integer vector specifying which columns of `object` should be used with `FUN`. Defaults to all columns in `object`.
`FUN`	function to apply to the distinct sets of rows of the data frame `object` defined by the values of `groups`.
`form`	an optional one-sided formula that defines the groups. When this formula is given the right-hand side is evaluated in `object`, converted to a factor if necessary, and the unique levels are used to define the groups. Defaults to `formula(object)`.
`level`	an optional positive integer giving the level of grouping to be used in an object with multiple nested grouping levels. Defaults to the highest or innermost level of grouping.
`groups`	an optional factor that will be used to split the rows into groups. Defaults to `getGroups(object, form, level)`.
`...`	optional additional arguments to the summary function `FUN`. Often it is helpful to specify `na.rm = TRUE`.

Value

Returns a data frame with as many rows as there are levels in the `groups` argument.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer, esp. sec. 3.4.

See Also

gsummary

Examples

```
## Find number of non-missing "conc" observations for each Subject
gapply( Phenobarb, FUN = function(x) sum(!is.na(x$conc)) )

# Pinheiro and Bates, p. 127
table( gapply(Quinidine, "conc", function(x) sum(!is.na(x))) )
changeRecords <- gapply( Quinidine, FUN = function(frm)
    any(is.na(frm[["conc"]]) & is.na(frm[["dose"]])) )
```

Gasoline *Refinery yield of gasoline*

Description

The Gasoline data frame has 32 rows and 6 columns.

Format

This data frame contains the following columns:

yield a numeric vector giving the percentage of crude oil converted to gasoline after distillation and fractionation

endpoint a numeric vector giving the temperature (degrees F) at which all the gasoline is vaporized

Sample an ordered factor giving the inferred crude oil sample number

API a numeric vector giving the crude oil gravity (degrees API)

vapor a numeric vector giving the vapor pressure of the crude oil $(\mathrm{lbf}/\mathrm{in}^2)$

ASTM a numeric vector giving the crude oil 10% point ASTM—the temperature at which 10% of the crude oil has become vapor.

Details

Prater (1955) provides data on crude oil properties and gasoline yields. Atkinson (1985) uses these data to illustrate the use of diagnostics in multiple regression analysis. Three of the covariates—API, vapor, and ASTM—measure characteristics of the crude oil used to produce the gasoline. The other covariate — endpoint—is a characteristic of the refining process. Daniel and Wood (1980) notice that the covariates characterizing the crude oil occur in only ten distinct groups and conclude that the data represent responses measured on ten different crude oil samples.

Source

Prater, N. H. (1955), Estimate gasoline yields from crudes, *Petroleum Refiner*, **35** (5).

Atkinson, A. C. (1985), *Plots, Transformations, and Regression*, Oxford Press, New York.

Daniel, C. and Wood, F. S. (1980), *Fitting Equations to Data*, Wiley, New York

Venables, W. N. and Ripley, B. D. (1999) *Modern Applied Statistics with S-PLUS (3rd ed)*, Springer, New York.

`getCovariate` *Extract Covariate from an Object*

Description

This function is generic; method functions can be written to handle specific classes of objects. Classes which already have methods for this function include `corStruct`, `corSpatial`, `data.frame`, and `varFunc`.

Usage

```
getCovariate(object, form, data)
```

Arguments

`object`	any object with a `covariate` component
`form`	an optional one-sided formula specifying the covariate(s) to be extracted. Defaults to `formula(object)`.
`data`	a data frame in which to evaluate the variables defined in `form`.

Value

will depend on the method function used; see the appropriate documentation.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer, esp. p. 100.

See Also

`getCovariate.corStruct`, `getCovariate.data.frame`, `getCovariate.varFunc`, `getCovariateFormula`

Examples

```
## see the method function documentation
```

`getCovariate.corStruct` *Extract corStruct Object Covariate*

Description

This method function extracts the covariate(s) associated with `object`.

Usage

```
## S3 method for class 'corStruct':
getCovariate(object, form, data)
```

Arguments

`object`	an object inheriting from class `corStruct` representing a correlation structure.
`form`	this argument is included to make the method function compatible with the generic. It will be assigned the value of `formula(object)` and should not be modified.
`data`	an optional data frame in which to evaluate the variables defined in `form`, in case `object` is not initialized and the covariate needs to be evaluated.

Value

when the correlation structure does not include a grouping factor, the returned value will be a vector or a matrix with the covariate(s) associated with `object`. If a grouping factor is present, the returned value will be a list of vectors or matrices with the covariate(s) corresponding to each grouping level.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

`getCovariate`

Examples

```
cs1 <- corAR1(form = ~ 1 | Subject)
getCovariate(cs1, data = Orthodont)
```

getCovariate.data.frame
Extract Data Frame Covariate

Description

The right hand side of `form`, stripped of any conditioning expression (i.e. an expression following a `|` operator), is evaluated in `object`.

Usage

```
## S3 method for class 'data.frame':
getCovariate(object, form, data)
```

Arguments

`object`	an object inheriting from class `data.frame`.
`form`	an optional formula specifying the covariate to be evaluated in `object`. Defaults to `formula(object)`.
`data`	some methods for this generic require a separate data frame. Not used in this method.

Value

the value of the right hand side of `form`, stripped of any conditional expression, evaluated in `object`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`getCovariateFormula`

Examples

```
getCovariate(Orthodont)
```

getCovariate.varFunc
Extract varFunc Covariate

Description

This method function extracts the covariate(s) associated with the variance function represented by `object`, if any is present.

Usage

```
## S3 method for class 'varFunc':
getCovariate(object, form, data)
```

Arguments

`object`	an object inheriting from class `varFunc`, representing a variance function structure.
`form`	an optional formula specifying the covariate to be evaluated in `object`. Defaults to `formula(object)`.
`data`	some methods for this generic require a `data` object. Not used in this method.

Value

if `object` has a `covariate` attribute, its value is returned; else `NULL` is returned.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`covariate<-.varFunc`

Examples

```
vf1 <- varPower(1.1, form = ~age)
covariate(vf1) <- Orthodont[["age"]]
getCovariate(vf1)
```

`getCovariateFormula` *Extract Covariates Formula*

Description

The right hand side of `formula(object)`, without any conditioning expressions (i.e. any expressions after a `|` operator) is returned as a one-sided formula.

Usage

```
getCovariateFormula(object)
```

Arguments

`object`	any object from which a formula can be extracted.

Value

a one-sided formula describing the covariates associated with `formula(object)`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`getCovariate`

Examples

```
getCovariateFormula(y ~ x | g)
getCovariateFormula(y ~ x)
```

---

getData *Extract Data from an Object*

Description

This function is generic; method functions can be written to handle specific classes of objects. Classes which already have methods for this function include `gls`, `lme`, and `lmList`.

Usage

```
getData(object)
```

Arguments

`object`	an object from which a data.frame can be extracted, generally a fitted model object.

Value

will depend on the method function used; see the appropriate documentation.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`getData.gls`, `getData.lme`, `getData.lmList`

Examples

```
## see the method function documentation
```

getData.gls *Extract gls Object Data*

Description

If present in the calling sequence used to produce `object`, the data frame used to fit the model is obtained.

Usage

```
## S3 method for class 'gls':
getData(object)
```

Arguments

`object`	an object inheriting from class `gls`, representing a generalized least squares fitted linear model.

Value

if a `data` argument is present in the calling sequence that produced `object`, the corresponding data frame (with `na.action` and `subset` applied to it, if also present in the call that produced `object`) is returned; else, `NULL` is returned.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`gls`, `getData`

Examples

```
fm1 <- gls(follicles ~ sin(2*pi*Time) + cos(2*pi*Time), data = Ovary,
           correlation = corAR1(form = ~ 1 | Mare))
getData(fm1)
```

getData.lme *Extract lme Object Data*

Description

If present in the calling sequence used to produce `object`, the data frame used to fit the model is obtained.

Usage

```
## S3 method for class 'lme':
getData(object)
```

Arguments

object	an object inheriting from class lme, representing a linear mixed-effects fitted model.

Value

if a data argument is present in the calling sequence that produced object, the corresponding data frame (with na.action and subset applied to it, if also present in the call that produced object) is returned; else, NULL is returned.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

lme, getData

Examples

```
fm1 <- lme(follicles ~ sin(2*pi*Time) + cos(2*pi*Time), data = Ovary,
           random = ~ sin(2*pi*Time))
getData(fm1)
```

---

getData.lmList *Extract lmList Object Data*

Description

If present in the calling sequence used to produce object, the data frame used to fit the model is obtained.

Usage

```
## S3 method for class 'lmList':
getData(object)
```

Arguments

object	an object inheriting from class lmList, representing a list of lm objects with a common model.

Value

if a data argument is present in the calling sequence that produced object, the corresponding data frame (with na.action and subset applied to it, if also present in the call that produced object) is returned; else, NULL is returned.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

lmList, getData

Examples

```
fm1 <- lmList(distance ~ age | Subject, Orthodont)
getData(fm1)
```

---

getGroups *Extract Grouping Factors from an Object*

Description

This function is generic; method functions can be written to handle specific classes of objects. Classes which already have methods for this function include corStruct, data.frame, gls, lme, lmList, and varFunc.

Usage

```
getGroups(object, form, level, data, sep)
```

Arguments

object	any object
form	an optional formula with a conditioning expression on its right hand side (i.e. an expression involving the | operator). Defaults to formula(object).
level	a positive integer vector with the level(s) of grouping to be used when multiple nested levels of grouping are present. This argument is optional for most methods of this generic function and defaults to all levels of nesting.
data	a data frame in which to interpret the variables named in form. Optional for most methods.
sep	character, the separator to use between group levels when multiple levels are collapsed. The default is '/'.

Value

will depend on the method function used; see the appropriate documentation.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer, esp. pp. 100, 461.

See Also

getGroupsFormula, getGroups.data.frame, getGroups.gls, getGroups.lmList, getGroups.lme

Examples

```
## see the method function documentation
```

`getGroups.corStruct` *Extract corStruct Groups*

Description

This method function extracts the grouping factor associated with `object`, if any is present.

Usage

```
## S3 method for class 'corStruct':
getGroups(object, form, level, data, sep)
```

Arguments

`object`	an object inheriting from class `corStruct` representing a correlation structure.
`form`	this argument is included to make the method function compatible with the generic. It will be assigned the value of `formula(object)` and should not be modified.
`level`	this argument is included to make the method function compatible with the generic and is not used.
`data`	an optional data frame in which to evaluate the variables defined in `form`, in case `object` is not initialized and the grouping factor needs to be evaluated.
`sep`	character, the separator to use between group levels when multiple levels are collapsed. The default is `'/'`.

Value

if a grouping factor is present in the correlation structure represented by `object`, the function returns the corresponding factor vector; else the function returns `NULL`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`getGroups`

Examples

```
cs1 <- corAR1(form = ~ 1 | Subject)
getGroups(cs1, data = Orthodont)
```

`getGroups.data.frame` *Extract Groups from a Data Frame*

Description

Each variable named in the expression after the `|` operator on the right hand side of `form` is evaluated in `object`. If more than one variable is indicated in `level` they are combined into a data frame; else the selected variable is returned as a vector. When multiple grouping levels are defined in `form` and `level > 1`, the levels of the returned factor are obtained by pasting together the levels of the grouping factors of level greater or equal to `level`, to ensure their uniqueness.

Usage

```
## S3 method for class 'data.frame':
getGroups(object, form, level, data, sep)
```

Arguments

`object`	an object inheriting from class `data.frame`.
`form`	an optional formula with a conditioning expression on its right hand side (i.e. an expression involving the `|` operator). Defaults to `formula(object)`.
`level`	a positive integer vector with the level(s) of grouping to be used when multiple nested levels of grouping are present. Defaults to all levels of nesting.
`data`	unused
`sep`	character, the separator to use between group levels when multiple levels are collapsed. The default is `'/'`.

Value

either a data frame with columns given by the grouping factors indicated in `level`, from outer to inner, or, when a single level is requested, a factor representing the selected grouping factor.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer, esp. pp. 100, 461.

See Also

`getGroupsFormula`

Examples

```
getGroups(Pixel)
getGroups(Pixel, level = 2)
```

`getGroups.gls` *Extract gls Object Groups*

Description

If present, the grouping factor associated to the correlation structure for the linear model represented by `object` is extracted.

Usage

```
## S3 method for class 'gls':
getGroups(object, form, level, data, sep)
```

Arguments

`object`	an object inheriting from class `gls`, representing a generalized least squares fitted linear model.
`form`	an optional formula with a conditioning expression on its right hand side (i.e. an expression involving the `|` operator). Defaults to `formula(object)`. Not used.
`level`	a positive integer vector with the level(s) of grouping to be used when multiple nested levels of grouping are present. This argument is optional for most methods of this generic function and defaults to all levels of nesting. Not used.
`data`	a data frame in which to interpret the variables named in `form`. Optional for most methods. Not used.
`sep`	character, the separator to use between group levels when multiple levels are collapsed. The default is `'/'`. Not used.

Value

if the linear model represented by `object` incorporates a correlation structure and the corresponding `corStruct` object has a grouping factor, a vector with the group values is returned; else, `NULL` is returned.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`gls`, `corClasses`

Examples

```
fm1 <- gls(follicles ~ sin(2*pi*Time) + cos(2*pi*Time), Ovary,
           correlation = corAR1(form = ~ 1 | Mare))
getGroups(fm1)
```

`getGroups.lme` *Extract lme Object Groups*

Description

The grouping factors corresponding to the linear mixed-effects model represented by `object` are extracted. If more than one level is indicated in `level`, the corresponding grouping factors are combined into a data frame; else the selected grouping factor is returned as a vector.

Usage

```
## S3 method for class 'lme':
getGroups(object, form, level, data, sep)
```

Arguments

`object`	an object inheriting from class `lme`, representing a fitted linear mixed-effects model.
`form`	this argument is included to make the method function compatible with the generic and is ignored in this method.
`level`	an optional integer vector giving the level(s) of grouping to be extracted from `object`. Defaults to the highest or innermost level of grouping.
`data`	unused
`sep`	character, the separator to use between group levels when multiple levels are collapsed. The default is `'/'`.

Value

either a data frame with columns given by the grouping factors indicated in `level`, or, when a single level is requested, a factor representing the selected grouping factor.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`lme`

Examples

```
fm1 <- lme(pixel ~ day + day^2, Pixel,
  random = list(Dog = ~day, Side = ~1))
getGroups(fm1, level = 1:2)
```

`getGroups.lmList` *Extract lmList Object Groups*

Description

The grouping factor determining the partitioning of the observations used to produce the `lm` components of `object` is extracted.

Usage

```
## S3 method for class 'lmList':
getGroups(object, form, level, data, sep)
```

Arguments

`object`	an object inheriting from class `lmList`, representing a list of `lm` objects with a common model.
`form`	an optional formula with a conditioning expression on its right hand side (i.e. an expression involving the `|` operator). Defaults to `formula(object)`. Not used.
`level`	a positive integer vector with the level(s) of grouping to be used when multiple nested levels of grouping are present. This argument is optional for most methods of this generic function and defaults to all levels of nesting. Not used.
`data`	a data frame in which to interpret the variables named in `form`. Optional for most methods. Not used.
`sep`	character, the separator to use between group levels when multiple levels are collapsed. The default is `'/'`. Not used.

Value

a vector with the grouping factor corresponding to the `lm` components of `object`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`lmList`

Examples

```
fm1 <- lmList(distance ~ age | Subject, Orthodont)
getGroups(fm1)
```

`getGroups.varFunc` *Extract varFunc Groups*

Description

This method function extracts the grouping factor associated with the variance function represented by `object`, if any is present.

Usage

```
## S3 method for class 'varFunc':
getGroups(object, form, level, data, sep)
```

Arguments

`object`	an object inheriting from class `varFunc`, representing a variance function structure.
`form`	an optional formula with a conditioning expression on its right hand side (i.e. an expression involving the `|` operator). Defaults to `formula(object)`. Not used.
`level`	a positive integer vector with the level(s) of grouping to be used when multiple nested levels of grouping are present. This argument is optional for most methods of this generic function and defaults to all levels of nesting. Not used.
`data`	a data frame in which to interpret the variables named in `form`. Optional for most methods. Not used.
`sep`	character, the separator to use between group levels when multiple levels are collapsed. The default is `'/'`. Not used.

Value

if `object` has a `groups` attribute, its value is returned; else `NULL` is returned.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

Examples

```
vf1 <- varPower(form = ~ age | Sex)
vf1 <- Initialize(vf1, Orthodont)
getGroups(vf1)
```

getGroupsFormula *Extract Grouping Formula*

Description

The conditioning expression associated with `formula(object)` (i.e. the expression after the `|` operator) is returned either as a named list of one-sided formulas, or a single one-sided formula, depending on the value of `asList`. The components of the returned list are ordered from outermost to innermost level and are named after the grouping factor expression.

Usage

```
getGroupsFormula(object, asList, sep)
```

Arguments

`object`	any object from which a formula can be extracted.
`asList`	an optional logical value. If `TRUE` the returned value with be a list of formulas; else, if `FALSE` the returned value will be a one-sided formula. Defaults to `FALSE`.
`sep`	character, the separator to use between group levels when multiple levels are collapsed. The default is `'/'`.

Value

a one-sided formula, or a list of one-sided formulas, with the grouping structure associated with `formula(object)`. If no conditioning expression is present in `formula(object)` a `NULL` value is returned.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`getGroupsFormula.gls`, `getGroupsFormula.lmList`, `getGroupsFormula.lme`, `getGroupsFormula.reStruct`, `getGroups`

Examples

```
getGroupsFormula(y ~ x | g1/g2)
```

getResponse *Extract Response Variable from an Object*

Description

This function is generic; method functions can be written to handle specific classes of objects. Classes which already have methods for this function include `data.frame`, `gls`, `lme`, and `lmList`.

Usage

```
getResponse(object, form)
```

Arguments

`object`	any object
`form`	an optional two-sided formula. Defaults to `formula(object)`.

Value

will depend on the method function used; see the appropriate documentation.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`getResponseFormula`

Examples

```
getResponse(Orthodont)
```

getResponseFormula *Extract Formula Specifying Response Variable*

Description

The left hand side of `formula{object}` is returned as a one-sided formula.

Usage

```
getResponseFormula(object)
```

Arguments

`object`	any object from which a formula can be extracted.

Value

a one-sided formula with the response variable associated with `formula{object}`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`getResponse`

Examples

```
getResponseFormula(y ~ x | g)
```

---

getVarCov *Extract variance-covariance matrix*

Description

Extract the variance-covariance matrix from a fitted model, such as a mixed-effects model.

Usage

```
getVarCov(obj, ...)
## S3 method for class 'lme':
getVarCov(obj, individuals,
    type = c("random.effects", "conditional", "marginal"), ...)
## S3 method for class 'gls':
getVarCov(obj, individual = 1, ...)
```

Arguments

`obj`	A fitted model. Methods are available for models fit by `lme` and by `gls`
`individuals`	For models fit by `lme` a vector of levels of the grouping factor can be specified for the conditional or marginal variance-covariance matrices.
`individual`	For models fit by `gls` the only type of variance-covariance matrix provided is the marginal variance-covariance of the responses by group. The optional argument `individual` specifies the group of responses.
`type`	For models fit by `lme` the `type` argument specifies the type of variance-covariance matrix, either `"random.effects"` for the random-effects variance-covariance (the default), or `"conditional"` for the conditional. variance-covariance of the responses or `"marginal"` for the the marginal variance-covariance of the responses.
`...`	Optional arguments for some methods, as described above

Value

A variance-covariance matrix or a list of variance-covariance matrices.

Author(s)

Mary Lindstrom ⟨lindstro@biostat.wisc.edu⟩

See Also

`lme`, `gls`

Examples

```
fm1 <- lme(distance ~ age, data = Orthodont, subset = Sex == "Female")
getVarCov(fm1)
getVarCov(fm1, individual = "F01", type = "marginal")
getVarCov(fm1, type = "conditional")
fm2 <- gls(follicles ~ sin(2*pi*Time) + cos(2*pi*Time), Ovary,
           correlation = corAR1(form = ~ 1 | Mare))
getVarCov(fm2)
```

`gls` *Fit Linear Model Using Generalized Least Squares*

Description

This function fits a linear model using generalized least squares. The errors are allowed to be correlated and/or have unequal variances.

Usage

```
gls(model, data, correlation, weights, subset, method, na.action,
    control, verbose)
## S3 method for class 'gls':
update(object, model., ..., evaluate = TRUE)
```

Arguments

`object`	an object inheriting from class `gls`, representing a generalized least squares fitted linear model.
`model`	a two-sided linear formula object describing the model, with the response on the left of a ~ operator and the terms, separated by + operators, on the right.
`model.`	Changes to the model – see `update.formula` for details.
`data`	an optional data frame containing the variables named in `model`, `correlation`, `weights`, and `subset`. By default the variables are taken from the environment from which `gls` is called.
`correlation`	an optional `corStruct` object describing the within-group correlation structure. See the documentation of `corClasses` for a description of the available `corStruct` classes. If a grouping variable is to be used, it must be specified in the `form` argument to the `corStruct` constructor. Defaults to `NULL`, corresponding to uncorrelated errors.

weights	an optional varFunc object or one-sided formula describing the within-group heteroscedasticity structure. If given as a formula, it is used as the argument to varFixed, corresponding to fixed variance weights. See the documentation on varClasses for a description of the available varFunc classes. Defaults to NULL, corresponding to homoscesdatic errors.
subset	an optional expression indicating which subset of the rows of data should be used in the fit. This can be a logical vector, or a numeric vector indicating which observation numbers are to be included, or a character vector of the row names to be included. All observations are included by default.
method	a character string. If "REML" the model is fit by maximizing the restricted log-likelihood. If "ML" the log-likelihood is maximized. Defaults to "REML".
na.action	a function that indicates what should happen when the data contain NAs. The default action (na.fail) causes gls to print an error message and terminate if there are any incomplete observations.
control	a list of control values for the estimation algorithm to replace the default values returned by the function glsControl. Defaults to an empty list.
verbose	an optional logical value. If TRUE information on the evolution of the iterative algorithm is printed. Default is FALSE.
...	some methods for this generic require additional arguments. None are used in this method.
evaluate	If TRUE evaluate the new call else return the call.

Value

an object of class gls representing the linear model fit. Generic functions such as print, plot, and summary have methods to show the results of the fit. See glsObject for the components of the fit. The functions resid, coef, and fitted can be used to extract some of its components.

Author(s)

Jose Pinheiro ⟨jcp@research.bell-labs.com⟩, Douglas Bates ⟨bates@stat.wisc.edu⟩

References

The different correlation structures available for the correlation argument are described in Box, G.E.P., Jenkins, G.M., and Reinsel G.C. (1994), Littel, R.C., Milliken, G.A., Stroup, W.W., and Wolfinger, R.D. (1996), and Venables, W.N. and Ripley, B.D. (1997). The use of variance functions for linear and nonlinear models is presented in detail in Carroll, R.J. and Ruppert, D. (1988) and Davidian, M. and Giltinan, D.M. (1995).

Box, G.E.P., Jenkins, G.M., and Reinsel G.C. (1994) "Time Series Analysis: Forecasting and Control", 3rd Edition, Holden-Day.

Carroll, R.J. and Ruppert, D. (1988) "Transformation and Weighting in Regression", Chapman and Hall.

Davidian, M. and Giltinan, D.M. (1995) "Nonlinear Mixed Effects Models for Repeated Measurement Data", Chapman and Hall.

Littel, R.C., Milliken, G.A., Stroup, W.W., and Wolfinger, R.D. (1996) "SAS Systems for Mixed Models", SAS Institute.

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer, esp. pp. 100, 461.

Venables, W.N. and Ripley, B.D. (1997) "Modern Applied Statistics with S-PLUS", 2nd Edition, Springer-Verlag.

See Also

corClasses, glsControl, glsObject, glsStruct, plot.gls, predict.gls, qqnorm.gls, residuals.gls, summary.gls, varClasses, varFunc

Examples

```
# AR(1) errors within each Mare
fm1 <- gls(follicles ~ sin(2*pi*Time) + cos(2*pi*Time), Ovary,
           correlation = corAR1(form = ~ 1 | Mare))
# variance increases as a power of the absolute fitted values
fm2 <- update(fm1, weights = varPower())
```

---

glsControl *Control Values for gls Fit*

Description

The values supplied in the function call replace the defaults and a list with all possible arguments is returned. The returned list is used as the `control` argument to the `gls` function.

Usage

```
glsControl(maxIter, msMaxIter, tolerance, msTol, msScale, msVerbose,
           singular.ok, qrTol, returnObject, apVar, .relStep,
           nlmStepMax, opt=c("nlminb", "optim"), optimMethod,
           minAbsParApVar, natural)
```

Arguments

`maxIter`	maximum number of iterations for the `gls` optimization algorithm. Default is 50.
`msMaxIter`	maximum number of iterations for the optimization step inside the `gls` optimization. Default is 50.
`tolerance`	tolerance for the convergence criterion in the `gls` algorithm. Default is 1e-6.
`msTol`	tolerance for the convergence criterion in `ms`, passed as the `rel.tolerance` argument to the function (see documentation on `ms`). Default is 1e-7.
`msScale`	scale function passed as the `scale` argument to the `ms` function (see documentation on that function). Default is `lmeScale`.
`msVerbose`	a logical value passed as the `trace` argument to `ms` (see documentation on that function). Default is `FALSE`.
`singular.ok`	a logical value indicating whether non-estimable coefficients (resulting from linear dependencies among the columns of the regression matrix) should be allowed. Default is `FALSE`.
`qrTol`	a tolerance for detecting linear dependencies among the columns of the regression matrix in its QR decomposition. Default is `.Machine$single.eps`.

`returnObject` a logical value indicating whether the fitted object should be returned when the maximum number of iterations is reached without convergence of the algorithm. Default is `FALSE`.

`apVar` a logical value indicating whether the approximate covariance matrix of the variance-covariance parameters should be calculated. Default is `TRUE`.

`.relStep` relative step for numerical derivatives calculations. Default is `.Machine$double.eps^(1/3)`.

`nlmStepMax` stepmax value to be passed to nlm. See `nlm` for details. Default is 100.0

`opt` the optimizer to be used, either `nlminb` (the default since (R 2.2.0) or `optim` (the previous default).

`optimMethod` character - the optimization method to be used with the `optim` optimizer. The default is `"BFGS"`. An alternative is `"L-BFGS-B"`.

`minAbsParApVar` numeric value - minimum absolute parameter value in the approximate variance calculation. The default is `0.05`.

`natural` logical. Should the natural parameterization be used for the approximate variance calculations? Default is `TRUE`.

Value

a list with components for each of the possible arguments.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`gls`, `lmeScale`

Examples

```
# decrease the maximum number iterations in the optimization call and
# request that information on the evolution of the ms iterations be printed
glsControl(msMaxIter = 20, msVerbose = TRUE)
```

---

glsObject *Fitted gls Object*

Description

An object returned by the `gls` function, inheriting from class `gls` and representing a generalized least squares fitted linear model. Objects of this class have methods for the generic functions `anova`, `coef`, `fitted`, `formula`, `getGroups`, `getResponse`, `intervals`, `logLik`, `plot`, `predict`, `print`, `residuals`, `summary`, and `update`.

Value

The following components must be included in a legitimate `gls` object.

`apVar`	an approximate covariance matrix for the variance-covariance coefficients. If `apVar = FALSE` in the list of control values used in the call to `gls`, this component is equal to `NULL`.
`call`	a list containing an image of the `gls` call that produced the object.
`coefficients`	a vector with the estimated linear model coefficients.
`contrasts`	a list with the contrasts used to represent factors in the model formula. This information is important for making predictions from a new data frame in which not all levels of the original factors are observed. If no factors are used in the model, this component will be an empty list.
`dims`	a list with basic dimensions used in the model fit, including the components `N` - the number of observations in the data and `p` - the number of coefficients in the linear model.
`fitted`	a vector with the fitted values..
`glsStruct`	an object inheriting from class `glsStruct`, representing a list of linear model components, such as `corStruct` and `varFunc` objects.
`groups`	a vector with the correlation structure grouping factor, if any is present.
`logLik`	the log-likelihood at convergence.
`method`	the estimation method: either `"ML"` for maximum likelihood, or `"REML"` for restricted maximum likelihood.
`numIter`	the number of iterations used in the iterative algorithm.
`residuals`	a vector with the residuals.
`sigma`	the estimated residual standard error.
`varBeta`	an approximate covariance matrix of the coefficients estimates.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`gls`, `glsStruct`

`glsStruct` *Generalized Least Squares Structure*

Description

A generalized least squares structure is a list of model components representing different sets of parameters in the linear model. A `glsStruct` may contain `corStruct` and `varFunc` objects. `NULL` arguments are not included in the `glsStruct` list.

Usage

```
glsStruct(corStruct, varStruct)
```

Arguments

corStruct	an optional corStruct object, representing a correlation structure. Default is NULL.
varStruct	an optional varFunc object, representing a variance function structure. Default is NULL.

Value

a list of model variance-covariance components determining the parameters to be estimated for the associated linear model.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

corClasses, gls, residuals.glsStruct, varFunc

Examples

```
gls1 <- glsStruct(corAR1(), varPower())
```

Glucose *Glucose levels over time*

Description

The Glucose data frame has 378 rows and 4 columns.

Format

This data frame contains the following columns:

Subject an ordered factor with levels 6 < 2 < 3 < 5 < 1 < 4

Time a numeric vector

conc a numeric vector of glucose levels

Meal an ordered factor with levels 2am < 6am < 10am < 2pm < 6pm < 10pm

Source

Hand, D. and Crowder, M. (1996), *Practical Longitudinal Data Analysis*, Chapman and Hall, London.

Glucose2 — *Glucose Levels Following Alcohol Ingestion*

Description

The `Glucose2` data frame has 196 rows and 4 columns.

Format

This data frame contains the following columns:

Subject a factor with levels `1` to `7` identifying the subject whose glucose level is measured.

Date a factor with levels `1 2` indicating the occasion in which the experiment was conducted.

Time a numeric vector giving the time since alcohol ingestion (in min/10).

glucose a numeric vector giving the blood glucose level (in mg/dl).

Details

Hand and Crowder (Table A.14, pp. 180-181, 1996) describe data on the blood glucose levels measured at 14 time points over 5 hours for 7 volunteers who took alcohol at time 0. The same experiment was repeated on a second date with the same subjects but with a dietary additive used for all subjects.

Source

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York. (Appendix A.10)

Hand, D. and Crowder, M. (1996), *Practical Longitudinal Data Analysis*, Chapman and Hall, London.

gnls — *Fit Nonlinear Model Using Generalized Least Squares*

Description

This function fits a nonlinear model using generalized least squares. The errors are allowed to be correlated and/or have unequal variances.

Usage

```
gnls(model, data, params, start, correlation, weights, subset,
     na.action, naPattern, control, verbose)
```

Arguments

model	a two-sided formula object describing the model, with the response on the left of a ~ operator and a nonlinear expression involving parameters and covariates on the right. If data is given, all names used in the formula should be defined as parameters or variables in the data frame.
data	an optional data frame containing the variables named in model, correlation, weights, subset, and naPattern. By default the variables are taken from the environment from which gnls is called.
params	an optional two-sided linear formula of the form p1+...+pn~x1+...+xm, or list of two-sided formulas of the form p1~x1+...+xm, with possibly different models for each parameter. The p1,...,pn represent parameters included on the right hand side of model and x1+...+xm define a linear model for the parameters (when the left hand side of the formula contains several parameters, they are all assumed to follow the same linear model described by the right hand side expression). A 1 on the right hand side of the formula(s) indicates a single fixed effects for the corresponding parameter(s). By default, the parameters are obtained from the names of start.
start	an optional named list, or numeric vector, with the initial values for the parameters in model. It can be omitted when a selfStarting function is used in model, in which case the starting estimates will be obtained from a single call to the nls function.
correlation	an optional corStruct object describing the within-group correlation structure. See the documentation of corClasses for a description of the available corStruct classes. If a grouping variable is to be used, it must be specified in the form argument to the corStruct constructor. Defaults to NULL, corresponding to uncorrelated errors.
weights	an optional varFunc object or one-sided formula describing the within-group heteroscedasticity structure. If given as a formula, it is used as the argument to varFixed, corresponding to fixed variance weights. See the documentation on varClasses for a description of the available varFunc classes. Defaults to NULL, corresponding to homoscesdatic errors.
subset	an optional expression indicating which subset of the rows of data should be used in the fit. This can be a logical vector, or a numeric vector indicating which observation numbers are to be included, or a character vector of the row names to be included. All observations are included by default.
na.action	a function that indicates what should happen when the data contain NAs. The default action (na.fail) causes gnls to print an error message and terminate if there are any incomplete observations.
naPattern	an expression or formula object, specifying which returned values are to be regarded as missing.
control	a list of control values for the estimation algorithm to replace the default values returned by the function gnlsControl. Defaults to an empty list.
verbose	an optional logical value. If TRUE information on the evolution of the iterative algorithm is printed. Default is FALSE.
...	some methods for this generic require additional arguments. None are used in this method.

Value

an object of class `gnls`, also inheriting from class `gls`, representing the nonlinear model fit. Generic functions such as `print`, `plot` and `summary` have methods to show the results of the fit. See `gnlsObject` for the components of the fit. The functions `resid`, `coef`, and `fitted` can be used to extract some of its components.

Author(s)

Jose Pinheiro ⟨jose.pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

The different correlation structures available for the `correlation` argument are described in Box, G.E.P., Jenkins, G.M., and Reinsel G.C. (1994), Littel, R.C., Milliken, G.A., Stroup, W.W., and Wolfinger, R.D. (1996), and Venables, W.N. and Ripley, B.D. (1997). The use of variance functions for linear and nonlinear models is presented in detail in Carrol, R.J. and Rupert, D. (1988) and Davidian, M. and Giltinan, D.M. (1995).

Box, G.E.P., Jenkins, G.M., and Reinsel G.C. (1994) "Time Series Analysis: Forecasting and Control", 3rd Edition, Holden-Day.

Carrol, R.J. and Rupert, D. (1988) "Transformation and Weighting in Regression", Chapman and Hall.

Davidian, M. and Giltinan, D.M. (1995) "Nonlinear Mixed Effects Models for Repeated Measurement Data", Chapman and Hall.

Littel, R.C., Milliken, G.A., Stroup, W.W., and Wolfinger, R.D. (1996) "SAS Systems for Mixed Models", SAS Institute.

Venables, W.N. and Ripley, B.D. (1997) "Modern Applied Statistics with S-plus", 2nd Edition, Springer-Verlag.

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

`corClasses`, `gnlsControl`, `gnlsObject`, `gnlsStruct`, `predict.gnls`, `varClasses`, `varFunc`

Examples

```
# variance increases with a power of the absolute fitted values
fm1 <- gnls(weight ~ SSlogis(Time, Asym, xmid, scal), Soybean,
            weights = varPower())
summary(fm1)
```

---

gnlsControl *Control Values for gnls Fit*

Description

The values supplied in the function call replace the defaults and a list with all possible arguments is returned. The returned list is used as the `control` argument to the `gnls` function.

Usage

```
gnlsControl(maxIter, nlsMaxIter, msMaxIter, minScale, tolerance,
            nlsTol, msTol, msScale, returnObject, msVerbose,
            apVar, .relStep, nlmStepMax,
            opt = c("nlminb", "optim"), optimMethod,
            minAbsParApVar)
```

Arguments

maxIter	maximum number of iterations for the gnls optimization algorithm. Default is 50.
nlsMaxIter	maximum number of iterations for the nls optimization step inside the gnls optimization. Default is 7.
msMaxIter	maximum number of iterations for the ms optimization step inside the gnls optimization. Default is 50.
minScale	minimum factor by which to shrink the default step size in an attempt to decrease the sum of squares in the nls step. Default 0.001.
tolerance	tolerance for the convergence criterion in the gnls algorithm. Default is 1e-6.
nlsTol	tolerance for the convergence criterion in nls step. Default is 1e-3.
msTol	tolerance for the convergence criterion in ms, passed as the rel.tolerance argument to the function (see documentation on ms). Default is 1e-7.
msScale	scale function passed as the scale argument to the ms function (see documentation on that function). Default is lmeScale.
returnObject	a logical value indicating whether the fitted object should be returned when the maximum number of iterations is reached without convergence of the algorithm. Default is FALSE.
msVerbose	a logical value passed as the trace argument to ms (see documentation on that function). Default is FALSE.
apVar	a logical value indicating whether the approximate covariance matrix of the variance-covariance parameters should be calculated. Default is TRUE.
.relStep	relative step for numerical derivatives calculations. Default is .Machine$double.eps^(1/3).
opt	the optimizer to be used, either nlminb (the default since (R 2.2.0) or optim (the previous default).
optimMethod	character - the optimization method to be used with the optim optimizer. The default is "BFGS". An alternative is "L-BFGS-B".
nlmStepMax	stepmax value to be passed to nlm. See nlm for details. Default is 100.0
minAbsParApVar	numeric value - minimum absolute parameter value in the approximate variance calculation. The default is 0.05.

Value

a list with components for each of the possible arguments.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

gnls, lmeScale

Examples

```
# decrease the maximum number iterations in the ms call and
# request that information on the evolution of the ms iterations be printed
gnlsControl(msMaxIter = 20, msVerbose = TRUE)
```

---

gnlsObject *Fitted gnls Object*

Description

An object returned by the gnls function, inheriting from class gnls and also from class gls, and representing a generalized nonlinear least squares fitted model. Objects of this class have methods for the generic functions anova, coef, fitted, formula, getGroups, getResponse, intervals, logLik, plot, predict, print, residuals, summary, and update.

Value

The following components must be included in a legitimate gnls object.

apVar	an approximate covariance matrix for the variance-covariance coefficients. If apVar = FALSE in the list of control values used in the call to gnls, this component is equal to NULL.
call	a list containing an image of the gnls call that produced the object.
coefficients	a vector with the estimated nonlinear model coefficients.
contrasts	a list with the contrasts used to represent factors in the model formula. This information is important for making predictions from a new data frame in which not all levels of the original factors are observed. If no factors are used in the model, this component will be an empty list.
dims	a list with basic dimensions used in the model fit, including the components N - the number of observations used in the fit and p - the number of coefficients in the nonlinear model.
fitted	a vector with the fitted values.
modelStruct	an object inheriting from class gnlsStruct, representing a list of model components, such as corStruct and varFunc objects.
groups	a vector with the correlation structure grouping factor, if any is present.
logLik	the log-likelihood at convergence.
numIter	the number of iterations used in the iterative algorithm.
plist	
pmap	
residuals	a vector with the residuals.
sigma	the estimated residual standard error.
varBeta	an approximate covariance matrix of the coefficients estimates.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

gnls, gnlsStruct

---

gnlsStruct *Generalized Nonlinear Least Squares Structure*

Description

A generalized nonlinear least squares structure is a list of model components representing different sets of parameters in the nonlinear model. A gnlsStruct may contain corStruct and varFunc objects. NULL arguments are not included in the gnlsStruct list.

Usage

```
gnlsStruct(corStruct, varStruct)
```

Arguments

corStruct	an optional corStruct object, representing a correlation structure. Default is NULL.
varStruct	an optional varFunc object, representing a variance function structure. Default is NULL.

Value

a list of model variance-covariance components determining the parameters to be estimated for the associated nonlinear model.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

gnls, corClasses, residuals.gnlsStruct varFunc

Examples

```
gnls1 <- gnlsStruct(corAR1(), varPower())
```

groupedData *Construct a groupedData Object*

Description

An object of the groupedData class is constructed from the formula and data by attaching the formula as an attribute of the data, along with any of outer, inner, labels, and units that are given. If order.groups is TRUE the grouping factor is converted to an ordered factor with the ordering determined by FUN. Depending on the number of grouping levels and the type of primary covariate, the returned object will be of one of three classes: nfnGroupedData - numeric covariate, single level of nesting; nffGroupedData - factor covariate, single level of nesting; and nmGroupedData - multiple levels of nesting. Several modeling and plotting functions can use the formula stored with a groupedData object to construct default plots and models.

Usage

```
groupedData(formula, data, order.groups, FUN, outer, inner,
 labels, units)
## S3 method for class 'groupedData':
update(object, formula, data, order.groups, FUN,
outer, inner, labels, units, ...)
```

Arguments

object	an object inheriting from class groupedData.
formula	a formula of the form resp ~ cov | group where resp is the response, cov is the primary covariate, and group is the grouping factor. The expression 1 can be used for the primary covariate when there is no other suitable candidate. Multiple nested grouping factors can be listed separated by the / symbol as in fact1/fact2. In an expression like this the fact2 factor is nested within the fact1 factor.
data	a data frame in which the expressions in formula can be evaluated. The resulting groupedData object will consist of the same data values in the same order but with additional attributes.
order.groups	an optional logical value, or list of logical values, indicating if the grouping factors should be converted to ordered factors according to the function FUN applied to the response from each group. If multiple levels of grouping are present, this argument can be either a single logical value (which will be repeated for all grouping levels) or a list of logical values. If no names are assigned to the list elements, they are assumed in the same order as the group levels (outermost to innermost grouping). Ordering within a level of grouping is done within the levels of the grouping factors which are outer to it. Changing the grouping factor to an ordered factor does not affect the ordering of the rows in the data frame but it does affect the order of the panels in a trellis display of the data or models fitted to the data. Defaults to TRUE.
FUN	an optional summary function that will be applied to the values of the response for each level of the grouping factor, when order.groups = TRUE, to determine the ordering. Defaults to the max function.

outer	an optional one-sided formula, or list of one-sided formulas, indicating covariates that are outer to the grouping factor(s). If multiple levels of grouping are present, this argument can be either a single one-sided formula, or a list of one-sided formulas. If no names are assigned to the list elements, they are assumed in the same order as the group levels (outermost to innermost grouping). An outer covariate is invariant within the sets of rows defined by the grouping factor. Ordering of the groups is done in such a way as to preserve adjacency of groups with the same value of the outer variables. When plotting a groupedData object, the argument `outer = TRUE` causes the panels to be determined by the `outer` formula. The points within the panels are associated by level of the grouping factor. Defaults to `NULL`, meaning that no outer covariates are present.
inner	an optional one-sided formula, or list of one-sided formulas, indicating covariates that are inner to the grouping factor(s). If multiple levels of grouping are present, this argument can be either a single one-sided formula, or a list of one-sided formulas. If no names are assigned to the list elements, they are assumed in the same order as the group levels (outermost to innermost grouping). An inner covariate can change within the sets of rows defined by the grouping factor. An inner formula can be used to associate points in a plot of a groupedData object. Defaults to `NULL`, meaning that no inner covariates are present.
labels	an optional list of character strings giving labels for the response and the primary covariate. The label for the primary covariate is named `x` and that for the response is named `y`. Either label can be omitted.
units	an optional list of character strings giving the units for the response and the primary covariate. The units string for the primary covariate is named `x` and that for the response is named `y`. Either units string can be omitted.
...	some methods for this generic require additional arguments. None are used in this method.

Value

an object of one of the classes `nfnGroupedData`, `nffGroupedData`, or `nmGroupedData`, and also inheriting from classes `groupedData` and `data.frame`.

Author(s)

Douglas Bates and Jose Pinheiro

References

Bates, D.M. and Pinheiro, J.C. (1997), "Software Design for Longitudinal Data", in "Modelling Longitudinal and Spatially Correlated Data: Methods, Applications and Future Directions", T.G. Gregoire (ed.), Springer-Verlag, New York.

Pinheiro, J.C. and Bates, D.M. (1997) "Future Directions in Mixed-Effects Software: Design of NLME 3.0" available at http://nlme.stat.wisc.edu/

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

`formula`, `gapply`, `gsummary`, `lme`, `plot.nffGroupedData`, `plot.nfnGroupedData`, `plot.nmGroupedData`, `reStruct`

Examples

```
Orth.new <-  # create a new copy of the groupedData object
  groupedData( distance ~ age | Subject,
               data = as.data.frame( Orthodont ),
               FUN = mean,
               outer = ~ Sex,
               labels = list( x = "Age",
                 y = "Distance from pituitary to pterygomaxillary fissure" ),
               units = list( x = "(yr)", y = "(mm)") )
## Not run:
plot( Orth.new )          # trellis plot by Subject
## End(Not run)
formula( Orth.new )       # extractor for the formula
gsummary( Orth.new )      # apply summary by Subject
fm1 <- lme( Orth.new )    # fixed and groups formulae extracted from object
Orthodont2 <- update(Orthodont, FUN = mean)
```

---

gsummary *Summarize by Groups*

Description

Provide a summary of the variables in a data frame by groups of rows. This is most useful with a `groupedData` object to examine the variables by group.

Usage

```
gsummary(object, FUN, omitGroupingFactor, form, level,
   groups, invariantsOnly, ...)
```

Arguments

`object`	an object to be summarized - usually a `groupedData` object or a `data.frame`.
`FUN`	an optional summary function or a list of summary functions to be applied to each variable in the frame. The function or functions are applied only to variables in `object` that vary within the groups defined by `groups`. Invariant variables are always summarized by group using the unique value that they assume within that group. If `FUN` is a single function it will be applied to each non-invariant variable by group to produce the summary for that variable. If `FUN` is a list of functions, the names in the list should designate classes of variables in the frame such as `ordered`, `factor`, or `numeric`. The indicated function will be applied to any non-invariant variables of that class. The default functions to be used are `mean` for numeric factors, and `Mode` for both `factor` and `ordered`. The `Mode` function, defined internally in `gsummary`, returns the modal or most popular value of the variable. It is different from the `mode` function that returns the S-language mode of the variable.
`omitGroupingFactor`	an optional logical value. When `TRUE` the grouping factor itself will be omitted from the group-wise summary but the levels of the grouping factor will continue to be used as the row names for the data frame that is produced by the summary. Defaults to `FALSE`.

`form` an optional one-sided formula that defines the groups. When this formula is given, the right-hand side is evaluated in `object`, converted to a factor if necessary, and the unique levels are used to define the groups. Defaults to `formula(object)`.

`level` an optional positive integer giving the level of grouping to be used in an object with multiple nested grouping levels. Defaults to the highest or innermost level of grouping.

`groups` an optional factor that will be used to split the rows into groups. Defaults to `getGroups(object, form, level)`.

`invariantsOnly`
an optional logical value. When `TRUE` only those covariates that are invariant within each group will be summarized. The summary value for the group is always the unique value taken on by that covariate within the group. The columns in the summary are of the same class as the corresponding columns in `object`. By definition, the grouping factor itself must be an invariant. When combined with `omitGroupingFactor = TRUE`, this option can be used to discover is there are invariant covariates in the data frame. Defaults to `FALSE`.

`...` optional additional arguments to the summary functions that are invoked on the variables by group. Often it is helpful to specify `na.rm = TRUE`.

Value

A `data.frame` with one row for each level of the grouping factor. The number of columns is at most the number of columns in `object`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

`summary`, `groupedData`, `getGroups`

Examples

```
gsummary(Orthodont)  # default summary by Subject
## gsummary with invariantsOnly = TRUE and omitGroupingFactor = TRUE
## determines whether there are covariates like Sex that are invariant
## within the repeated observations on the same Subject.
gsummary(Orthodont, inv = TRUE, omit = TRUE)
```

Gun *Methods for firing naval guns*

Description

The `Gun` data frame has 36 rows and 4 columns.

Format

This data frame contains the following columns:

rounds a numeric vector

Method a factor with levels `M1` `M2`

Team an ordered factor with levels `T1S` < `T3S` < `T2S` < `T1A` < `T2A` < `T3A` < `T1H` < `T3H` < `T2H`

Physique an ordered factor with levels `Slight` < `Average` < `Heavy`

Details

Hicks (p.180, 1993) reports data from an experiment on methods for firing naval guns. Gunners of three different physiques (slight, average, and heavy) tested two firing methods. Both methods were tested twice by each of nine teams of three gunners with identical physique. The response was the number of rounds fired per minute.

Source

Hicks, C. R. (1993), *Fundamental Concepts in the Design of Experiments (4th ed)*, Harcourt Brace, New York.

IGF *Radioimmunoassay of IGF-I Protein*

Description

The `IGF` data frame has 237 rows and 3 columns.

Format

This data frame contains the following columns:

Lot an ordered factor giving the radioactive tracer lot.

age a numeric vector giving the age (in days) of the radioactive tracer.

conc a numeric vector giving the estimated concentration of IGF-I protein (ng/ml)

Details

Davidian and Giltinan (1995) describe data obtained during quality control radioimmunoassays for ten different lots of radioactive tracer used to calibrate the Insulin-like Growth Factor (IGF-I) protein concentration measurements.

Source

Davidian, M. and Giltinan, D. M. (1995), *Nonlinear Models for Repeated Measurement Data*, Chapman and Hall, London.

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York. (Appendix A.11)

`Initialize` *Initialize Object*

Description

This function is generic; method functions can be written to handle specific classes of objects. Classes which already have methods for this function include: `corStruct`, `lmeStruct`, `reStruct`, and `varFunc`.

Usage

```
Initialize(object, data, ...)
```

Arguments

`object`	any object requiring initialization, e.g. "plug-in" structures such as `corStruct` and `varFunc` objects.
`data`	a data frame to be used in the initialization procedure.
`...`	some methods for this generic function require additional arguments.

Value

an initialized object with the same class as `object`. Changes introduced by the initialization procedure will depend on the method function used; see the appropriate documentation.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

`Initialize.corStruct`, `Initialize.lmeStruct`, `Initialize.glsStruct`, `Initialize.varFunc`, `isInitialized`

Examples

```
## see the method function documentation
```

`Initialize.corStruct` *Initialize corStruct Object*

Description

This method initializes `object` by evaluating its associated covariate(s) and grouping factor, if any is present, in `data`, calculating various dimensions and constants used by optimization algorithms involving `corStruct` objects (see the appropriate `Dim` method documentation), and assigning initial values for the coefficients in `object`, if none were present.

Usage

```
## S3 method for class 'corStruct':
Initialize(object, data, ...)
```

Arguments

`object`	an object inheriting from class `corStruct` representing a correlation structure.
`data`	a data frame in which to evaluate the variables defined in `formula(object)`.
`...`	this argument is included to make this method compatible with the generic.

Value

an initialized object with the same class as `object` representing a correlation structure.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

`Dim.corStruct`

Examples

```
cs1 <- corAR1(form = ~ 1 | Subject)
cs1 <- Initialize(cs1, data = Orthodont)
```

Initialize.glsStruct

Initialize a glsStruct Object

Description

The individual linear model components of the `glsStruct` list are initialized.

Usage

```
## S3 method for class 'glsStruct':
Initialize(object, data, control, ...)
```

Arguments

`object`	an object inheriting from class `glsStruct`, representing a list of linear model components, such as `corStruct` and `varFunc` objects.
`data`	a data frame in which to evaluate the variables defined in `formula(object)`.
`control`	an optional list with control parameters for the initialization and optimization algorithms used in `gls`. Defaults to `list(singular.ok = FALSE, qrTol = .Machine$single.eps)`, implying that linear dependencies are not allowed in the model and that the tolerance for detecting linear dependencies among the columns of the regression matrix is `.Machine$single.eps`.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

a `glsStruct` object similar to `object`, but with initialized model components.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`gls`, `Initialize.corStruct`, `Initialize.varFunc`, `Initialize`

Initialize.lmeStruct

Initialize an lmeStruct Object

Description

The individual linear mixed-effects model components of the `lmeStruct` list are initialized.

Usage

```
## S3 method for class 'lmeStruct':
Initialize(object, data, groups, conLin, control, ...)
```

Arguments

`object`	an object inheriting from class `lmeStruct`, representing a list of linear mixed-effects model components, such as `reStruct`, `corStruct`, and `varFunc` objects.
`data`	a data frame in which to evaluate the variables defined in `formula(object)`.
`groups`	a data frame with the grouping factors corresponding to the lme model associated with `object` as columns, sorted from innermost to outermost grouping level.
`conLin`	an optional condensed linear model object, consisting of a list with components `"Xy"`, corresponding to a regression matrix (`X`) combined with a response vector (`y`), and `"logLik"`, corresponding to the log-likelihood of the underlying lme model. Defaults to `attr(object, "conLin")`.
`control`	an optional list with control parameters for the initialization and optimization algorithms used in `lme`. Defaults to `list(niterEM=20, gradHess=TRUE)`, implying that 20 EM iterations are to be used in the derivation of initial estimates for the coefficients of the `reStruct` component of `object` and, if possible, numerical gradient vectors and Hessian matrices for the log-likelihood function are to be used in the optimization algorithm.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

an `lmeStruct` object similar to `object`, but with initialized model components.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`lme`, `Initialize.reStruct`, `Initialize.corStruct`, `Initialize.varFunc`, `Initialize`

`Initialize.reStruct` *Initialize reStruct Object*

Description

Initial estimates for the parameters in the `pdMat` objects forming `object`, which have not yet been initialized, are obtained using the methodology described in Bates and Pinheiro (1998). These estimates may be refined using a series of EM iterations, as described in Bates and Pinheiro (1998). The number of EM iterations to be used is defined in `control`.

Usage

```
## S3 method for class 'reStruct':
Initialize(object, data, conLin, control, ...)
```

Arguments

object	an object inheriting from class reStruct, representing a random effects structure and consisting of a list of pdMat objects.
data	a data frame in which to evaluate the variables defined in formula(object).
conLin	a condensed linear model object, consisting of a list with components "Xy", corresponding to a regression matrix (X) combined with a response vector (y), and "logLik", corresponding to the log-likelihood of the underlying model.
control	an optional list with a single component niterEM controlling the number of iterations for the EM algorithm used to refine initial parameter estimates. It is given as a list for compatibility with other Initialize methods. Defaults to list(niterEM = 20).
...	some methods for this generic require additional arguments. None are used in this method.

Value

an reStruct object similar to object, but with all pdMat components initialized.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Bates, D.M. and Pinheiro, J.C. (1998) "Computational methods for multilevel models" available in PostScript or PDF formats at http://franz.stat.wisc.edu/pub/NLME/

See Also

reStruct, pdMat, Initialize

Initialize.varFunc *Initialize varFunc Object*

Description

This method initializes object by evaluating its associated covariate(s) and grouping factor, if any is present, in data; determining if the covariate(s) need to be updated when the values of the coefficients associated with object change; initializing the log-likelihood and the weights associated with object; and assigning initial values for the coefficients in object, if none were present. The covariate(s) will only be initialized if no update is needed when coef(object) changes.

Usage

```
## S3 method for class 'varFunc':
Initialize(object, data, ...)
```

Arguments

object	an object inheriting from class varFunc, representing a variance function structure.
data	a data frame in which to evaluate the variables named in formula(object).
...	this argument is included to make this method compatible with the generic.

Value

an initialized object with the same class as object representing a variance function structure.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

Initialize

Examples

```
vf1 <- varPower( form = ~ age | Sex )
vf1 <- Initialize( vf1, Orthodont )
```

---

intervals *Confidence Intervals on Coefficients*

Description

Confidence intervals on the parameters associated with the model represented by object are obtained. This function is generic; method functions can be written to handle specific classes of objects. Classes which already have methods for this function include: gls, lme, and lmList.

Usage

```
intervals(object, level, ...)
```

Arguments

object	a fitted model object from which parameter estimates can be extracted.
level	an optional numeric value for the interval confidence level. Defaults to 0.95.
...	some methods for the generic may require additional arguments.

Value

will depend on the method function used; see the appropriate documentation.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

intervals.lme, intervals.lmList, intervals.gls

Examples

```
## see the method documentation
```

---

intervals.gls *Confidence Intervals on gls Parameters*

Description

Approximate confidence intervals for the parameters in the linear model represented by `object` are obtained, using a normal approximation to the distribution of the (restricted) maximum likelihood estimators (the estimators are assumed to have a normal distribution centered at the true parameter values and with covariance matrix equal to the negative inverse Hessian matrix of the (restricted) log-likelihood evaluated at the estimated parameters). Confidence intervals are obtained in an unconstrained scale first, using the normal approximation, and, if necessary, transformed to the constrained scale.

Usage

```
## S3 method for class 'gls':
intervals(object, level, which, ...)
```

Arguments

`object`	an object inheriting from class `gls`, representing a generalized least squares fitted linear model.
`level`	an optional numeric value for the interval confidence level. Defaults to 0.95.
`which`	an optional character string specifying the subset of parameters for which to construct the confidence intervals. Possible values are `"all"` for all parameters, `"var-cov"` for the variance-covariance parameters only, and `"coef"` for the linear model coefficients only. Defaults to `"all"`.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

a list with components given by data frames with rows corresponding to parameters and columns `lower`, `est.`, and `upper` representing respectively lower confidence limits, the estimated values, and upper confidence limits for the parameters. Possible components are:

`coef`	linear model coefficients, only present when `which` is not equal to `"var-cov"`.
`corStruct`	correlation parameters, only present when `which` is not equal to `"coef"` and a correlation structure is used in `object`.

varFunc	variance function parameters, only present when which is not equal to "coef" and a variance function structure is used in object.
sigma	residual standard error.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

gls, intervals, print.intervals.gls

Examples

```
fm1 <- gls(follicles ~ sin(2*pi*Time) + cos(2*pi*Time), Ovary,
           correlation = corAR1(form = ~ 1 | Mare))
intervals(fm1)
```

---

intervals.lme *Confidence Intervals on lme Parameters*

Description

Approximate confidence intervals for the parameters in the linear mixed-effects model represented by object are obtained, using a normal approximation to the distribution of the (restricted) maximum likelihood estimators (the estimators are assumed to have a normal distribution centered at the true parameter values and with covariance matrix equal to the negative inverse Hessian matrix of the (restricted) log-likelihood evaluated at the estimated parameters). Confidence intervals are obtained in an unconstrained scale first, using the normal approximation, and, if necessary, transformed to the constrained scale. The pdNatural parametrization is used for general positive-definite matrices.

Usage

```
## S3 method for class 'lme':
intervals(object, level, which, ...)
```

Arguments

object	an object inheriting from class lme, representing a fitted linear mixed-effects model.
level	an optional numeric value with the confidence level for the intervals. Defaults to 0.95.
which	an optional character string specifying the subset of parameters for which to construct the confidence intervals. Possible values are "all" for all parameters, "var-cov" for the variance-covariance parameters only, and "fixed" for the fixed effects only. Defaults to "all".
...	some methods for this generic require additional arguments. None are used in this method.

Value

a list with components given by data frames with rows corresponding to parameters and columns `lower`, `est.`, and `upper` representing respectively lower confidence limits, the estimated values, and upper confidence limits for the parameters. Possible components are:

`fixed`	fixed effects, only present when `which` is not equal to `"var-cov"`.
`reStruct`	random effects variance-covariance parameters, only present when `which` is not equal to `"fixed"`.
`corStruct`	within-group correlation parameters, only present when `which` is not equal to `"fixed"` and a correlation structure is used in `object`.
`varFunc`	within-group variance function parameters, only present when `which` is not equal to `"fixed"` and a variance function structure is used in `object`.
`sigma`	within-group standard deviation.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

`lme`, `intervals`, `print.intervals.lme`, `pdNatural`

Examples

```
fm1 <- lme(distance ~ age, Orthodont, random = ~ age | Subject)
intervals(fm1)
```

`intervals.lmList` *Confidence Intervals on lmList Coefficients*

Description

Confidence intervals on the linear model coefficients are obtained for each `lm` component of `object` and organized into a three dimensional array. The first dimension corresponding to the names of the `object` components. The second dimension is given by `lower`, `est.`, and `upper` corresponding, respectively, to the lower confidence limit, estimated coefficient, and upper confidence limit. The third dimension is given by the coefficients names.

Usage

```
## S3 method for class 'lmList':
intervals(object, level, pool, ...)
```

Arguments

object	an object inheriting from class lmList, representing a list of lm objects with a common model.
level	an optional numeric value with the confidence level for the intervals. Defaults to 0.95.
pool	an optional logical value indicating whether a pooled estimate of the residual standard error should be used. Default is attr(object, "pool").
...	some methods for this generic require additional arguments. None are used in this method.

Value

a three dimensional array with the confidence intervals and estimates for the coefficients of each lm component of object.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

lmList, intervals, plot.intervals.lmList

Examples

```
fm1 <- lmList(distance ~ age | Subject, Orthodont)
intervals(fm1)
```

---

isBalanced *Check a Design for Balance*

Description

Check the design of the experiment or study for balance.

Usage

```
isBalanced(object, countOnly, level)
```

Arguments

object	A groupedData object containing a data frame and a formula that describes the roles of variables in the data frame. The object will have one or more nested grouping factors and a primary covariate.
countOnly	A logical value indicating if the check for balance should only consider the number of observations at each level of the grouping factor(s). Defaults to FALSE.
level	an optional integer vector specifying the desired prediction levels. Levels increase from outermost to innermost grouping, with level 0 representing the population (fixed effects) predictions. Defaults to the innermost level.

Details

A design is balanced with respect to the grouping factor(s) if there are the same number of observations at each distinct value of the grouping factor or each combination of distinct levels of the nested grouping factors. If `countOnly` is `FALSE` the design is also checked for balance with respect to the primary covariate, which is often the time of the observation. A design is balanced with respect to the grouping factor and the covariate if the number of observations at each distinct level (or combination of levels for nested factors) is constant and the times at which the observations are taken (in general, the values of the primary covariates) also are constant.

Value

`TRUE` or `FALSE` according to whether the data are balanced or not

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`table`, `groupedData`

Examples

```
isBalanced(Orthodont)                      # should return TRUE
isBalanced(Orthodont, countOnly = TRUE)    # should return TRUE
isBalanced(Pixel)                          # should return FALSE
isBalanced(Pixel, level = 1)               # should return FALSE
```

`isInitialized` *Check if Object is Initialized*

Description

Checks if `object` has been initialized (generally through a call to `Initialize`), by searching for components and attributes which are modified during initialization.

Usage

```
isInitialized(object)
```

Arguments

`object`	any object requiring initialization.

Value

a logical value indicating whether `object` has been initialized.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

Initialize

Examples

```
pd1 <- pdDiag(~age)
isInitialized(pd1)
```

LDEsysMat *Generate system matrix for LDEs*

Description

Generate the system matrix for the linear differential equations determined by a compartment model.

Usage

```
LDEsysMat(pars, incidence)
```

Arguments

`pars`	a numeric vector of parameter values.
`incidence`	an integer matrix with columns named `From`, `To`, and `Par`. Values in the `Par` column must be in the range 1 to `length(pars)`. Values in the `From` column must be between 1 and the number of compartments. Values in the `To` column must be between 0 and the number of compartments.

Details

A compartment model describes material transfer between `k` in a system of `k` compartments to a linear system of differential equations. Given a description of the system and a vector of parameter values this function returns the system matrix.

This function is intended for use in a general system for solving compartment models, as described in Bates and Watts (1988).

Value

A `k` by `k` numeric matrix.

Author(s)

Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Bates, D. M. and Watts, D. G. (1988), *Nonlinear Regression Analysis and Its Applications*, Wiley, New York.

Examples

```
# incidence matrix for a two compartment open system
incidence <-
  matrix(c(1,1,2,2,2,1,3,2,0), ncol = 3, byrow = TRUE,
   dimnames = list(NULL, c("Par", "From", "To")))
incidence
LDEsysMat(c(1.2, 0.3, 0.4), incidence)
```

---

lme *Linear Mixed-Effects Models*

Description

This generic function fits a linear mixed-effects model in the formulation described in Laird and Ware (1982) but allowing for nested random effects. The within-group errors are allowed to be correlated and/or have unequal variances.

Usage

```
lme(fixed, data, random, correlation, weights, subset, method,
    na.action, control, contrasts = NULL, keep.data = TRUE)
## S3 method for class 'lme':
update(object, fixed., ..., evaluate = TRUE)
```

Arguments

`object`	an object inheriting from class `lme`, representing a fitted linear mixed-effects model.
`fixed`	a two-sided linear formula object describing the fixed-effects part of the model, with the response on the left of a ~ operator and the terms, separated by + operators, on the right, an `lmList` object, or a `groupedData` object. The method functions `lme.lmList` and `lme.groupedData` are documented separately.
`fixed.`	Changes to the fixed-effects formula – see `update.formula` for details.
`data`	an optional data frame containing the variables named in `fixed`, `random`, `correlation`, `weights`, and `subset`. By default the variables are taken from the environment from which `lme` is called.
`random`	optionally, any of the following: (i) a one-sided formula of the form `~x1+...+xn | g1/.../gm`, with `x1+...+xn` specifying the model for the random effects and `g1/.../gm` the grouping structure (`m` may be equal to 1, in which case no `/` is required). The random effects formula will be repeated for all levels of grouping, in the case of multiple levels of grouping; (ii) a list of one-sided formulas of the form `~x1+...+xn | g`, with possibly different random effects models for each grouping level. The order of nesting will be assumed the same as the order of the elements in the list; (iii) a one-sided formula of the form `~x1+...+xn`, or a `pdMat` object with a formula (i.e. a non-`NULL` value for `formula(object)`), or a list of such formulas or `pdMat` objects. In this case, the grouping structure formula will be derived from the data used to fit the linear mixed-effects model, which should inherit from class `groupedData`; (iv) a named list of formulas or `pdMat` objects as in (iii), with the grouping factors as names. The order of nesting will be assumed the same as

the order of the order of the elements in the list; (v) an `reStruct` object. See the documentation on `pdClasses` for a description of the available `pdMat` classes. Defaults to a formula consisting of the right hand side of `fixed`.

`correlation` an optional `corStruct` object describing the within-group correlation structure. See the documentation of `corClasses` for a description of the available `corStruct` classes. Defaults to `NULL`, corresponding to no within-group correlations.

`weights` an optional `varFunc` object or one-sided formula describing the within-group heteroscedasticity structure. If given as a formula, it is used as the argument to `varFixed`, corresponding to fixed variance weights. See the documentation on `varClasses` for a description of the available `varFunc` classes. Defaults to `NULL`, corresponding to homocesdatic within-group errors.

`subset` an optional expression indicating the subset of the rows of `data` that should be used in the fit. This can be a logical vector, or a numeric vector indicating which observation numbers are to be included, or a character vector of the row names to be included. All observations are included by default.

`method` a character string. If `"REML"` the model is fit by maximizing the restricted log-likelihood. If `"ML"` the log-likelihood is maximized. Defaults to `"REML"`.

`na.action` a function that indicates what should happen when the data contain `NA`s. The default action (`na.fail`) causes `lme` to print an error message and terminate if there are any incomplete observations.

`control` a list of control values for the estimation algorithm to replace the default values returned by the function `lmeControl`. Defaults to an empty list.

`contrasts` an optional list. See the `contrasts.arg` of `model.matrix.default`.

`keep.data` logical: should the `data` argument (if supplied and a data frame) be saved as part of the model object?

`...` some methods for this generic require additional arguments. None are used in this method.

`evaluate` If `TRUE` evaluate the new call else return the call.

Value

an object of class `lme` representing the linear mixed-effects model fit. Generic functions such as `print`, `plot` and `summary` have methods to show the results of the fit. See `lmeObject` for the components of the fit. The functions `resid`, `coef`, `fitted`, `fixed.effects`, and `random.effects` can be used to extract some of its components.

Author(s)

Jose Pinheiro ⟨jose.pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

The computational methods are described in Bates, D.M. and Pinheiro (1998) and follow on the general framework of Lindstrom, M.J. and Bates, D.M. (1988). The model formulation is described in Laird, N.M. and Ware, J.H. (1982). The variance-covariance parametrizations are described in <Pinheiro, J.C. and Bates., D.M. (1996). The different correlation structures available for the `correlation` argument are described in Box, G.E.P., Jenkins, G.M., and Reinsel G.C. (1994), Littel, R.C., Milliken, G.A., Stroup, W.W., and Wolfinger, R.D. (1996), and Venables, W.N. and

Ripley, B.D. (1997). The use of variance functions for linear and nonlinear mixed effects models is presented in detail in Davidian, M. and Giltinan, D.M. (1995).

Bates, D.M. and Pinheiro, J.C. (1998) "Computational methods for multilevel models" available in PostScript or PDF formats at http://franz.stat.wisc.edu/pub/NLME/

Box, G.E.P., Jenkins, G.M., and Reinsel G.C. (1994) "Time Series Analysis: Forecasting and Control", 3rd Edition, Holden-Day.

Davidian, M. and Giltinan, D.M. (1995) "Nonlinear Mixed Effects Models for Repeated Measurement Data", Chapman and Hall.

Laird, N.M. and Ware, J.H. (1982) "Random-Effects Models for Longitudinal Data", Biometrics, 38, 963-974.

Lindstrom, M.J. and Bates, D.M. (1988) "Newton-Raphson and EM Algorithms for Linear Mixed-Effects Models for Repeated-Measures Data", Journal of the American Statistical Association, 83, 1014-1022.

Littel, R.C., Milliken, G.A., Stroup, W.W., and Wolfinger, R.D. (1996) "SAS Systems for Mixed Models", SAS Institute.

Pinheiro, J.C. and Bates., D.M. (1996) "Unconstrained Parametrizations for Variance-Covariance Matrices", Statistics and Computing, 6, 289-296.

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

Venables, W.N. and Ripley, B.D. (1997) "Modern Applied Statistics with S-plus", 2nd Edition, Springer-Verlag.

See Also

corClasses, lme.lmList, lme.groupedData, lmeControl, lmeObject, lmeStruct, lmList, pdClasses, plot.lme, predict.lme, qqnorm.lme, residuals.lme, reStruct, simulate.lme, summary.lme, varClasses, varFunc

Examples

```
fm1 <- lme(distance ~ age, data = Orthodont) # random is ~ age
fm2 <- lme(distance ~ age + Sex, data = Orthodont, random = ~ 1)
summary(fm1)
summary(fm2)
```

---

lme.groupedData *LME fit from groupedData Object*

Description

The response variable and primary covariate in `formula(fixed)` are used to construct the fixed effects model formula. This formula and the `groupedData` object are passed as the `fixed` and `data` arguments to `lme.formula`, together with any other additional arguments in the function call. See the documentation on `lme.formula` for a description of that function.

Usage

```
## S3 method for class 'groupedData':
lme(fixed, data, random, correlation, weights,
    subset, method, na.action, control, contrasts, keep.data = TRUE)
```

Arguments

fixed	a data frame inheriting from class groupedData.
data	this argument is included for consistency with the generic function. It is ignored in this method function.
random	optionally, any of the following: (i) a one-sided formula of the form ~x1+...+xn | g1/.../gm, with x1+...+xn specifying the model for the random effects and g1/.../gm the grouping structure (m may be equal to 1, in which case no / is required). The random effects formula will be repeated for all levels of grouping, in the case of multiple levels of grouping; (ii) a list of one-sided formulas of the form ~x1+...+xn | g, with possibly different random effects models for each grouping level. The order of nesting will be assumed the same as the order of the elements in the list; (iii) a one-sided formula of the form ~x1+...+xn, or a pdMat object with a formula (i.e. a non-NULL value for formula(object)), or a list of such formulas or pdMat objects. In this case, the grouping structure formula will be derived from the data used to fit the linear mixed-effects model, which should inherit from class groupedData; (iv) a named list of formulas or pdMat objects as in (iii), with the grouping factors as names. The order of nesting will be assumed the same as the order of the order of the elements in the list; (v) an reStruct object. See the documentation on pdClasses for a description of the available pdMat classes. Defaults to a formula consisting of the right hand side of fixed.
correlation	an optional corStruct object describing the within-group correlation structure. See the documentation of corClasses for a description of the available corStruct classes. Defaults to NULL, corresponding to no within-group correlations.
weights	an optional varFunc object or one-sided formula describing the within-group heteroscedasticity structure. If given as a formula, it is used as the argument to varFixed, corresponding to fixed variance weights. See the documentation on varClasses for a description of the available varFunc classes. Defaults to NULL, corresponding to homocesdatic within-group errors.
subset	an optional expression indicating the subset of the rows of data that should be used in the fit. This can be a logical vector, or a numeric vector indicating which observation numbers are to be included, or a character vector of the row names to be included. All observations are included by default.
method	a character string. If "REML" the model is fit by maximizing the restricted log-likelihood. If "ML" the log-likelihood is maximized. Defaults to "REML".
na.action	a function that indicates what should happen when the data contain NAs. The default action (na.fail) causes lme to print an error message and terminate if there are any incomplete observations.
control	a list of control values for the estimation algorithm to replace the default values returned by the function lmeControl. Defaults to an empty list.
contrasts	an optional list. See the contrasts.arg of model.matrix.default.
keep.data	logical: should the data argument (if supplied and a data frame) be saved as part of the model object?

Value

an object of class lme representing the linear mixed-effects model fit. Generic functions such as print, plot and summary have methods to show the results of the fit. See lmeObject

for the components of the fit. The functions `resid`, `coef`, `fitted`, `fixed.effects`, and `random.effects` can be used to extract some of its components.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

The computational methods are described in Bates, D.M. and Pinheiro (1998) and follow on the general framework of Lindstrom, M.J. and Bates, D.M. (1988). The model formulation is described in Laird, N.M. and Ware, J.H. (1982). The variance-covariance parametrizations are described in Pinheiro, J.C. and Bates., D.M. (1996). The different correlation structures available for the `correlation` argument are described in Box, G.E.P., Jenkins, G.M., and Reinsel G.C. (1994), Littel, R.C., Milliken, G.A., Stroup, W.W., and Wolfinger, R.D. (1996), and Venables, W.N. and Ripley, B.D. (1997). The use of variance functions for linear and nonlinear mixed effects models is presented in detail in Davidian, M. and Giltinan, D.M. (1995).

Bates, D.M. and Pinheiro, J.C. (1998) "Computational methods for multilevel models" available in PostScript or PDF formats at http://franz.stat.wisc.edu/pub/NLME/

Box, G.E.P., Jenkins, G.M., and Reinsel G.C. (1994) "Time Series Analysis: Forecasting and Control", 3rd Edition, Holden-Day.

Davidian, M. and Giltinan, D.M. (1995) "Nonlinear Mixed Effects Models for Repeated Measurement Data", Chapman and Hall.

Laird, N.M. and Ware, J.H. (1982) "Random-Effects Models for Longitudinal Data", Biometrics, 38, 963-974.

Lindstrom, M.J. and Bates, D.M. (1988) "Newton-Raphson and EM Algorithms for Linear Mixed-Effects Models for Repeated-Measures Data", Journal of the American Statistical Association, 83, 1014-1022.

Littel, R.C., Milliken, G.A., Stroup, W.W., and Wolfinger, R.D. (1996) "SAS Systems for Mixed Models", SAS Institute.

Pinheiro, J.C. and Bates., D.M. (1996) "Unconstrained Parametrizations for Variance-Covariance Matrices", Statistics and Computing, 6, 289-296.

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

Venables, W.N. and Ripley, B.D. (1997) "Modern Applied Statistics with S-plus", 2nd Edition, Springer-Verlag.

See Also

`lme`, `groupedData`, `lmeObject`

Examples

```
fm1 <- lme(Orthodont)
summary(fm1)
```

`lme.lmList` *LME fit from lmList Object*

Description

If the random effects names defined in `random` are a subset of the `lmList` object coefficient names, initial estimates for the covariance matrix of the random effects are obtained (overwriting any values given in `random`). `formula(fixed)` and the `data` argument in the calling sequence used to obtain `fixed` are passed as the `fixed` and `data` arguments to `lme.formula`, together with any other additional arguments in the function call. See the documentation on `lme.formula` for a description of that function.

Usage

```
## S3 method for class 'lmList':
lme(fixed, data, random, correlation, weights, subset, method,
    na.action, control, contrasts, keep.data)
```

Arguments

`fixed`	an object inheriting from class `lmList`, representing a list of `lm` fits with a common model.
`data`	this argument is included for consistency with the generic function. It is ignored in this method function.
`random`	an optional one-sided linear formula with no conditioning expression, or a `pdMat` object with a `formula` attribute. Multiple levels of grouping are not allowed with this method function. Defaults to a formula consisting of the right hand side of `formula(fixed)`.
`correlation`	an optional `corStruct` object describing the within-group correlation structure. See the documentation of `corClasses` for a description of the available `corStruct` classes. Defaults to `NULL`, corresponding to no within-group correlations.
`weights`	an optional `varFunc` object or one-sided formula describing the within-group heteroscedasticity structure. If given as a formula, it is used as the argument to `varFixed`, corresponding to fixed variance weights. See the documentation on `varClasses` for a description of the available `varFunc` classes. Defaults to `NULL`, corresponding to homocesdatic within-group errors.
`subset`	an optional expression indicating the subset of the rows of `data` that should be used in the fit. This can be a logical vector, or a numeric vector indicating which observation numbers are to be included, or a character vector of the row names to be included. All observations are included by default.
`method`	a character string. If `"REML"` the model is fit by maximizing the restricted log-likelihood. If `"ML"` the log-likelihood is maximized. Defaults to `"REML"`.
`na.action`	a function that indicates what should happen when the data contain `NA`s. The default action (`na.fail`) causes `lme` to print an error message and terminate if there are any incomplete observations.
`control`	a list of control values for the estimation algorithm to replace the default values returned by the function `lmeControl`. Defaults to an empty list.

contrasts	an optional list. See the contrasts.arg of model.matrix.default.
keep.data	logical: should the data argument (if supplied and a data frame) be saved as part of the model object?

Value

an object of class lme representing the linear mixed-effects model fit. Generic functions such as print, plot and summary have methods to show the results of the fit. See lmeObject for the components of the fit. The functions resid, coef, fitted, fixed.effects, and random.effects can be used to extract some of its components.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

The computational methods are described in Bates, D.M. and Pinheiro (1998) and follow on the general framework of Lindstrom, M.J. and Bates, D.M. (1988). The model formulation is described in Laird, N.M. and Ware, J.H. (1982). The variance-covariance parametrizations are described in <Pinheiro, J.C. and Bates., D.M. (1996). The different correlation structures available for the correlation argument are described in Box, G.E.P., Jenkins, G.M., and Reinsel G.C. (1994), Littel, R.C., Milliken, G.A., Stroup, W.W., and Wolfinger, R.D. (1996), and Venables, W.N. and Ripley, B.D. (1997). The use of variance functions for linear and nonlinear mixed effects models is presented in detail in Davidian, M. and Giltinan, D.M. (1995).

Bates, D.M. and Pinheiro, J.C. (1998) "Computational methods for multilevel models" available in PostScript or PDF formats at http://franz.stat.wisc.edu/pub/NLME/

Box, G.E.P., Jenkins, G.M., and Reinsel G.C. (1994) "Time Series Analysis: Forecasting and Control", 3rd Edition, Holden-Day.

Davidian, M. and Giltinan, D.M. (1995) "Nonlinear Mixed Effects Models for Repeated Measurement Data", Chapman and Hall.

Laird, N.M. and Ware, J.H. (1982) "Random-Effects Models for Longitudinal Data", Biometrics, 38, 963-974.

Lindstrom, M.J. and Bates, D.M. (1988) "Newton-Raphson and EM Algorithms for Linear Mixed-Effects Models for Repeated-Measures Data", Journal of the American Statistical Association, 83, 1014-1022.

Littel, R.C., Milliken, G.A., Stroup, W.W., and Wolfinger, R.D. (1996) "SAS Systems for Mixed Models", SAS Institute.

Pinheiro, J.C. and Bates., D.M. (1996) "Unconstrained Parametrizations for Variance-Covariance Matrices", Statistics and Computing, 6, 289-296.

Venables, W.N. and Ripley, B.D. (1997) "Modern Applied Statistics with S-plus", 2nd Edition, Springer-Verlag.

See Also

lme, lmList, lmeObject

Examples

```
fm1 <- lmList(Orthodont)
fm2 <- lme(fm1)
summary(fm1)
summary(fm2)
```

---

lmeControl *Control Values for lme Fit*

Description

The values supplied in the function call replace the defaults and a list with all possible arguments is returned. The returned list is used as the `control` argument to the `lme` function.

Usage

```
lmeControl(maxIter, msMaxIter, tolerance, niterEM, msMaxEval, msTol,
           msScale, msVerbose, returnObject, gradHess, apVar,
           .relStep, minAbsParApVar, nlmStepMax,
           opt = c("nlminb", "optim"), optimMethod,
           natural)
```

Arguments

`maxIter`	maximum number of iterations for the `lme` optimization algorithm. Default is 50.
`msMaxIter`	maximum number of iterations for the `nlm` optimization step inside the `lme` optimization. Default is 50.
`tolerance`	tolerance for the convergence criterion in the `lme` algorithm. Default is 1e-6.
`niterEM`	number of iterations for the EM algorithm used to refine the initial estimates of the random effects variance-covariance coefficients. Default is 25.
`msMaxEval`	maximum number of evaluations of the objective function permitted for nlminb. Default is 200.
`msTol`	tolerance for the convergence criterion in `nlm`, passed as the `rel.tolerance` argument to the function (see documentation on `nlm`). Default is 1e-7.
`msScale`	scale function passed as the `scale` argument to the `nlm` function (see documentation on that function). Default is `lmeScale`.
`msVerbose`	a logical value passed as the `trace` argument to `nlm` (see documentation on that function). Default is `FALSE`.
`returnObject`	a logical value indicating whether the fitted object should be returned when the maximum number of iterations is reached without convergence of the algorithm. Default is `FALSE`.
`gradHess`	a logical value indicating whether numerical gradient vectors and Hessian matrices of the log-likelihood function should be used in the `nlm` optimization. This option is only available when the correlation structure (`corStruct`) and the variance function structure (`varFunc`) have no "varying" parameters and the `pdMat` classes used in the random effects structure are `pdSymm` (general positive-definite), `pdDiag` (diagonal), `pdIdent` (multiple of the identity), or `pdCompSymm` (compound symmetry). Default is `TRUE`.

apVar	a logical value indicating whether the approximate covariance matrix of the variance-covariance parameters should be calculated. Default is TRUE.
.relStep	relative step for numerical derivatives calculations. Default is .Machine$double.eps^(1/3).
nlmStepMax	stepmax value to be passed to nlm. See nlm for details. Default is 100.0
opt	the optimizer to be used, either nlminb (the default since (R 2.2.0) or optim (the previous default).
optimMethod	character - the optimization method to be used with the optim optimizer. The default is "BFGS". An alternative is "L-BFGS-B".
minAbsParApVar	numeric value - minimum absolute parameter value in the approximate variance calculation. The default is 0.05.
natural	a logical value indicating whether the pdNatural parametrization should be used for general positive-definite matrices (pdSymm) in reStruct, when the approximate covariance matrix of the estimators is calculated. Default is TRUE.

Value

a list with components for each of the possible arguments.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

lme, nlm, optim, lmeScale

Examples

```
# decrease the maximum number iterations in the ms call and
# request that information on the evolution of the ms iterations be printed
lmeControl(msMaxIter = 20, msVerbose = TRUE)
```

---

lmeObject *Fitted lme Object*

Description

An object returned by the lme function, inheriting from class lme and representing a fitted linear mixed-effects model. Objects of this class have methods for the generic functions anova, coef, fitted, fixed.effects, formula, getGroups, getResponse, intervals, logLik, pairs, plot, predict, print, random.effects, residuals, summary, and update.

Value

The following components must be included in a legitimate `lme` object.

`apVar`	an approximate covariance matrix for the variance-covariance coefficients. If `apVar = FALSE` in the list of control values used in the call to `lme`, this component is equal to `NULL`.
`call`	a list containing an image of the `lme` call that produced the object.
`coefficients`	a list with two components, `fixed` and `random`, where the first is a vector containing the estimated fixed effects and the second is a list of matrices with the estimated random effects for each level of grouping. For each matrix in the `random` list, the columns refer to the random effects and the rows to the groups.
`contrasts`	a list with the contrasts used to represent factors in the fixed effects formula and/or random effects formula. This information is important for making predictions from a new data frame in which not all levels of the original factors are observed. If no factors are used in the lme model, this component will be an empty list.
`dims`	a list with basic dimensions used in the lme fit, including the components `N` - the number of observations in the data, `Q` - the number of grouping levels, `qvec` - the number of random effects at each level from innermost to outermost (last two values are equal to zero and correspond to the fixed effects and the response), `ngrps` - the number of groups at each level from innermost to outermost (last two values are one and correspond to the fixed effects and the response), and `ncol` - the number of columns in the model matrix for each level of grouping from innermost to outermost (last two values are equal to the number of fixed effects and one).
`fitted`	a data frame with the fitted values as columns. The leftmost column corresponds to the population fixed effects (corresponding to the fixed effects only) and successive columns from left to right correspond to increasing levels of grouping.
`fixDF`	a list with components `X` and `terms` specifying the denominator degrees of freedom for, respectively, t-tests for the individual fixed effects and F-tests for the fixed-effects terms in the models.
`groups`	a data frame with the grouping factors as columns. The grouping level increases from left to right.
`logLik`	the (restricted) log-likelihood at convergence.
`method`	the estimation method: either `"ML"` for maximum likelihood, or `"REML"` for restricted maximum likelihood.
`modelStruct`	an object inheriting from class `lmeStruct`, representing a list of mixed-effects model components, such as `reStruct`, `corStruct`, and `varFunc` objects.
`numIter`	the number of iterations used in the iterative algorithm.
`residuals`	a data frame with the residuals as columns. The leftmost column corresponds to the population residuals and successive columns from left to right correspond to increasing levels of grouping.
`sigma`	the estimated within-group error standard deviation.
`varFix`	an approximate covariance matrix of the fixed effects estimates.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

lme, lmeStruct

`lmeScale` *Scale for lme Optimization*

Description

This function calculates the scales to be used for each coefficient estimated through an `nlm` optimization in the `lme` function. If all initial values are zero, the scale is set to one for all coefficients; else, the scale for a coefficient with non-zero initial value is equal to the inverse of its initial value and the scale for the coefficients with initial value equal to zero is set to the median of the non-zero initial value coefficients.

Usage

```
lmeScale(start)
```

Arguments

`start`	the starting values for the coefficients to be estimated.

Value

a vector with the scales to be used in `nlm` for estimating the coefficients.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

nlm

`lmeStruct` *Linear Mixed-Effects Structure*

Description

A linear mixed-effects structure is a list of model components representing different sets of parameters in the linear mixed-effects model. An `lmeStruct` list must contain at least a `reStruct` object, but may also contain `corStruct` and `varFunc` objects. `NULL` arguments are not included in the `lmeStruct` list.

Usage

```
lmeStruct(reStruct, corStruct, varStruct)
```

Arguments

reStruct	a reStruct representing a random effects structure.
corStruct	an optional corStruct object, representing a correlation structure. Default is NULL.
varStruct	an optional varFunc object, representing a variance function structure. Default is NULL.

Value

a list of model components determining the parameters to be estimated for the associated linear mixed-effects model.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

corClasses, lme, residuals.lmeStruct, reStruct, varFunc

Examples

```
lms1 <- lmeStruct(reStruct(~age), corAR1(), varPower())
```

---

lmList — *List of lm Objects with a Common Model*

Description

Data is partitioned according to the levels of the grouping factor g and individual lm fits are obtained for each data partition, using the model defined in object.

Usage

```
lmList(object, data, level, subset, na.action, pool)
## S3 method for class 'lmList':
update(object, formula., ..., evaluate = TRUE)
## S3 method for class 'lmList':
print(x, pool, ...)
```

Arguments

object	For lmList, either a linear formula object of the form y ~ x1+...+xn | g or a groupedData object. In the formula object, y represents the response, x1,...,xn the covariates, and g the grouping factor specifying the partitioning of the data according to which different lm fits should be performed. The grouping factor g may be omitted from the formula, in which case the grouping structure will be obtained from data, which must inherit from class groupedData. The method function lmList.groupedData is documented separately. For the method update.lmList, object is an object inheriting from class lmList.

formula	(used in update.lmList only) a two-sided linear formula with the common model for the individuals lm fits.
formula.	Changes to the formula – see update.formula for details.
data	a data frame in which to interpret the variables named in object.
level	an optional integer specifying the level of grouping to be used when multiple nested levels of grouping are present.
subset	an optional expression indicating which subset of the rows of data should be used in the fit. This can be a logical vector, or a numeric vector indicating which observation numbers are to be included, or a character vector of the row names to be included. All observations are included by default.
na.action	a function that indicates what should happen when the data contain NAs. The default action (na.fail) causes lmList to print an error message and terminate if there are any incomplete observations.
pool	an optional logical value indicating whether a pooled estimate of the residual standard error should be used in calculations of standard deviations or standard errors for summaries.
x	an object inheriting from class lmList to be printed.
...	some methods for this generic require additional arguments. None are used in this method.
evaluate	If TRUE evaluate the new call else return the call.

Value

a list of lm objects with as many components as the number of groups defined by the grouping factor. Generic functions such as coef, fixed.effects, lme, pairs, plot, predict, random.effects, summary, and update have methods that can be applied to an lmList object.

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

lm, lme.lmList, plot.lmList, pooledSD, predict.lmList, residuals.lmList, summary.lmList

Examples

```
fm1 <- lmList(distance ~ age | Subject, Orthodont)
summary(fm1)
```

lmList.groupedData *lmList Fit from a groupedData Object*

Description

The response variable and primary covariate in `formula(object)` are used to construct the linear model formula. This formula and the `groupedData` object are passed as the `object` and `data` arguments to `lmList.formula`, together with any other additional arguments in the function call. See the documentation on `lmList.formula` for a description of that function.

Usage

```
## S3 method for class 'groupedData':
lmList(object, data, level, subset, na.action, pool)
```

Arguments

`object`	a data frame inheriting from class `groupedData`.
`data`	this argument is included for consistency with the generic function. It is ignored in this method function.
`level`	an optional integer specifying the level of grouping to be used when multiple nested levels of grouping are present.
`subset`	an optional expression indicating which subset of the rows of `data` should be used in the fit. This can be a logical vector, or a numeric vector indicating which observation numbers are to be included, or a character vector of the row names to be included. All observations are included by default.
`na.action`	a function that indicates what should happen when the data contain `NA`s. The default action (`na.fail`) causes `lmList` to print an error message and terminate if there are any incomplete observations.
`pool`	an optional logical value that is preserved as an attribute of the returned value. This will be used as the default for `pool` in calculations of standard deviations or standard errors for summaries.

Value

a list of `lm` objects with as many components as the number of groups defined by the grouping factor. Generic functions such as `coef`, `fixed.effects`, `lme`, `pairs`, `plot`, `predict`, `random.effects`, `summary`, and `update` have methods that can be applied to an `lmList` object.

See Also

`groupedData`, `lm`, `lme.lmList`, `lmList`, `lmList.formula`

Examples

```
fm1 <- lmList(Orthodont)
summary(fm1)
```

logDet *Extract the Logarithm of the Determinant*

Description

This function is generic; method functions can be written to handle specific classes of objects. Classes which already have methods for this function include: `corStruct`, several `pdMat` classes, and `reStruct`.

Usage

```
logDet(object, ...)
```

Arguments

`object`	any object from which a matrix, or list of matrices, can be extracted
`...`	some methods for this generic function require additional arguments.

Value

will depend on the method function used; see the appropriate documentation.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`logLik`, `logDet.corStruct`, `logDet.pdMat`, `logDet.reStruct`

Examples

```
## see the method function documentation
```

logDet.corStruct *Extract corStruct Log-Determinant*

Description

This method function extracts the logarithm of the determinant of a square-root factor of the correlation matrix associated with `object`, or the sum of the log-determinants of square-root factors of the list of correlation matrices associated with `object`.

Usage

```
## S3 method for class 'corStruct':
logDet(object, covariate, ...)
```

Arguments

object	an object inheriting from class corStruct, representing a correlation structure.
covariate	an optional covariate vector (matrix), or list of covariate vectors (matrices), at which values the correlation matrix, or list of correlation matrices, are to be evaluated. Defaults to getCovariate(object).
...	some methods for this generic require additional arguments. None are used in this method.

Value

the log-determinant of a square-root factor of the correlation matrix associated with object, or the sum of the log-determinants of square-root factors of the list of correlation matrices associated with object.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

logLik.corStruct, corMatrix.corStruct, logDet

Examples

```
cs1 <- corAR1(0.3)
logDet(cs1, covariate = 1:4)
```

---

logDet.pdMat *Extract Log-Determinant from a pdMat Object*

Description

This method function extracts the logarithm of the determinant of a square-root factor of the positive-definite matrix represented by object.

Usage

```
## S3 method for class 'pdMat':
logDet(object, ...)
```

Arguments

object	an object inheriting from class pdMat, representing a positive definite matrix.
...	some methods for this generic require additional arguments. None are used in this method.

Value

the log-determinant of a square-root factor of the positive-definite matrix represented by object.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

pdMat, logDet

Examples

```
pd1 <- pdSymm(diag(1:3))
logDet(pd1)
```

---

logDet.reStruct *Extract reStruct Log-Determinants*

Description

Calculates, for each of the pdMat components of object, the logarithm of the determinant of a square-root factor.

Usage

```
## S3 method for class 'reStruct':
logDet(object, ...)
```

Arguments

object	an object inheriting from class reStruct, representing a random effects structure and consisting of a list of pdMat objects.
...	some methods for this generic require additional arguments. None are used in this method.

Value

a vector with the log-determinants of square-root factors of the pdMat components of object.

Author(s)

Jose Pinheiro

See Also

reStruct, pdMat, logDet

Examples

```
rs1 <- reStruct(list(A = pdSymm(diag(1:3), form = ~Score),
  B = pdDiag(2 * diag(4), form = ~Educ)))
logDet(rs1)
```

`logLik.corStruct` *Extract corStruct Log-Likelihood*

Description

This method function extracts the component of a Gaussian log-likelihood associated with the correlation structure, which is equal to the negative of the logarithm of the determinant (or sum of the logarithms of the determinants) of the matrix (or matrices) represented by `object`.

Usage

```
## S3 method for class 'corStruct':
logLik(object, data, ...)
```

Arguments

`object`	an object inheriting from class `corStruct`, representing a correlation structure.
`data`	this argument is included to make this method function compatible with other `logLik` methods and will be ignored.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

the negative of the logarithm of the determinant (or sum of the logarithms of the determinants) of the correlation matrix (or matrices) represented by `object`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`logDet.corStruct`, `logLik.lme`,

Examples

```
cs1 <- corAR1(0.2)
cs1 <- Initialize(cs1, data = Orthodont)
logLik(cs1)
```

logLik.glsStruct *Log-Likelihood of a glsStruct Object*

Description

`Pars` is used to update the coefficients of the model components of `object` and the individual (restricted) log-likelihood contributions of each component are added together. The type of log-likelihood (restricted or not) is determined by the `settings` attribute of `object`.

Usage

```
## S3 method for class 'glsStruct':
logLik(object, Pars, conLin, ...)
```

Arguments

`object`	an object inheriting from class `glsStruct`, representing a list of linear model components, such as `corStruct` and `varFunc` objects.
`Pars`	the parameter values at which the (restricted) log-likelihood is to be evaluated.
`conLin`	an optional condensed linear model object, consisting of a list with components `"Xy"`, corresponding to a regression matrix (`X`) combined with a response vector (`y`), and `"logLik"`, corresponding to the log-likelihood of the underlying linear model. Defaults to `attr(object, "conLin")`.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

the (restricted) log-likelihood for the linear model described by `object`, evaluated at `Pars`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`gls`, `glsStruct`, `logLik.lme`

logLik.gnls *Log-Likelihood of a gnls Object*

Description

Returns the log-likelihood value of the nonlinear model represented by `object` evaluated at the estimated coefficients.

Usage

```
## S3 method for class 'gnls':
logLik(object, REML, ...)
```

Arguments

object	an object inheriting from class gnls, representing a generalized nonlinear least squares fitted model.
REML	an optional logical value. If TRUE the restricted log-likelihood is returned, else, if FALSE, the log-likelihood is returned. Defaults to FALSE.
...	some methods for this generic require additional arguments. None are used in this method.

Value

the log-likelihood of the linear model represented by object evaluated at the estimated coefficients.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

gnls, logLik.lme

Examples

```
fm1 <- gnls(weight ~ SSlogis(Time, Asym, xmid, scal), Soybean,
            weights = varPower())
logLik(fm1)
```

---

logLik.gnlsStruct *Log-Likelihood of a gnlsStruct Object*

Description

Pars is used to update the coefficients of the model components of object and the individual log-likelihood contributions of each component are added together.

Usage

```
## S3 method for class 'gnlsStruct':
logLik(object, Pars, conLin, ...)
```

Arguments

object	an object inheriting from class gnlsStruct, representing a list of model components, such as corStruct and varFunc objects, and attributes specifying the underlying nonlinear model and the response variable.
Pars	the parameter values at which the log-likelihood is to be evaluated.
conLin	an optional condensed linear model object, consisting of a list with components "Xy", corresponding to a regression matrix (X) combined with a response vector (y), and "logLik", corresponding to the log-likelihood of the underlying nonlinear model. Defaults to attr(object, "conLin").
...	some methods for this generic require additional arguments. None are used in this method.

Value

the log-likelihood for the linear model described by `object`, evaluated at `Pars`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`gnls`, `gnlsStruct`, `logLik.gnls`

---

logLik.lme *Log-Likelihood of an lme Object*

Description

If `REML=FALSE`, returns the log-likelihood value of the linear mixed-effects model represented by `object` evaluated at the estimated coefficients; else, the restricted log-likelihood evaluated at the estimated coefficients is returned.

Usage

```
## S3 method for class 'lme':
logLik(object, REML, ...)
```

Arguments

`object`	an object inheriting from class `lme`, representing a fitted linear mixed-effects model.
`REML`	an optional logical value. If `TRUE` the restricted log-likelihood is returned, else, if `FALSE`, the log-likelihood is returned. Defaults to `FALSE`.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

the (restricted) log-likelihood of the model represented by `object` evaluated at the estimated coefficients.

Author(s)

Jose Pinheiro and Douglas Bates

References

Harville, D.A. (1974) “Bayesian Inference for Variance Components Using Only Error Contrasts”, *Biometrika*, **61**, 383–385.

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

lme,gls, logLik.corStruct, logLik.glsStruct, logLik.lmeStruct, logLik.lmList, logLik.reStruct, logLik.varFunc,

Examples

```
fm1 <- lme(distance ~ Sex * age, Orthodont, random = ~ age, method = "ML")
logLik(fm1)
logLik(fm1, REML = TRUE)
```

---

logLik.lmeStruct *Log-Likelihood of an lmeStruct Object*

Description

Pars is used to update the coefficients of the model components of object and the individual (restricted) log-likelihood contributions of each component are added together. The type of log-likelihood (restricted or not) is determined by the settings attribute of object.

Usage

```
## S3 method for class 'lmeStruct':
logLik(object, Pars, conLin, ...)
```

Arguments

object	an object inheriting from class lmeStruct, representing a list of linear mixed-effects model components, such as reStruct, corStruct, and varFunc objects.
Pars	the parameter values at which the (restricted) log-likelihood is to be evaluated.
conLin	an optional condensed linear model object, consisting of a list with components "Xy", corresponding to a regression matrix (X) combined with a response vector (y), and "logLik", corresponding to the log-likelihood of the underlying lme model. Defaults to attr(object, "conLin").
...	some methods for this generic require additional arguments. None are used in this method.

Value

the (restricted) log-likelihood for the linear mixed-effects model described by object, evaluated at Pars.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

lme, lmeStruct, logLik.lme

`logLik.lmList` *Log-Likelihood of an lmList Object*

Description

If `pool=FALSE`, the (restricted) log-likelihoods of the `lm` components of `object` are summed together. Else, the (restricted) log-likelihood of the `lm` fit with different coefficients for each level of the grouping factor associated with the partitioning of the `object` components is obtained.

Usage

```
## S3 method for class 'lmList':
logLik(object, REML, pool, ...)
```

Arguments

`object`	an object inheriting from class `lmList`, representing a list of `lm` objects with a common model.
`REML`	an optional logical value. If `TRUE` the restricted log-likelihood is returned, else, if `FALSE`, the log-likelihood is returned. Defaults to `FALSE`.
`pool`	an optional logical value indicating whether all `lm` components of `object` may be assumed to have the same error variance. Default is `attr(object, "pool")`.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

either the sum of the (restricted) log-likelihoods of each `lm` component in `object`, or the (restricted) log-likelihood for the `lm` fit with separate coefficients for each component of `object`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`lmList`, `logLik.lme`,

Examples

```
fm1 <- lmList(distance ~ age | Subject, Orthodont)
logLik(fm1)   # returns NA when it should not
```

`logLik.reStruct` *Calculate reStruct Log-Likelihood*

Description

Calculates the log-likelihood, or restricted log-likelihood, of the Gaussian linear mixed-effects model represented by `object` and `conLin` (assuming spherical within-group covariance structure), evaluated at `coef(object)`. The `settings` attribute of `object` determines whether the log-likelihood, or the restricted log-likelihood, is to be calculated. The computational methods are described in Bates and Pinheiro (1998).

Usage

```
## S3 method for class 'reStruct':
logLik(object, conLin, ...)
```

Arguments

`object`	an object inheriting from class `reStruct`, representing a random effects structure and consisting of a list of `pdMat` objects.
`conLin`	a condensed linear model object, consisting of a list with components `"Xy"`, corresponding to a regression matrix (`X`) combined with a response vector (`y`), and `"logLik"`, corresponding to the log-likelihood of the underlying model.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

the log-likelihood, or restricted log-likelihood, of linear mixed-effects model represented by `object` and `conLin`, evaluated at `coef{object}`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Bates, D.M. and Pinheiro, J.C. (1998) "Computational methods for multilevel models" available in PostScript or PDF formats at http://franz.stat.wisc.edu/pub/NLME/

See Also

`reStruct`, `pdMat`, `logLik.lme`

`logLik.varFunc` *Extract varFunc logLik*

Description

This method function extracts the component of a Gaussian log-likelihood associated with the variance function structure represented by `object`, which is equal to the sum of the logarithms of the corresponding weights.

Usage

```
## S3 method for class 'varFunc':
logLik(object, data, ...)
```

Arguments

`object`	an object inheriting from class `varFunc`, representing a variance function structure.
`data`	this argument is included to make this method function compatible with other `logLik` methods and will be ignored.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

the sum of the logarithms of the weights corresponding to the variance function structure represented by `object`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`logLik.lme`

Examples

```
vf1 <- varPower(form = ~age)
vf1 <- Initialize(vf1, Orthodont)
coef(vf1) <- 0.1
logLik(vf1)
```

Machines *Productivity Scores for Machines and Workers*

Description

The `Machines` data frame has 54 rows and 3 columns.

Format

This data frame contains the following columns:

Worker an ordered factor giving the unique identifier for the worker.

Machine a factor with levels `A`, `B`, and `C` identifying the machine brand.

score a productivity score.

Details

Data on an experiment to compare three brands of machines used in an industrial process are presented in Milliken and Johnson (p. 285, 1992). Six workers were chosen randomly among the employees of a factory to operate each machine three times. The response is an overall productivity score taking into account the number and quality of components produced.

Source

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York. (Appendix A.14)

Milliken, G. A. and Johnson, D. E. (1992), *Analysis of Messy Data, Volume I: Designed Experiments*, Chapman and Hall, London.

MathAchieve *Mathematics achievement scores*

Description

The `MathAchieve` data frame has 7185 rows and 6 columns.

Format

This data frame contains the following columns:

School an ordered factor identifying the school that the student attends

Minority a factor with levels `No Yes` indicating if the student is a member of a minority racial group.

Sex a factor with levels `Male Female`

SES a numeric vector of socio-economic status.

MathAch a numeric vector of mathematics achievement scores.

MEANSES a numeric vector of the mean SES for the school.

Details

Each row in this data frame contains the data for one student.

Examples

```
summary(MathAchieve)
```

MathAchSchool *School demographic data for MathAchieve*

Description

The `MathAchSchool` data frame has 160 rows and 7 columns.

Format

This data frame contains the following columns:

School a factor giving the school on which the measurement is made.

Size a numeric vector giving the number of students in the school

Sector a factor with levels `Public` `Catholic`

PRACAD a numeric vector giving the percentage of students on the academic track

DISCLIM a numeric vector measuring the discrimination climate

HIMINTY a factor with levels `0` `1`

MEANSES a numeric vector giving the mean SES score.

Details

These variables give the school-level demographic data to accompany the `MathAchieve` data.

Matrix *Assign Matrix Values*

Description

This function is generic; method functions can be written to handle specific classes of objects. Classes which already have methods for this function include `pdMat`, `pdBlocked`, and `reStruct`.

Usage

```
matrix(object) <- value
```

Arguments

`object`	any object to which `as.matrix` can be applied.
`value`	a matrix, or list of matrices, with the same dimensions as `as.matrix(object)` with the new values to be assigned to the matrix associated with `object`.

Value

will depend on the method function; see the appropriate documentation.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`as.matrix`

Examples

```
## see the method function documentation
```

Matrix.pdMat *Assign Matrix to a pdMat Object*

Description

The positive-definite matrix represented by `object` is replaced by `value`. If the original matrix had row and/or column names, the corresponding names for `value` can either be `NULL`, or a permutation of the original names.

Usage

```
## S3 method for class 'pdMat':
matrix(object) <- value
```

Arguments

`object`	an object inheriting from class `pdMat`, representing a positive definite matrix.
`value`	a matrix with the new values to be assigned to the positive-definite matrix represented by `object`. Must have the same dimensions as `as.matrix(object)`.

Value

a `pdMat` object similar to `object`, but with its coefficients modified to produce the matrix in `value`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`pdMat`,`"matrix<-"`

Examples

```
pd1 <- pdSymm(diag(3))
matrix(pd1) <- diag(1:3)
pd1
```

Matrix.reStruct *Assign reStruct Matrices*

Description

The individual matrices in `value` are assigned to each `pdMat` component of `object`, in the order they are listed. The new matrices must have the same dimensions as the matrices they are meant to replace.

Usage

```
## S3 method for class 'reStruct':
matrix(object) <-  value
```

Arguments

`object`	an object inheriting from class `reStruct`, representing a random effects structure and consisting of a list of `pdMat` objects.
`value`	a matrix, or list of matrices, with the new values to be assigned to the matrices associated with the `pdMat` components of `object`.

Value

an `reStruct` object similar to `object`, but with the coefficients of the individual `pdMat` components modified to produce the matrices listed in `value`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`reStruct`, `pdMat`, `"matrix<-"`

Examples

```
rs1 <- reStruct(list(Dog = ~day, Side = ~1), data = Pixel)
matrix(rs1) <- list(diag(2), 3)
```

Meat *Tenderness of meat*

Description

The `Meat` data frame has 30 rows and 4 columns.

Format

This data frame contains the following columns:

Storage an ordered factor specifying the storage treatment - 1 (0 days), 2 (1 day), 3 (2 days), 4 (4 days), 5 (9 days), and 6 (18 days)

score a numeric vector giving the tenderness score of beef roast.

Block an ordered factor identifying the muscle from which the roast was extracted with levels `II` < `V` < `I` < `III` < `IV`

Pair an ordered factor giving the unique identifier for each pair of beef roasts with levels `II-1` < ... < `IV-1`

Details

Cochran and Cox (section 11.51, 1957) describe data from an experiment conducted at Iowa State College (Paul, 1943) to compare the effects of length of cold storage on the tenderness of beef roasts. Six storage periods ranging from 0 to 18 days were used. Thirty roasts were scored by four judges on a scale from 0 to 10, with the score increasing with tenderness. The response was the sum of all four scores. Left and right roasts from the same animal were grouped into pairs, which were further grouped into five blocks, according to the muscle from which they were extracted. Different storage periods were applied to each roast within a pair according to a balanced incomplete block design.

Source

Cochran, W. G. and Cox, G. M. (1957), *Experimental Designs*, Wiley, New York.

`Milk` *Protein content of cows' milk*

Description

The `Milk` data frame has 1337 rows and 4 columns.

Format

This data frame contains the following columns:

protein a numeric vector giving the protein content of the milk.

Time a numeric vector giving the time since calving (weeks).

Cow an ordered factor giving a unique identifier for each cow.

Diet a factor with levels `barley`, `barley+lupins`, and `lupins` identifying the diet for each cow.

Details

Diggle, Liang, and Zeger (1994) describe data on the protein content of cows' milk in the weeks following calving. The cattle are grouped according to whether they are fed a diet with barley alone, with barley and lupins, or with lupins alone.

Source

Diggle, Peter J., Liang, Kung-Yee and Zeger, Scott L. (1994), *Analysis of longitudinal data*, Oxford University Press, Oxford.

`model.matrix.reStruct` *reStruct Model Matrix*

Description

The model matrices for each element of `formula(object)`, calculated using `data`, are bound together column-wise. When multiple grouping levels are present (i.e. when `length(object) > 1`), the individual model matrices are combined from innermost (at the leftmost position) to outermost (at the rightmost position).

Usage

```
## S3 method for class 'reStruct':
model.matrix(object, data, contrast, ...)
```

Arguments

`object`	an object inheriting from class `reStruct`, representing a random effects structure and consisting of a list of `pdMat` objects.
`data`	a data frame in which to evaluate the variables defined in `formula(object)`.
`contrast`	an optional named list specifying the contrasts to be used for representing the `factor` variables in `data`. The components names should match the names of the variables in `data` for which the contrasts are to be specified. The components of this list will be used as the `contrasts` attribute of the corresponding factor. If missing, the default contrast specification is used.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

a matrix obtained by binding together, column-wise, the model matrices for each element of `formula(object)`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`model.matrix`, `contrasts`, `reStruct`, `formula.reStruct`

Examples

```
rs1 <- reStruct(list(Dog = ~day, Side = ~1), data = Pixel)
model.matrix(rs1, Pixel)
```

Muscle — *Contraction of heart muscle sections*

Description

The `Muscle` data frame has 60 rows and 3 columns.

Format

This data frame contains the following columns:

Strip an ordered factor indicating the strip of muscle being measured.

conc a numeric vector giving the concentration of CaCl2

length a numeric vector giving the shortening ofthe heart muscle strip.

Details

Baumann and Waldvogel (1963) describe data on the shortening of heart muscle strips dipped in a $CaCl_2$ solution. The muscle strips are taken from the left auricle of a rat's heart.

Source

Baumann, F. and Waldvogel, F. (1963), La restitution pastsystolique de la contraction de l'oreillette gauche du rat. Effets de divers ions et de l'acetylcholine, *Helvetica Physiologica Acta*, **21**.

Names — *Names Associated with an Object*

Description

This function is generic; method functions can be written to handle specific classes of objects. Classes which already have methods for this function include: `formula`, `modelStruct`, `pdBlocked`, `pdMat`, and `reStruct`.

Usage

```
Names(object, ...)
Names(object, ...) <- value
```

Arguments

`object`	any object for which names can be extracted and/or assigned.
`...`	some methods for this generic function require additional arguments.
`value`	names to be assigned to `object`.

Value

will depend on the method function used; see the appropriate documentation.

SIDE EFFECTS

On the left side of an assignment, sets the names associated with `object` to `value`, which must have an appropriate length.

Note

If `names` were generic, there would be no need for this generic function.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`Names.formula`, `Names.pdMat`

Examples

```
## see the method function documentation
```

`Names.formula` *Extract Names from a formula*

Description

This method function returns the names of the terms corresponding to the right hand side of `object` (treated as a linear formula), obtained as the column names of the corresponding `model.matrix`.

Usage

```
## S3 method for class 'formula':
Names(object, data, exclude, ...)
```

Arguments

`object`	an object inheriting from class `formula`.
`data`	an optional data frame containing the variables specified in `object`. By default the variables are taken from the environment from which `Names.formula` is called.
`exclude`	an optional character vector with names to be excluded from the returned value. Default is `c("pi",".")`.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

a character vector with the column names of the `model.matrix` corresponding to the right hand side of `object` which are not listed in `excluded`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

model.matrix, terms, Names

Examples

```
Names(distance ~ Sex * age, data = Orthodont)
```

Names.pdBlocked *Names of a pdBlocked Object*

Description

This method function extracts the first element of the Dimnames attribute, which contains the column names, for each block diagonal element in the matrix represented by object.

Usage

```
## S3 method for class 'pdBlocked':
Names(object, asList, ...)
```

Arguments

object	an object inheriting from class pdBlocked representing a positive-definite matrix with block diagonal structure
asList	a logical value. If TRUE a list with the names for each block diagonal element is returned. If FALSE a character vector with all column names is returned. Defaults to FALSE.
...	some methods for this generic require additional arguments. None are used in this method.

Value

if asList is FALSE, a character vector with column names of the matrix represented by object; otherwise, if asList is TRUE, a list with components given by the column names of the individual block diagonal elements in the matrix represented by object.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

Names, Names.pdMat

Examples

```
pd1 <- pdBlocked(list(~Sex - 1, ~age - 1), data = Orthodont)
Names(pd1)
```

Names.pdMat *Names of a pdMat Object*

Description

This method function returns the fist element of the Dimnames attribute of object, which contains the column names of the matrix represented by object.

Usage

```
## S3 method for class 'pdMat':
Names(object, ...)
## S3 method for class 'pdMat':
Names(object, ...) <- value
```

Arguments

object	an object inheriting from class pdMat, representing a positive-definite matrix.
value	a character vector with the replacement values for the column and row names of the matrix represented by object. It must have length equal to the dimension of the matrix represented by object and, if names have been previously assigned to object, it must correspond to a permutation of the original names.
...	some methods for this generic require additional arguments. None are used in this method.

Value

if object has a Dimnames attribute then the first element of this attribute is returned; otherwise NULL.

SIDE EFFECTS

On the left side of an assignment, sets the Dimnames attribute of object to list(value, value).

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

Names, Names.pdBlocked

Examples

```
pd1 <- pdSymm(~age, data = Orthodont)
Names(pd1)
```

`Names.reStruct` *Names of an reStruct Object*

Description

This method function extracts the column names of each of the positive-definite matrices represented the `pdMat` elements of `object`.

Usage

```
## S3 method for class 'reStruct':
Names(object, ...)
## S3 method for class 'reStruct':
Names(object, ...) <- value
```

Arguments

`object`	an object inheriting from class `reStruct`, representing a random effects structure and consisting of a list of `pdMat` objects.
`value`	a list of character vectors with the replacement values for the names of the individual `pdMat` objects that form `object`. It must have the same length as `object`.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

a list containing the column names of each of the positive-definite matrices represented by the `pdMat` elements of `object`.

SIDE EFFECTS

On the left side of an assignment, sets the `Names` of the `pdMat` elements of `object` to the corresponding element of `value`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`reStruct`, `pdMat`, `Names.pdMat`

Examples

```
rs1 <- reStruct(list(Dog = ~day, Side = ~1), data = Pixel)
Names(rs1)
```

needUpdate *Check if Update is Needed*

Description

This function is generic; method functions can be written to handle specific classes of objects. By default, it tries to extract a `needUpdate` attribute of `object`. If this is `NULL` or `FALSE` it returns `FALSE`; else it returns `TRUE`. Updating of objects usually takes place in iterative algorithms in which auxiliary quantities associated with the object, and not being optimized over, may change.

Usage

```
needUpdate(object)
```

Arguments

`object`	any object

Value

a logical value indicating whether `object` needs to be updated.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`needUpdate.modelStruct`

Examples

```
vf1 <- varExp()
vf1 <- Initialize(vf1, data = Orthodont)
needUpdate(vf1)
```

needUpdate.modelStruct *Check if a modelStruct Object Needs Updating*

Description

This method function checks if any of the elements of `object` needs to be updated. Updating of objects usually takes place in iterative algorithms in which auxiliary quantities associated with the object, and not being optimized over, may change.

Usage

```
## S3 method for class 'modelStruct':
needUpdate(object)
```

Arguments

`object`	an object inheriting from class `modelStruct`, representing a list of model components, such as `corStruct` and `varFunc` objects.

Value

a logical value indicating whether any element of `object` needs to be updated.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`needUpdate`

Examples

```
lms1 <- lmeStruct(reStruct = reStruct(pdDiag(diag(2), ~age)),
   varStruct = varPower(form = ~age))
needUpdate(lms1)
```

`Nitrendipene` *Assay of nitrendipene*

Description

The `Nitrendipene` data frame has 89 rows and 4 columns.

Format

This data frame contains the following columns:

activity a numeric vector

NIF a numeric vector

Tissue an ordered factor with levels `2` < `1` < `3` < `4`

log.NIF a numeric vector

Details

Source

Bates, D. M. and Watts, D. G. (1988), *Nonlinear Regression Analysis and Its Applications*, Wiley, New York.

nlme *Nonlinear Mixed-Effects Models*

Description

This generic function fits a nonlinear mixed-effects model in the formulation described in Lindstrom and Bates (1990) but allowing for nested random effects. The within-group errors are allowed to be correlated and/or have unequal variances.

Usage

```
nlme(model, data, fixed, random, groups, start, correlation, weights,
     subset, method, na.action, naPattern, control, verbose)
```

Arguments

model	a nonlinear model formula, with the response on the left of a ~ operator and an expression involving parameters and covariates on the right, or an nlsList object. If data is given, all names used in the formula should be defined as parameters or variables in the data frame. The method function nlme.nlsList is documented separately.
data	an optional data frame containing the variables named in model, fixed, random, correlation, weights, subset, and naPattern. By default the variables are taken from the environment from which nlme is called.
fixed	a two-sided linear formula of the form f1+...+fn~x1+...+xm, or a list of two-sided formulas of the form f1~x1+...+xm, with possibly different models for different parameters. The f1,...,fn are the names of parameters included on the right hand side of model and the x1+...+xm expressions define linear models for these parameters (when the left hand side of the formula contains several parameters, they all are assumed to follow the same linear model, described by the right hand side expression). A 1 on the right hand side of the formula(s) indicates a single fixed effects for the corresponding parameter(s).
random	optionally, any of the following: (i) a two-sided formula of the form r1+...+rn~x1+...+xm | g1/.../gQ, with r1,...,rn naming parameters included on the right hand side of model, x1+...+xm specifying the random-effects model for these parameters and g1/.../gQ the grouping structure (Q may be equal to 1, in which case no / is required). The random effects formula will be repeated for all levels of grouping, in the case of multiple levels of grouping; (ii) a two-sided formula of the form r1+...+rn~x1+..+xm, a list of two-sided formulas of the form r1~x1+...+xm, with possibly different random-effects models for different parameters, a pdMat object with a two-sided formula, or list of two-sided formulas (i.e. a non-NULL value for formula(random)), or a list of pdMat objects with two-sided formulas, or lists of two-sided formulas. In this case, the grouping structure formula will be given in groups, or derived from the data used to fit the nonlinear mixed-effects model, which should inherit from class groupedData,; (iii) a named list of formulas, lists of formulas, or pdMat objects as in (ii), with the grouping factors as names. The order of nesting will be assumed the same as the order of the order of the elements in the list; (iv) an reStruct object. See the documentation on pdClasses for a description of

the available pdMat classes. Defaults to fixed, resulting in all fixed effects having also random effects.

groups — an optional one-sided formula of the form ~g1 (single level of nesting) or ~g1/.../gQ (multiple levels of nesting), specifying the partitions of the data over which the random effects vary. g1,...,gQ must evaluate to factors in data. The order of nesting, when multiple levels are present, is taken from left to right (i.e. g1 is the first level, g2 the second, etc.).

start — an optional numeric vector, or list of initial estimates for the fixed effects and random effects. If declared as a numeric vector, it is converted internally to a list with a single component fixed, given by the vector. The fixed component is required, unless the model function inherits from class selfStart, in which case initial values will be derived from a call to nlsList. An optional random component is used to specify initial values for the random effects and should consist of a matrix, or a list of matrices with length equal to the number of grouping levels. Each matrix should have as many rows as the number of groups at the corresponding level and as many columns as the number of random effects in that level.

correlation — an optional corStruct object describing the within-group correlation structure. See the documentation of corClasses for a description of the available corStruct classes. Defaults to NULL, corresponding to no within-group correlations.

weights — an optional varFunc object or one-sided formula describing the within-group heteroscedasticity structure. If given as a formula, it is used as the argument to varFixed, corresponding to fixed variance weights. See the documentation on varClasses for a description of the available varFunc classes. Defaults to NULL, corresponding to homoscesdatic within-group errors.

subset — an optional expression indicating the subset of the rows of data that should be used in the fit. This can be a logical vector, or a numeric vector indicating which observation numbers are to be included, or a character vector of the row names to be included. All observations are included by default.

method — a character string. If "REML" the model is fit by maximizing the restricted log-likelihood. If "ML" the log-likelihood is maximized. Defaults to "ML".

na.action — a function that indicates what should happen when the data contain NAs. The default action (na.fail) causes nlme to print an error message and terminate if there are any incomplete observations.

naPattern — an expression or formula object, specifying which returned values are to be regarded as missing.

control — a list of control values for the estimation algorithm to replace the default values returned by the function nlmeControl. Defaults to an empty list.

verbose — an optional logical value. If TRUE information on the evolution of the iterative algorithm is printed. Default is FALSE.

Value

an object of class nlme representing the nonlinear mixed-effects model fit. Generic functions such as print, plot and summary have methods to show the results of the fit. See nlmeObject for the components of the fit. The functions resid, coef, fitted, fixed.effects, and random.effects can be used to extract some of its components.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

The model formulation and computational methods are described in Lindstrom, M.J. and Bates, D.M. (1990). The variance-covariance parametrizations are described in Pinheiro, J.C. and Bates., D.M. (1996). The different correlation structures available for the `correlation` argument are described in Box, G.E.P., Jenkins, G.M., and Reinsel G.C. (1994), Littel, R.C., Milliken, G.A., Stroup, W.W., and Wolfinger, R.D. (1996), and Venables, W.N. and Ripley, B.D. (1997). The use of variance functions for linear and nonlinear mixed effects models is presented in detail in Davidian, M. and Giltinan, D.M. (1995).

Box, G.E.P., Jenkins, G.M., and Reinsel G.C. (1994) "Time Series Analysis: Forecasting and Control", 3rd Edition, Holden-Day.

Davidian, M. and Giltinan, D.M. (1995) "Nonlinear Mixed Effects Models for Repeated Measurement Data", Chapman and Hall.

Laird, N.M. and Ware, J.H. (1982) "Random-Effects Models for Longitudinal Data", Biometrics, 38, 963-974.

Littel, R.C., Milliken, G.A., Stroup, W.W., and Wolfinger, R.D. (1996) "SAS Systems for Mixed Models", SAS Institute.

Lindstrom, M.J. and Bates, D.M. (1990) "Nonlinear Mixed Effects Models for Repeated Measures Data", Biometrics, 46, 673-687.

Pinheiro, J.C. and Bates., D.M. (1996) "Unconstrained Parametrizations for Variance-Covariance Matrices", Statistics and Computing, 6, 289-296.

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

Venables, W.N. and Ripley, B.D. (1997) "Modern Applied Statistics with S-plus", 2nd Edition, Springer-Verlag.

See Also

`nlmeControl`, `nlme.nlsList`, `nlmeObject`, `nlsList`, `nlmeStruct`, `pdClasses`, `reStruct`, `varFunc`, `corClasses`, `varClasses`

Examples

```
fm1 <- nlme(height ~ SSasymp(age, Asym, R0, lrc),
            data = Loblolly,
            fixed = Asym + R0 + lrc ~ 1,
            random = Asym ~ 1,
            start = c(Asym = 103, R0 = -8.5, lrc = -3.3))
summary(fm1)
fm2 <- update(fm1, random = pdDiag(Asym + lrc ~ 1))
summary(fm2)
```

nlme.nlsList *NLME fit from nlsList Object*

Description

If the random effects names defined in `random` are a subset of the `lmList` object coefficient names, initial estimates for the covariance matrix of the random effects are obtained (overwriting any values given in `random`). `formula(fixed)` and the `data` argument in the calling sequence used to obtain `fixed` are passed as the `fixed` and `data` arguments to `nlme.formula`, together with any other additional arguments in the function call. See the documentation on `nlme.formula` for a description of that function.

Usage

```
## S3 method for class 'nlsList':
nlme(model, data, fixed, random, groups, start, correlation, weights,
     subset, method, na.action, naPattern, control, verbose)
```

Arguments

`model`	an object inheriting from class `nlsList`, representing a list of `nls` fits with a common model.
`data`	this argument is included for consistency with the generic function. It is ignored in this method function.
`fixed`	this argument is included for consistency with the generic function. It is ignored in this method function.
`random`	an optional one-sided linear formula with no conditioning expression, or a `pdMat` object with a `formula` attribute. Multiple levels of grouping are not allowed with this method function. Defaults to a formula consisting of the right hand side of `formula(fixed)`.
`groups`	an optional one-sided formula of the form `~g1` (single level of nesting) or `~g1/.../gQ` (multiple levels of nesting), specifying the partitions of the data over which the random effects vary. `g1,...,gQ` must evaluate to factors in `data`. The order of nesting, when multiple levels are present, is taken from left to right (i.e. `g1` is the first level, `g2` the second, etc.).
`start`	an optional numeric vector, or list of initial estimates for the fixed effects and random effects. If declared as a numeric vector, it is converted internally to a list with a single component `fixed`, given by the vector. The `fixed` component is required, unless the model function inherits from class `selfStart`, in which case initial values will be derived from a call to `nlsList`. An optional `random` component is used to specify initial values for the random effects and should consist of a matrix, or a list of matrices with length equal to the number of grouping levels. Each matrix should have as many rows as the number of groups at the corresponding level and as many columns as the number of random effects in that level.
`correlation`	an optional `corStruct` object describing the within-group correlation structure. See the documentation of `corClasses` for a description of the available `corStruct` classes. Defaults to `NULL`, corresponding to no within-group correlations.

weights	an optional varFunc object or one-sided formula describing the within-group heteroscedasticity structure. If given as a formula, it is used as the argument to varFixed, corresponding to fixed variance weights. See the documentation on varClasses for a description of the available varFunc classes. Defaults to NULL, corresponding to homoscesdatic within-group errors.
subset	an optional expression indicating the subset of the rows of data that should be used in the fit. This can be a logical vector, or a numeric vector indicating which observation numbers are to be included, or a character vector of the row names to be included. All observations are included by default.
method	a character string. If "REML" the model is fit by maximizing the restricted log-likelihood. If "ML" the log-likelihood is maximized. Defaults to "ML".
na.action	a function that indicates what should happen when the data contain NAs. The default action (na.fail) causes nlme to print an error message and terminate if there are any incomplete observations.
naPattern	an expression or formula object, specifying which returned values are to be regarded as missing.
control	a list of control values for the estimation algorithm to replace the default values returned by the function nlmeControl. Defaults to an empty list.
verbose	an optional logical value. If TRUE information on the evolution of the iterative algorithm is printed. Default is FALSE.

Value

an object of class nlme representing the linear mixed-effects model fit. Generic functions such as print, plot and summary have methods to show the results of the fit. See nlmeObject for the components of the fit. The functions resid, coef, fitted, fixed.effects, and random.effects can be used to extract some of its components.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

The computational methods are described in Bates, D.M. and Pinheiro (1998) and follow on the general framework of Lindstrom, M.J. and Bates, D.M. (1988). The model formulation is described in Laird, N.M. and Ware, J.H. (1982). The variance-covariance parametrizations are described in <Pinheiro, J.C. and Bates., D.M. (1996). The different correlation structures available for the correlation argument are described in Box, G.E.P., Jenkins, G.M., and Reinsel G.C. (1994), Littel, R.C., Milliken, G.A., Stroup, W.W., and Wolfinger, R.D. (1996), and Venables, W.N. and Ripley, B.D. (1997). The use of variance functions for linear and nonlinear mixed effects models is presented in detail in Davidian, M. and Giltinan, D.M. (1995).

Bates, D.M. and Pinheiro, J.C. (1998) "Computational methods for multilevel models" available in PostScript or PDF formats at http://franz.stat.wisc.edu/pub/NNLME/

Box, G.E.P., Jenkins, G.M., and Reinsel G.C. (1994) "Time Series Analysis: Forecasting and Control", 3rd Edition, Holden-Day.

Davidian, M. and Giltinan, D.M. (1995) "Nonlinear Mixed Effects Models for Repeated Measurement Data", Chapman and Hall.

Laird, N.M. and Ware, J.H. (1982) "Random-Effects Models for Longitudinal Data", Biometrics, 38, 963-974.

Lindstrom, M.J. and Bates, D.M. (1988) "Newton-Raphson and EM Algorithms for Linear Mixed-Effects Models for Repeated-Measures Data", Journal of the American Statistical Association, 83, 1014-1022.

Littel, R.C., Milliken, G.A., Stroup, W.W., and Wolfinger, R.D. (1996) "SAS Systems for Mixed Models", SAS Institute.

Pinheiro, J.C. and Bates., D.M. (1996) "Unconstrained Parametrizations for Variance-Covariance Matrices", Statistics and Computing, 6, 289-296.

Venables, W.N. and Ripley, B.D. (1997) "Modern Applied Statistics with S-plus", 2nd Edition, Springer-Verlag.

See Also

nlme, lmList, nlmeObject

Examples

```
fm1 <- nlsList(SSasymp, data = Loblolly)
fm2 <- nlme(fm1, random = Asym ~ 1)
summary(fm1)
summary(fm2)
```

---

nlmeControl *Control Values for nlme Fit*

Description

The values supplied in the function call replace the defaults and a list with all possible arguments is returned. The returned list is used as the `control` argument to the `nlme` function.

Usage

```
nlmeControl(maxIter, pnlsMaxIter, msMaxIter, minScale,
            tolerance, niterEM, pnlsTol, msTol, msScale,
            returnObject, msVerbose, gradHess, apVar, .relStep,
            nlmStepMax = 100.0, minAbsParApVar = 0.05,
            opt = c("nlminb", "nlm"), natural = TRUE)
```

Arguments

`maxIter`	maximum number of iterations for the `nlme` optimization algorithm. Default is 50.
`pnlsMaxIter`	maximum number of iterations for the `PNLS` optimization step inside the `nlme` optimization. Default is 7.
`msMaxIter`	maximum number of iterations for the `nlm` optimization step inside the `nlme` optimization. Default is 50.
`minScale`	minimum factor by which to shrink the default step size in an attempt to decrease the sum of squares in the `PNLS` step. Default 0.001.
`tolerance`	tolerance for the convergence criterion in the `nlme` algorithm. Default is 1e-6.
`niterEM`	number of iterations for the EM algorithm used to refine the initial estimates of the random effects variance-covariance coefficients. Default is 25.

pnlsTol	tolerance for the convergence criterion in PNLS step. Default is 1e-3.
msTol	tolerance for the convergence criterion in nlm, passed as the rel.tolerance argument to the function (see documentation on nlm). Default is 1e-7.
msScale	scale function passed as the scale argument to the nlm function (see documentation on that function). Default is lmeScale.
returnObject	a logical value indicating whether the fitted object should be returned when the maximum number of iterations is reached without convergence of the algorithm. Default is FALSE.
msVerbose	a logical value passed as the trace argument to nlm (see documentation on that function). Default is FALSE.
gradHess	a logical value indicating whether numerical gradient vectors and Hessian matrices of the log-likelihood function should be used in the nlm optimization. This option is only available when the correlation structure (corStruct) and the variance function structure (varFunc) have no "varying" parameters and the pdMat classes used in the random effects structure are pdSymm (general positive-definite), pdDiag (diagonal), pdIdent (multiple of the identity), or pdCompSymm (compound symmetry). Default is TRUE.
apVar	a logical value indicating whether the approximate covariance matrix of the variance-covariance parameters should be calculated. Default is TRUE.
.relStep	relative step for numerical derivatives calculations. Default is .Machine$double.eps^(1/3).
nlmStepMax	stepmax value to be passed to nlm. See nlm for details. Default is 100.0
minAbsParApVar	numeric value - minimum absolute parameter value in the approximate variance calculation. The default is 0.05.
opt	the optimizer to be used, either nlminb (the default since (R 2.2.0) or nlm (the previous default).
natural	a logical value indicating whether the pdNatural parametrization should be used for general positive-definite matrices (pdSymm) in reStruct, when the approximate covariance matrix of the estimators is calculated. Default is TRUE.

Value

a list with components for each of the possible arguments.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

nlme, nlm, optim, nlmeStruct

Examples

```
# decrease the maximum number iterations in the ms call and
# request that information on the evolution of the ms iterations be printed
nlmeControl(msMaxIter = 20, msVerbose = TRUE)
```

nlmeObject *Fitted nlme Object*

Description

An object returned by the nlme function, inheriting from class nlme, also inheriting from class lme, and representing a fitted nonlinear mixed-effects model. Objects of this class have methods for the generic functions anova, coef, fitted, fixed.effects, formula, getGroups, getResponse, intervals, logLik, pairs, plot, predict, print, random.effects, residuals, summary, and update.

Value

The following components must be included in a legitimate nlme object.

apVar	an approximate covariance matrix for the variance-covariance coefficients. If apVar = FALSE in the list of control values used in the call to nlme, this component is equal to NULL.
call	a list containing an image of the nlme call that produced the object.
coefficients	a list with two components, fixed and random, where the first is a vector containing the estimated fixed effects and the second is a list of matrices with the estimated random effects for each level of grouping. For each matrix in the random list, the columns refer to the random effects and the rows to the groups.
contrasts	a list with the contrasts used to represent factors in the fixed effects formula and/or random effects formula. This information is important for making predictions from a new data frame in which not all levels of the original factors are observed. If no factors are used in the nlme model, this component will be an empty list.
dims	a list with basic dimensions used in the nlme fit, including the components N - the number of observations in the data, Q - the number of grouping levels, qvec - the number of random effects at each level from innermost to outermost (last two values are equal to zero and correspond to the fixed effects and the response), ngrps - the number of groups at each level from innermost to outermost (last two values are one and correspond to the fixed effects and the response), and ncol - the number of columns in the model matrix for each level of grouping from innermost to outermost (last two values are equal to the number of fixed effects and one).
fitted	a data frame with the fitted values as columns. The leftmost column corresponds to the population fixed effects (corresponding to the fixed effects only) and successive columns from left to right correspond to increasing levels of grouping.
fixDF	a list with components X and terms specifying the denominator degrees of freedom for, respectively, t-tests for the individual fixed effects and F-tests for the fixed-effects terms in the models.
groups	a data frame with the grouping factors as columns. The grouping level increases from left to right.
logLik	the (restricted) log-likelihood at convergence.
map	a list with components fmap, rmap, rmapRel, and bmap, specifying various mappings for the fixed and random effects, used to generate predictions from the fitted object.

method	the estimation method: either "ML" for maximum likelihood, or "REML" for restricted maximum likelihood.
modelStruct	an object inheriting from class nlmeStruct, representing a list of mixed-effects model components, such as reStruct, corStruct, and varFunc objects.
numIter	the number of iterations used in the iterative algorithm.
residuals	a data frame with the residuals as columns. The leftmost column corresponds to the population residuals and successive columns from left to right correspond to increasing levels of grouping.
sigma	the estimated within-group error standard deviation.
varFix	an approximate covariance matrix of the fixed effects estimates.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

nlme, nlmeStruct

---

nlmeStruct *Nonlinear Mixed-Effects Structure*

Description

A nonlinear mixed-effects structure is a list of model components representing different sets of parameters in the nonlinear mixed-effects model. An nlmeStruct list must contain at least a reStruct object, but may also contain corStruct and varFunc objects. NULL arguments are not included in the nlmeStruct list.

Usage

```
nlmeStruct(reStruct, corStruct, varStruct)
```

Arguments

reStruct	a reStruct representing a random effects structure.
corStruct	an optional corStruct object, representing a correlation structure. Default is NULL.
varStruct	an optional varFunc object, representing a variance function structure. Default is NULL.

Value

a list of model components determining the parameters to be estimated for the associated nonlinear mixed-effects model.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

corClasses, nlme, residuals.nlmeStruct, reStruct, varFunc

Examples

```
nlms1 <- nlmeStruct(reStruct(~age), corAR1(), varPower())
```

---

nlsList *List of nls Objects with a Common Model*

Description

Data is partitioned according to the levels of the grouping factor defined in model and individual nls fits are obtained for each data partition, using the model defined in model.

Usage

```
nlsList(model, data, start, control, level, subset, na.action, pool)
## S3 method for class 'nlsList':
update(object, model., ..., evaluate = TRUE)
```

Arguments

object	an object inheriting from class nlsList, representing a list of fitted nls objects.
model	either a nonlinear model formula, with the response on the left of a ~ operator and an expression involving parameters, covariates, and a grouping factor separated by the | operator on the right, or a selfStart function. The method function nlsList.selfStart is documented separately.
model.	Changes to the model – see update.formula for details.
data	a data frame in which to interpret the variables named in model.
start	an optional named list with initial values for the parameters to be estimated in model. It is passed as the start argument to each nls call and is required when the nonlinear function in model does not inherit from class selfStart.
control	a list of control values passed as the control argument to nls. Defaults to an empty list.
level	an optional integer specifying the level of grouping to be used when multiple nested levels of grouping are present.
subset	an optional expression indicating the subset of the rows of data that should be used in the fit. This can be a logical vector, or a numeric vector indicating which observation numbers are to be included, or a character vector of the row names to be included. All observations are included by default.
na.action	a function that indicates what should happen when the data contain NAs. The default action (na.fail) causes nlsList to print an error message and terminate if there are any incomplete observations.
pool	an optional logical value that is preserved as an attribute of the returned value. This will be used as the default for pool in calculations of standard deviations or standard errors for summaries.

...	some methods for this generic require additional arguments. None are used in this method.
evaluate	If TRUE evaluate the new call else return the call.

Value

a list of nls objects with as many components as the number of groups defined by the grouping factor. Generic functions such as coef, fixed.effects, lme, pairs, plot, predict, random.effects, summary, and update have methods that can be applied to an nlsList object.

References

Pinheiro, J.C., and Bates, D.M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer.

See Also

nls, nlme.nlsList, nlsList.selfStart, summary.nlsList

Examples

```
fm1 <- nlsList(uptake ~ SSasympOff(conc, Asym, lrc, c0),
   data = CO2, start = c(Asym = 30, lrc = -4.5, c0 = 52))
summary(fm1)
```

---

nlsList.selfStart *nlsList Fit from a selfStart Function*

Description

The response variable and primary covariate in formula(data) are used together with model to construct the nonlinear model formula. This is used in the nls calls and, because a selfStarting model function can calculate initial estimates for its parameters from the data, no starting estimates need to be provided.

Usage

```
## S3 method for class 'selfStart':
nlsList(model, data, start, control, level, subset, na.action, pool)
```

Arguments

model	a selfStart model function, which calculates initial estimates for the model parameters from data.
data	a data frame in which to interpret the variables in model. Because no grouping factor can be specified in model, data must inherit from class groupedData.
start	an optional named list with initial values for the parameters to be estimated in model. It is passed as the start argument to each nls call and is required when the nonlinear function in model does not inherit from class selfStart.

control	a list of control values passed as the control argument to nls. Defaults to an empty list.
level	an optional integer specifying the level of grouping to be used when multiple nested levels of grouping are present.
subset	an optional expression indicating the subset of the rows of data that should be used in the fit. This can be a logical vector, or a numeric vector indicating which observation numbers are to be included, or a character vector of the row names to be included. All observations are included by default.
na.action	a function that indicates what should happen when the data contain NAs. The default action (na.fail) causes nlsList to print an error message and terminate if there are any incomplete observations.
pool	an optional logical value that is preserved as an attribute of the returned value. This will be used as the default for pool in calculations of standard deviations or standard errors for summaries.

Value

a list of nls objects with as many components as the number of groups defined by the grouping factor. A NULL value is assigned to the components corresponding to clusters for which the nls algorithm failed to converge. Generic functions such as coef, fixed.effects, lme, pairs, plot, predict, random.effects, summary, and update have methods that can be applied to an nlsList object.

See Also

selfStart, groupedData, nls, nlsList, nlme.nlsList, nlsList.formula

Examples

```
fm1 <- nlsList(SSasympOff, CO2)
summary(fm1)
```

Oats — *Split-plot Experiment on Varieties of Oats*

Description

The Oats data frame has 72 rows and 4 columns.

Format

This data frame contains the following columns:

Block an ordered factor with levels VI < V < III < IV < II < I

Variety a factor with levels Golden Rain Marvellous Victory

nitro a numeric vector

yield a numeric vector

Details

These data have been introduced by Yates (1935) as an example of a split-plot design. The treatment structure used in the experiment was a 3×4 full factorial, with three varieties of oats and four concentrations of nitrogen. The experimental units were arranged into six blocks, each with three whole-plots subdivided into four subplots. The varieties of oats were assigned randomly to the whole-plots and the concentrations of nitrogen to the subplots. All four concentrations of nitrogen were used on each whole-plot.

Source

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York. (Appendix A.15)

Venables, W. N. and Ripley, B. D. (1999) *Modern Applied Statistics with S-PLUS (3rd ed)*, Springer, New York.

Orthodont *Growth curve data on an orthdontic measurement*

Description

The `Orthodont` data frame has 108 rows and 4 columns of the change in an orthdontic measurement over time for several young subjects.

Format

This data frame contains the following columns:

distance a numeric vector of distances from the pituitary to the pterygomaxillary fissure (mm). These distances are measured on x-ray images of the skull.

age a numeric vector of ages of the subject (yr).

Subject an ordered factor indicating the subject on which the measurement was made. The levels are labelled `M01` to `M16` for the males and `F01` to `F13` for the females. The ordering is by increasing average distance within sex.

Sex a factor with levels `Male` and `Female`

Details

Investigators at the University of North Carolina Dental School followed the growth of 27 children (16 males, 11 females) from age 8 until age 14. Every two years they measured the distance between the pituitary and the pterygomaxillary fissure, two points that are easily identified on x-ray exposures of the side of the head.

Source

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York. (Appendix A.17)

Potthoff, R. F. and Roy, S. N. (1964), "A generalized multivariate analysis of variance model useful especially for growth curve problems", *Biometrika*, **51**, 313–326.

Examples

```
formula(Orthodont)
plot(Orthodont)
```

Ovary *Counts of Ovarian Follicles*

Description

The `Ovary` data frame has 308 rows and 3 columns.

Format

This data frame contains the following columns:

Mare an ordered factor indicating the mare on which the measurement is made.

Time time in the estrus cycle. The data were recorded daily from 3 days before ovulation until 3 days after the next ovulation. The measurement times for each mare are scaled so that the ovulations for each mare occur at times 0 and 1.

follicles the number of ovarian follicles greater than 10 mm in diameter.

Details

Pierson and Ginther (1987) report on a study of the number of large ovarian follicles detected in different mares at several times in their estrus cycles.

Source

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York. (Appendix A.18)

Pierson, R. A. and Ginther, O. J. (1987), Follicular population dynamics during the estrus cycle of the mare, *Animal Reproduction Science*, **14**, 219-231.

Oxboys *Heights of Boys in Oxford*

Description

The `Oxboys` data frame has 234 rows and 4 columns.

Format

This data frame contains the following columns:

Subject an ordered factor giving a unique identifier for each boy in the experiment

age a numeric vector giving the standardized age (dimensionless)

height a numeric vector giving the height of the boy (cm)

Occasion an ordered factor - the result of converting `age` from a continuous variable to a count so these slightly unbalanced data can be analyzed as balanced.

Details

These data are described in Goldstein (1987) as data on the height of a selection of boys from Oxford, England versus a standardized age.

Source

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York. (Appendix A.19)

`Oxide` *Variability in Semiconductor Manufacturing*

Description

The `Oxide` data frame has 72 rows and 5 columns.

Format

This data frame contains the following columns:

Source a factor with levels `1` and `2`

Lot a factor giving a unique identifier for each lot.

Wafer a factor giving a unique identifier for each wafer within a lot.

Site a factor with levels `1`, `2`, and `3`

Thickness a numeric vector giving the thickness of the oxide layer.

Details

These data are described in Littell et al. (1996, p. 155) as coming "from a passive data collection study in the semiconductor industry where the objective is to estimate the variance components to determine the assignable causes of the observed variability." The observed response is the thickness of the oxide layer on silicon wafers, measured at three different sites of each of three wafers selected from each of eight lots sampled from the population of lots.

Source

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York. (Appendix A.20)

Littell, R. C., Milliken, G. A., Stroup, W. W. and Wolfinger, R. D. (1996), *SAS System for Mixed Models*, SAS Institute, Cary, NC.

`pairs.compareFits` *Pairs Plot of compareFits Object*

Description

Scatter plots of the values being compared are generated for each pair of coefficients in `x`. Different symbols (colors) are used for each object being compared and values corresponding to the same group are joined by a line, to facilitate comparison of fits. If only two coefficients are present, the `trellis` function `xyplot` is used; otherwise the `trellis` function `splom` is used.

Usage

```
## S3 method for class 'compareFits':
pairs(x, subset, key, ...)
```

Arguments

`x`	an object of class `compareFits`.
`subset`	an optional logical or integer vector specifying which rows of `x` should be used in the plots. If missing, all rows are used.
`key`	an optional logical value, or list. If `TRUE`, a legend is included at the top of the plot indicating which symbols (colors) correspond to which objects being compared. If `FALSE`, no legend is included. If given as a list, `key` is passed down as an argument to the `trellis` function generating the plots (`splom` or `xyplot`). Defaults to `TRUE`.
`...`	optional arguments passed down to the `trellis` function generating the plots.

Value

Pairwise scatter plots of the values being compared, with different symbols (colors) used for each object under comparison.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

`compareFits`, `plot.compareFits`, `pairs.lme`, `pairs.lmList`, `xyplot`, `splom`

Examples

```
fm1 <- lmList(Orthodont)
fm2 <- lme(Orthodont)
pairs(compareFits(coef(fm1), coef(fm2)))
```

`pairs.lme` *Pairs Plot of an lme Object*

Description

Diagnostic plots for the linear mixed-effects fit are obtained. The `form` argument gives considerable flexibility in the type of plot specification. A conditioning expression (on the right side of a `|` operator) always implies that different panels are used for each level of the conditioning factor, according to a Trellis display. The expression on the right hand side of the formula, before a `|` operator, must evaluate to a data frame with at least two columns. If the data frame has two columns, a scatter plot of the two variables is displayed (the Trellis function `xyplot` is used). Otherwise, if more than two columns are present, a scatter plot matrix with pairwise scatter plots of the columns in the data frame is displayed (the Trellis function `splom` is used).

Usage

```
## S3 method for class 'lme':
pairs(x, form, label, id, idLabels, grid, ...)
```

Arguments

`x`	an object inheriting from class `lme`, representing a fitted linear mixed-effects model.
`form`	an optional one-sided formula specifying the desired type of plot. Any variable present in the original data frame used to obtain `x` can be referenced. In addition, `x` itself can be referenced in the formula using the symbol `"."`. Conditional expressions on the right of a `|` operator can be used to define separate panels in a Trellis display. The expression on the right hand side of `form`, and to the left of the `|` operator, must evaluate to a data frame with at least two columns. Default is `~ coef(.)` , corresponding to a pairs plot of the coefficients evaluated at the innermost level of nesting.
`label`	an optional character vector of labels for the variables in the pairs plot.
`id`	an optional numeric value, or one-sided formula. If given as a value, it is used as a significance level for an outlier test based on the Mahalanobis distances of the estimated random effects. Groups with random effects distances greater than the $1 - value$ percentile of the appropriate chi-square distribution are identified in the plot using `idLabels`. If given as a one-sided formula, its right hand side must evaluate to a logical, integer, or character vector which is used to identify points in the plot. If missing, no points are identified.
`idLabels`	an optional vector, or one-sided formula. If given as a vector, it is converted to character and used to label the points identified according to `id`. If given as a one-sided formula, its right hand side must evaluate to a vector which is converted to character and used to label the identified points. Default is the innermost grouping factor.
`grid`	an optional logical value indicating whether a grid should be added to plot. Default is `FALSE`.
`...`	optional arguments passed to the Trellis plot function.

Value

a diagnostic Trellis plot.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

lme, pairs.compareFits, pairs.lmList, xyplot, splom

Examples

```
fm1 <- lme(distance ~ age, Orthodont, random = ~ age | Subject)
# scatter plot of coefficients by gender, identifying unusual subjects
pairs(fm1, ~coef(., augFrame = TRUE) | Sex, id = 0.1, adj = -0.5)
# scatter plot of estimated random effects
## Not run:
pairs(fm1, ~ranef(.))
## End(Not run)
```

---

pairs.lmList *Pairs Plot of an lmList Object*

Description

Diagnostic plots for the linear model fits corresponding to the x components are obtained. The form argument gives considerable flexibility in the type of plot specification. A conditioning expression (on the right side of a | operator) always implies that different panels are used for each level of the conditioning factor, according to a Trellis display. The expression on the right hand side of the formula, before a | operator, must evaluate to a data frame with at least two columns. If the data frame has two columns, a scatter plot of the two variables is displayed (the Trellis function xyplot is used). Otherwise, if more than two columns are present, a scatter plot matrix with pairwise scatter plots of the columns in the data frame is displayed (the Trellis function splom is used).

Usage

```
## S3 method for class 'lmList':
pairs(x, form, label, id, idLabels, grid, ...)
```

Arguments

x	an object inheriting from class lmList, representing a list of lm objects with a common model.
form	an optional one-sided formula specifying the desired type of plot. Any variable present in the original data frame used to obtain x can be referenced. In addition, x itself can be referenced in the formula using the symbol ".". Conditional expressions on the right of a | operator can be used to define separate panels in a Trellis display. The expression on the right hand side of form, and to the left of the | operator, must evaluate to a data frame with at least two columns. Default is ~ coef(.) , corresponding to a pairs plot of the coefficients of x.

label	an optional character vector of labels for the variables in the pairs plot.
id	an optional numeric value, or one-sided formula. If given as a value, it is used as a significance level for an outlier test based on the Mahalanobis distances of the estimated random effects. Groups with random effects distances greater than the $1 - value$ percentile of the appropriate chi-square distribution are identified in the plot using idLabels. If given as a one-sided formula, its right hand side must evaluate to a logical, integer, or character vector which is used to identify points in the plot. If missing, no points are identified.
idLabels	an optional vector, or one-sided formula. If given as a vector, it is converted to character and used to label the points identified according to id. If given as a one-sided formula, its right hand side must evaluate to a vector which is converted to character and used to label the identified points. Default is the innermost grouping factor.
grid	an optional logical value indicating whether a grid should be added to plot. Default is FALSE.
...	optional arguments passed to the Trellis plot function.

Value

a diagnostic Trellis plot.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

lmList, pairs.lme, pairs.compareFits, xyplot, splom

Examples

```
fm1 <- lmList(distance ~ age | Subject, Orthodont)
# scatter plot of coefficients by gender, identifying unusual subjects
pairs(fm1, ~coef(.) | Sex, id = 0.1, adj = -0.5)
# scatter plot of estimated random effects
## Not run:
pairs(fm1, ~ranef(.))
## End(Not run)
```

PBG *Effect of Phenylbiguanide on Blood Pressure*

Description

The PBG data frame has 60 rows and 5 columns.

Format

This data frame contains the following columns:

deltaBP a numeric vector

dose a numeric vector

Run an ordered factor with levels `T5` < `T4` < `T3` < `T2` < `T1` < `P5` < `P3` < `P2` < `P4` < `P1`

Treatment a factor with levels `MDL 72222` `Placebo`

Rabbit an ordered factor with levels `5` < `3` < `2` < `4` < `1`

Details

Data on an experiment to examine the effect of a antagonist MDL 72222 on the change in blood pressure experienced with increasing dosage of phenylbiguanide are described in Ludbrook (1994) and analyzed in Venables and Ripley (1999, section 8.8). Each of five rabbits was exposed to increasing doses of phenylbiguanide after having either a placebo or the HD5-antagonist MDL 72222 administered.

Source

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York. (Appendix A.21)

Venables, W. N. and Ripley, B. D. (1999) *Modern Applied Statistics with S-PLUS (3rd ed)*, Springer, New York.

Ludbrook, J. (1994), Repeated measurements and multiple comparisons in cardiovascular research, *Cardiovascular Research*, **28**, 303-311.

`pdBlocked` *Positive-Definite Block Diagonal Matrix*

Description

This function is a constructor for the `pdBlocked` class, representing a positive-definite block-diagonal matrix. Each block-diagonal element of the underlying matrix is itself a positive-definite matrix and is represented internally as an individual `pdMat` object. When `value` is `numeric(0)`, a list of uninitialized `pdMat` objects, a list of one-sided formulas, or a list of vectors of character strings, `object` is returned as an uninitialized `pdBlocked` object (with just some of its attributes and its class defined) and needs to have its coefficients assigned later, generally using the `coef` or `matrix` replacement functions. If `value` is a list of initialized `pdMat` objects, `object` will be constructed from the list obtained by applying `as.matrix` to each of the `pdMat` elements of `value`. Finally, if `value` is a list of numeric vectors, they are assumed to represent the unrestricted coefficients of the block-diagonal elements of the underlying positive-definite matrix.

Usage

```
pdBlocked(value, form, nam, data, pdClass)
```

Arguments

value	an optional list with elements to be used as the value argument to other pdMat constructors. These include: pdMat objects, positive-definite matrices, one-sided linear formulas, vectors of character strings, or numeric vectors. All elements in the list must be similar (e.g. all one-sided formulas, or all numeric vectors). Defaults to numeric(0), corresponding to an uninitialized object.
form	an optional list of one-sided linear formulas specifying the row/column names for the block-diagonal elements of the matrix represented by object. Because factors may be present in form, the formulas needs to be evaluated on a data.frame to resolve the names they define. This argument is ignored when value is a list of one-sided formulas. Defaults to NULL.
nam	an optional list of vector of character strings specifying the row/column names for the block-diagonal elements of the matrix represented by object. Each of its components must have length equal to the dimension of the corresponding block-diagonal element and unreplicated elements. This argument is ignored when value is a list of vector of character strings. Defaults to NULL.
data	an optional data frame in which to evaluate the variables named in value and form. It is used to obtain the levels for factors, which affect the dimensions and the row/column names of the underlying matrix. If NULL, no attempt is made to obtain information on any factors appearing in the formulas. Defaults to the parent frame from which the function was called.
pdClass	an optional vector of character strings naming the pdMat classes to be assigned to the individual blocks in the underlying matrix. If a single class is specified, it is used for all block-diagonal elements. This argument will only be used when value is missing, or its elements are not pdMat objects. Defaults to "pdSymm".

Value

a pdBlocked object representing a positive-definite block-diagonal matrix, also inheriting from class pdMat.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer, esp. p. 162.

See Also

as.matrix.pdMat, coef.pdMat, pdClasses, matrix<-.pdMat

Examples

```
pd1 <- pdBlocked(list(diag(1:2), diag(c(0.1, 0.2, 0.3))),
                 nam = list(c("A","B"), c("a1", "a2", "a3")))
pd1
```

`pdClasses` *Positive-Definite Matrix Classes*

Description

Standard classes of positive-definite matrices (`pdMat`) structures available in the `nlme` library.

Value

Available standard classes:

`pdSymm`	general positive-definite matrix, with no additional structure
`pdLogChol`	general positive-definite matrix, with no additional structure, using a log-Cholesky parameterization
`pdDiag`	diagonal
`pdIdent`	multiple of an identity
`pdCompSymm`	compound symmetry structure (constant diagonal and constant off-diagonal elements)
`pdBlocked`	block-diagonal matrix, with diagonal blocks of any "atomic" `pdMat` class
`pdNatural`	general positive-definite matrix in natural parametrization (i.e. parametrized in terms of standard deviations and correlations). The underlying coefficients are not unrestricted, so this class should NOT be used for optimization.

Note

Users may define their own `pdMat` classes by specifying a `constructor` function and, at a minimum, methods for the functions `pdConstruct`, `pdMatrix` and `coef`. For examples of these functions, see the methods for classes `pdSymm` and `pdDiag`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

`pdBlocked`, `pdCompSymm`, `pdDiag`, `pdFactor`, `pdIdent`, `pdMat`, `pdMatrix`, `pdNatural`, `pdSymm`, `pdLogChol`

pdCompSymm *Positive-Definite Matrix with Compound Symmetry Structure*

Description

This function is a constructor for the pdCompSymm class, representing a positive-definite matrix with compound symmetry structure (constant diagonal and constant off-diagonal elements). The underlying matrix is represented by 2 unrestricted parameters. When value is numeric(0), an unitialized pdMat object, a one-sided formula, or a vector of character strings, object is returned as an uninitialized pdCompSymm object (with just some of its attributes and its class defined) and needs to have its coefficients assigned later, generally using the coef or matrix replacement functions. If value is an initialized pdMat object, object will be constructed from as.matrix(value). Finally, if value is a numeric vector of length 2, it is assumed to represent the unrestricted coefficients of the underlying positive-definite matrix.

Usage

```
pdCompSymm(value, form, nam, data)
```

Arguments

value	an optional initialization value, which can be any of the following: a pdMat object, a positive-definite matrix, a one-sided linear formula (with variables separated by +), a vector of character strings, or a numeric vector of length 2. Defaults to numeric(0), corresponding to an uninitialized object.
form	an optional one-sided linear formula specifying the row/column names for the matrix represented by object. Because factors may be present in form, the formula needs to be evaluated on a data.frame to resolve the names it defines. This argument is ignored when value is a one-sided formula. Defaults to NULL.
nam	an optional vector of character strings specifying the row/column names for the matrix represented by object. It must have length equal to the dimension of the underlying positive-definite matrix and unreplicated elements. This argument is ignored when value is a vector of character strings. Defaults to NULL.
data	an optional data frame in which to evaluate the variables named in value and form. It is used to obtain the levels for factors, which affect the dimensions and the row/column names of the underlying matrix. If NULL, no attempt is made to obtain information on factors appearing in the formulas. Defaults to the parent frame from which the function was called.

Value

a pdCompSymm object representing a positive-definite matrix with compound symmetry structure, also inheriting from class pdMat.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer, esp. p. 161.

See Also

as.matrix.pdMat, coef.pdMat, matrix<-.pdMat, pdClasses

Examples

```
pd1 <- pdCompSymm(diag(3) + 1, nam = c("A","B","C"))
pd1
```

---

pdConstruct *Construct pdMat Objects*

Description

This function is an alternative constructor for the pdMat class associated with object and is mostly used internally in other functions. See the documentation on the principal constructor function, generally with the same name as the pdMat class of object.

Usage

```
pdConstruct(object, value, form, nam, data, ...)
```

Arguments

object	an object inheriting from class pdMat, representing a positive definite matrix.
value	an optional initialization value, which can be any of the following: a pdMat object, a positive-definite matrix, a one-sided linear formula (with variables separated by +), a vector of character strings, or a numeric vector. Defaults to numeric(0), corresponding to an uninitialized object.
form	an optional one-sided linear formula specifying the row/column names for the matrix represented by object. Because factors may be present in form, the formula needs to be evaluated on a data.frame to resolve the names it defines. This argument is ignored when value is a one-sided formula. Defaults to NULL.
nam	an optional vector of character strings specifying the row/column names for the matrix represented by object. It must have length equal to the dimension of the underlying positive-definite matrix and unreplicated elements. This argument is ignored when value is a vector of character strings. Defaults to NULL.
data	an optional data frame in which to evaluate the variables named in value and form. It is used to obtain the levels for factors, which affect the dimensions and the row/column names of the underlying matrix. If NULL, no attempt is made to obtain information on factors appearing in the formulas. Defaults to the parent frame from which the function was called.
...	optional arguments for some methods.

Value

a pdMat object representing a positive-definite matrix, inheriting from the same classes as object.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

pdCompSymm, pdDiag, pdIdent, pdNatural, pdSymm

Examples

```
pd1 <- pdSymm()
pdConstruct(pd1, diag(1:4))
```

---

pdConstruct.pdBlocked

Construct pdBlocked Objects

Description

This function give an alternative constructor for the pdBlocked class, representing a positive-definite block-diagonal matrix. Each block-diagonal element of the underlying matrix is itself a positive-definite matrix and is represented internally as an individual pdMat object. When value is numeric(0), a list of uninitialized pdMat objects, a list of one-sided formulas, or a list of vectors of character strings, object is returned as an uninitialized pdBlocked object (with just some of its attributes and its class defined) and needs to have its coefficients assigned later, generally using the coef or matrix replacement functions. If value is a list of initialized pdMat objects, object will be constructed from the list obtained by applying as.matrix to each of the pdMat elements of value. Finally, if value is a list of numeric vectors, they are assumed to represent the unrestricted coefficients of the block-diagonal elements of the underlying positive-definite matrix.

Usage

```
## S3 method for class 'pdBlocked':
pdConstruct(object, value, form, nam, data, pdClass,
...)
```

Arguments

object	an object inheriting from class pdBlocked, representing a positive definite block-diagonal matrix.
value	an optional list with elements to be used as the value argument to other pdMat constructors. These include: pdMat objects, positive-definite matrices, one-sided linear formulas, vectors of character strings, or numeric vectors. All elements in the list must be similar (e.g. all one-sided formulas, or all numeric vectors). Defaults to numeric(0), corresponding to an uninitialized object.

form	an optional list of one-sided linear formula specifying the row/column names for the block-diagonal elements of the matrix represented by object. Because factors may be present in form, the formulas needs to be evaluated on a data.frame to resolve the names they defines. This argument is ignored when value is a list of one-sided formulas. Defaults to NULL.
nam	an optional list of vector of character strings specifying the row/column names for the block-diagonal elements of the matrix represented by object. Each of its components must have length equal to the dimension of the corresponding block-diagonal element and unreplicated elements. This argument is ignored when value is a list of vector of character strings. Defaults to NULL.
data	an optional data frame in which to evaluate the variables named in value and form. It is used to obtain the levels for factors, which affect the dimensions and the row/column names of the underlying matrix. If NULL, no attempt is made to obtain information on factors appearing in the formulas. Defaults to the parent frame from which the function was called.
pdClass	an optional vector of character strings naming the pdMat classes to be assigned to the individual blocks in the underlying matrix. If a single class is specified, it is used for all block-diagonal elements. This argument will only be used when value is missing, or its elements are not pdMat objects. Defaults to "pdSymm".
...	some methods for this generic require additional arguments. None are used in this method.

Value

a pdBlocked object representing a positive-definite block-diagonal matrix, also inheriting from class pdMat.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

as.matrix.pdMat, coef.pdMat, pdBlocked, pdClasses, pdConstruct, matrix<-.pdMat

Examples

```
pd1 <- pdBlocked(list(c("A","B"), c("a1", "a2", "a3")))
pdConstruct(pd1, list(diag(1:2), diag(c(0.1, 0.2, 0.3))))
```

pdDiag *Diagonal Positive-Definite Matrix*

Description

This function is a constructor for the pdDiag class, representing a diagonal positive-definite matrix. If the matrix associated with object is of dimension n, it is represented by n unrestricted parameters, given by the logarithm of the square-root of the diagonal values. When value is numeric(0), an uninitialized pdMat object, a one-sided formula, or a vector of character strings, object is returned as an uninitialized pdDiag object (with just some of its attributes and its class defined) and needs to have its coefficients assigned later, generally using the coef or matrix replacement functions. If value is an initialized pdMat object, object will be constructed from as.matrix(value). Finally, if value is a numeric vector, it is assumed to represent the unrestricted coefficients of the underlying positive-definite matrix.

Usage

```
pdDiag(value, form, nam, data)
```

Arguments

value	an optional initialization value, which can be any of the following: a pdMat object, a positive-definite matrix, a one-sided linear formula (with variables separated by +), a vector of character strings, or a numeric vector of length equal to the dimension of the underlying positive-definite matrix. Defaults to numeric(0), corresponding to an uninitialized object.
form	an optional one-sided linear formula specifying the row/column names for the matrix represented by object. Because factors may be present in form, the formula needs to be evaluated on a data.frame to resolve the names it defines. This argument is ignored when value is a one-sided formula. Defaults to NULL.
nam	an optional vector of character strings specifying the row/column names for the matrix represented by object. It must have length equal to the dimension of the underlying positive-definite matrix and unreplicated elements. This argument is ignored when value is a vector of character strings. Defaults to NULL.
data	an optional data frame in which to evaluate the variables named in value and form. It is used to obtain the levels for factors, which affect the dimensions and the row/column names of the underlying matrix. If NULL, no attempt is made to obtain information on factors appearing in the formulas. Defaults to the parent frame from which the function was called.

Value

a pdDiag object representing a diagonal positive-definite matrix, also inheriting from class pdMat.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

as.matrix.pdMat, coef.pdMat, pdClasses, matrix<-.pdMat

Examples

```
pd1 <- pdDiag(diag(1:3), nam = c("A","B","C"))
pd1
```

---

pdFactor *Square-Root Factor of a Positive-Definite Matrix*

Description

A square-root factor of the positive-definite matrix represented by `object` is obtained. Letting Σ denote a positive-definite matrix, a square-root factor of Σ is any square matrix L such that $\Sigma = L'L$. This function extracts L.

Usage

```
pdFactor(object)
```

Arguments

`object`	an object inheriting from class `pdMat`, representing a positive definite matrix, which must have been initialized (i.e. `length(coef(object)) > 0`).

Value

a vector with a square-root factor of the positive-definite matrix associated with `object` stacked column-wise.

Note

This function is used intensively in optimization algorithms and its value is returned as a vector for efficiency reasons. The `pdMatrix` function can be used to obtain square-root factors in matrix form.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

pdMatrix

Examples

```
pd1 <- pdCompSymm(4 * diag(3) + 1)
pdFactor(pd1)
```

---

pdFactor.reStruct *Extract Square-Root Factor from Components of an reStruct Object*

Description

This method function extracts square-root factors of the positive-definite matrices corresponding to the pdMat elements of object.

Usage

```
## S3 method for class 'reStruct':
pdFactor(object)
```

Arguments

object	an object inheriting from class reStruct, representing a random effects structure and consisting of a list of pdMat objects.

Value

a vector with square-root factors of the positive-definite matrices corresponding to the elements of object stacked column-wise.

Note

This function is used intensively in optimization algorithms and its value is returned as a vector for efficiency reasons. The pdMatrix function can be used to obtain square-root factors in matrix form.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

pdFactor, pdMatrix.reStruct, pdFactor.pdMat

Examples

```
rs1 <- reStruct(pdSymm(diag(3), ~age+Sex, data = Orthodont))
pdFactor(rs1)
```

pdIdent *Multiple of the Identity Positive-Definite Matrix*

Description

This function is a constructor for the pdIdent class, representing a multiple of the identity positive-definite matrix. The matrix associated with object is represented by 1 unrestricted parameter, given by the logarithm of the square-root of the diagonal value. When value is numeric(0), an uninitialized pdMat object, a one-sided formula, or a vector of character strings, object is returned as an uninitialized pdIdent object (with just some of its attributes and its class defined) and needs to have its coefficients assigned later, generally using the coef or matrix replacement functions. If value is an initialized pdMat object, object will be constructed from as.matrix(value). Finally, if value is a numeric value, it is assumed to represent the unrestricted coefficient of the underlying positive-definite matrix.

Usage

```
pdIdent(value, form, nam, data)
```

Arguments

value	an optional initialization value, which can be any of the following: a pdMat object, a positive-definite matrix, a one-sided linear formula (with variables separated by +), a vector of character strings, or a numeric value. Defaults to numeric(0), corresponding to an uninitialized object.
form	an optional one-sided linear formula specifying the row/column names for the matrix represented by object. Because factors may be present in form, the formula needs to be evaluated on a data.frame to resolve the names it defines. This argument is ignored when value is a one-sided formula. Defaults to NULL.
nam	an optional vector of character strings specifying the row/column names for the matrix represented by object. It must have length equal to the dimension of the underlying positive-definite matrix and unreplicated elements. This argument is ignored when value is a vector of character strings. Defaults to NULL.
data	an optional data frame in which to evaluate the variables named in value and form. It is used to obtain the levels for factors, which affect the dimensions and the row/column names of the underlying matrix. If NULL, no attempt is made to obtain information on factors appearing in the formulas. Defaults to the parent frame from which the function was called.

Value

a pdIdent object representing a multiple of the identity positive-definite matrix, also inheriting from class pdMat.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

as.matrix.pdMat, coef.pdMat, pdClasses, matrix<-.pdMat

Examples

```
pd1 <- pdIdent(4 * diag(3), nam = c("A","B","C"))
pd1
```

---

pdLogChol *General Positive-Definite Matrix*

Description

This function is a constructor for the pdLogChol class, representing a general positive-definite matrix. If the matrix associated with object is of dimension n, it is represented by $n(n+1)/2$ unrestricted parameters, using the log-Cholesky parametrization described in Pinheiro and Bates (1996). When value is numeric(0), an uninitialized pdMat object, a one-sided formula, or a vector of character strings, object is returned as an uninitialized pdLogChol object (with just some of its attributes and its class defined) and needs to have its coefficients assigned later, generally using the coef or matrix replacement functions. If value is an initialized pdMat object, object will be constructed from as.matrix(value). Finally, if value is a numeric vector, it is assumed to represent the unrestricted coefficients of the matrix-logarithm parametrization of the underlying positive-definite matrix.

Usage

```
pdLogChol(value, form, nam, data)
```

Arguments

value	an optional initialization value, which can be any of the following: a pdMat object, a positive-definite matrix, a one-sided linear formula (with variables separated by +), a vector of character strings, or a numeric vector. Defaults to numeric(0), corresponding to an uninitialized object.
form	an optional one-sided linear formula specifying the row/column names for the matrix represented by object. Because factors may be present in form, the formula needs to be evaluated on a data.frame to resolve the names it defines. This argument is ignored when value is a one-sided formula. Defaults to NULL.
nam	an optional vector of character strings specifying the row/column names for the matrix represented by object. It must have length equal to the dimension of the underlying positive-definite matrix and unreplicated elements. This argument is ignored when value is a vector of character strings. Defaults to NULL.
data	an optional data frame in which to evaluate the variables named in value and form. It is used to obtain the levels for factors, which affect the dimensions and the row/column names of the underlying matrix. If NULL, no attempt is made to obtain information on factors appearing in the formulas. Defaults to the parent frame from which the function was called.

Value

a pdLogChol object representing a general positive-definite matrix, also inheriting from class pdMat.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C. and Bates., D.M. (1996) "Unconstrained Parametrizations for Variance-Covariance Matrices", Statistics and Computing, 6, 289-296.

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

as.matrix.pdMat, coef.pdMat, pdClasses, matrix<-.pdMat

Examples

```
pd1 <- pdLogChol(diag(1:3), nam = c("A","B","C"))
pd1
```

---

pdMat *Positive-Definite Matrix*

Description

This function gives an alternative way of constructing an object inheriting from the pdMat class named in pdClass, or from data.class(object) if object inherits from pdMat, and is mostly used internally in other functions. See the documentation on the principal constructor function, generally with the same name as the pdMat class of object.

Usage

```
pdMat(value, form, nam, data, pdClass)
```

Arguments

value	an optional initialization value, which can be any of the following: a pdMat object, a positive-definite matrix, a one-sided linear formula (with variables separated by +), a vector of character strings, or a numeric vector. Defaults to numeric(0), corresponding to an uninitialized object.
form	an optional one-sided linear formula specifying the row/column names for the matrix represented by object. Because factors may be present in form, the formula needs to be evaluated on a data.frame to resolve the names it defines. This argument is ignored when value is a one-sided formula. Defaults to NULL.

nam	an optional vector of character strings specifying the row/column names for the matrix represented by object. It must have length equal to the dimension of the underlying positive-definite matrix and unreplicated elements. This argument is ignored when value is a vector of character strings. Defaults to NULL.
data	an optional data frame in which to evaluate the variables named in value and form. It is used to obtain the levels for factors, which affect the dimensions and the row/column names of the underlying matrix. If NULL, no attempt is made to obtain information on factors appearing in the formulas. Defaults to the parent frame from which the function was called.
pdClass	an optional character string naming the pdMat class to be assigned to the returned object. This argument will only be used when value is not a pdMat object. Defaults to "pdSymm".

Value

a pdMat object representing a positive-definite matrix, inheriting from the class named in pdClass, or from class(object), if object inherits from pdMat.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

pdClasses, pdCompSymm, pdDiag, pdIdent, pdNatural, pdSymm, reStruct, solve.pdMat, summary.pdMat

Examples

```
pd1 <- pdMat(diag(1:4), pdClass = "pdDiag")
pd1
```

---

pdMatrix *Extract Matrix or Square-Root Factor from a pdMat Object*

Description

The positive-definite matrix represented by object, or a square-root factor of it is obtained. Letting Σ denote a positive-definite matrix, a square-root factor of Σ is any square matrix L such that $\Sigma = L'L$. This function extracts S or L.

Usage

```
pdMatrix(object, factor)
```

Arguments

`object`	an object inheriting from class `pdMat`, representing a positive definite matrix.
`factor`	an optional logical value. If `TRUE`, a square-root factor of the positive-definite matrix represented by `object` is returned; else, if `FALSE`, the positive-definite matrix is returned. Defaults to `FALSE`.

Value

if `fact` is `FALSE` the positive-definite matrix represented by `object` is returned; else a square-root of the positive-definite matrix is returned.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer, esp. p. 162.

See Also

`as.matrix.pdMat`, `pdClasses`, `pdFactor`, `pdMat`, `pdMatrix.reStruct`, `corMatrix`

Examples

```
pd1 <- pdSymm(diag(1:4))
pdMatrix(pd1)
```

`pdMatrix.reStruct` *Extract Matrix or Square-Root Factor from Components of an reStruct Object*

Description

This method function extracts the positive-definite matrices corresponding to the `pdMat` elements of `object`, or square-root factors of the positive-definite matrices.

Usage

```
## S3 method for class 'reStruct':
pdMatrix(object, factor)
```

Arguments

`object`	an object inheriting from class `reStruct`, representing a random effects structure and consisting of a list of `pdMat` objects.
`factor`	an optional logical value. If `TRUE`, square-root factors of the positive-definite matrices represented by the elements of `object` are returned; else, if `FALSE`, the positive-definite matrices are returned. Defaults to `FALSE`.

Value

a list with components given by the positive-definite matrices corresponding to the elements of `object`, or square-root factors of the positive-definite matrices.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer, esp. p. 162.

See Also

`as.matrix.reStruct`, `reStruct`, `pdMat`, `pdMatrix`, `pdMatrix.pdMat`

Examples

```
rs1 <- reStruct(pdSymm(diag(3), ~age+Sex, data = Orthodont))
pdMatrix(rs1)
```

---

pdNatural *General Positive-Definite Matrix in Natural Parametrization*

Description

This function is a constructor for the `pdNatural` class, representing a general positive-definite matrix, using a natural parametrization . If the matrix associated with `object` is of dimension n, it is represented by $n(n+1)/2$ parameters. Letting σ_{ij} denote the ij-th element of the underlying positive definite matrix and $\rho_{ij} = \sigma_i/\sqrt{\sigma_{ii}\sigma_{jj}},\ i \neq j$ denote the associated "correlations", the "natural" parameters are given by $\sqrt{\sigma_{ii}},\ i = 1, \ldots, n$ and $\log((1+\rho_{ij})/(1-\rho_{ij})),\ i \neq j$. Note that all natural parameters are individually unrestricted, but not jointly unrestricted (meaning that not all unrestricted vectors would give positive-definite matrices). Therefore, this parametrization should NOT be used for optimization. It is mostly used for deriving approximate confidence intervals on parameters following the optimization of an objective function. When `value` is `numeric(0)`, an uninitialized `pdMat` object, a one-sided formula, or a vector of character strings, `object` is returned as an uninitialized `pdSymm` object (with just some of its attributes and its class defined) and needs to have its coefficients assigned later, generally using the `coef` or `matrix` replacement functions. If `value` is an initialized `pdMat` object, `object` will be constructed from `as.matrix(value)`. Finally, if `value` is a numeric vector, it is assumed to represent the natural parameters of the underlying positive-definite matrix.

Usage

```
pdNatural(value, form, nam, data)
```

Arguments

`value`	an optional initialization value, which can be any of the following: a `pdMat` object, a positive-definite matrix, a one-sided linear formula (with variables separated by +), a vector of character strings, or a numeric vector. Defaults to `numeric(0)`, corresponding to an uninitialized object.
`form`	an optional one-sided linear formula specifying the row/column names for the matrix represented by `object`. Because factors may be present in `form`, the formula needs to be evaluated on a data.frame to resolve the names it defines. This argument is ignored when `value` is a one-sided formula. Defaults to `NULL`.
`nam`	an optional vector of character strings specifying the row/column names for the matrix represented by object. It must have length equal to the dimension of the underlying positive-definite matrix and unreplicated elements. This argument is ignored when `value` is a vector of character strings. Defaults to `NULL`.
`data`	an optional data frame in which to evaluate the variables named in `value` and `form`. It is used to obtain the levels for `factors`, which affect the dimensions and the row/column names of the underlying matrix. If `NULL`, no attempt is made to obtain information on `factors` appearing in the formulas. Defaults to the parent frame from which the function was called.

Value

a `pdNatural` object representing a general positive-definite matrix in natural parametrization, also inheriting from class `pdMat`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer, esp. p. 162.

See Also

`as.matrix.pdMat`, `coef.pdMat`, `pdClasses`, `matrix<-.pdMat`

Examples

```
pdNatural(diag(1:3))
```

`pdSymm` *General Positive-Definite Matrix*

Description

This function is a constructor for the pdSymm class, representing a general positive-definite matrix. If the matrix associated with object is of dimension n, it is represented by $n(n+1)/2$ unrestricted parameters, using the matrix-logarithm parametrization described in Pinheiro and Bates (1996). When value is numeric(0), an uninitialized pdMat object, a one-sided formula, or a vector of character strings, object is returned as an uninitialized pdSymm object (with just some of its attributes and its class defined) and needs to have its coefficients assigned later, generally using the coef or matrix replacement functions. If value is an initialized pdMat object, object will be constructed from as.matrix(value). Finally, if value is a numeric vector, it is assumed to represent the unrestricted coefficients of the matrix-logarithm parametrization of the underlying positive-definite matrix.

Usage

```
pdSymm(value, form, nam, data)
```

Arguments

value	an optional initialization value, which can be any of the following: a pdMat object, a positive-definite matrix, a one-sided linear formula (with variables separated by +), a vector of character strings, or a numeric vector. Defaults to numeric(0), corresponding to an uninitialized object.
form	an optional one-sided linear formula specifying the row/column names for the matrix represented by object. Because factors may be present in form, the formula needs to be evaluated on a data.frame to resolve the names it defines. This argument is ignored when value is a one-sided formula. Defaults to NULL.
nam	an optional vector of character strings specifying the row/column names for the matrix represented by object. It must have length equal to the dimension of the underlying positive-definite matrix and unreplicated elements. This argument is ignored when value is a vector of character strings. Defaults to NULL.
data	an optional data frame in which to evaluate the variables named in value and form. It is used to obtain the levels for factors, which affect the dimensions and the row/column names of the underlying matrix. If NULL, no attempt is made to obtain information on factors appearing in the formulas. Defaults to the parent frame from which the function was called.

Value

a pdSymm object representing a general positive-definite matrix, also inheriting from class pdMat.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C. and Bates., D.M. (1996) "Unconstrained Parametrizations for Variance-Covariance Matrices", Statistics and Computing, 6, 289-296.

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

as.matrix.pdMat, coef.pdMat, pdClasses, matrix<-.pdMat

Examples

```
pd1 <- pdSymm(diag(1:3), nam = c("A","B","C"))
pd1
```

Phenobarb *Phenobarbitol Kinetics*

Description

The Phenobarb data frame has 744 rows and 7 columns.

Format

This data frame contains the following columns:

Subject an ordered factor identifying the infant.

Wt a numeric vector giving the birth weight of the infant (kg).

Apgar an ordered factor giving the the 5-minute Apgar score for the infant. This is an indication of health of the newborn infant.

ApgarInd a factor indicating whether the 5-minute Apgar score is < 5 or >= 5.

time a numeric vector giving the time when the sample is drawn or drug administered (hr).

dose a numeric vector giving the dose of drug administered (ug/kg).

conc a numeric vector giving the phenobarbital concentration in the serum (ug/L).

Details

Data from a pharmacokinetics study of phenobarbital in neonatal infants. During the first few days of life the infants receive multiple doses of phenobarbital for prevention of seizures. At irregular intervals blood samples are drawn and serum phenobarbital concentrations are determined. The data were originally given in Grasela and Donn(1985) and are analyzed in Boeckmann, Sheiner and Beal (1994), in Davidian and Giltinan (1995), and in Littell et al. (1996).

Source

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York. (Appendix A.23)

Davidian, M. and Giltinan, D. M. (1995), *Nonlinear Models for Repeated Measurement Data*, Chapman and Hall, London. (section 6.6)

Grasela and Donn (1985), Neonatal population pharmacokinetics of phenobarbital derived from routine clinical data, *Developmental Pharmacology and Therapeutics*, **8**, 374-383.

Boeckmann, A. J., Sheiner, L. B., and Beal, S. L. (1994), *NONMEM Users Guide: Part V*, University of California, San Francisco.

Littell, R. C., Milliken, G. A., Stroup, W. W. and Wolfinger, R. D. (1996), *SAS System for Mixed Models*, SAS Institute, Cary, NC.

`phenoModel` *Model function for the Phenobarb data*

Description

A model function for a model used with the `Phenobarb` data. This function uses compiled C code to improve execution speed.

Usage

```
phenoModel(Subject, time, dose, lCl, lV)
```

Arguments

`Subject`	an integer vector of subject identifiers. These should be sorted in increasing order.
`time`	numeric. A vector of the times at which the sample was drawn or the drug administered (hr).
`dose`	numeric. A vector of doses of drug administered (ug/kg).
`lCl`	numeric. A vector of values of the natural log of the clearance parameter according to `Subject` and `time`.
`lV`	numeric. A vector of values of the natural log of the effective volume of distribution according to `Subject` and `time`.

Details

See the details section of `Phenobarb` for a description of the model function that `phenoModel` evaluates.

Value

a numeric vector of predicted phenobarbital concentrations.

Author(s)

Jose Pinheiro ⟨jose.pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J. C. and Bates, D. M. (2000) *Mixed-effects Models in S and S-PLUS*, Springer. (section 6.4)

Examples

Pixel — *X-ray pixel intensities over time*

Description

The `Pixel` data frame has 102 rows and 4 columns of data on the pixel intensities of CT scans of dogs over time

Format

This data frame contains the following columns:

Dog a factor with levels `1` to `10` designating the dog on which the scan was made

Side a factor with levels `L` and `R` designating the side of the dog being scanned

day a numeric vector giving the day post injection of the contrast on which the scan was made

pixel a numeric vector of pixel intensities

Source

Pinheiro, J. C. and Bates, D. M. (2000) *Mixed-effects Models in S and S-PLUS*, Springer.

Examples

```
fm1 <- lme(pixel ~ day + I(day^2), data = Pixel,
           random = list(Dog = ~ day, Side = ~ 1))
summary(fm1)
VarCorr(fm1)
```

plot.ACF — *Plot an ACF Object*

Description

an `xyplot` of the autocorrelations versus the lags, with `type = "h"`, is produced. If `alpha > 0`, curves representing the critical limits for a two-sided test of level `alpha` for the autocorrelations are added to the plot.

Usage

```
## S3 method for class 'ACF':
plot(x, alpha, xlab, ylab, grid, ...)
```

Arguments

x	an object inheriting from class ACF, consisting of a data frame with two columns named lag and ACF, representing the autocorrelation values and the corresponding lags.
alpha	an optional numeric value with the significance level for testing if the autocorrelations are zero. Lines corresponding to the lower and upper critical values for a test of level alpha are added to the plot. Default is 0, in which case no lines are plotted.
xlab,ylab	optional character strings with the x- and y-axis labels. Default respectively to "Lag" and "Autocorrelation".
grid	an optional logical value indicating whether a grid should be added to plot. Default is FALSE.
...	optional arguments passed to the xyplot function.

Value

an xyplot Trellis plot.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

ACF, xyplot

Examples

```
fm1 <- lme(follicles ~ sin(2*pi*Time) + cos(2*pi*Time), Ovary)
plot(ACF(fm1, maxLag = 10), alpha = 0.01)
```

---

plot.augPred *Plot an augPred Object*

Description

A Trellis xyplot of predictions versus the primary covariate is generated, with a different panel for each value of the grouping factor. Predicted values are joined by lines, with different line types (colors) being used for each level of grouping. Original observations are represented by circles.

Usage

```
## S3 method for class 'augPred':
plot(x, key, grid, ...)
```

Arguments

x	an object of class augPred.
key	an optional logical value, or list. If TRUE, a legend is included at the top of the plot indicating which symbols (colors) correspond to which prediction levels. If FALSE, no legend is included. If given as a list, key is passed down as an argument to the trellis function generating the plots (xyplot). Defaults to TRUE.
grid	an optional logical value indicating whether a grid should be added to plot. Default is FALSE.
...	optional arguments passed down to the trellis function generating the plots.

Value

A Trellis plot of predictions versus the primary covariate, with panels determined by the grouping factor.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

augPred, xyplot

Examples

```
fm1 <- lme(Orthodont)
plot(augPred(fm1, level = 0:1, length.out = 2))
```

---

plot.compareFits *Plot a compareFits Object*

Description

A Trellis dotplot of the values being compared, with different rows per group, is generated, with a different panel for each coefficient. Different symbols (colors) are used for each object being compared.

Usage

```
## S3 method for class 'compareFits':
plot(x, subset, key, mark, ...)
```

Arguments

x	an object of class compareFits.
subset	an optional logical or integer vector specifying which rows of x should be used in the plots. If missing, all rows are used.

key	an optional logical value, or list. If TRUE, a legend is included at the top of the plot indicating which symbols (colors) correspond to which objects being compared. If FALSE, no legend is included. If given as a list, key is passed down as an argument to the trellis function generating the plots (dotplot). Defaults to TRUE.
mark	an optional numeric vector, of length equal to the number of coefficients being compared, indicating where vertical lines should be drawn in the plots. If missing, no lines are drawn.
...	optional arguments passed down to the trellis function generating the plots.

Value

A Trellis dotplot of the values being compared, with rows determined by the groups and panels by the coefficients.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

compareFits, pairs.compareFits, dotplot

Examples

```
## Not run:
fm1 <- lmList(Orthodont)
fm2 <- lme(Orthodont)
plot(compareFits(coef(fm1), coef(fm2)))
## End(Not run)
```

---

plot.gls *Plot a gls Object*

Description

Diagnostic plots for the linear model fit are obtained. The form argument gives considerable flexibility in the type of plot specification. A conditioning expression (on the right side of a | operator) always implies that different panels are used for each level of the conditioning factor, according to a Trellis display. If form is a one-sided formula, histograms of the variable on the right hand side of the formula, before a | operator, are displayed (the Trellis function histogram is used). If form is two-sided and both its left and right hand side variables are numeric, scatter plots are displayed (the Trellis function xyplot is used). Finally, if form is two-sided and its left had side variable is a factor, box-plots of the right hand side variable by the levels of the left hand side variable are displayed (the Trellis function bwplot is used).

Usage

```
## S3 method for class 'gls':
plot(x, form, abline, id, idLabels, idResType, grid, ...)
```

Arguments

x	an object inheriting from class gls, representing a generalized least squares fitted linear model.
form	an optional formula specifying the desired type of plot. Any variable present in the original data frame used to obtain x can be referenced. In addition, x itself can be referenced in the formula using the symbol ".". Conditional expressions on the right of a | operator can be used to define separate panels in a Trellis display. Default is resid(., type = "p") ~ fitted(.) , corresponding to a plot of the standardized residuals versus fitted values, both evaluated at the innermost level of nesting.
abline	an optional numeric value, or numeric vector of length two. If given as a single value, a horizontal line will be added to the plot at that coordinate; else, if given as a vector, its values are used as the intercept and slope for a line added to the plot. If missing, no lines are added to the plot.
id	an optional numeric value, or one-sided formula. If given as a value, it is used as a significance level for a two-sided outlier test for the standardized residuals. Observations with absolute standardized residuals greater than the $1 - value/2$ quantile of the standard normal distribution are identified in the plot using idLabels. If given as a one-sided formula, its right hand side must evaluate to a logical, integer, or character vector which is used to identify observations in the plot. If missing, no observations are identified.
idLabels	an optional vector, or one-sided formula. If given as a vector, it is converted to character mode and used to label the observations identified according to id. If given as a one-sided formula, its right hand side must evaluate to a vector which is converted to character mode and used to label the identified observations. Default is the innermost grouping factor.
idResType	an optional character string specifying the type of residuals to be used in identifying outliers, when id is a numeric value. If "pearson", the standardized residuals (raw residuals divided by the corresponding standard errors) are used; else, if "normalized", the normalized residuals (standardized residuals premultiplied by the inverse square-root factor of the estimated error correlation matrix) are used. Partial matching of arguments is used, so only the first character needs to be provided. Defaults to "pearson".
grid	an optional logical value indicating whether a grid should be added to plot. Default depends on the type of Trellis plot used: if xyplot defaults to TRUE, else defaults to FALSE.
...	optional arguments passed to the Trellis plot function.

Value

a diagnostic Trellis plot.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

gls, xyplot, bwplot, histogram

Examples

```
fm1 <- gls(follicles ~ sin(2*pi*Time) + cos(2*pi*Time), Ovary,
           correlation = corAR1(form = ~ 1 | Mare))
# standardized residuals versus fitted values by Mare
plot(fm1, resid(., type = "p") ~ fitted(.) | Mare, abline = 0)
# box-plots of residuals by Mare
plot(fm1, Mare ~ resid(.))
# observed versus fitted values by Mare
plot(fm1, follicles ~ fitted(.) | Mare, abline = c(0,1))
```

---

plot.intervals.lmList

Plot lmList Confidence Intervals

Description

A Trellis dot-plot of the confidence intervals on the linear model coefficients is generated, with a different panel for each coefficient. Rows in the dot-plot correspond to the names of the `lm` components of the `lmList` object used to produce `x`. The lower and upper confidence limits are connected by a line segment and the estimated coefficients are marked with a `"+"`. The Trellis function `dotplot` is used in this method function.

Usage

```
## S3 method for class 'intervals.lmList':
plot(x, ...)
```

Arguments

`x`	an object inheriting from class `intervals.lmList`, representing confidence intervals and estimates for the coefficients in the `lm` components of the `lmList` object used to produce `x`.
`...`	optional arguments passed to the Trellis `dotplot` function.

Value

a Trellis plot with the confidence intervals on the coefficients of the individual `lm` components of the `lmList` that generated `x`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`intervals.lmList`, `lmList`, `dotplot`

Examples

```
fm1 <- lmList(distance ~ age | Subject, Orthodont)
plot(intervals(fm1))
```

`plot.lme` *Plot an lme or nls object*

Description

Diagnostic plots for the linear mixed-effects fit are obtained. The `form` argument gives considerable flexibility in the type of plot specification. A conditioning expression (on the right side of a `|` operator) always implies that different panels are used for each level of the conditioning factor, according to a Trellis display. If `form` is a one-sided formula, histograms of the variable on the right hand side of the formula, before a `|` operator, are displayed (the Trellis function `histogram` is used). If `form` is two-sided and both its left and right hand side variables are numeric, scatter plots are displayed (the Trellis function `xyplot` is used). Finally, if `form` is two-sided and its left had side variable is a factor, box-plots of the right hand side variable by the levels of the left hand side variable are displayed (the Trellis function `bwplot` is used).

Usage

```
## S3 method for class 'lme':
plot(x, form, abline, id, idLabels, idResType, grid, ...)
## S3 method for class 'nls':
plot(x, form, abline, id, idLabels, idResType, grid, ...)
```

Arguments

`x`	an object inheriting from class `lme`, representing a fitted linear mixed-effects model, or from `nls`, representing an fitted nonlinear least squares model.
`form`	an optional formula specifying the desired type of plot. Any variable present in the original data frame used to obtain `x` can be referenced. In addition, `x` itself can be referenced in the formula using the symbol `"."`. Conditional expressions on the right of a `|` operator can be used to define separate panels in a Trellis display. Default is `resid(., type = "p") ~ fitted(.)` , corresponding to a plot of the standardized residuals versus fitted values, both evaluated at the innermost level of nesting.
`abline`	an optional numeric value, or numeric vector of length two. If given as a single value, a horizontal line will be added to the plot at that coordinate; else, if given as a vector, its values are used as the intercept and slope for a line added to the plot. If missing, no lines are added to the plot.
`id`	an optional numeric value, or one-sided formula. If given as a value, it is used as a significance level for a two-sided outlier test for the standardized, or normalized residuals. Observations with absolute standardized (normalized) residuals greater than the $1 - value/2$ quantile of the standard normal distribution are identified in the plot using `idLabels`. If given as a one-sided formula, its right hand side must evaluate to a logical, integer, or character vector which is used to identify observations in the plot. If missing, no observations are identified.
`idLabels`	an optional vector, or one-sided formula. If given as a vector, it is converted to character and used to label the observations identified according to `id`. If given as a one-sided formula, its right hand side must evaluate to a vector which is converted to character and used to label the identified observations. Default is the innermost grouping factor.

idResType	an optional character string specifying the type of residuals to be used in identifying outliers, when id is a numeric value. If "pearson", the standardized residuals (raw residuals divided by the corresponding standard errors) are used; else, if "normalized", the normalized residuals (standardized residuals premultiplied by the inverse square-root factor of the estimated error correlation matrix) are used. Partial matching of arguments is used, so only the first character needs to be provided. Defaults to "pearson".
grid	an optional logical value indicating whether a grid should be added to plot. Default depends on the type of Trellis plot used: if xyplot defaults to TRUE, else defaults to FALSE.
...	optional arguments passed to the Trellis plot function.

Value

a diagnostic Trellis plot.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

lme, xyplot, bwplot, histogram

Examples

```
fm1 <- lme(distance ~ age, Orthodont, random = ~ age | Subject)
# standardized residuals versus fitted values by gender
plot(fm1, resid(., type = "p") ~ fitted(.) | Sex, abline = 0)
# box-plots of residuals by Subject
plot(fm1, Subject ~ resid(.))
# observed versus fitted values by Subject
plot(fm1, distance ~ fitted(.) | Subject, abline = c(0,1))
```

---

plot.lmList *Plot an lmList Object*

Description

Diagnostic plots for the linear model fits corresponding to the x components are obtained. The form argument gives considerable flexibility in the type of plot specification. A conditioning expression (on the right side of a | operator) always implies that different panels are used for each level of the conditioning factor, according to a Trellis display. If form is a one-sided formula, histograms of the variable on the right hand side of the formula, before a | operator, are displayed (the Trellis function histogram is used). If form is two-sided and both its left and right hand side variables are numeric, scatter plots are displayed (the Trellis function xyplot is used). Finally, if form is two-sided and its left had side variable is a factor, box-plots of the right hand side variable by the levels of the left hand side variable are displayed (the Trellis function bwplot is used).

Usage

```
## S3 method for class 'lmList':
plot(x, form, abline, id, idLabels, grid, ...)
```

Arguments

x	an object inheriting from class lmList, representing a list of lm objects with a common model.
form	an optional formula specifying the desired type of plot. Any variable present in the original data frame used to obtain x can be referenced. In addition, x itself can be referenced in the formula using the symbol ".". Conditional expressions on the right of a | operator can be used to define separate panels in a Trellis display. Default is resid(., type = "pool") ~ fitted(.) , corresponding to a plot of the standardized residuals (using a pooled estimate for the residual standard error) versus fitted values.
abline	an optional numeric value, or numeric vector of length two. If given as a single value, a horizontal line will be added to the plot at that coordinate; else, if given as a vector, its values are used as the intercept and slope for a line added to the plot. If missing, no lines are added to the plot.
id	an optional numeric value, or one-sided formula. If given as a value, it is used as a significance level for a two-sided outlier test for the standardized residuals. Observations with absolute standardized residuals greater than the $1 - value/2$ quantile of the standard normal distribution are identified in the plot using idLabels. If given as a one-sided formula, its right hand side must evaluate to a logical, integer, or character vector which is used to identify observations in the plot. If missing, no observations are identified.
idLabels	an optional vector, or one-sided formula. If given as a vector, it is converted to character and used to label the observations identified according to id. If given as a one-sided formula, its right hand side must evaluate to a vector which is converted to character and used to label the identified observations. Default is getGroups(x).
grid	an optional logical value indicating whether a grid should be added to plot. Default depends on the type of Trellis plot used: if xyplot defaults to TRUE, else defaults to FALSE.
...	optional arguments passed to the Trellis plot function.

Value

a diagnostic Trellis plot.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

lmList,predict.lm, xyplot, bwplot, histogram

Examples

```
fm1 <- lmList(distance ~ age | Subject, Orthodont)
# standardized residuals versus fitted values by gender
plot(fm1, resid(., type = "pool") ~ fitted(.) | Sex, abline = 0, id = 0.05)
# box-plots of residuals by Subject
plot(fm1, Subject ~ resid(.))
# observed versus fitted values by Subject
plot(fm1, distance ~ fitted(.) | Subject, abline = c(0,1))
```

plot.nffGroupedData *Plot an nffGroupedData Object*

Description

A Trellis dot-plot of the response by group is generated. If outer variables are specified, the combination of their levels are used to determine the panels of the Trellis display. The Trellis function `dotplot` is used.

Usage

```
## S3 method for class 'nffGroupedData':
plot(x, outer, inner, innerGroups, xlab, ylab, strip, panel, key,
     grid, ...)
```

Arguments

`x`	an object inheriting from class `nffGroupedData`, representing a `groupedData` object with a factor primary covariate and a single grouping level.
`outer`	an optional logical value or one-sided formula, indicating covariates that are outer to the grouping factor, which are used to determine the panels of the Trellis plot. If equal to `TRUE`, `attr(object, "outer")` is used to indicate the outer covariates. An outer covariate is invariant within the sets of rows defined by the grouping factor. Ordering of the groups is done in such a way as to preserve adjacency of groups with the same value of the outer variables. Defaults to `NULL`, meaning that no outer covariates are to be used.
`inner`	an optional logical value or one-sided formula, indicating a covariate that is inner to the grouping factor, which is used to associate points within each panel of the Trellis plot. If equal to `TRUE`, `attr(object, "inner")` is used to indicate the inner covariate. An inner covariate can change within the sets of rows defined by the grouping factor. Defaults to `NULL`, meaning that no inner covariate is present.
`innerGroups`	an optional one-sided formula specifying a factor to be used for grouping the levels of the `inner` covariate. Different colors, or symbols, are used for each level of the `innerGroups` factor. Default is `NULL`, meaning that no `innerGroups` covariate is present.
`xlab`	an optional character string with the label for the horizontal axis. Default is the `y` elements of `attr(object, "labels")` and `attr(object, "units")` pasted together.
`ylab`	an optional character string with the label for the vertical axis. Default is the grouping factor name.
`strip`	an optional function passed as the `strip` argument to the `dotplot` function. Default is `strip.default(..., style = 1)` (see `trellis.args`).
`panel`	an optional function used to generate the individual panels in the Trellis display, passed as the `panel` argument to the `dotplot` function.

key	an optional logical function or function. If TRUE and either inner or innerGroups are non-NULL, a legend for the different inner (innerGroups) levels is included at the top of the plot. If given as a function, it is passed as the key argument to the dotplot function. Default is TRUE is either inner or innerGroups are non-NULL and FALSE otherwise.
grid	this argument is included for consistency with the plot.nfnGroupedData method calling sequence. It is ignored in this method function.
...	optional arguments passed to the dotplot function.

Value

a Trellis dot-plot of the response by group.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Bates, D.M. and Pinheiro, J.C. (1997), "Software Design for Longitudinal Data", in "Modelling Longitudinal and Spatially Correlated Data: Methods, Applications and Future Directions", T.G. Gregoire (ed.), Springer-Verlag, New York.

Pinheiro, J.C. and Bates, D.M. (1997) "Future Directions in Mixed-Effects Software: Design of NLME 3.0" available at http://franz.stat.wisc.edu/pub/NLME.

See Also

groupedData, dotplot

Examples

```
plot(Machines)
plot(Machines, inner = TRUE)
```

---

plot.nfnGroupedData

Plot an nfnGroupedData Object

Description

A Trellis plot of the response versus the primary covariate is generated. If outer variables are specified, the combination of their levels are used to determine the panels of the Trellis display. Otherwise, the levels of the grouping variable determine the panels. A scatter plot of the response versus the primary covariate is displayed in each panel, with observations corresponding to same inner group joined by line segments. The Trellis function xyplot is used.

Usage

```
## S3 method for class 'nfnGroupedData':
plot(x, outer, inner, innerGroups, xlab, ylab, strip, aspect, panel,
     key, grid, ...)
```

Arguments

x	an object inheriting from class nfnGroupedData, representing a groupedData object with a numeric primary covariate and a single grouping level.
outer	an optional logical value or one-sided formula, indicating covariates that are outer to the grouping factor, which are used to determine the panels of the Trellis plot. If equal to TRUE, attr(object, "outer") is used to indicate the outer covariates. An outer covariate is invariant within the sets of rows defined by the grouping factor. Ordering of the groups is done in such a way as to preserve adjacency of groups with the same value of the outer variables. Defaults to NULL, meaning that no outer covariates are to be used.
inner	an optional logical value or one-sided formula, indicating a covariate that is inner to the grouping factor, which is used to associate points within each panel of the Trellis plot. If equal to TRUE, attr(object, "inner") is used to indicate the inner covariate. An inner covariate can change within the sets of rows defined by the grouping factor. Defaults to NULL, meaning that no inner covariate is present.
innerGroups	an optional one-sided formula specifying a factor to be used for grouping the levels of the inner covariate. Different colors, or line types, are used for each level of the innerGroups factor. Default is NULL, meaning that no innerGroups covariate is present.
xlab, ylab	optional character strings with the labels for the plot. Default is the corresponding elements of attr(object, "labels") and attr(object, "units") pasted together.
strip	an optional function passed as the strip argument to the xyplot function. Default is strip.default(..., style = 1) (see trellis.args).
aspect	an optional character string indicating the aspect ratio for the plot passed as the aspect argument to the xyplot function. Default is "xy" (see trellis.args).
panel	an optional function used to generate the individual panels in the Trellis display, passed as the panel argument to the xyplot function.
key	an optional logical function or function. If TRUE and innerGroups is non-NULL, a legend for the different innerGroups levels is included at the top of the plot. If given as a function, it is passed as the key argument to the xyplot function. Default is TRUE if innerGroups is non-NULL and FALSE otherwise.
grid	an optional logical value indicating whether a grid should be added to plot. Default is TRUE.
...	optional arguments passed to the xyplot function.

Value

a Trellis plot of the response versus the primary covariate.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Bates, D.M. and Pinheiro, J.C. (1997), "Software Design for Longitudinal Data", in "Modelling Longitudinal and Spatially Correlated Data: Methods, Applications and Future Directions", T.G. Gregoire (ed.), Springer-Verlag, New York.

Pinheiro, J.C. and Bates, D.M. (1997) "Future Directions in Mixed-Effects Software: Design of NLME 3.0" available at http://franz.stat.wisc.edu/pub/NLME.

See Also

`groupedData`, `xyplot`

Examples

```
# different panels per Subject
plot(Orthodont)
# different panels per gender
plot(Orthodont, outer = TRUE)
```

`plot.nmGroupedData` *Plot an nmGroupedData Object*

Description

The `groupedData` object is summarized by the values of the `displayLevel` grouping factor (or the combination of its values and the values of the covariate indicated in `preserve`, if any is present). The collapsed data is used to produce a new `groupedData` object, with grouping factor given by the `displayLevel` factor, which is plotted using the appropriate `plot` method for `groupedData` objects with single level of grouping.

Usage

```
## S3 method for class 'nmGroupedData':
plot(x, collapseLevel, displayLevel, outer, inner,
    preserve, FUN, subset, key, grid, ...)
```

Arguments

`x`	an object inheriting from class `nmGroupedData`, representing a `groupedData` object with multiple grouping factors.
`collapseLevel`	an optional positive integer or character string indicating the grouping level to use when collapsing the data. Level values increase from outermost to innermost grouping. Default is the highest or innermost level of grouping.
`displayLevel`	an optional positive integer or character string indicating the grouping level to use for determining the panels in the Trellis display, when `outer` is missing. Default is `collapseLevel`.
`outer`	an optional logical value or one-sided formula, indicating covariates that are outer to the `displayLevel` grouping factor, which are used to determine the panels of the Trellis plot. If equal to `TRUE`, the `displayLevel` element `attr(object, "outer")` is used to indicate the outer covariates. An outer

covariate is invariant within the sets of rows defined by the grouping factor. Ordering of the groups is done in such a way as to preserve adjacency of groups with the same value of the outer variables. Defaults to NULL, meaning that no outer covariates are to be used.

inner — an optional logical value or one-sided formula, indicating a covariate that is inner to the displayLevel grouping factor, which is used to associate points within each panel of the Trellis plot. If equal to TRUE, attr(object, "outer") is used to indicate the inner covariate. An inner covariate can change within the sets of rows defined by the grouping factor. Defaults to NULL, meaning that no inner covariate is present.

preserve — an optional one-sided formula indicating a covariate whose levels should be preserved when collapsing the data according to the collapseLevel grouping factor. The collapsing factor is obtained by pasting together the levels of the collapseLevel grouping factor and the values of the covariate to be preserved. Default is NULL, meaning that no covariates need to be preserved.

FUN — an optional summary function or a list of summary functions to be used for collapsing the data. The function or functions are applied only to variables in object that vary within the groups defined by collapseLevel. Invariant variables are always summarized by group using the unique value that they assume within that group. If FUN is a single function it will be applied to each non-invariant variable by group to produce the summary for that variable. If FUN is a list of functions, the names in the list should designate classes of variables in the data such as ordered, factor, or numeric. The indicated function will be applied to any non-invariant variables of that class. The default functions to be used are mean for numeric factors, and Mode for both factor and ordered. The Mode function, defined internally in gsummary, returns the modal or most popular value of the variable. It is different from the mode function that returns the S-language mode of the variable.

subset — an optional named list. Names can be either positive integers representing grouping levels, or names of grouping factors. Each element in the list is a vector indicating the levels of the corresponding grouping factor to be used for plotting the data. Default is NULL, meaning that all levels are used.

key — an optional logical value, or list. If TRUE, a legend is included at the top of the plot indicating which symbols (colors) correspond to which prediction levels. If FALSE, no legend is included. If given as a list, key is passed down as an argument to the trellis function generating the plots (xyplot). Defaults to TRUE.

grid — an optional logical value indicating whether a grid should be added to plot. Default is TRUE.

... — optional arguments passed to the Trellis plot function.

Value

a Trellis display of the data collapsed over the values of the collapseLevel grouping factor and grouped according to the displayLevel grouping factor.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Bates, D.M. and Pinheiro, J.C. (1997), "Software Design for Longitudinal Data", in "Modelling Longitudinal and Spatially Correlated Data: Methods, Applications and Future Directions", T.G. Gregoire (ed.), Springer-Verlag, New York.

Pinheiro, J.C. and Bates, D.M. (1997) "Future Directions in Mixed-Effects Software: Design of NLME 3.0" available at http://franz.stat.wisc.edu/pub/NLME.

See Also

groupedData, collapse.groupedData, plot.nfnGroupedData, plot.nffGroupedData

Examples

```
# no collapsing, panels by Dog
plot(Pixel, display = "Dog", inner = ~Side)
# collapsing by Dog, preserving day
plot(Pixel, collapse = "Dog", preserve = ~day)
```

---

plot.ranef.lme *Plot a ranef.lme Object*

Description

If form is missing, or is given as a one-sided formula, a Trellis dot-plot of the random effects is generated, with a different panel for each random effect (coefficient). Rows in the dot-plot are determined by the form argument (if not missing) or by the row names of the random effects (coefficients). If a single factor is specified in form, its levels determine the dot-plot rows (with possibly multiple dots per row); otherwise, if form specifies a crossing of factors, the dot-plot rows are determined by all combinations of the levels of the individual factors in the formula. The Trellis function dotplot is used in this method function.

If form is a two-sided formula, a Trellis display is generated, with a different panel for each variable listed in the right hand side of form. Scatter plots are generated for numeric variables and boxplots are generated for categorical (factor or ordered) variables.

Usage

```
## S3 method for class 'ranef.lme':
plot(x, form, omitFixed, level, grid, control, ...)
```

Arguments

x	an object inheriting from class ranef.lme, representing the estimated coefficients or estimated random effects for the lme object from which it was produced.
form	an optional formula specifying the desired type of plot. If given as a one-sided formula, a dotplot of the estimated random effects (coefficients) grouped according to all combinations of the levels of the factors named in form is returned. Single factors (~g) or crossed factors (~g1*g2) are allowed. If given as a two-sided formula, the left hand side must be a single random effects (coefficient) and the right hand side is formed by covariates in x separated by +.

A Trellis display of the random effect (coefficient) versus the named covariates is returned in this case. Default is NULL, in which case the row names of the random effects (coefficients) are used.

omitFixed — an optional logical value indicating whether columns with values that are constant across groups should be omitted. Default is TRUE.

level — an optional integer value giving the level of grouping to be used for x. Only used when x is a list with different components for each grouping level. Defaults to the highest or innermost level of grouping.

grid — an optional logical value indicating whether a grid should be added to plot. Only applies to plots associated with two-sided formulas in form. Default is FALSE.

control — an optional list with control values for the plot, when form is given as a two-sided formula. The control values are referenced by name in the control list and only the ones to be modified from the default need to be specified. Available values include: drawLine, a logical value indicating whether a loess smoother should be added to the scatter plots and a line connecting the medians should be added to the boxplots (default is TRUE); span.loess, used as the span argument in the call to panel.loess (default is 2/3); degree.loess, used as the degree argument in the call to panel.loess (default is 1); cex.axis, the character expansion factor for the x-axis (default is 0.8); srt.axis, the rotation factor for the x-axis (default is 0); and mgp.axis, the margin parameters for the x-axis (default is c(2, 0.5, 0)).

... — optional arguments passed to the Trellis dotplot function.

Value

a Trellis plot of the estimated random-effects (coefficients) versus covariates, or groups.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

ranef.lme, lme, dotplot

Examples

```
## Not run:
fm1 <- lme(distance ~ age, Orthodont, random = ~ age | Subject)
plot(ranef(fm1))
fm1RE <- ranef(fm1, aug = TRUE)
plot(fm1RE, form = ~ Sex)
plot(fm1RE, form = age ~ Sex)
## End(Not run)
```

plot.ranef.lmList *Plot a ranef.lmList Object*

Description

If form is missing, or is given as a one-sided formula, a Trellis dot-plot of the random effects is generated, with a different panel for each random effect (coefficient). Rows in the dot-plot are determined by the form argument (if not missing) or by the row names of the random effects (coefficients). If a single factor is specified in form, its levels determine the dot-plot rows (with possibly multiple dots per row); otherwise, if form specifies a crossing of factors, the dot-plot rows are determined by all combinations of the levels of the individual factors in the formula. The Trellis function dotplot is used in this method function.

If form is a two-sided formula, a Trellis display is generated, with a different panel for each variable listed in the right hand side of form. Scatter plots are generated for numeric variables and boxplots are generated for categorical (factor or ordered) variables.

Usage

```
## S3 method for class 'ranef.lmList':
plot(x, form, grid, control, ...)
```

Arguments

x	an object inheriting from class ranef.lmList, representing the estimated coefficients or estimated random effects for the lmList object from which it was produced.
form	an optional formula specifying the desired type of plot. If given as a one-sided formula, a dotplot of the estimated random effects (coefficients) grouped according to all combinations of the levels of the factors named in form is returned. Single factors (~g) or crossed factors (~g1*g2) are allowed. If given as a two-sided formula, the left hand side must be a single random effects (coefficient) and the right hand side is formed by covariates in x separated by +. A Trellis display of the random effect (coefficient) versus the named covariates is returned in this case. Default is NULL, in which case the row names of the random effects (coefficients) are used.
grid	an optional logical value indicating whether a grid should be added to plot. Only applies to plots associated with two-sided formulas in form. Default is FALSE.
control	an optional list with control values for the plot, when form is given as a two-sided formula. The control values are referenced by name in the control list and only the ones to be modified from the default need to be specified. Available values include: drawLine, a logical value indicating whether a loess smoother should be added to the scatter plots and a line connecting the medians should be added to the boxplots (default is TRUE); span.loess, used as the span argument in the call to panel.loess (default is 2/3); degree.loess, used as the degree argument in the call to panel.loess (default is 1); cex.axis, the character expansion factor for the x-axis (default is 0.8); srt.axis, the rotation factor for the x-axis (default is 0); and mgp.axis, the margin parameters for the x-axis (default is c(2, 0.5, 0)).
...	optional arguments passed to the Trellis dotplot function.

Value

a Trellis plot of the estimated random-effects (coefficients) versus covariates, or groups.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

lmList, dotplot

Examples

```
fm1 <- lmList(distance ~ age | Subject, Orthodont)
plot(ranef(fm1))
fm1RE <- ranef(fm1, aug = TRUE)
plot(fm1RE, form = ~ Sex)
## Not run:
plot(fm1RE, form = age ~ Sex)
## End(Not run)
```

---

plot.Variogram *Plot a Variogram Object*

Description

an xyplot of the semi-variogram versus the distances is produced. If smooth = TRUE, a loess smoother is added to the plot. If showModel = TRUE and x includes an "modelVariog" attribute, the corresponding semi-variogram is added to the plot.

Usage

```
## S3 method for class 'Variogram':
plot(x, smooth, showModel, sigma, span, xlab,
     ylab, type, ylim, grid, ...)
```

Arguments

x	an object inheriting from class Variogram, consisting of a data frame with two columns named variog and dist, representing the semi-variogram values and the corresponding distances.
smooth	an optional logical value controlling whether a loess smoother should be added to the plot. Defaults to TRUE, when showModel is FALSE.
showModel	an optional logical value controlling whether the semi-variogram corresponding to an "modelVariog" attribute of x, if any is present, should be added to the plot. Defaults to TRUE, when the "modelVariog" attribute is present.
sigma	an optional numeric value used as the height of a horizontal line displayed in the plot. Can be used to represent the process standard deviation. Default is NULL, implying that no horizontal line is drawn.
span	an optional numeric value with the smoothing parameter for the loess fit. Default is 0.6.

xlab,ylab	optional character strings with the x- and y-axis labels. Default respectively to "Distance" and "SemiVariogram".
type	an optional character indicating the type of plot. Defaults to "p".
ylim	an optional numeric vector with the limits for the y-axis. Defaults to c(0, max(x$variog)).
grid	an optional logical value indicating whether a grid should be added to plot. Default is FALSE.
...	optional arguments passed to the Trellis xyplot function.

Value

an xyplot Trellis plot.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

Variogram, xyplot, loess

Examples

```
fm1 <- lme(follicles ~ sin(2*pi*Time) + cos(2*pi*Time), Ovary)
plot(Variogram(fm1, form = ~ Time | Mare, maxDist = 0.7))
```

---

pooledSD *Extract Pooled Standard Deviation*

Description

The pooled estimated standard deviation is obtained by adding together the residual sum of squares for each non-null element of object, dividing by the sum of the corresponding residual degrees-of-freedom, and taking the square-root.

Usage

```
pooledSD(object)
```

Arguments

object	an object inheriting from class lmList.

Value

the pooled standard deviation for the non-null elements of object, with an attribute df with the number of degrees-of-freedom used in the estimation.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

lmList, lm

Examples

```
fm1 <- lmList(Orthodont)
pooledSD(fm1)
```

`predict.gls` *Predictions from a gls Object*

Description

The predictions for the linear model represented by `object` are obtained at the covariate values defined in `newdata`.

Usage

```
## S3 method for class 'gls':
predict(object, newdata, na.action, ...)
```

Arguments

`object`	an object inheriting from class `gls`, representing a generalized least squares fitted linear model.
`newdata`	an optional data frame to be used for obtaining the predictions. All variables used in the linear model must be present in the data frame. If missing, the fitted values are returned.
`na.action`	a function that indicates what should happen when `newdata` contains `NA`s. The default action (`na.fail`) causes the function to print an error message and terminate if there are any incomplete observations.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

a vector with the predicted values.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

gls

Examples

```
fm1 <- gls(follicles ~ sin(2*pi*Time) + cos(2*pi*Time), Ovary,
           correlation = corAR1(form = ~ 1 | Mare))
newOvary <- data.frame(Time = c(-0.75, -0.5, 0, 0.5, 0.75))
predict(fm1, newOvary)
```

predict.gnls *Predictions from a gnls Object*

Description

The predictions for the nonlinear model represented by `object` are obtained at the covariate values defined in `newdata`.

Usage

```
## S3 method for class 'gnls':
predict(object, newdata, na.action, naPattern, ...)
```

Arguments

`object`	an object inheriting from class `gnls`, representing a generalized nonlinear least squares fitted model.
`newdata`	an optional data frame to be used for obtaining the predictions. All variables used in the nonlinear model must be present in the data frame. If missing, the fitted values are returned.
`na.action`	a function that indicates what should happen when `newdata` contains `NA`s. The default action (`na.fail`) causes the function to print an error message and terminate if there are any incomplete observations.
`naPattern`	an expression or formula object, specifying which returned values are to be regarded as missing.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

a vector with the predicted values.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`gnls`

Examples

```
fm1 <- gnls(weight ~ SSlogis(Time, Asym, xmid, scal), Soybean,
            weights = varPower())
newSoybean <- data.frame(Time = c(10,30,50,80,100))
predict(fm1, newSoybean)
```

`predict.lme` *Predictions from an lme Object*

Description

The predictions at level i are obtained by adding together the population predictions (based only on the fixed effects estimates) and the estimated contributions of the random effects to the predictions at grouping levels less or equal to i. The resulting values estimate the best linear unbiased predictions (BLUPs) at level i. If group values not included in the original grouping factors are present in `newdata`, the corresponding predictions will be set to `NA` for levels greater or equal to the level at which the unknown groups occur.

Usage

```
## S3 method for class 'lme':
predict(object, newdata, level, asList, na.action, ...)
```

Arguments

`object`	an object inheriting from class `lme`, representing a fitted linear mixed-effects model.
`newdata`	an optional data frame to be used for obtaining the predictions. All variables used in the fixed and random effects models, as well as the grouping factors, must be present in the data frame. If missing, the fitted values are returned.
`level`	an optional integer vector giving the level(s) of grouping to be used in obtaining the predictions. Level values increase from outermost to innermost grouping, with level zero corresponding to the population predictions. Defaults to the highest or innermost level of grouping.
`asList`	an optional logical value. If `TRUE` and a single value is given in `level`, the returned object is a list with the predictions split by groups; else the returned value is either a vector or a data frame, according to the length of `level`.
`na.action`	a function that indicates what should happen when `newdata` contains `NA`s. The default action (`na.fail`) causes the function to print an error message and terminate if there are any incomplete observations.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

if a single level of grouping is specified in `level`, the returned value is either a list with the predictions split by groups (`asList = TRUE`) or a vector with the predictions (`asList = FALSE`); else, when multiple grouping levels are specified in `level`, the returned object is a data frame with columns given by the predictions at different levels and the grouping factors.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`lme`, `fitted.lme`

Examples

```
fm1 <- lme(distance ~ age, Orthodont, random = ~ age | Subject)
newOrth <- data.frame(Sex = c("Male","Male","Female","Female","Male","Male"),
                      age = c(15, 20, 10, 12, 2, 4),
                      Subject = c("M01","M01","F30","F30","M04","M04"))
predict(fm1, newOrth, level = 0:1)
```

---

predict.lmList *Predictions from an lmList Object*

Description

If the grouping factor corresponding to `object` is included in `newdata`, the data frame is partitioned according to the grouping factor levels; else, `newdata` is repeated for all `lm` components. The predictions and, optionally, the standard errors for the predictions, are obtained for each `lm` component of `object`, using the corresponding element of the partitioned `newdata`, and arranged into a list with as many components as `object`, or combined into a single vector or data frame (if `se.fit=TRUE`).

Usage

```
## S3 method for class 'lmList':
predict(object, newdata, subset, pool, asList, se.fit, ...)
```

Arguments

`object`	an object inheriting from class `lmList`, representing a list of `lm` objects with a common model.
`newdata`	an optional data frame to be used for obtaining the predictions. All variables used in the `object` model formula must be present in the data frame. If missing, the same data frame used to produce `object` is used.
`subset`	an optional character or integer vector naming the `lm` components of `object` from which the predictions are to be extracted. Default is `NULL`, in which case all components are used.
`asList`	an optional logical value. If `TRUE`, the returned object is a list with the predictions split by groups; else the returned value is a vector. Defaults to `FALSE`.
`pool`	an optional logical value indicating whether a pooled estimate of the residual standard error should be used. Default is `attr(object, "pool")`.
`se.fit`	an optional logical value indicating whether pointwise standard errors should be computed along with the predictions. Default is `FALSE`.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

a list with components given by the predictions (and, optionally, the standard errors for the predictions) from each `lm` component of `object`, a vector with the predictions from all `lm` components of `object`, or a data frame with columns given by the predictions and their corresponding standard errors.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

lmList, predict.lm

Examples

```
fm1 <- lmList(distance ~ age | Subject, Orthodont)
predict(fm1, se.fit = TRUE)
```

---

predict.nlme *Predictions from an nlme Object*

Description

The predictions at level i are obtained by adding together the contributions from the estimated fixed effects and the estimated random effects at levels less or equal to i and evaluating the model function at the resulting estimated parameters. If group values not included in the original grouping factors are present in newdata, the corresponding predictions will be set to NA for levels greater or equal to the level at which the unknown groups occur.

Usage

```
## S3 method for class 'nlme':
predict(object, newdata, level, asList, na.action,
naPattern, ...)
```

Arguments

object	an object inheriting from class nlme, representing a fitted nonlinear mixed-effects model.
newdata	an optional data frame to be used for obtaining the predictions. All variables used in the nonlinear model, the fixed and the random effects models, as well as the grouping factors, must be present in the data frame. If missing, the fitted values are returned.
level	an optional integer vector giving the level(s) of grouping to be used in obtaining the predictions. Level values increase from outermost to innermost grouping, with level zero corresponding to the population predictions. Defaults to the highest or innermost level of grouping.
asList	an optional logical value. If TRUE and a single value is given in level, the returned object is a list with the predictions split by groups; else the returned value is either a vector or a data frame, according to the length of level.
na.action	a function that indicates what should happen when newdata contains NAs. The default action (na.fail) causes the function to print an error message and terminate if there are any incomplete observations.
naPattern	an expression or formula object, specifying which returned values are to be regarded as missing.
...	some methods for this generic require additional arguments. None are used in this method.

Value

if a single level of grouping is specified in `level`, the returned value is either a list with the predictions split by groups (`asList = TRUE`) or a vector with the predictions (`asList = FALSE`); else, when multiple grouping levels are specified in `level`, the returned object is a data frame with columns given by the predictions at different levels and the grouping factors.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`nlme`, `fitted.lme`

Examples

```
fm1 <- nlme(height ~ SSasymp(age, Asym, R0, lrc),
            data = Loblolly,
            fixed = Asym + R0 + lrc ~ 1,
            random = Asym ~ 1,
            start = c(Asym = 103, R0 = -8.5, lrc = -3.3))
newLoblolly <- data.frame(age = c(5,10,15,20,25,30),
                          Seed = rep(301,6))
predict(fm1, newLoblolly, level = 0:1)
```

`print.summary.pdMat` *Print a summary.pdMat Object*

Description

The standard deviations and correlations associated with the positive-definite matrix represented by `object` (considered as a variance-covariance matrix) are printed, together with the formula and the grouping level associated `object`, if any are present.

Usage

```
## S3 method for class 'summary.pdMat':
print(x, sigma, rdig, Level, resid, ...)
```

Arguments

`x`	an object inheriting from class `summary.pdMat`, generally resulting from applying `summary` to an object inheriting from class `pdMat`.
`sigma`	an optional numeric value used as a multiplier for the square-root factor of the positive-definite matrix represented by `object` (usually the estimated within-group standard deviation from a mixed-effects model). Defaults to 1.
`rdig`	an optional integer value with the number of significant digits to be used in printing correlations. Defaults to 3.

Level	an optional character string with a description of the grouping level associated with object (generally corresponding to levels of grouping in a mixed-effects model). Defaults to NULL.
resid	an optional logical value. If TRUE an extra row with the "residual" standard deviation given in sigma will be included in the output. Defaults to FALSE.
...	optional arguments passed to print.default; see the documentation on that method function.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

summary.pdMat,pdMat

Examples

```
pd1 <- pdCompSymm(3 * diag(2) + 1, form = ~age + age^2,
         data = Orthodont)
print(summary(pd1), sigma = 1.2, resid = TRUE)
```

---

print.varFunc *Print a varFunc Object*

Description

The class and the coefficients associated with x are printed.

Usage

```
## S3 method for class 'varFunc':
print(x, ...)
```

Arguments

x	an object inheriting from class varFunc, representing a variance function structure.
...	optional arguments passed to print.default; see the documentation on that method function.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

summary.varFunc

Examples

```
vf1 <- varPower(0.3, form = ~age)
vf1 <- Initialize(vf1, Orthodont)
print(vf1)
```

`qqnorm.gls` *Normal Plot of Residuals from a gls Object*

Description

Diagnostic plots for assessing the normality of residuals the generalized least squares fit are obtained. The `form` argument gives considerable flexibility in the type of plot specification. A conditioning expression (on the right side of a `|` operator) always implies that different panels are used for each level of the conditioning factor, according to a Trellis display.

Usage

```
## S3 method for class 'gls':
qqnorm(y, form, abline, id, idLabels, grid, ...)
```

Arguments

`y`	an object inheriting from class `gls`, representing a generalized least squares fitted model.
`form`	an optional one-sided formula specifying the desired type of plot. Any variable present in the original data frame used to obtain `y` can be referenced. In addition, `y` itself can be referenced in the formula using the symbol `"."`. Conditional expressions on the right of a `|` operator can be used to define separate panels in a Trellis display. The expression on the right hand side of `form` and to the left of a `|` operator must evaluate to a residuals vector. Default is `~ resid(., type = "p")`, corresponding to a normal plot of the standardized residuals.
`abline`	an optional numeric value, or numeric vector of length two. If given as a single value, a horizontal line will be added to the plot at that coordinate; else, if given as a vector, its values are used as the intercept and slope for a line added to the plot. If missing, no lines are added to the plot.
`id`	an optional numeric value, or one-sided formula. If given as a value, it is used as a significance level for a two-sided outlier test for the standardized residuals (random effects). Observations with absolute standardized residuals (random effects) greater than the $1 - value/2$ quantile of the standard normal distribution are identified in the plot using `idLabels`. If given as a one-sided formula, its right hand side must evaluate to a logical, integer, or character vector which is used to identify observations in the plot. If missing, no observations are identified.
`idLabels`	an optional vector, or one-sided formula. If given as a vector, it is converted to character and used to label the observations identified according to `id`. If given as a one-sided formula, its right hand side must evaluate to a vector which is converted to character and used to label the identified observations. Default is the innermost grouping factor.
`grid`	an optional logical value indicating whether a grid should be added to plot. Default depends on the type of Trellis plot used: if `xyplot` defaults to `TRUE`, else defaults to `FALSE`.
`...`	optional arguments passed to the Trellis plot function.

Value

a diagnostic Trellis plot for assessing normality of residuals.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

gls, plot.gls

Examples

```
fm1 <- gls(follicles ~ sin(2*pi*Time) + cos(2*pi*Time), Ovary,
           correlation = corAR1(form = ~ 1 | Mare))
qqnorm(fm1, abline = c(0,1))
```

---

qqnorm.lme *Normal Plot of Residuals or Random Effects from an lme Object*

Description

Diagnostic plots for assessing the normality of residuals and random effects in the linear mixed-effects fit are obtained. The `form` argument gives considerable flexibility in the type of plot specification. A conditioning expression (on the right side of a `|` operator) always implies that different panels are used for each level of the conditioning factor, according to a Trellis display.

Usage

```
## S3 method for class 'lme':
qqnorm(y, form, abline, id, idLabels, grid, ...)
```

Arguments

`y`	an object inheriting from class `lme`, representing a fitted linear mixed-effects model or from class `lmList`, representing a list of `lm` objects, or from class `lm`, representing a fitted linear model, or from class `nls`, representing a nonlinear least squares fitted model.
`form`	an optional one-sided formula specifying the desired type of plot. Any variable present in the original data frame used to obtain `y` can be referenced. In addition, `y` itself can be referenced in the formula using the symbol `"."`. Conditional expressions on the right of a `|` operator can be used to define separate panels in a Trellis display. The expression on the right hand side of `form` and to the left of a `|` operator must evaluate to a residuals vector, or a random effects matrix. Default is `~ resid(., type = "p")`, corresponding to a normal plot of the standardized residuals evaluated at the innermost level of nesting.
`abline`	an optional numeric value, or numeric vector of length two. If given as a single value, a horizontal line will be added to the plot at that coordinate; else, if given as a vector, its values are used as the intercept and slope for a line added to the plot. If missing, no lines are added to the plot.

`id`	an optional numeric value, or one-sided formula. If given as a value, it is used as a significance level for a two-sided outlier test for the standardized residuals (random effects). Observations with absolute standardized residuals (random effects) greater than the $1 - value/2$ quantile of the standard normal distribution are identified in the plot using `idLabels`. If given as a one-sided formula, its right hand side must evaluate to a logical, integer, or character vector which is used to identify observations in the plot. If missing, no observations are identified.
`idLabels`	an optional vector, or one-sided formula. If given as a vector, it is converted to character and used to label the observations identified according to `id`. If given as a one-sided formula, its right hand side must evaluate to a vector which is converted to character and used to label the identified observations. Default is the innermost grouping factor.
`grid`	an optional logical value indicating whether a grid should be added to plot. Default is `FALSE`.
`...`	optional arguments passed to the Trellis plot function.

Value

a diagnostic Trellis plot for assessing normality of residuals or random effects.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`lme`, `plot.lme`

Examples

```
## Not run:
fm1 <- lme(distance ~ age, Orthodont, random = ~ age | Subject)
# normal plot of standardized residuals by gender
qqnorm(fm1, ~ resid(., type = "p") | Sex, abline = c(0, 1))
# normal plots of random effects
qqnorm(fm1, ~ranef(.))
## End(Not run)
```

`Quinidine` *Quinidine Kinetics*

Description

The `Quinidine` data frame has 1471 rows and 14 columns.

Format

This data frame contains the following columns:

Subject a factor identifying the patient on whom the data were collected.

time a numeric vector giving the time (hr) at which the drug was administered or the blood sample drawn. This is measured from the time the patient entered the study.

conc a numeric vector giving the serum quinidine concentration (mg/L).

dose a numeric vector giving the dose of drug administered (mg). Although there were two different forms of quinidine administered, the doses were adjusted for differences in salt content by conversion to milligrams of quinidine base.

interval a numeric vector giving the when the drug has been given at regular intervals for a sufficiently long period of time to assume steady state behavior, the interval is recorded.

Age a numeric vector giving the age of the subject on entry to the study (yr).

Height a numeric vector giving the height of the subject on entry to the study (in.).

Weight a numeric vector giving the body weight of the subject (kg).

Race a factor with levels `Caucasian`, `Latin`, and `Black` identifying the race of the subject.

Smoke a factor with levels `no` and `yes` giving smoking status at the time of the measurement.

Ethanol a factor with levels `none`, `current`, `former` giving ethanol (alcohol) abuse status at the time of the measurement.

Heart a factor with levels `No/Mild`, `Moderate`, and `Severe` indicating congestive heart failure for the subject.

Creatinine an ordered factor with levels `< 50` < `>= 50` indicating the creatine clearance (mg/min).

glyco a numeric vector giving the alpha-1 acid glycoprotein concentration (mg/dL). Often measured at the same time as the quinidine concentration.

Details

Verme et al. (1992) analyze routine clinical data on patients receiving the drug quinidine as a treatment for cardiac arrythmia (atrial fibrillation of ventricular arrythmias). All patients were receiving oral quinidine doses. At irregular intervals blood samples were drawn and serum concentrations of quinidine were determined. These data are analyzed in several publications, including Davidian and Giltinan (1995, section 9.3).

Source

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York. (Appendix A.25)

Davidian, M. and Giltinan, D. M. (1995), *Nonlinear Models for Repeated Measurement Data*, Chapman and Hall, London.

Verme, C. N., Ludden, T. M., Clementi, W. A. and Harris, S. C. (1992), Pharmacokinetics of quinidine in male patients: A population analysis, *Clinical Pharmacokinetics*, **22**, 468-480.

quinModel *Model function for the Quinidine data*

Description

A model function for a model used with the `Quinidine` data. This function calls compiled C code.

Usage

```
quinModel(Subject, time, conc, dose, interval, lV, lKa, lCl)
```

Arguments

`Subject`	a factor identifying the patient on whom the data were collected.
`time`	a numeric vector giving the time (hr) at which the drug was administered or the blood sample drawn. This is measured from the time the patient entered the study.
`conc`	a numeric vector giving the serum quinidine concentration (mg/L).
`dose`	a numeric vector giving the dose of drug administered (mg). Although there were two different forms of quinidine administered, the doses were adjusted for differences in salt content by conversion to milligrams of quinidine base.
`interval`	a numeric vector giving the when the drug has been given at regular intervals for a sufficiently long period of time to assume steady state behavior, the interval is recorded.
`lV`	numeric. A vector of values of the natural log of the effective volume of distribution according to `Subject` and `time`.
`lKa`	numeric. A vector of values of the natural log of the absorption rate constant according to `Subject` and `time`.
`lCl`	numeric. A vector of values of the natural log of the clearance parameter according to `Subject` and `time`.

Details

See the details section of `Quinidine` for a description of the model function that `quinModel` evaluates.

Value

a numeric vector of predicted quinidine concentrations.

Author(s)

Jose Pinheiro ⟨jose.pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J. C. and Bates, D. M. (2000) *Mixed-effects Models in S and S-PLUS*, Springer. (section 8.2)

Examples

`Rail` *Evaluation of Stress in Railway Rails*

Description

The `Rail` data frame has 18 rows and 2 columns.

Format

This data frame contains the following columns:

Rail an ordered factor identifying the rail on which the measurement was made.

travel a numeric vector giving the travel time for ultrasonic head-waves in the rail (nanoseconds). The value given is the original travel time minus 36,100 nanoseconds.

Details

Devore (2000, Example 10.10, p. 427) cites data from an article in *Materials Evaluation* on "a study of travel time for a certain type of wave that results from longitudinal stress of rails used for railroad track."

Source

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York. (Appendix A.26)

Devore, J. L. (2000), *Probability and Statistics for Engineering and the Sciences (5th ed)*, Duxbury, Boston, MA.

`random.effects` *Extract Random Effects*

Description

This function is generic; method functions can be written to handle specific classes of objects. Classes which already have methods for this function include `lmList` and `lme`.

Usage

```
random.effects(object, ...)
ranef(object, ...)
```

Arguments

`object`	any fitted model object from which random effects estimates can be extracted.
`...`	some methods for this generic function require additional arguments.

Value

will depend on the method function used; see the appropriate documentation.

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer, esp. pp. 100, 461.

See Also

ranef.lmList,ranef.lme

Examples

```
## see the method function documentation
```

---

ranef.lme *Extract lme Random Effects*

Description

The estimated random effects at level i are represented as a data frame with rows given by the different groups at that level and columns given by the random effects. If a single level of grouping is specified, the returned object is a data frame; else, the returned object is a list of such data frames. Optionally, the returned data frame(s) may be augmented with covariates summarized over groups.

Usage

```
## S3 method for class 'lme':
ranef(object, augFrame, level, data, which, FUN,
      standard, omitGroupingFactor, subset, ...)
```

Arguments

object	an object inheriting from class lme, representing a fitted linear mixed-effects model.
augFrame	an optional logical value. If TRUE, the returned data frame is augmented with variables defined in data; else, if FALSE, only the coefficients are returned. Defaults to FALSE.
level	an optional vector of positive integers giving the levels of grouping to be used in extracting the random effects from an object with multiple nested grouping levels. Defaults to all levels of grouping.
data	an optional data frame with the variables to be used for augmenting the returned data frame when augFrame = TRUE. Defaults to the data frame used to fit object.
which	an optional positive integer vector specifying which columns of data should be used in the augmentation of the returned data frame. Defaults to all columns in data.

FUN	an optional summary function or a list of summary functions to be applied to group-varying variables, when collapsing data by groups. Group-invariant variables are always summarized by the unique value that they assume within that group. If FUN is a single function it will be applied to each non-invariant variable by group to produce the summary for that variable. If FUN is a list of functions, the names in the list should designate classes of variables in the frame such as ordered, factor, or numeric. The indicated function will be applied to any group-varying variables of that class. The default functions to be used are mean for numeric factors, and Mode for both factor and ordered. The Mode function, defined internally in gsummary, returns the modal or most popular value of the variable. It is different from the mode function that returns the S-language mode of the variable.
standard	an optional logical value indicating whether the estimated random effects should be "standardized" (i.e. divided by the estimate of the standard deviation of that group of random effects). Defaults to FALSE.
omitGroupingFactor	an optional logical value. When TRUE the grouping factor itself will be omitted from the group-wise summary of data but the levels of the grouping factor will continue to be used as the row names for the returned data frame. Defaults to FALSE.
subset	an optional expression indicating for which rows the random effects should be extracted.
...	some methods for this generic require additional arguments. None are used in this method.

Value

a data frame, or list of data frames, with the estimated random effects at the grouping level(s) specified in level and, optionally, other covariates summarized over groups. The returned object inherits from classes random.effects.lme and data.frame.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer, esp. pp. 100, 461.

See Also

coef.lme, gsummary, lme, plot.ranef.lme, random.effects

Examples

```
fm1 <- lme(distance ~ age, Orthodont, random = ~ age | Subject)
ranef(fm1)
random.effects(fm1)              # same as above
random.effects(fm1, augFrame = TRUE)
```

`ranef.lmList` *Extract lmList Random Effects*

Description

The difference between the individual `lm` components coefficients and their average is calculated.

Usage

```
## S3 method for class 'lmList':
ranef(object, augFrame, data, which, FUN, standard,
                  omitGroupingFactor, ...)
```

Arguments

`object`	an object inheriting from class `lmList`, representing a list of `lm` objects with a common model.
`augFrame`	an optional logical value. If `TRUE`, the returned data frame is augmented with variables defined in `data`; else, if `FALSE`, only the coefficients are returned. Defaults to `FALSE`.
`data`	an optional data frame with the variables to be used for augmenting the returned data frame when `augFrame = TRUE`. Defaults to the data frame used to fit `object`.
`which`	an optional positive integer vector specifying which columns of `data` should be used in the augmentation of the returned data frame. Defaults to all columns in `data`.
`FUN`	an optional summary function or a list of summary functions to be applied to group-varying variables, when collapsing `data` by groups. Group-invariant variables are always summarized by the unique value that they assume within that group. If `FUN` is a single function it will be applied to each non-invariant variable by group to produce the summary for that variable. If `FUN` is a list of functions, the names in the list should designate classes of variables in the frame such as `ordered`, `factor`, or `numeric`. The indicated function will be applied to any group-varying variables of that class. The default functions to be used are `mean` for numeric factors, and `Mode` for both `factor` and `ordered`. The `Mode` function, defined internally in `gsummary`, returns the modal or most popular value of the variable. It is different from the `mode` function that returns the S-language mode of the variable.
`standard`	an optional logical value indicating whether the estimated random effects should be "standardized" (i.e. divided by the corresponding estimated standard error). Defaults to `FALSE`.
`omitGroupingFactor`	an optional logical value. When `TRUE` the grouping factor itself will be omitted from the group-wise summary of `data` but the levels of the grouping factor will continue to be used as the row names for the returned data frame. Defaults to `FALSE`.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

a vector with the differences between the individual `lm` coefficients in `object` and their average.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer, esp. pp. 100, 461.

See Also

`fixed.effects.lmList`, `lmList`, `random.effects`

Examples

```
fm1 <- lmList(distance ~ age | Subject, Orthodont)
ranef(fm1)
random.effects(fm1)              # same as above
```

`RatPupWeight` *The weight of rat pups*

Description

The `RatPupWeight` data frame has 322 rows and 5 columns.

Format

This data frame contains the following columns:

weight a numeric vector

sex a factor with levels `Male` `Female`

Litter an ordered factor with levels `9` < `8` < `7` < `4` < `2` < `10` < `1` < `3` < `5` < `6` < `21` < `22` < `24` < `27` < `26` < `25` < `23` < `17` < `11` < `14` < `13` < `15` < `16` < `20` < `19` < `18` < `12`

Lsize a numeric vector

Treatment an ordered factor with levels `Control` < `Low` < `High`

Details

Source

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York.

recalc *Recalculate Condensed Linear Model Object*

Description

This function is generic; method functions can be written to handle specific classes of objects. Classes which already have methods for this function include: `corStruct`, `modelStruct`, `reStruct`, and `varFunc`.

Usage

```
recalc(object, conLin, ...)
```

Arguments

`object`	any object which induces a recalculation of the condensed linear model object `conLin`.
`conLin`	a condensed linear model object, consisting of a list with components `"Xy"`, corresponding to a regression matrix (`X`) combined with a response vector (`y`), and `"logLik"`, corresponding to the log-likelihood of the underlying model.
`...`	some methods for this generic can take additional arguments.

Value

the recalculated condensed linear model object.

Note

This function is only used inside model fitting functions, such as `lme` and `gls`, that require recalculation of a condensed linear model object.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`recalc.corStruct`, `recalc.modelStruct`, `recalc.reStruct`, `recalc.varFunc`

Examples

```
## see the method function documentation
```

`recalc.corStruct` *Recalculate for corStruct Object*

Description

This method function pre-multiples the `"Xy"` component of `conLin` by the transpose square-root factor(s) of the correlation matrix (matrices) associated with `object` and adds the log-likelihood contribution of `object`, given by `logLik(object)`, to the `"logLik"` component of `conLin`.

Usage

```
## S3 method for class 'corStruct':
recalc(object, conLin, ...)
```

Arguments

`object`	an object inheriting from class `corStruct`, representing a correlation structure.
`conLin`	a condensed linear model object, consisting of a list with components `"Xy"`, corresponding to a regression matrix (`X`) combined with a response vector (`y`), and `"logLik"`, corresponding to the log-likelihood of the underlying model.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

the recalculated condensed linear model object.

Note

This method function is only used inside model fitting functions, such as `lme` and `gls`, that allow correlated error terms.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`corFactor`, `logLik.corStruct`

`recalc.modelStruct` *Recalculate for a modelStruct Object*

Description

This method function recalculates the condensed linear model object using each element of `object` sequentially from last to first.

Usage

```
## S3 method for class 'modelStruct':
recalc(object, conLin, ...)
```

Arguments

`object`	an object inheriting from class `modelStruct`, representing a list of model components, such as `corStruct` and `varFunc` objects.
`conLin`	an optional condensed linear model object, consisting of a list with components `"Xy"`, corresponding to a regression matrix (`X`) combined with a response vector (`y`), and `"logLik"`, corresponding to the log-likelihood of the underlying model. Defaults to `attr(object, "conLin")`.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

the recalculated condensed linear model object.

Note

This method function is generally only used inside model fitting functions, such as `lme` and `gls`, that allow model components, such as correlated error terms and variance functions.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`recalc.corStruct`, `recalc.reStruct`, `recalc.varFunc`

`recalc.reStruct` *Recalculate for an reStruct Object*

Description

The log-likelihood, or restricted log-likelihood, of the Gaussian linear mixed-effects model represented by `object` and `conLin` (assuming spherical within-group covariance structure), evaluated at `coef(object)` is calculated and added to the `logLik` component of `conLin`. The `settings` attribute of `object` determines whether the log-likelihood, or the restricted log-likelihood, is to be calculated. The computational methods for the (restricted) log-likelihood calculations are described in Bates and Pinheiro (1998).

Usage

```
## S3 method for class 'reStruct':
recalc(object, conLin, ...)
```

Arguments

`object`	an object inheriting from class `reStruct`, representing a random effects structure and consisting of a list of `pdMat` objects.
`conLin`	a condensed linear model object, consisting of a list with components `"Xy"`, corresponding to a regression matrix (`X`) combined with a response vector (`y`), and `"logLik"`, corresponding to the log-likelihood of the underlying model.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

the condensed linear model with its `logLik` component updated.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Bates, D.M. and Pinheiro, J.C. (1998) "Computational methods for multilevel models" available in PostScript or PDF formats at http://franz.stat.wisc.edu/pub/NLME/

See Also

`logLik`, `lme`, `recalc`, `reStruct`

`recalc.varFunc` *Recalculate for varFunc Object*

Description

This method function pre-multiples the `"Xy"` component of `conLin` by a diagonal matrix with diagonal elements given by the weights corresponding to the variance structure represented by `object`e and adds the log-likelihood contribution of `object`, given by `logLik(object)`, to the `"logLik"` component of `conLin`.

Usage

```
## S3 method for class 'varFunc':
recalc(object, conLin, ...)
```

Arguments

`object`	an object inheriting from class `varFunc`, representing a variance function structure.
`conLin`	a condensed linear model object, consisting of a list with components `"Xy"`, corresponding to a regression matrix (`X`) combined with a response vector (`y`), and `"logLik"`, corresponding to the log-likelihood of the underlying model.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

the recalculated condensed linear model object.

Note

This method function is only used inside model fitting functions, such as `lme` and `gls`, that allow heteroscedastic error terms.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`recalc`, `varWeights`, `logLik.varFunc`

Relaxin *Assay for Relaxin*

Description

The `Relaxin` data frame has 198 rows and 3 columns.

Format

This data frame contains the following columns:

Run an ordered factor with levels `5` < `8` < `9` < `3` < `4` < `2` < `7` < `1` < `6`

conc a numeric vector

cAMP a numeric vector

Details

Source

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York.

Remifentanil *Pharmacokinetics of remifentanil*

Description

The `Remifentanil` data frame has 2107 rows and 12 columns.

Format

This data frame contains the following columns:

ID a numeric vector

Subject an ordered factor

Time a numeric vector

conc a numeric vector

Rate a numeric vector

Amt a numeric vector

Age a numeric vector

Sex a factor with levels `Female` `Male`

Ht a numeric vector

Wt a numeric vector

BSA a numeric vector

LBM a numeric vector

Details

Source

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York.

`residuals.gls` *Extract gls Residuals*

Description

The residuals for the linear model represented by `object` are extracted.

Usage

```
## S3 method for class 'gls':
residuals(object, type, ...)
```

Arguments

`object`	an object inheriting from class `gls`, representing a generalized least squares fitted linear model, or from class `gnls`, representing a generalized nonlinear least squares fitted linear model.
`type`	an optional character string specifying the type of residuals to be used. If `"response"`, the "raw" residuals (observed - fitted) are used; else, if `"pearson"`, the standardized residuals (raw residuals divided by the corresponding standard errors) are used; else, if `"normalized"`, the normalized residuals (standardized residuals pre-multiplied by the inverse square-root factor of the estimated error correlation matrix) are used. Partial matching of arguments is used, so only the first character needs to be provided. Defaults to `"pearson"`.
`...`	some methods for this generic function require additional arguments. None are used in this method.

Value

a vector with the residuals for the linear model represented by `object`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`gls`

Examples

```
fm1 <- gls(follicles ~ sin(2*pi*Time) + cos(2*pi*Time), Ovary,
           correlation = corAR1(form = ~ 1 | Mare))
residuals(fm1)
```

`residuals.glsStruct` *Calculate glsStruct Residuals*

Description

The residuals for the linear model represented by `object` are extracted.

Usage

```
## S3 method for class 'glsStruct':
residuals(object, glsFit, ...)
```

Arguments

`object`	an object inheriting from class `glsStruct`, representing a list of linear model components, such as `corStruct` and `varFunc` objects.
`glsFit`	an optional list with components `logLik` (log-likelihood), `beta` (coefficients), `sigma` (standard deviation for error term), `varBeta` (coefficients' covariance matrix), `fitted` (fitted values), and `residuals` (residuals). Defaults to `attr(object, "glsFit")`.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

a vector with the residuals for the linear model represented by `object`.

Note

This method function is primarily used inside `gls` and `residuals.gls`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`gls`, `glsStruct`, `residuals.gls`, `fitted.glsStruct`

residuals.gnlsStruct *Calculate gnlsStruct Residuals*

Description

The residuals for the nonlinear model represented by `object` are extracted.

Usage

```
## S3 method for class 'gnlsStruct':
residuals(object, ...)
```

Arguments

`object`	an object inheriting from class `gnlsStruct`, representing a list of model components, such as `corStruct` and `varFunc` objects, and attributes specifying the underlying nonlinear model and the response variable.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

a vector with the residuals for the nonlinear model represented by `object`.

Note

This method function is primarily used inside `gnls` and `residuals.gnls`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`gnls`, `residuals.gnls`, `fitted.gnlsStruct`

residuals.lme *Extract lme Residuals*

Description

The residuals at level i are obtained by subtracting the fitted levels at that level from the response vector (and dividing by the estimated within-group standard error, if `type="pearson"`). The fitted values at level i are obtained by adding together the population fitted values (based only on the fixed effects estimates) and the estimated contributions of the random effects to the fitted values at grouping levels less or equal to i.

Usage

```
## S3 method for class 'lme':
residuals(object, level, type, asList, ...)
```

Arguments

object	an object inheriting from class lme, representing a fitted linear mixed-effects model.
level	an optional integer vector giving the level(s) of grouping to be used in extracting the residuals from object. Level values increase from outermost to innermost grouping, with level zero corresponding to the population residuals. Defaults to the highest or innermost level of grouping.
type	an optional character string specifying the type of residuals to be used. If "response", the "raw" residuals (observed - fitted) are used; else, if "pearson", the standardized residuals (raw residuals divided by the corresponding standard errors) are used; else, if "normalized", the normalized residuals (standardized residuals pre-multiplied by the inverse square-root factor of the estimated error correlation matrix) are used. Partial matching of arguments is used, so only the first character needs to be provided. Defaults to "pearson".
asList	an optional logical value. If TRUE and a single value is given in level, the returned object is a list with the residuals split by groups; else the returned value is either a vector or a data frame, according to the length of level. Defaults to FALSE.
...	some methods for this generic require additional arguments. None are used in this method.

Value

if a single level of grouping is specified in level, the returned value is either a list with the residuals split by groups (asList = TRUE) or a vector with the residuals (asList = FALSE); else, when multiple grouping levels are specified in level, the returned object is a data frame with columns given by the residuals at different levels and the grouping factors.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Bates, D.M. and Pinheiro, J.C. (1998) "Computational methods for multilevel models" available in PostScript or PDF formats at http://franz.stat.wisc.edu/pub/NLME/

See Also

lme, fitted.lme

Examples

```
fm1 <- lme(distance ~ age + Sex, data = Orthodont, random = ~ 1)
residuals(fm1, level = 0:1)
```

`residuals.lmeStruct` *Calculate lmeStruct Residuals*

Description

The residuals at level i are obtained by subtracting the fitted values at that level from the response vector. The fitted values at level i are obtained by adding together the population fitted values (based only on the fixed effects estimates) and the estimated contributions of the random effects to the fitted values at grouping levels less or equal to i.

Usage

```
## S3 method for class 'lmeStruct':
residuals(object, level, conLin, lmeFit, ...)
```

Arguments

`object`	an object inheriting from class `lmeStruct`, representing a list of linear mixed-effects model components, such as `reStruct`, `corStruct`, and `varFunc` objects.
`level`	an optional integer vector giving the level(s) of grouping to be used in extracting the residuals from `object`. Level values increase from outermost to innermost grouping, with level zero corresponding to the population fitted values. Defaults to the highest or innermost level of grouping.
`conLin`	an optional condensed linear model object, consisting of a list with components `"Xy"`, corresponding to a regression matrix (`X`) combined with a response vector (`y`), and `"logLik"`, corresponding to the log-likelihood of the underlying lme model. Defaults to `attr(object, "conLin")`.
`lmeFit`	an optional list with components `beta` and `b` containing respectively the fixed effects estimates and the random effects estimates to be used to calculate the residuals. Defaults to `attr(object, "lmeFit")`.
`...`	some methods for this generic accept optional arguments.

Value

if a single level of grouping is specified in `level`, the returned value is a vector with the residuals at the desired level; else, when multiple grouping levels are specified in `level`, the returned object is a matrix with columns given by the residuals at different levels.

Note

This method function is primarily used within the `lme` function.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Bates, D.M. and Pinheiro, J.C. (1998) "Computational methods for multilevel models" available in PostScript or PDF formats at http://franz.stat.wisc.edu/pub/NLME/

See Also

`lme`, `residuals.lme`, `fitted.lmeStruct`

`residuals.lmList` *Extract lmList Residuals*

Description

The residuals are extracted from each `lm` component of `object` and arranged into a list with as many components as `object`, or combined into a single vector.

Usage

```
## S3 method for class 'lmList':
residuals(object, type, subset, asList, ...)
```

Arguments

`object`	an object inheriting from class `lmList`, representing a list of `lm` objects with a common model.
`subset`	an optional character or integer vector naming the `lm` components of `object` from which the residuals are to be extracted. Default is `NULL`, in which case all components are used.
`type`	an optional character string specifying the type of residuals to be extracted. Options include `"response"` for the "raw" residuals (observed - fitted), `"pearson"` for the standardized residuals (raw residuals divided by the estimated residual standard error) using different standard errors for each `lm` fit, and `"pooled.pearson"` for the standardized residuals using a pooled estimate of the residual standard error. Partial matching of arguments is used, so only the first character needs to be provided. Defaults to `"response"`.
`asList`	an optional logical value. If `TRUE`, the returned object is a list with the residuals split by groups; else the returned value is a vector. Defaults to `FALSE`.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

a list with components given by the residuals of each `lm` component of `object`, or a vector with the residuals for all `lm` components of `object`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

lmList, fitted.lmList

Examples

```
fm1 <- lmList(distance ~ age | Subject, Orthodont)
residuals(fm1)
```

`residuals.nlmeStruct` *Calculate nlmeStruct Residuals*

Description

The residuals at level i are obtained by subtracting the fitted values at that level from the response vector. The fitted values at level i are obtained by adding together the contributions from the estimated fixed effects and the estimated random effects at levels less or equal to i and evaluating the model function at the resulting estimated parameters.

Usage

```
## S3 method for class 'nlmeStruct':
residuals(object, level, conLin, ...)
```

Arguments

`object`	an object inheriting from class `nlmeStruct`, representing a list of mixed-effects model components, such as `reStruct`, `corStruct`, and `varFunc` objects.
`level`	an optional integer vector giving the level(s) of grouping to be used in extracting the residuals from `object`. Level values increase from outermost to innermost grouping, with level zero corresponding to the population fitted values. Defaults to the highest or innermost level of grouping.
`conLin`	an optional condensed linear model object, consisting of a list with components `"Xy"`, corresponding to a regression matrix (`X`) combined with a response vector (`y`), and `"logLik"`, corresponding to the log-likelihood of the underlying nlme model. Defaults to `attr(object, "conLin")`.
`...`	optional arguments to the residuals generic. Not used.

Value

if a single level of grouping is specified in `level`, the returned value is a vector with the residuals at the desired level; else, when multiple grouping levels are specified in `level`, the returned object is a matrix with columns given by the residuals at different levels.

Note

This method function is primarily used within the `nlme` function.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Bates, D.M. and Pinheiro, J.C. (1998) "Computational methods for multilevel models" available in PostScript or PDF formats at http://nlme.stat.wisc.edu

See Also

nlme, fitted.nlmeStruct

---

reStruct *Random Effects Structure*

Description

This function is a constructor for the reStruct class, representing a random effects structure and consisting of a list of pdMat objects, plus a settings attribute containing information for the optimization algorithm used to fit the associated mixed-effects model.

Usage

```
reStruct(object, pdClass, REML, data)
## S3 method for class 'reStruct':
print(x, sigma, reEstimates, verbose, ...)
```

Arguments

object	any of the following: (i) a one-sided formula of the form ~x1+...+xn | g1/.../gm, with x1+...+xn specifying the model for the random effects and g1/.../gm the grouping structure (m may be equal to 1, in which case no / is required). The random effects formula will be repeated for all levels of grouping, in the case of multiple levels of grouping; (ii) a list of one-sided formulas of the form ~x1+...+xn | g, with possibly different random effects models for each grouping level. The order of nesting will be assumed the same as the order of the elements in the list; (iii) a one-sided formula of the form ~x1+...+xn, or a pdMat object with a formula (i.e. a non-NULL value for formula(object)), or a list of such formulas or pdMat objects. In this case, the grouping structure formula will be derived from the data used to to fit the mixed-effects model, which should inherit from class groupedData; (iv) a named list of formulas or pdMat objects as in (iii), with the grouping factors as names. The order of nesting will be assumed the same as the order of the order of the elements in the list; (v) an reStruct object.
pdClass	an optional character string with the name of the pdMat class to be used for the formulas in object. Defaults to "pdSymm" which corresponds to a general positive-definite matrix.
REML	an optional logical value. If TRUE, the associated mixed-effects model will be fitted using restricted maximum likelihood; else, if FALSE, maximum likelihood will be used. Defaults to FALSE.

data	an optional data frame in which to evaluate the variables used in the random effects formulas in object. It is used to obtain the levels for factors, which affect the dimensions and the row/column names of the underlying pdMat objects. If NULL, no attempt is made to obtain information on factors appearing in the formulas. Defaults to the parent frame from which the function was called.
x	an object inheriting from class reStruct to be printed.
sigma	an optional numeric value used as a multiplier for the square-root factors of the pdMat components (usually the estimated within-group standard deviation from a mixed-effects model). Defaults to 1.
reEstimates	an optional list with the random effects estimates for each level of grouping. Only used when verbose = TRUE.
verbose	an optional logical value determining if the random effects estimates should be printed. Defaults to FALSE.
...	Optional arguments can be given to other methods for this generic. None are used in this method.

Value

an object inheriting from class reStruct, representing a random effects structure.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

groupedData, lme, pdMat, solve.reStruct, summary.reStruct, update.reStruct

Examples

```
rs1 <- reStruct(list(Dog = ~day, Side = ~1), data = Pixel)
rs1
```

---

simulate.lme *Simulate results from lme models*

Description

The model object is fit to the data. Using the fitted values of the parameters, nsim new data vectors from this model are simulated. Both m1 and m2 are fit by maximum likelihood (ML) and/or by restricted maximum likelihood (REML) to each of the simulated data vectors.

Usage

```
simulate.lme(object, nsim, seed, m2, method, niterEM, useGen, ...)
```

Arguments

object	an object inheriting from class lme, representing a fitted linear mixed-effects model, or a list containing an lme model specification. If given as a list, it should contain components fixed, data, and random with values suitable for a call to lme. This argument defines the null model.
m2	an lme object, or a list, like m1 containing a second lme model specification. This argument defines the alternative model. If given as a list, only those parts of the specification that change between model m1 and m2 need to be specified.
seed	an optional integer that is passed to set.seed. Defaults to a random integer.
method	an optional character array. If it includes "REML" the models are fit by maximizing the restricted log-likelihood. If it includes "ML" the log-likelihood is maximized. Defaults to c("REML", "ML"), in which case both methods are used.
nsim	an optional positive integer specifying the number of simulations to perform. Defaults to **1**. **This has changed. Previously the default was 1000.**
niterEM	an optional integer vector of length 2 giving the number of iterations of the EM algorithm to apply when fitting the m1 and m2 to each simulated set of data. Defaults to c(40,200).
useGen	an optional logical value. If TRUE, numerical derivatives are used to obtain the gradient and the Hessian of the log-likelihood in the optimization algorithm in the ms function. If FALSE, the default algorithm in ms for functions that do not incorporate gradient and Hessian attributes is used. Default depends on the pdMat classes used in m1 and m2: if both are standard classes (see pdClasses) then defaults to TRUE, otherwise defaults to FALSE.
...	optional additional arguments. None are used.

Value

an object of class simulate.lme with components null and alt. Each of these has components ML and/or REML which are matrices. An attribute called Random.seed contains the seed that was used for the random number generator.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

lme, set.seed

Examples

```
## Not run:
orthSim <-
   simulate.lme(list(fixed = distance ~ age, data = Orthodont,
                     random = ~ 1 | Subject), nsim = 1000,
                m2 = list(random = ~ age | Subject))
## End(Not run)
```

solve.pdMat *Calculate Inverse of a Positive-Definite Matrix*

Description

The positive-definite matrix represented by `a` is inverted and assigned to `a`.

Usage

```
## S3 method for class 'pdMat':
solve(a, b, ...)
```

Arguments

`a`	an object inheriting from class `pdMat`, representing a positive definite matrix.
`b`	this argument is only included for consistency with the generic function and is not used in this method function.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

a `pdMat` object similar to `a`, but with coefficients corresponding to the inverse of the positive-definite matrix represented by `a`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`pdMat`

Examples

```
pd1 <- pdCompSymm(3 * diag(3) + 1)
solve(pd1)
```

solve.reStruct *Apply Solve to an reStruct Object*

Description

`Solve` is applied to each `pdMat` component of `a`, which results in inverting the positive-definite matrices they represent.

Usage

```
## S3 method for class 'reStruct':
solve(a, b, ...)
```

Arguments

a	an object inheriting from class reStruct, representing a random effects structure and consisting of a list of pdMat objects.
b	this argument is only included for consistency with the generic function and is not used in this method function.
...	some methods for this generic require additional arguments. None are used in this method.

Value

an reStruct object similar to a, but with the pdMat components representing the inverses of the matrices represented by the components of a.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

solve.pdMat, reStruct

Examples

```
rs1 <- reStruct(list(A = pdSymm(diag(1:3), form = ~Score),
  B = pdDiag(2 * diag(4), form = ~Educ)))
solve(rs1)
```

Soybean *Growth of soybean plants*

Description

The Soybean data frame has 412 rows and 5 columns.

Format

This data frame contains the following columns:

Plot a factor giving a unique identifier for each plot.

Variety a factor indicating the variety; Forrest (F) or Plant Introduction #416937 (P).

Year a factor indicating the year the plot was planted.

Time a numeric vector giving the time the sample was taken (days after planting).

weight a numeric vector giving the average leaf weight per plant (g).

Details

These data are described in Davidian and Giltinan (1995, 1.1.3, p.7) as "Data from an experiment to compare growth patterns of two genotypes of soybeans: Plant Introduction #416937 (P), an experimental strain, and Forrest (F), a commercial variety."

Source

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York. (Appendix A.27)

Davidian, M. and Giltinan, D. M. (1995), *Nonlinear Models for Repeated Measurement Data*, Chapman and Hall, London.

Examples

```
summary(fm1 <- nlsList(SSlogis, data = Soybean))
```

---

splitFormula *Split a Formula*

Description

Splits the right hand side of `form` into a list of subformulas according to the presence of `sep`. The left hand side of `form`, if present, will be ignored. The length of the returned list will be equal to the number of occurrences of `sep` in `form` plus one.

Usage

```
splitFormula(form, sep)
```

Arguments

`form`	a `formula` object.
`sep`	an optional character string specifying the separator to be used for splitting the formula. Defaults to `"/"`.

Value

a list of formulas, corresponding to the split of `form` according to `sep`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`formula`

Examples

```
splitFormula(~ g1/g2/g3)
```

Spruce *Growth of Spruce Trees*

Description

The Spruce data frame has 1027 rows and 4 columns.

Format

This data frame contains the following columns:

Tree a factor giving a unique identifier for each tree.

days a numeric vector giving the number of days since the beginning of the experiment.

logSize a numeric vector giving the logarithm of an estimate of the volume of the tree trunk.

plot a factor identifying the plot in which the tree was grown.

Details

Diggle, Liang, and Zeger (1994, Example 1.3, page 5) describe data on the growth of spruce trees that have been exposed to an ozone-rich atmosphere or to a normal atmosphere.

Source

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York. (Appendix A.28)

Diggle, Peter J., Liang, Kung-Yee and Zeger, Scott L. (1994), *Analysis of longitudinal data*, Oxford University Press, Oxford.

summary.corStruct *Summarize a corStruct Object*

Description

This method function prepares object to be printed using the print.summary method, by changing its class and adding a structName attribute to it.

Usage

```
## S3 method for class 'corStruct':
summary(object, structName, ...)
```

Arguments

object	an object inheriting from class corStruct, representing a correlation structure.
structName	an optional character string defining the type of correlation structure associated with object, to be used in the print.summary method. Defaults to class(object)[1].
...	some methods for this generic require additional arguments. None are used in this method.

Value

an object identical to `object`, but with its class changed to `summary.corStruct` and an additional attribute `structName`. The returned value inherits from the same classes as `object`.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

`corClasses`, `corNatural`, `Initialize.corStruct`, `summary`

Examples

```
cs1 <- corAR1(0.2)
summary(cs1)
```

`summary.gls` *Summarize a gls Object*

Description

Additional information about the linear model fit represented by `object` is extracted and included as components of `object`.

Usage

```
## S3 method for class 'gls':
summary(object, verbose, ...)
```

Arguments

`object`	an object inheriting from class `gls`, representing a generalized least squares fitted linear model.
`verbose`	an optional logical value used to control the amount of output when the object is printed. Defaults to `FALSE`.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

an object inheriting from class `summary.gls` with all components included in `object` (see `glsObject` for a full description of the components) plus the following components:

`corBeta`	approximate correlation matrix for the coefficients estimates
`tTable`	a data frame with columns `Value`, `Std. Error`, `t-value`, and `p-value` representing respectively the coefficients estimates, their approximate standard errors, the ratios between the estimates and their standard errors, and the associated p-value under a t approximation. Rows correspond to the different coefficients.

residuals	if more than five observations are used in the gls fit, a vector with the minimum, first quartile, median, third quartile, and maximum of the residuals distribution; else the residuals.
AIC	the Akaike Information Criterion corresponding to object.
BIC	the Bayesian Information Criterion corresponding to object.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

AIC, BIC, gls, summary

Examples

```
fm1 <- gls(follicles ~ sin(2*pi*Time) + cos(2*pi*Time), Ovary,
           correlation = corAR1(form = ~ 1 | Mare))
summary(fm1)
```

---

summary.lme *Summarize an lme Object*

Description

Additional information about the linear mixed-effects fit represented by object is extracted and included as components of object. The returned object is suitable for printing with the print.summary.lme method.

Usage

```
## S3 method for class 'lme':
summary(object, adjustSigma, verbose, ...)
```

Arguments

object	an object inheriting from class lme, representing a fitted linear mixed-effects model.
adjustSigma	an optional logical value. If TRUE and the estimation method used to obtain object was maximum likelihood, the residual standard error is multiplied by $\sqrt{n_{obs}/(n_{obs} - n_{par})}$, converting it to a REML-like estimate. This argument is only used when a single fitted object is passed to the function. Default is TRUE.
verbose	an optional logical value used to control the amount of output in the print.summary.lme method. Defaults to FALSE.
...	some methods for this generic require additional arguments. None are used in this method.

Value

an object inheriting from class `summary.lme` with all components included in `object` (see `lmeObject` for a full description of the components) plus the following components:

`corFixed`	approximate correlation matrix for the fixed effects estimates
`tTable`	a data frame with columns `Value`, `Std. Error`, `DF`, `t-value`, and `p-value` representing respectively the fixed effects estimates, their approximate standard errors, the denominator degrees of freedom, the ratios between the estimates and their standard errors, and the associated p-value from a t distribution. Rows correspond to the different fixed effects.
`residuals`	if more than five observations are used in the `lme` fit, a vector with the minimum, first quartile, median, third quartile, and maximum of the innermost grouping level residuals distribution; else the innermost grouping level residuals.
`AIC`	the Akaike Information Criterion corresponding to `object`.
`BIC`	the Bayesian Information Criterion corresponding to `object`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`AIC`, `BIC`, `lme`, `print.summary.lme`

Examples

```
fm1 <- lme(distance ~ age, Orthodont, random = ~ age | Subject)
summary(fm1)
```

`summary.lmList` *Summarize an lmList Object*

Description

The `summary.lm` method is applied to each `lm` component of `object` to produce summary information on the individual fits, which is organized into a list of summary statistics. The returned object is suitable for printing with the `print.summary.lmList` method.

Usage

```
## S3 method for class 'lmList':
summary(object, pool, ...)
```

Arguments

`object`	an object inheriting from class `lmList`, representing a list of `lm` fitted objects.
`pool`	an optional logical value indicating whether a pooled estimate of the residual standard error should be used. Default is `attr(object, "pool")`.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

a list with summary statistics obtained by applying `summary.lm` to the elements of `object`, inheriting from class `summary.lmList`. The components of `value` are:

`call`	a list containing an image of the `lmList` call that produced `object`.
`coefficients`	a three dimensional array with summary information on the `lm` coefficients. The first dimension corresponds to the names of the `object` components, the second dimension is given by `"Value"`, `"Std. Error"`, `"t value"`, and `"Pr(>|t|)"`, corresponding, respectively, to the coefficient estimates and their associated standard errors, t-values, and p-values. The third dimension is given by the coefficients names.
`correlation`	a three dimensional array with the correlations between the individual `lm` coefficient estimates. The first dimension corresponds to the names of the `object` components. The third dimension is given by the coefficients names. For each coefficient, the rows of the associated array give the correlations between that coefficient and the remaining coefficients, by `lm` component.
`cov.unscaled`	a three dimensional array with the unscaled variances/covariances for the individual `lm` coefficient estimates (giving the estimated variance/covariance for the coefficients, when multiplied by the estimated residual errors). The first dimension corresponds to the names of the `object` components. The third dimension is given by the coefficients names. For each coefficient, the rows of the associated array give the unscaled covariances between that coefficient and the remaining coefficients, by `lm` component.
`df`	an array with the number of degrees of freedom for the model and for residuals, for each `lm` component.
`df.residual`	the total number of degrees of freedom for residuals, corresponding to the sum of residuals df of all `lm` components.
`fstatistics`	an array with the F test statistics and corresponding degrees of freedom, for each `lm` component.
`pool`	the value of the `pool` argument to the function.
`r.squared`	a vector with the multiple R-squared statistics for each `lm` component.
`residuals`	a list with components given by the residuals from individual `lm` fits.
`RSE`	the pooled estimate of the residual standard error.
`sigma`	a vector with the residual standard error estimates for the individual `lm` fits.
`terms`	the terms object used in fitting the individual `lm` components.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`lmList`, `summary`

Examples

```
fm1 <- lmList(distance ~ age | Subject, Orthodont)
summary(fm1)
```

summary.modelStruct *Summarize a modelStruct Object*

Description

This method function applies `summary` to each element of `object`.

Usage

```
## S3 method for class 'modelStruct':
summary(object, ...)
```

Arguments

`object`	an object inheriting from class `modelStruct`, representing a list of model components, such as `reStruct`, `corStruct` and `varFunc` objects.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

a list with elements given by the summarized components of `object`. The returned value is of class `summary.modelStruct`, also inheriting from the same classes as `object`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`reStruct`, `summary`

Examples

```
lms1 <- lmeStruct(reStruct = reStruct(pdDiag(diag(2), ~age)),
   corStruct = corAR1(0.3))
summary(lms1)
```

summary.nlsList *Summarize an nlsList Object*

Description

The `summary` function is applied to each `nls` component of `object` to produce summary information on the individual fits, which is organized into a list of summary statistics. The returned object is suitable for printing with the `print.summary.nlsList` method.

Usage

```
## S3 method for class 'nlsList':
summary(object, ...)
```

Arguments

object	an object inheriting from class nlsList, representing a list of nls fitted objects.
...	optional arguments to the summary.lmList method. One such optional argument is pool, a logical value indicating whether a pooled estimate of the residual standard error should be used. Default is attr(object, "pool").

Value

a list with summary statistics obtained by applying summary to the elements of object, inheriting from class summary.nlsList. The components of value are:

call	a list containing an image of the nlsList call that produced object.
parameters	a three dimensional array with summary information on the nls coefficients. The first dimension corresponds to the names of the object components, the second dimension is given by "Value", "Std. Error", "t value", and "Pr(>|t|)", corresponding, respectively, to the coefficient estimates and their associated standard errors, t-values, and p-values. The third dimension is given by the coefficients names.
correlation	a three dimensional array with the correlations between the individual nls coefficient estimates. The first dimension corresponds to the names of the object components. The third dimension is given by the coefficients names. For each coefficient, the rows of the associated array give the correlations between that coefficient and the remaining coefficients, by nls component.
cov.unscaled	a three dimensional array with the unscaled variances/covariances for the individual lm coefficient estimates (giving the estimated variance/covariance for the coefficients, when multiplied by the estimated residual errors). The first dimension corresponds to the names of the object components. The third dimension is given by the coefficients names. For each coefficient, the rows of the associated array give the unscaled covariances between that coefficient and the remaining coefficients, by nls component.
df	an array with the number of degrees of freedom for the model and for residuals, for each nls component.
df.residual	the total number of degrees of freedom for residuals, corresponding to the sum of residuals df of all nls components.
pool	the value of the pool argument to the function.
RSE	the pooled estimate of the residual standard error.
sigma	a vector with the residual standard error estimates for the individual lm fits.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

nlsList, summary

Examples

```
fm1 <- nlsList(SSasymp, Loblolly)
summary(fm1)
```

`summary.pdMat` *Summarize a pdMat Object*

Description

Attributes `structName` and `noCorrelation`, with the values of the corresponding arguments to the method function, are appended to `object` and its class is changed to `summary.pdMat`.

Usage

```
## S3 method for class 'pdMat':
summary(object, structName, noCorrelation, ...)
```

Arguments

`object`	an object inheriting from class `pdMat`, representing a positive definite matrix.
`structName`	an optional character string with a description of the `pdMat` class. Default depends on the method function: `"Blocked"` for `pdBlocked`, `"Compound Symmetry"` for `pdCompSymm`, `"Diagonal"` for `pdDiag`, `"Multiple of an Identity"` for `pdIdent`, `"General Positive-Definite, Natural Parametrization"` for `pdNatural`, `"General Positive-Definite"` for `pdSymm`, and `data.class(object)` for `pdMat`.
`noCorrelation`	an optional logical value indicating whether correlations are to be printed in `print.summary.pdMat`. Default depends on the method function: `FALSE` for `pdDiag` and `pdIdent`, and `TRUE` for all other classes.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

an object similar to `object`, with additional attributes `structName` and `noCorrelation`, inheriting from class `summary.pdMat`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`print.summary.pdMat`, `pdMat`

Examples

```
summary(pdSymm(diag(4)))
```

summary.varFunc *Summarize varFunc Object*

Description

A structName attribute, with the value of corresponding argument, is appended to object and its class is changed to summary.varFunc.

Usage

```
## S3 method for class 'varFunc':
summary(object, structName, ...)
```

Arguments

object	an object inheriting from class varFunc, representing a variance function structure.
structName	an optional character string with a description of the varFunc class. Default depends on the method function: "Combination of variance functions" for varComb, "Constant plus power of covariate" for varConstPower, "Exponential of variance covariate" for varExp, "Different standard deviations per stratum" for varIdent, "Power of variance covariate" for varPower, and data.class(object) for varFunc.
...	some methods for this generic require additional arguments. None are used in this method.

Value

an object similar to object, with an additional attribute structName, inheriting from class summary.varFunc.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

varClasses, varFunc

Examples

```
vf1 <- varPower(0.3, form = ~age)
vf1 <- Initialize(vf1, Orthodont)
summary(vf1)
```

`Tetracycline1` *Pharmacokinetics of tetracycline*

Description

The `Tetracycline1` data frame has 40 rows and 4 columns.

Format

This data frame contains the following columns:

conc a numeric vector

Time a numeric vector

Subject an ordered factor with levels `5` < `3` < `2` < `4` < `1`

Formulation a factor with levels `tetrachel` `tetracyn`

Details

Source

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York.

`Tetracycline2` *Pharmacokinetics of tetracycline*

Description

The `Tetracycline2` data frame has 40 rows and 4 columns.

Format

This data frame contains the following columns:

conc a numeric vector

Time a numeric vector

Subject an ordered factor with levels `4` < `5` < `2` < `1` < `3`

Formulation a factor with levels `Berkmycin` `tetramycin`

Details

Source

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York.

update.modelStruct *Update a modelStruct Object*

Description

This method function updates each element of `object`, allowing the access to `data`.

Usage

```
## S3 method for class 'modelStruct':
update(object, data, ...)
```

Arguments

`object`	an object inheriting from class `modelStruct`, representing a list of model components, such as `corStruct` and `varFunc` objects.
`data`	a data frame in which to evaluate the variables needed for updating the elements of `object`.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

an object similar to `object` (same class, length, and names), but with updated elements.

Note

This method function is primarily used within model fitting functions, such as `lme` and `gls`, that allow model components such as variance functions.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`reStruct`

update.varFunc *Update varFunc Object*

Description

If the `formula(object)` includes a `"."` term, representing a fitted object, the variance covariate needs to be updated upon completion of an optimization cycle (in which the variance function weights are kept fixed). This method function allows a reevaluation of the variance covariate using the current fitted object and, optionally, other variables in the original data.

Usage

```
## S3 method for class 'varFunc':
update(object, data, ...)
```

Arguments

object	an object inheriting from class varFunc, representing a variance function structure.
data	a list with a component named "." with the current version of the fitted object (from which fitted values, coefficients, and residuals can be extracted) and, if necessary, other variables used to evaluate the variance covariate(s).
...	some methods for this generic require additional arguments. None are used in this method.

Value

if formula(object) includes a "." term, an varFunc object similar to object, but with the variance covariate reevaluated at the current fitted object value; else object is returned unchanged.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

needUpdate, covariate<-.varFunc

---

varClasses *Variance Function Classes*

Description

Standard classes of variance function structures (varFunc) available in the nlme library. Covariates included in the variance function, denoted by variance covariates, may involve functions of the fitted model object, such as the fitted values and the residuals. Different coefficients may be assigned to the levels of a classification factor.

Value

Available standard classes ():

varExp	exponential of a variance covariate.
varPower	power of a variance covariate.
varConstPower	constant plus power of a variance covariate.
varIdent	constant variance(s), generally used to allow different variances according to the levels of a classification factor.
varFixed	fixed weights, determined by a variance covariate.
varComb	combination of variance functions.

Note

Users may define their own varFunc classes by specifying a constructor function and, at a minimum, methods for the functions coef, coef<-, and initialize. For examples of these functions, see the methods for class varPower.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

varComb, varConstPower, varExp, varFixed, varIdent, varPower, summary.varFunc

---

varComb *Combination of Variance Functions*

Description

This function is a constructor for the varComb class, representing a combination of variance functions. The corresponding variance function is equal to the product of the variance functions of the varFunc objects listed in ....

Usage

```
varComb(...)
```

Arguments

...	objects inheriting from class varFunc representing variance function structures.

Value

a varComb object representing a combination of variance functions, also inheriting from class varFunc.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

varClasses, varWeights.varComb, coef.varComb

Examples

```
vf1 <- varComb(varIdent(form = ~1|Sex), varPower())
```

---

varConstPower *Constant Plus Power Variance Function*

Description

This function is a constructor for the varConstPower class, representing a constant plus power variance function structure. Letting v denote the variance covariate and $\sigma^2(v)$ denote the variance function evaluated at v, the constant plus power variance function is defined as $\sigma^2(v) = (\theta_1 + |v|^{\theta}_2)^2$, where θ_1, θ_2 are the variance function coefficients. When a grouping factor is present, different θ_1, θ_2 are used for each factor level.

Usage

```
varConstPower(const, power, form, fixed)
```

Arguments

const, power	optional numeric vectors, or lists of numeric values, with, respectively, the coefficients for the constant and the power terms. Both arguments must have length one, unless a grouping factor is specified in form. If either argument has length greater than one, it must have names which identify its elements to the levels of the grouping factor defined in form. If a grouping factor is present in form and the argument has length one, its value will be assigned to all grouping levels. Only positive values are allowed for const. Default is numeric(0), which results in a vector of zeros of appropriate length being assigned to the coefficients when object is initialized (corresponding to constant variance equal to one).
form	an optional one-sided formula of the form ~ v, or ~ v | g, specifying a variance covariate v and, optionally, a grouping factor g for the coefficients. The variance covariate must evaluate to a numeric vector and may involve expressions using ".", representing a fitted model object from which fitted values (fitted(.)) and residuals (resid(.)) can be extracted (this allows the variance covariate to be updated during the optimization of an object function). When a grouping factor is present in form, a different coefficient value is used for each of its levels. Several grouping variables may be simultaneously specified, separated by the * operator, as in ~ v | g1 * g2 * g3. In this case, the levels of each grouping variable are pasted together and the resulting factor is used to group the observations. Defaults to ~ fitted(.) representing a variance covariate given by the fitted values of a fitted model object and no grouping factor.
fixed	an optional list with components const and/or power, consisting of numeric vectors, or lists of numeric values, specifying the values at which some or all of the coefficients in the variance function should be fixed. If a grouping factor is specified in form, the components of fixed must have names identifying which coefficients are to be fixed. Coefficients included in fixed are not allowed to vary during the optimization of an objective function. Defaults to NULL, corresponding to no fixed coefficients.

Value

a varConstPower object representing a constant plus power variance function structure, also inheriting from class varFunc.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

varClasses, varWeights.varFunc, coef.varConstPower

Examples

```
vf1 <- varConstPower(1.2, 0.2, form = ~age|Sex)
```

VarCorr *Extract variance and correlation components*

Description

This function calculates the estimated variances, standard deviations, and correlations between the random-effects terms in a linear mixed-effects model, of class lme, or a nonlinear mixed-effects model, of class nlme. The within-group error variance and standard deviation are also calculated.

Usage

```
VarCorr(x, sigma, rdig)
```

Arguments

x	a fitted model object, usually an object inheriting from class lme.
sigma	an optional numeric value used as a multiplier for the standard deviations. Default is 1.
rdig	an optional integer value specifying the number of digits used to represent correlation estimates. Default is 3.

Value

a matrix with the estimated variances, standard deviations, and correlations for the random effects. The first two columns, named Variance and StdDev, give, respectively, the variance and the standard deviations. If there are correlation components in the random effects model, the third column, named Corr, and the remaining unnamed columns give the estimated correlations among random effects within the same level of grouping. The within-group error variance and standard deviation are included as the last row in the matrix.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer, esp. pp. 100, 461.

See Also

lme, nlme

Examples

```
fm1 <- lme(distance ~ age, data = Orthodont, random = ~age)
VarCorr(fm1)
```

---

varExp *Exponential Variance Function*

Description

This function is a constructor for the varExp class, representing an exponential variance function structure. Letting v denote the variance covariate and $\sigma^2(v)$ denote the variance function evaluated at v, the exponential variance function is defined as $\sigma^2(v) = \exp(2\theta v)$, where θ is the variance function coefficient. When a grouping factor is present, a different θ is used for each factor level.

Usage

```
varExp(value, form, fixed)
```

Arguments

value	an optional numeric vector, or list of numeric values, with the variance function coefficients. Value must have length one, unless a grouping factor is specified in form. If value has length greater than one, it must have names which identify its elements to the levels of the grouping factor defined in form. If a grouping factor is present in form and value has length one, its value will be assigned to all grouping levels. Default is numeric(0), which results in a vector of zeros of appropriate length being assigned to the coefficients when object is initialized (corresponding to constant variance equal to one).
form	an optional one-sided formula of the form ~ v, or ~ v | g, specifying a variance covariate v and, optionally, a grouping factor g for the coefficients. The variance covariate must evaluate to a numeric vector and may involve expressions using ".", representing a fitted model object from which fitted values (fitted(.)) and residuals (resid(.)) can be extracted (this allows the variance covariate to be updated during the optimization of an object function). When a grouping factor is present in form, a different coefficient value is used for each of its levels. Several grouping variables may be simultaneously specified, separated by the * operator, like in ~ v | g1 * g2 * g3. In this case, the levels of each grouping variable are pasted together and the resulting

factor is used to group the observations. Defaults to ~ fitted(.) representing a variance covariate given by the fitted values of a fitted model object and no grouping factor.

fixed — an optional numeric vector, or list of numeric values, specifying the values at which some or all of the coefficients in the variance function should be fixed. If a grouping factor is specified in form, fixed must have names identifying which coefficients are to be fixed. Coefficients included in fixed are not allowed to vary during the optimization of an objective function. Defaults to NULL, corresponding to no fixed coefficients.

Value

a varExp object representing an exponential variance function structure, also inheriting from class varFunc.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

varClasses, varWeights.varFunc, coef.varExp

Examples

```
vf1 <- varExp(0.2, form = ~age|Sex)
```

---

varFixed *Fixed Variance Function*

Description

This function is a constructor for the varFixed class, representing a variance function with fixed variances. Letting v denote the variance covariate defined in value, the variance function $\sigma^2(v)$ for this class is $\sigma^2(v) = |v|$. The variance covariate v is evaluated once at initialization and remains fixed thereafter. No coefficients are required to represent this variance function.

Usage

```
varFixed(value)
```

Arguments

value — a one-sided formula of the form ~ v specifying a variance covariate v. Grouping factors are ignored.

Value

a `varFixed` object representing a fixed variance function structure, also inheriting from class `varFunc`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

`varClasses`, `varWeights.varFunc`, `varFunc`

Examples

```
vf1 <- varFixed(~age)
```

`varFunc` *Variance Function Structure*

Description

If `object` is a one-sided formula, it is used as the argument to `varFixed` and the resulting object is returned. Else, if `object` inherits from class `varFunc`, it is returned unchanged.

Usage

```
varFunc(object)
```

Arguments

`object`	either an one-sided formula specifying a variance covariate, or an object inheriting from class `varFunc`, representing a variance function structure.

Value

an object from class `varFunc`, representing a variance function structure.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

`summary.varFunc`, `varFixed`, `varWeights.varFunc`, `coef.varFunc`

Examples

```
vf1 <- varFunc(~age)
```

varIdent *Constant Variance Function*

Description

This function is a constructor for the varIdent class, representing a constant variance function structure. If no grouping factor is present in form, the variance function is constant and equal to one, and no coefficients required to represent it. When form includes a grouping factor with $M > 1$ levels, the variance function allows M different variances, one for each level of the factor. For identifiability reasons, the coefficients of the variance function represent the ratios between the variances and a reference variance (corresponding to a reference group level). Therefore, only $M - 1$ coefficients are needed to represent the variance function. By default, if the elements in value are unnamed, the first group level is taken as the reference level.

Usage

```
varIdent(value, form, fixed)
```

Arguments

value	an optional numeric vector, or list of numeric values, with the variance function coefficients. If no grouping factor is present in form, this argument is ignored, as the resulting variance function contains no coefficients. If value has length one, its value is repeated for all coefficients in the variance function. If value has length greater than one, it must have length equal to the number of grouping levels minus one and names which identify its elements to the levels of the grouping factor. Only positive values are allowed for this argument. Default is numeric(0), which results in a vector of zeros of appropriate length being assigned to the coefficients when object is initialized (corresponding to constant variance equal to one).
form	an optional one-sided formula of the form ~ v, or ~ v | g, specifying a variance covariate v and, optionally, a grouping factor g for the coefficients. The variance covariate is ignored in this variance function. When a grouping factor is present in form, a different coefficient value is used for each of its levels less one reference level (see description section below). Several grouping variables may be simultaneously specified, separated by the * operator, like in ~ v | g1 * g2 * g3. In this case, the levels of each grouping variable are pasted together and the resulting factor is used to group the observations. Defaults to ~ 1.
fixed	an optional numeric vector, or list of numeric values, specifying the values at which some or all of the coefficients in the variance function should be fixed. It must have names identifying which coefficients are to be fixed. Coefficients included in fixed are not allowed to vary during the optimization of an objective function. Defaults to NULL, corresponding to no fixed coefficients.

Value

a varIdent object representing a constant variance function structure, also inheriting from class varFunc.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

`varClasses`, `varWeights.varFunc`, `coef.varIdent`

Examples

```
vf1 <- varIdent(c(Female = 0.5), form = ~ 1 | Sex)
```

`Variogram` *Calculate Semi-variogram*

Description

This function is generic; method functions can be written to handle specific classes of objects. Classes which already have methods for this function include `default`, `gls` and `lme`. See the appropriate method documentation for a description of the arguments.

Usage

```
Variogram(object, distance, ...)
```

Arguments

`object`	a numeric vector with the values to be used for calculating the semi-variogram, usually a residual vector from a fitted model.
`distance`	a numeric vector with the pairwise distances corresponding to the elements of `object`. The order of the elements in `distance` must correspond to the pairs `(1,2), (1,3), ..., (n-1,n)`, with `n` representing the length of `object`, and must have length `n(n-1)/2`.
`...`	some methods for this generic function require additional arguments.

Value

will depend on the method function used; see the appropriate documentation.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Cressie, N.A.C. (1993), "Statistics for Spatial Data", J. Wiley & Sons.

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

Variogram.corExp, Variogram.corGaus, Variogram.corLin, Variogram.corRatio, Variogram.corSpatial, Variogram.corSpher, Variogram.default, Variogram.gls, Variogram.lme, plot.Variogram

Examples

```
## see the method function documentation
```

Variogram.corExp *Calculate Semi-variogram for a corExp Object*

Description

This method function calculates the semi-variogram values corresponding to the Exponential correlation model, using the estimated coefficients corresponding to `object`, at the distances defined by `distance`.

Usage

```
## S3 method for class 'corExp':
Variogram(object, distance, sig2, length.out, ...)
```

Arguments

`object`	an object inheriting from class `corExp`, representing an exponential spatial correlation structure.
`distance`	an optional numeric vector with the distances at which the semi-variogram is to be calculated. Defaults to `NULL`, in which case a sequence of length `length.out` between the minimum and maximum values of `getCovariate(object)` is used.
`sig2`	an optional numeric value representing the process variance. Defaults to `1`.
`length.out`	an optional integer specifying the length of the sequence of distances to be used for calculating the semi-variogram, when `distance = NULL`. Defaults to `50`.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

a data frame with columns `variog` and `dist` representing, respectively, the semi-variogram values and the corresponding distances. The returned value inherits from class `Variogram`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Cressie, N.A.C. (1993), "Statistics for Spatial Data", J. Wiley & Sons.

See Also

corExp, plot.Variogram, Variogram

Examples

```
stopifnot(require("stats", quietly = TRUE))
cs1 <- corExp(3, form = ~ Time | Rat)
cs1 <- Initialize(cs1, BodyWeight)
Variogram(cs1)[1:10,]
```

Variogram.corGaus *Calculate Semi-variogram for a corGaus Object*

Description

This method function calculates the semi-variogram values corresponding to the Gaussian correlation model, using the estimated coefficients corresponding to `object`, at the distances defined by `distance`.

Usage

```
## S3 method for class 'corGaus':
Variogram(object, distance, sig2, length.out, ...)
```

Arguments

`object`	an object inheriting from class `corGaus`, representing an Gaussian spatial correlation structure.
`distance`	an optional numeric vector with the distances at which the semi-variogram is to be calculated. Defaults to `NULL`, in which case a sequence of length `length.out` between the minimum and maximum values of `getCovariate(object)` is used.
`sig2`	an optional numeric value representing the process variance. Defaults to `1`.
`length.out`	an optional integer specifying the length of the sequence of distances to be used for calculating the semi-variogram, when `distance = NULL`. Defaults to `50`.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

a data frame with columns `variog` and `dist` representing, respectively, the semi-variogram values and the corresponding distances. The returned value inherits from class `Variogram`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Cressie, N.A.C. (1993), "Statistics for Spatial Data", J. Wiley & Sons.

See Also

corGaus, plot.Variogram, Variogram

Examples

```
cs1 <- corGaus(3, form = ~ Time | Rat)
cs1 <- Initialize(cs1, BodyWeight)
Variogram(cs1)[1:10,]
```

Variogram.corLin *Calculate Semi-variogram for a corLin Object*

Description

This method function calculates the semi-variogram values corresponding to the Linear correlation model, using the estimated coefficients corresponding to `object`, at the distances defined by `distance`.

Usage

```
## S3 method for class 'corLin':
Variogram(object, distance, sig2, length.out, ...)
```

Arguments

`object`	an object inheriting from class `corLin`, representing an Linear spatial correlation structure.
`distance`	an optional numeric vector with the distances at which the semi-variogram is to be calculated. Defaults to `NULL`, in which case a sequence of length `length.out` between the minimum and maximum values of `getCovariate(object)` is used.
`sig2`	an optional numeric value representing the process variance. Defaults to `1`.
`length.out`	an optional integer specifying the length of the sequence of distances to be used for calculating the semi-variogram, when `distance = NULL`. Defaults to `50`.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

a data frame with columns `variog` and `dist` representing, respectively, the semi-variogram values and the corresponding distances. The returned value inherits from class `Variogram`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Cressie, N.A.C. (1993), "Statistics for Spatial Data", J. Wiley & Sons.

See Also

corLin, plot.Variogram, Variogram

Examples

```
cs1 <- corLin(15, form = ~ Time | Rat)
cs1 <- Initialize(cs1, BodyWeight)
Variogram(cs1)[1:10,]
```

Variogram.corRatio *Calculate Semi-variogram for a corRatio Object*

Description

This method function calculates the semi-variogram values corresponding to the Rational Quadratic correlation model, using the estimated coefficients corresponding to `object`, at the distances defined by `distance`.

Usage

```
## S3 method for class 'corRatio':
Variogram(object, distance, sig2, length.out, ...)
```

Arguments

`object`	an object inheriting from class `corRatio`, representing an Rational Quadratic spatial correlation structure.
`distance`	an optional numeric vector with the distances at which the semi-variogram is to be calculated. Defaults to `NULL`, in which case a sequence of length `length.out` between the minimum and maximum values of `getCovariate(object)` is used.
`sig2`	an optional numeric value representing the process variance. Defaults to `1`.
`length.out`	an optional integer specifying the length of the sequence of distances to be used for calculating the semi-variogram, when `distance = NULL`. Defaults to `50`.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

a data frame with columns `variog` and `dist` representing, respectively, the semi-variogram values and the corresponding distances. The returned value inherits from class `Variogram`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Cressie, N.A.C. (1993), "Statistics for Spatial Data", J. Wiley & Sons.

See Also

corRatio, plot.Variogram Variogram

Examples

```
cs1 <- corRatio(7, form = ~ Time | Rat)
cs1 <- Initialize(cs1, BodyWeight)
Variogram(cs1)[1:10,]
```

Variogram.corSpatial

Calculate Semi-variogram for a corSpatial Object

Description

This method function calculates the semi-variogram values corresponding to the model defined in FUN, using the estimated coefficients corresponding to object, at the distances defined by distance.

Usage

```
## S3 method for class 'corSpatial':
Variogram(object, distance, sig2, length.out, FUN, ...)
```

Arguments

object	an object inheriting from class corSpatial, representing spatial correlation structure.
distance	an optional numeric vector with the distances at which the semi-variogram is to be calculated. Defaults to NULL, in which case a sequence of length length.out between the minimum and maximum values of getCovariate(object) is used.
sig2	an optional numeric value representing the process variance. Defaults to 1.
length.out	an optional integer specifying the length of the sequence of distances to be used for calculating the semi-variogram, when distance = NULL. Defaults to 50.
FUN	a function of two arguments, the distance and the range corresponding to object, specifying the semi-variogram model.
...	some methods for this generic require additional arguments. None are used in this method.

Value

a data frame with columns variog and dist representing, respectively, the semi-variogram values and the corresponding distances. The returned value inherits from class Variogram.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Cressie, N.A.C. (1993), "Statistics for Spatial Data", J. Wiley & Sons.

See Also

corSpatial, Variogram, Variogram.default, Variogram.corExp, Variogram.corGaus, Variogram.corLin, Variogram.corRatio, Variogram.corSpher, plot.Variogram

Examples

```
cs1 <- corExp(3, form = ~ Time | Rat)
cs1 <- Initialize(cs1, BodyWeight)
Variogram(cs1, FUN = function(x, y) (1 - exp(-x/y)))[1:10,]
```

Variogram.corSpher *Calculate Semi-variogram for a corSpher Object*

Description

This method function calculates the semi-variogram values corresponding to the Spherical correlation model, using the estimated coefficients corresponding to `object`, at the distances defined by `distance`.

Usage

```
## S3 method for class 'corSpher':
Variogram(object, distance, sig2, length.out, ...)
```

Arguments

`object`	an object inheriting from class `corSpher`, representing an Spherical spatial correlation structure.
`distance`	an optional numeric vector with the distances at which the semi-variogram is to be calculated. Defaults to `NULL`, in which case a sequence of length `length.out` between the minimum and maximum values of `getCovariate(object)` is used.
`sig2`	an optional numeric value representing the process variance. Defaults to `1`.
`length.out`	an optional integer specifying the length of the sequence of distances to be used for calculating the semi-variogram, when `distance = NULL`. Defaults to `50`.
`...`	some methods for this generic require additional arguments. None are used in this method.

Value

a data frame with columns `variog` and `dist` representing, respectively, the semi-variogram values and the corresponding distances. The returned value inherits from class `Variogram`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Cressie, N.A.C. (1993), "Statistics for Spatial Data", J. Wiley & Sons.

See Also

corSpher, plot.Variogram, Variogram

Examples

```
cs1 <- corSpher(15, form = ~ Time | Rat)
cs1 <- Initialize(cs1, BodyWeight)
Variogram(cs1)[1:10,]
```

Variogram.default *Calculate Semi-variogram*

Description

This method function calculates the semi-variogram for an arbitrary vector object, according to the distances in distance. For each pair of elements x, y in object, the corresponding semi-variogram is $(x - y)^2/2$. The semi-variogram is useful for identifying and modeling spatial correlation structures in observations with constant expectation and constant variance.

Usage

```
## Default S3 method:
Variogram(object, distance, ...)
```

Arguments

object	a numeric vector with the values to be used for calculating the semi-variogram, usually a residual vector from a fitted model.
distance	a numeric vector with the pairwise distances corresponding to the elements of object. The order of the elements in distance must correspond to the pairs (1,2), (1,3), ..., (n-1,n), with n representing the length of object, and must have length n(n-1)/2.
...	some methods for this generic require additional arguments. None are used in this method.

Value

a data frame with columns variog and dist representing, respectively, the semi-variogram values and the corresponding distances. The returned value inherits from class Variogram.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Cressie, N.A.C. (1993), "Statistics for Spatial Data", J. Wiley & Sons.

See Also

`Variogram`, `Variogram.gls`, `Variogram.lme`, `plot.Variogram`

Examples

```
## Not run:
fm1 <- lm(follicles ~ sin(2 * pi * Time) + cos(2 * pi * Time), Ovary,
          subset = Mare == 1)
Variogram(resid(fm1), dist(1:29))[1:10,]
## End(Not run)
```

`Variogram.gls` *Calculate Semi-variogram for Residuals from a gls Object*

Description

This method function calculates the semi-variogram for the residuals from a `gls` fit. The semi-variogram values are calculated for pairs of residuals within the same group level, if a grouping factor is present. If `collapse` is different from `"none"`, the individual semi-variogram values are collapsed using either a robust estimator (`robust = TRUE`) defined in Cressie (1993), or the average of the values within the same distance interval. The semi-variogram is useful for modeling the error term correlation structure.

Usage

```
## S3 method for class 'gls':
Variogram(object, distance, form, resType, data,
      na.action, maxDist, length.out, collapse, nint, breaks,
      robust, metric, ...)
```

Arguments

`object`	an object inheriting from class `gls`, representing a generalized least squares fitted model.
`distance`	an optional numeric vector with the distances between residual pairs. If a grouping variable is present, only the distances between residual pairs within the same group should be given. If missing, the distances are calculated based on the values of the arguments `form`, `data`, and `metric`, unless `object` includes a `corSpatial` element, in which case the associated covariate (obtained with the `getCovariate` method) is used.
`form`	an optional one-sided formula specifying the covariate(s) to be used for calculating the distances between residual pairs and, optionally, a grouping factor for partitioning the residuals (which must appear to the right of a `|` operator in `form`). Default is `~1`, implying that the observation order within the groups is used to obtain the distances.

resType — an optional character string specifying the type of residuals to be used. If "response", the "raw" residuals (observed - fitted) are used; else, if "pearson", the standardized residuals (raw residuals divided by the corresponding standard errors) are used; else, if "normalized", the normalized residuals (standardized residuals pre-multiplied by the inverse square-root factor of the estimated error correlation matrix) are used. Partial matching of arguments is used, so only the first character needs to be provided. Defaults to "pearson".

data — an optional data frame in which to interpret the variables in form. By default, the same data used to fit object is used.

na.action — a function that indicates what should happen when the data contain NAs. The default action (na.fail) causes an error message to be printed and the function to terminate, if there are any incomplete observations.

maxDist — an optional numeric value for the maximum distance used for calculating the semi-variogram between two residuals. By default all residual pairs are included.

length.out — an optional integer value. When object includes a corSpatial element, its semi-variogram values are calculated and this argument is used as the length.out argument to the corresponding Variogram method. Defaults to 50.

collapse — an optional character string specifying the type of collapsing to be applied to the individual semi-variogram values. If equal to "quantiles", the semi-variogram values are split according to quantiles of the distance distribution, with equal number of observations per group, with possibly varying distance interval lengths. Else, if "fixed", the semi-variogram values are divided according to distance intervals of equal lengths, with possibly different number of observations per interval. Else, if "none", no collapsing is used and the individual semi-variogram values are returned. Defaults to "quantiles".

nint — an optional integer with the number of intervals to be used when collapsing the semi-variogram values. Defaults to 20.

robust — an optional logical value specifying if a robust semi-variogram estimator should be used when collapsing the individual values. If TRUE the robust estimator is used. Defaults to FALSE.

breaks — an optional numeric vector with the breakpoints for the distance intervals to be used in collapsing the semi-variogram values. If not missing, the option specified in collapse is ignored.

metric — an optional character string specifying the distance metric to be used. The currently available options are "euclidean" for the root sum-of-squares of distances; "maximum" for the maximum difference; and "manhattan" for the sum of the absolute differences. Partial matching of arguments is used, so only the first three characters need to be provided. Defaults to "euclidean".

... — some methods for this generic require additional arguments. None are used in this method.

Value

a data frame with columns variog and dist representing, respectively, the semi-variogram values and the corresponding distances. If the semi-variogram values are collapsed, an extra column, n.pairs, with the number of residual pairs used in each semi-variogram calculation, is included in the returned data frame. If object includes a corSpatial element, a data frame with its

corresponding semi-variogram is included in the returned value, as an attribute `"modelVariog"`. The returned value inherits from class `Variogram`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Cressie, N.A.C. (1993), "Statistics for Spatial Data", J. Wiley & Sons.

See Also

`gls`, `Variogram`, `Variogram.default`, `Variogram.lme`, `plot.Variogram`

Examples

```
## Not run:
fm1 <- gls(weight ~ Time * Diet, BodyWeight)
Variogram(fm1, form = ~ Time | Rat)[1:10,]
## End(Not run)
```

`Variogram.lme` *Calculate Semi-variogram for Residuals from an lme Object*

Description

This method function calculates the semi-variogram for the within-group residuals from an `lme` fit. The semi-variogram values are calculated for pairs of residuals within the same group. If `collapse` is different from `"none"`, the individual semi-variogram values are collapsed using either a robust estimator (`robust = TRUE`) defined in Cressie (1993), or the average of the values within the same distance interval. The semi-variogram is useful for modeling the error term correlation structure.

Usage

```
## S3 method for class 'lme':
Variogram(object, distance, form, resType, data,
       na.action, maxDist, length.out, collapse, nint, breaks,
       robust, metric, ...)
```

Arguments

`object`	an object inheriting from class `lme`, representing a fitted linear mixed-effects model.
`distance`	an optional numeric vector with the distances between residual pairs. If a grouping variable is present, only the distances between residual pairs within the same group should be given. If missing, the distances are calculated based on the values of the arguments `form`, `data`, and `metric`, unless `object` includes a `corSpatial` element, in which case the associated covariate (obtained with the `getCovariate` method) is used.

`form` an optional one-sided formula specifying the covariate(s) to be used for calculating the distances between residual pairs and, optionally, a grouping factor for partitioning the residuals (which must appear to the right of a `|` operator in `form`). Default is `~1`, implying that the observation order within the groups is used to obtain the distances.

`resType` an optional character string specifying the type of residuals to be used. If `"response"`, the "raw" residuals (observed - fitted) are used; else, if `"pearson"`, the standardized residuals (raw residuals divided by the corresponding standard errors) are used; else, if `"normalized"`, the normalized residuals (standardized residuals pre-multiplied by the inverse square-root factor of the estimated error correlation matrix) are used. Partial matching of arguments is used, so only the first character needs to be provided. Defaults to `"pearson"`.

`data` an optional data frame in which to interpret the variables in `form`. By default, the same data used to fit `object` is used.

`na.action` a function that indicates what should happen when the data contain `NA`s. The default action (`na.fail`) causes an error message to be printed and the function to terminate, if there are any incomplete observations.

`maxDist` an optional numeric value for the maximum distance used for calculating the semi-variogram between two residuals. By default all residual pairs are included.

`length.out` an optional integer value. When `object` includes a `corSpatial` element, its semi-variogram values are calculated and this argument is used as the `length.out` argument to the corresponding `Variogram` method. Defaults to `50`.

`collapse` an optional character string specifying the type of collapsing to be applied to the individual semi-variogram values. If equal to `"quantiles"`, the semi-variogram values are split according to quantiles of the distance distribution, with equal number of observations per group, with possibly varying distance interval lengths. Else, if `"fixed"`, the semi-variogram values are divided according to distance intervals of equal lengths, with possibly different number of observations per interval. Else, if `"none"`, no collapsing is used and the individual semi-variogram values are returned. Defaults to `"quantiles"`.

`nint` an optional integer with the number of intervals to be used when collapsing the semi-variogram values. Defaults to `20`.

`robust` an optional logical value specifying if a robust semi-variogram estimator should be used when collapsing the individual values. If `TRUE` the robust estimator is used. Defaults to `FALSE`.

`breaks` an optional numeric vector with the breakpoints for the distance intervals to be used in collapsing the semi-variogram values. If not missing, the option specified in `collapse` is ignored.

`metric` an optional character string specifying the distance metric to be used. The currently available options are `"euclidean"` for the root sum-of-squares of distances; `"maximum"` for the maximum difference; and `"manhattan"` for the sum of the absolute differences. Partial matching of arguments is used, so only the first three characters need to be provided. Defaults to `"euclidean"`.

`...` some methods for this generic require additional arguments. None are used in this method.

Value

a data frame with columns `variog` and `dist` representing, respectively, the semi-variogram values and the corresponding distances. If the semi-variogram values are collapsed, an extra column, `n.pairs`, with the number of residual pairs used in each semi-variogram calculation, is included in the returned data frame. If `object` includes a `corSpatial` element, a data frame with its corresponding semi-variogram is included in the returned value, as an attribute `"modelVariog"`. The returned value inherits from class `Variogram`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Cressie, N.A.C. (1993), "Statistics for Spatial Data", J. Wiley & Sons.

See Also

`lme`, `Variogram`, `Variogram.default`, `Variogram.gls`, `plot.Variogram`

Examples

```
fm1 <- lme(weight ~ Time * Diet, data=BodyWeight, ~ Time | Rat)
Variogram(fm1, form = ~ Time | Rat, nint = 10, robust = TRUE)
```

---

varPower *Power Variance Function*

Description

This function is a constructor for the `varPower` class, representing a power variance function structure. Letting v denote the variance covariate and $\sigma^2(v)$ denote the variance function evaluated at v, the power variance function is defined as $\sigma^2(v) = |v|^{2\theta}$, where θ is the variance function coefficient. When a grouping factor is present, a different θ is used for each factor level.

Usage

```
varPower(value, form, fixed)
```

Arguments

`value`	an optional numeric vector, or list of numeric values, with the variance function coefficients. `Value` must have length one, unless a grouping factor is specified in `form`. If `value` has length greater than one, it must have names which identify its elements to the levels of the grouping factor defined in `form`. If a grouping factor is present in `form` and `value` has length one, its value will be assigned to all grouping levels. Default is `numeric(0)`, which results in a vector of zeros of appropriate length being assigned to the coefficients when `object` is initialized (corresponding to constant variance equal to one).

`form` an optional one-sided formula of the form `~ v`, or `~ v | g`, specifying a variance covariate `v` and, optionally, a grouping factor `g` for the coefficients. The variance covariate must evaluate to a numeric vector and may involve expressions using `"."`, representing a fitted model object from which fitted values (`fitted(.)`) and residuals (`resid(.)`) can be extracted (this allows the variance covariate to be updated during the optimization of an object function). When a grouping factor is present in `form`, a different coefficient value is used for each of its levels. Several grouping variables may be simultaneously specified, separated by the `*` operator, like in `~ v | g1 * g2 * g3`. In this case, the levels of each grouping variable are pasted together and the resulting factor is used to group the observations. Defaults to `~ fitted(.)` representing a variance covariate given by the fitted values of a fitted model object and no grouping factor.

`fixed` an optional numeric vector, or list of numeric values, specifying the values at which some or all of the coefficients in the variance function should be fixed. If a grouping factor is specified in `form`, `fixed` must have names identifying which coefficients are to be fixed. Coefficients included in `fixed` are not allowed to vary during the optimization of an objective function. Defaults to `NULL`, corresponding to no fixed coefficients.

Value

a `varPower` object representing a power variance function structure, also inheriting from class `varFunc`.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

`varWeights.varFunc`, `coef.varPower`

Examples

```
vf1 <- varPower(0.2, form = ~age|Sex)
```

---

varWeights *Extract Variance Function Weights*

Description

The inverse of the standard deviations corresponding to the variance function structure represented by `object` are returned.

Usage

```
varWeights(object)
```

Arguments

`object`	an object inheriting from class `varFunc`, representing a variance function structure.

Value

if `object` has a `weights` attribute, its value is returned; else `NULL` is returned.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

`logLik.varFunc`, `varWeights`

Examples

```
vf1 <- varPower(form=~age)
vf1 <- Initialize(vf1, Orthodont)
coef(vf1) <- 0.3
varWeights(vf1)[1:10]
```

`varWeights.glsStruct`

Variance Weights for glsStruct Object

Description

If `object` includes a `varStruct` component, the inverse of the standard deviations of the variance function structure represented by the corresponding `varFunc` object are returned; else, a vector of ones of length equal to the number of observations in the data frame used to fit the associated linear model is returned.

Usage

```
## S3 method for class 'glsStruct':
varWeights(object)
```

Arguments

`object`	an object inheriting from class `glsStruct`, representing a list of linear model components, such as `corStruct` and `varFunc` objects.

Value

if `object` includes a `varStruct` component, a vector with the corresponding variance weights; else, or a vector of ones.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

varWeights

---

varWeights.lmeStruct

Variance Weights for lmeStruct Object

Description

If `object` includes a `varStruct` component, the inverse of the standard deviations of the variance function structure represented by the corresponding `varFunc` object are returned; else, a vector of ones of length equal to the number of observations in the data frame used to fit the associated linear mixed-effects model is returned.

Usage

```
## S3 method for class 'lmeStruct':
varWeights(object)
```

Arguments

`object`	an object inheriting from class `lmeStruct`, representing a list of linear mixed-effects model components, such as `reStruct`, `corStruct`, and `varFunc` objects.

Value

if `object` includes a `varStruct` component, a vector with the corresponding variance weights; else, or a vector of ones.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

References

Pinheiro, J.C., and Bates, D.M. (2000) "Mixed-Effects Models in S and S-PLUS", Springer.

See Also

varWeights

`Wafer` *Modeling of Analog MOS Circuits*

Description

The `Wafer` data frame has 400 rows and 4 columns.

Format

This data frame contains the following columns:

Wafer a factor with levels `1 2 3 4 5 6 7 8 9 10`

Site a factor with levels `1 2 3 4 5 6 7 8`

voltage a numeric vector

current a numeric vector

Details

Source

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York.

`Wheat` *Yields by growing conditions*

Description

The `Wheat` data frame has 48 rows and 4 columns.

Format

This data frame contains the following columns:

Tray an ordered factor with levels `3 < 1 < 2 < 4 < 5 < 6 < 8 < 9 < 7 < 12 < 11 < 10`

Moisture a numeric vector

fertilizer a numeric vector

DryMatter a numeric vector

Details

Source

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York.

Wheat2 *Wheat Yield Trials*

Description

The Wheat2 data frame has 224 rows and 5 columns.

Format

This data frame contains the following columns:

Block an ordered factor with levels 4 < 2 < 3 < 1

variety a factor with levels ARAPAHOE BRULE BUCKSKIN CENTURA CENTURK78 CHEYENNE CODY COLT GAGE HOMESTEAD KS831374 LANCER LANCOTA NE83404 NE83406 NE83407 NE83432 NE83498 NE83T12 NE84557 NE85556 NE85623 NE86482 NE86501 NE86503 NE86507 NE86509 NE86527 NE86582 NE86606 NE86607 NE86T666 NE87403 NE87408 NE87409 NE87446 NE87451 NE87457 NE87463 NE87499 NE87512 NE87513 NE87522 NE87612 NE87613 NE87615 NE87619 NE87627 NORKAN REDLAND ROUGHRIDER SCOUT66 SIOUXLAND TAM107 TAM200 VONA

yield a numeric vector

latitude a numeric vector

longitude a numeric vector

Details

Source

Pinheiro, J. C. and Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York.

[.pdMat *Subscript a pdMat Object*

Description

This method function extracts sub-matrices from the positive-definite matrix represented by x.

Usage

```
## S3 method for class 'pdMat':
x[i, j, drop = TRUE]
## S3 method for class 'pdMat':
x[i, j] <- value
```

Arguments

x	an object inheriting from class pdMat representing a positive-definite matrix.
i, j	optional subscripts applying respectively to the rows and columns of the positive-definite matrix represented by object. When i (j) is omitted, all rows (columns) are extracted.
drop	a logical value. If TRUE, single rows or columns are converted to vectors. If FALSE the returned value retains its matrix representation.
value	a vector, or matrix, with the replacement values for the relevant piece of the matrix represented by x.

Value

if i and j are identical, the returned value will be pdMat object with the same class as x. Otherwise, the returned value will be a matrix. In the case a single row (or column) is selected, the returned value may be converted to a vector, according to the rules above.

Author(s)

Jose Pinheiro ⟨Jose.Pinheiro@pharma.novartis.com⟩ and Douglas Bates ⟨bates@stat.wisc.edu⟩

See Also

[, pdMat

Examples

```
pd1 <- pdSymm(diag(3))
pd1[1, , drop = FALSE]
pd1[1:2, 1:2] <- 3 * diag(2)
```

Chapter 19

The `nnet` package

`class.ind` *Generates Class Indicator Matrix from a Factor*

Description

Generates a class indicator function from a given factor.

Usage

```
class.ind(cl)
```

Arguments

`cl`	factor or vector of classes for cases.

Value

a matrix which is zero except for the column corresponding to the class.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

Examples

```
# The function is currently defined as
class.ind <- function(cl)
{
  n <- length(cl)
  cl <- as.factor(cl)
  x <- matrix(0, n, length(levels(cl)) )
  x[(1:n) + n*(unclass(cl)-1)] <- 1
  dimnames(x) <- list(names(cl), levels(cl))
  x
}
```

multinom *Fit Multinomial Log-linear Models*

Description

Fits multinomial log-linear models via neural networks.

Usage

```
multinom(formula, data, weights, subset, na.action,
         contrasts = NULL, Hess = FALSE, summ = 0, censored = FALSE,
         model = FALSE, ...)
```

Arguments

formula	a formula expression as for regression models, of the form response ~ predictors. The response should be a factor or a matrix with K columns, which will be interpreted as counts for each of K classes. A log-linear model is fitted, with coefficients zero for the first class. An offset can be included: it should be a numeric matrix with K columns if the response is either a matrix with K columns or a factor with K > 2 classes, or a numeric vector for a response factor with 2 levels. See the documentation of formula() for other details.
data	an optional data frame in which to interpret the variables occurring in formula.
weights	optional case weights in fitting.
subset	expression saying which subset of the rows of the data should be used in the fit. All observations are included by default.
na.action	a function to filter missing data.
contrasts	a list of contrasts to be used for some or all of the factors appearing as variables in the model formula.
Hess	logical for whether the Hessian (the observed/expected information matrix) should be returned.
summ	integer; if non-zero summarize by deleting duplicate rows and adjust weights. Methods 1 and 2 differ in speed (2 uses C); method 3 also combines rows with the same X and different Y, which changes the baseline for the deviance.
censored	If Y is a matrix with K > 2 columns, interpret the entries as one for possible classes, zero for impossible classes, rather than as counts.
model	logical. If true, the model frame is saved as component model of the returned object.
...	additional arguments for nnet

Details

multinom calls nnet. The variables on the rhs of the formula should be roughly scaled to [0,1] or the fit will be slow or may not converge at all.

Value

A `nnet` object with additional components:

`deviance`	the residual deviance, compared to the full saturated model (that explains individual observations exactly). Also, minus twice log-likelihood.
`edf`	the (effective) number of degrees of freedom used by the model
`AIC`	the AIC for this fit.
`Hessian`	(if `Hess` is true).
`model`	(if `model` is true).

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`nnet`

Examples

```
options(contrasts = c("contr.treatment", "contr.poly"))
library(MASS)
example(birthwt)
(bwt.mu <- multinom(low ~ ., bwt))
## Not run: Call:
multinom(formula = low ~ ., data = bwt)

Coefficients:
 (Intercept)         age         lwt raceblack raceother
   0.823477 -0.03724311 -0.01565475  1.192371 0.7406606
     smoke       ptd        ht        ui       ftv1     ftv2+
 0.7555234 1.343648 1.913213 0.6802007 -0.4363238 0.1789888

Residual Deviance: 195.4755
AIC: 217.4755
## End(Not run)
```

`nnet` *Fit Neural Networks*

Description

Fit single-hidden-layer neural network, possibly with skip-layer connections.

Usage

```
nnet(x, ...)

## S3 method for class 'formula':
nnet(formula, data, weights, ...,
     subset, na.action, contrasts = NULL)
```

```
## Default S3 method:
nnet(x, y, weights, size, Wts, mask,
     linout = FALSE, entropy = FALSE, softmax = FALSE,
     censored = FALSE, skip = FALSE, rang = 0.7, decay = 0,
     maxit = 100, Hess = FALSE, trace = TRUE, MaxNWts = 1000,
     abstol = 1.0e-4, reltol = 1.0e-8, ...)
```

Arguments

formula	A formula of the form `class ~ x1 + x2 + ...`
x	matrix or data frame of `x` values for examples.
y	matrix or data frame of target values for examples.
weights	(case) weights for each example – if missing defaults to 1.
size	number of units in the hidden layer. Can be zero if there are skip-layer units.
data	Data frame from which variables specified in `formula` are preferentially to be taken.
subset	An index vector specifying the cases to be used in the training sample. (NOTE: If given, this argument must be named.)
na.action	A function to specify the action to be taken if `NA`s are found. The default action is for the procedure to fail. An alternative is na.omit, which leads to rejection of cases with missing values on any required variable. (NOTE: If given, this argument must be named.)
contrasts	a list of contrasts to be used for some or all of the factors appearing as variables in the model formula.
Wts	initial parameter vector. If missing chosen at random.
mask	logical vector indicating which parameters should be optimized (default all).
linout	switch for linear output units. Default logistic output units.
entropy	switch for entropy (= maximum conditional likelihood) fitting. Default by least-squares.
softmax	switch for softmax (log-linear model) and maximum conditional likelihood fitting. `linout`, `entropy`, `softmax` and `censored` are mutually exclusive.
censored	A variant on `softmax`, in which non-zero targets mean possible classes. Thus for `softmax` a row of `(0, 1, 1)` means one example each of classes 2 and 3, but for `censored` it means one example whose class is only known to be 2 or 3.
skip	switch to add skip-layer connections from input to output.
rang	Initial random weights on [-`rang`, `rang`]. Value about 0.5 unless the inputs are large, in which case it should be chosen so that `rang` * max(`|x|`) is about 1.
decay	parameter for weight decay. Default 0.
maxit	maximum number of iterations. Default 100.
Hess	If true, the Hessian of the measure of fit at the best set of weights found is returned as component `Hessian`.
trace	switch for tracing optimization. Default `TRUE`.
MaxNWts	The maximum allowable number of weights. There is no intrinsic limit in the code, but increasing `MaxNWts` will probably allow fits that are very slow and time-consuming (and perhaps uninterruptable).

abstol	Stop if the fit criterion falls below abstol, indicating an essentially perfect fit.
reltol	Stop if the optimizer is unable to reduce the fit criterion by a factor of at least 1 - reltol.
...	arguments passed to or from other methods.

Details

If the response in formula is a factor, an appropriate classification network is constructed; this has one output and entropy fit if the number of levels is two, and a number of outputs equal to the number of classes and a softmax output stage for more levels. If the response is not a factor, it is passed on unchanged to nnet.default.

Optimization is done via the BFGS method of optim.

Value

object of class "nnet" or "nnet.formula". Mostly internal structure, but has components

wts	the best set of weights found
value	value of fitting criterion plus weight decay term.
fitted.values	the fitted values for the training data.
residuals	the residuals for the training data.

References

Ripley, B. D. (1996) *Pattern Recognition and Neural Networks.* Cambridge.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

predict.nnet, nnetHess

Examples

```
data(iris3)
# use half the iris data
ir <- rbind(iris3[,,1],iris3[,,2],iris3[,,3])
targets <- class.ind( c(rep("s", 50), rep("c", 50), rep("v", 50)) )
samp <- c(sample(1:50,25), sample(51:100,25), sample(101:150,25))
ir1 <- nnet(ir[samp,], targets[samp,], size = 2, rang = 0.1,
            decay = 5e-4, maxit = 200)
test.cl <- function(true, pred) {
    true <- max.col(true)
    cres <- max.col(pred)
    table(true, cres)
}
test.cl(targets[-samp,], predict(ir1, ir[-samp,]))

# or
ird <- data.frame(rbind(iris3[,,1], iris3[,,2], iris3[,,3]),
        species = factor(c(rep("s",50), rep("c", 50), rep("v", 50))))
ir.nn2 <- nnet(species ~ ., data = ird, subset = samp, size = 2, rang = 0.1,
               decay = 5e-4, maxit = 200)
table(ird$species[-samp], predict(ir.nn2, ird[-samp,], type = "class"))
```

nnetHess *Evaluates Hessian for a Neural Network*

Description

Evaluates the Hessian (matrix of second derivatives) of the specified neural network. Normally called via argument `Hess=TRUE` to `nnet` or via `vcov.multinom`.

Usage

```
nnetHess(net, x, y, weights)
```

Arguments

`net`	object of class `nnet` as returned by `nnet`.
`x`	training data.
`y`	classes for training data.
`weights`	the (case) weights used in the `nnet` fit.

Value

square symmetric matrix of the Hessian evaluated at the weights stored in the net.

References

Ripley, B. D. (1996) *Pattern Recognition and Neural Networks.* Cambridge.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`nnet`, `predict.nnet`

Examples

```
data(iris3)
# use half the iris data
ir <- rbind(iris3[,,1], iris3[,,2], iris3[,,3])
targets <- matrix(c(rep(c(1,0,0),50), rep(c(0,1,0),50), rep(c(0,0,1),50)),
150, 3, byrow=TRUE)
samp <- c(sample(1:50,25), sample(51:100,25), sample(101:150,25))
ir1 <- nnet(ir[samp,], targets[samp,], size=2, rang=0.1, decay=5e-4, maxit=200)
eigen(nnetHess(ir1, ir[samp,], targets[samp,]), TRUE)$values
```

`predict.nnet` *Predict New Examples by a Trained Neural Net*

Description

Predict new examples by a trained neural net.

Usage

```
## S3 method for class 'nnet':
predict(object, newdata, type = c("raw","class"), ...)
```

Arguments

`object`	an object of class `nnet` as returned by `nnet`.
`newdata`	matrix or data frame of test examples. A vector is considered to be a row vector comprising a single case.
`type`	Type of output
`...`	arguments passed to or from other methods.

Details

This function is a method for the generic function `predict()` for class `"nnet"`. It can be invoked by calling `predict(x)` for an object `x` of the appropriate class, or directly by calling `predict.nnet(x)` regardless of the class of the object.

Value

If `type = "raw"`, the matrix of values returned by the trained network; if `type = "class"`, the corresponding class (which is probably only useful if the net was generated by `nnet.formula`).

References

Ripley, B. D. (1996) *Pattern Recognition and Neural Networks.* Cambridge.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`nnet`, `which.is.max`

Examples

```
data(iris3)
# use half the iris data
ir <- rbind(iris3[,,1], iris3[,,2], iris3[,,3])
targets <- class.ind( c(rep("s", 50), rep("c", 50), rep("v", 50)) )
samp <- c(sample(1:50,25), sample(51:100,25), sample(101:150,25))
ir1 <- nnet(ir[samp,], targets[samp,],size = 2, rang = 0.1,
            decay = 5e-4, maxit = 200)
test.cl <- function(true, pred){
```

```
        true <- max.col(true)
        cres <- max.col(pred)
        table(true, cres)
}
test.cl(targets[-samp,], predict(ir1, ir[-samp,]))

# or
ird <- data.frame(rbind(iris3[,,1], iris3[,,2], iris3[,,3]),
        species=factor(c(rep("s",50), rep("c", 50), rep("v", 50))))
ir.nn2 <- nnet(species ~ ., data = ird, subset = samp, size = 2, rang = 0.1,
            decay = 5e-4, maxit = 200)
table(ird$species[-samp], predict(ir.nn2, ird[-samp,], type = "class"))
```

`which.is.max` *Find Maximum Position in Vector*

Description

Find the maximum position in a vector, breaking ties at random.

Usage

```
which.is.max(x)
```

Arguments

`x`	a vector

Details

Ties are broken at random.

Value

index of a maximal value.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*. Fourth edition. Springer.

See Also

`max.col`, `which.max` which takes the first of ties.

Examples

```
## Not run:
pred <- predict(nnet, test)
table(true, apply(pred,1,which.is.max))
## End(Not run)
```

Chapter 20

The `rpart` package

`car.test.frame` *Automobile Data from 'Consumer Reports' 1990*

Description

The `car.test.frame` data frame has 60 rows and 8 columns, giving data on makes of cars taken from the April, 1990 issue of *Consumer Reports*. This is part of a larger dataset, some columns of which are given in `cu.summary`.

Usage

```
car.test.frame
```

Format

This data frame contains the following columns:

`Price` a numeric vector giving the list price in US dollars of a standard model

`Country` of origin, a factor with levels `France` `Germany` `Japan` `Japan/USA` `Korea` `Mexico` `Sweden` `USA`

`Reliability` a numeric vector coded `1` to `5`.

`Mileage` fuel consumption miles per US gallon, as tested.

`Type` a factor with levels `Compact` `Large` `Medium` `Small` `Sporty` `Van`

`Weight` kerb weight in pounds.

`Disp.` the engine capacity (displacement) in litres.

`HP` the net horsepower of the vehicle.

Source

Consumer Reports, April, 1990, pp. 235–288 quoted in

John M. Chambers and Trevor J. Hastie eds. (1992) *Statistical Models in S*, Wadsworth and Brooks/Cole, Pacific Grove, CA 1992, pp. 46–47.

See Also

cu.summary

Examples

```
z.auto <- rpart(Mileage ~ Weight, car.test.frame)
summary(z.auto)
```

---

cu.summary *Automobile Data from 'Consumer Reports' 1990*

Description

The cu.summary data frame has 117 rows and 5 columns, giving data on makes of cars taken from the April, 1990 issue of *Consumer Reports*.

Usage

```
cu.summary
```

Format

This data frame contains the following columns:

Price a numeric vector giving the list price in US dollars of a standard model

Country of origin, a factor with levels Brazil England France Germany Japan Japan/USA Korea Mexico Sweden USA

Reliability an ordered factor with levels Much worse < worse < average < better < Much better

Mileage fuel consumption miles per US gallon, as tested.

Type a factor with levels Compact Large Medium Small Sporty Van

Source

Consumer Reports, April, 1990, pp. 235–288 quoted in

John M. Chambers and Trevor J. Hastie eds. (1992) *Statistical Models in S*, Wadsworth and Brooks/Cole, Pacific Grove, CA 1992, pp. 46–47.

See Also

car.test.frame

Examples

```
fit <- rpart(Price ~ Mileage + Type + Country, cu.summary)
plot(fit, compress=TRUE)
text(fit, use.n=TRUE)
```

kyphosis — *Data on Children who have had Corrective Spinal Surgery*

Description

The kyphosis data frame has 81 rows and 4 columns. representing data on children who have had corrective spinal surgery

Usage

```
kyphosis
```

Format

This data frame contains the following columns:

Kyphosis a factor with levels absent present indicating if a kyphosis (a type of deformation) was present after the operation.

Age in months

Number the number of vertebrae involved

Start the number of the first (topmost) vertebra operated on.

Source

John M. Chambers and Trevor J. Hastie eds. (1992) *Statistical Models in S*, Wadsworth and Brooks/Cole, Pacific Grove, CA 1992.

Examples

```
fit <- rpart(Kyphosis ~ Age + Number + Start, data=kyphosis)
fit2 <- rpart(Kyphosis ~ Age + Number + Start, data=kyphosis,
              parms=list(prior=c(.65,.35), split='information'))
fit3 <- rpart(Kyphosis ~ Age + Number + Start, data=kyphosis,
              control=rpart.control(cp=.05))
par(mfrow=c(1,2))
plot(fit)
text(fit, use.n=TRUE)
plot(fit2)
text(fit2, use.n=TRUE)
```

labels.rpart — *Create Split Labels For an Rpart Object*

Description

This function provides labels for the branches of an rpart tree.

Usage

```
## S3 method for class 'rpart':
labels(object, digits=4, minlength=1, pretty, collapse=TRUE, ...)
```

Arguments

object	fitted model object of class rpart. This is assumed to be the result of some function that produces an object with the same named components as that returned by the rpart function.
digits	the number of digits to be used for numeric values. All of the rpart functions that call labels explicitly set this value, with options("digits") as the default.
minlength	the minimum length for abbreviation of character or factor variables. If 0 no abbreviation is done; if 1 then single letters are used with "a" for the first level, "b" for the second and so on. If the value is greater than 1, the abbreviate function is used.
pretty	an argument included for backwards compatibility: pretty=0 implies minlength=0, pretty=NULL implies minlength=1, and pretty=TRUE implies minlength=4.
collapse	logical. The returned set of labels is always of the same length as the number of nodes in the tree. If collapse=TRUE (default), the returned value is a vector of labels for the branch leading into each node, with "root" as the label for the top node. If FALSE, the returned value is a two column matrix of labels for the left and right branches leading out from each node, with "leaf" as the branch labels for terminal nodes.
...	optional arguments to abbreviate.

Value

Vector of split labels (collapse=TRUE) or matrix of left and right splits (collapse=FALSE) for the supplied rpart object. This function is called by printing methods for rpart and is not intended to be called directly by the users.

See Also

abbreviate

---

meanvar.rpart *Mean-Variance Plot for an Rpart Object*

Description

Creates a plot on the current graphics device of the deviance of the node divided by the number of observations at the node. Also returns the node number.

Usage

```
meanvar(tree, ...)

## S3 method for class 'rpart':
meanvar(tree, xlab="ave(y)", ylab="ave(deviance)", ...)
```

Arguments

tree	fitted model object of class rpart. This is asumed to be the result of some function that produces an object with the same named components as that returned by the rpart function.
xlab	x-axis label for the plot.
ylab	y-axis label for the plot.
...	additional graphical parameters may be supplied as arguments to this function.

Value

an invisible list containing the following vectors is returned.

x	fitted value at terminal nodes (yval).
y	deviance of node divided by number of observations at node.
label	node number.

Side Effects

a plot is put on the current graphics device.

See Also

plot.rpart.

Examples

```
z.auto <- rpart(Mileage ~ Weight, car.test.frame)
meanvar(z.auto, log='xy')
```

---

na.rpart *Handles Missing Values in an Rpart Object*

Description

Handles missing values in an rpart object.

Usage

```
na.rpart(x)
```

Arguments

x	a model frame.

Details

Internal function that handles missing values when calling the function rpart.

path.rpart *Follow Paths to Selected Nodes of an Rpart Object*

Description

Returns a names list where each element contains the splits on the path from the root to the selected nodes.

Usage

```
path.rpart(tree, nodes, pretty=0, print.it=TRUE)
```

Arguments

tree	fitted model object of class rpart. This is assumed to be the result of some function that produces an object with the same named components as that returned by the rpart function.
nodes	an integer vector containing indices (node numbers) of all nodes for which paths are desired. If missing, user selects nodes as described below.
pretty	an integer denoting the extent to which factor levels in split labels will be abbreviated. A value of (0) signifies no abbreviation. A NULL, the default, signifies using elements of letters to represent the different factor levels.
print.it	Logical. Denotes whether paths will be printed out as nodes are interactively selected. Irrelevant if nodes argument is supplied.

Details

The function has a required argument as an rpart object and a list of nodes as optional arguments. Omitting a list of nodes will cause the function to wait for the user to select nodes from the dendogram. It will return a list, with one component for each node specified or selected. The component contains the sequence of splits leading to that node. In the graphical interation, the individual paths are printed out as nodes are selected.

Value

A named (by node) list, each element of which contains all the splits on the path from the root to the specified or selected nodes.

Graphical Interaction

A dendrogram of the rpart object is expected to be visible on the graphics device, and a graphics input device (eg a mouse) is required. Clicking (the selection button) on a node selects that node. This process may be repeated any number of times. Clicking the exit button will stop the selection process and return the list of paths.

References

This function was modified from path.tree in S.

See Also

rpart

Examples

```
fit <- rpart(Kyphosis ~ Age + Number + Start, data=kyphosis)
summary(fit)
path.rpart(fit, node=c(11, 22))
```

`plot.rpart` *Plot an Rpart Object*

Description

Plots an rpart object on the current graphics device.

Usage

```
## S3 method for class 'rpart':
plot(x, uniform=FALSE, branch=1, compress=FALSE, nspace,
     margin=0, minbranch=.3, ...)
```

Arguments

`x`	a fitted object of class `rpart`, containing a classification, regression, or rate tree.
`uniform`	if `TRUE`, uniform vertical spacing of the nodes is used; this may be less cluttered when fitting a large plot onto a page. The default is to use a non-uniform spacing proportional to the error in the fit.
`branch`	controls the shape of the branches from parent to child node. Any number from 0 to 1 is allowed. A value of 1 gives square shouldered branches, a value of 0 give V shaped branches, with other values being intermediate.
`compress`	if `FALSE`, the leaf nodes will be at the horzontal plot coordinates of `1:nleaves`. If `TRUE`, the routine attempts a more compact arrangement of the tree. The compaction algorithm assumes `uniform=TRUE`; surprisingly, the result is usually an improvement even when that is not the case.
`nspace`	the amount of extra space between a node with children and a leaf, as compared to the minimal space between leaves. Applies to compressed trees only. The default is the value of `branch`.
`margin`	an extra percentage of white space to leave around the borders of the tree. (Long labels sometimes get cut off by the default computation).
`minbranch`	set the minimum length for a branch to `minbranch` times the average branch length. This parameter is ignored if `uniform=TRUE`. Sometimes a split will give very little improvement, or even (in the classification case) no improvement at all. A tree with branch lengths strictly proportional to improvement leaves no room to squeeze in node labels.
`...`	arguments to be passed to or from other methods.

Details

This function is a method for the generic function `plot`, for objects of class `rpart`. The y-coordinate of the top node of the tree will always be 1.

Value

the coordinates of the nodes are returned as a list, with components `x` and `y`.

Side Effects

an unlabeled plot is produced on the current graphics device.

See Also

`rpart`, `text.rpart`

Examples

```
fit <- rpart(Price ~ Mileage + Type + Country, cu.summary)
plot(fit, compress=TRUE)
text(fit, use.n=TRUE)
```

---

plotcp *Plot a Complexity Parameter Table for an Rpart Fit*

Description

Gives a visual representation of the cross-validation results in an `rpart` object.

Usage

```
plotcp(x, minline = TRUE, lty = 3, col = 1,
       upper = c("size", "splits", "none"), ...)
```

Arguments

`x`	an object of class `rpart`
`minline`	whether a horizontal line is drawn 1SE above the minimum of the curve.
`lty`	line type for this line
`col`	colour for this line
`upper`	what is plotted on the top axis: the size of the tree (the number of leaves), the number of splits or nothing.
`...`	additional plotting parameters

Details

The set of possible cost-complexity prunings of a tree from a nested set. For the geometric means of the intervals of values of `cp` for which a pruning is optimal, a cross-validation has (usually) been done in the initial construction by `rpart`. The `cptable` in the fit contains the mean and standard deviation of the errors in the cross-validated prediction against each of the geometric means, and these are plotted by this function. A good choice of `cp` for pruning is often the leftmost value for which the mean lies below the horizontal line.

Value

None.

Side Effects

A plot is produced on the current graphical device.

See Also

rpart, printcp, rpart.object

---

post.rpart *PostScript Presentation Plot of an Rpart Object*

Description

Generates a PostScript presentation plot of an rpart object.

Usage

```
post(tree, ...)

## S3 method for class 'rpart':
post(tree, title.,
     filename = paste(deparse(substitute(tree)), ".ps", sep = ""),
     digits = getOption("digits") - 3, pretty = TRUE,
     use.n = TRUE, horizontal = TRUE, ...)
```

Arguments

tree	fitted model object of class rpart. This is assumed to be the result of some function that produces an object with the same named components as that returned by the rpart function.
title.	a title which appears at the top of the plot. By default, the name of the rpart endpoint is printed out.
filename	ASCII file to contain the output. By default, the name of the file is the name of the object given by rpart (with the suffix .ps added). If filename = "", the plot appears on the current graphical device.
digits	number of significant digits to include in numerical data.
pretty	an integer denoting the extent to which factor levels will be abbreviated in the character strings defining the splits; (0) signifies no abbreviation of levels. A NULL signifies using elements of letters to represent the different factor levels. The default (TRUE) indicates the maximum possible abbreviation.
use.n	Logical. If TRUE (default), adds to label (#events level1/ #events level2/etc. for method class, n for method anova, and #events/n for methods poisson and exp).
horizontal	Logical. If TRUE (default), plot is horizontal. If FALSE, plot appears as landscape.
...	other arguments to the postscript function.

Details

The plot created uses the functions `plot.rpart` and `text.rpart` (with the `fancy` option). The settings were chosen because they looked good to us, but other options may be better, depending on the `rpart` object. Users are encouraged to write their own function containing favorite options.

Side Effects

a plot of `rpart` is created using the `postscript` driver, or the current device if `filename = ""`.

See Also

`plot.rpart`, `rpart`, `text.rpart`, `abbreviate`

Examples

```
z.auto <- rpart(Mileage ~ Weight, car.test.frame)
post(z.auto, file = "")   # display tree on active device
   # now construct postscript version on file "pretty.ps"
   # with no title
post(z.auto, file = "pretty.ps", title = " ")
z.hp <- rpart(Mileage ~ Weight + HP, car.test.frame)
post(z.hp)
```

`predict.rpart` *Predictions from a Fitted Rpart Object*

Description

Returns a vector of predicted responses from a fitted `rpart` object.

Usage

```
## S3 method for class 'rpart':
predict(object, newdata = list(),
        type = c("vector", "prob", "class", "matrix"),
        na.action = na.pass, ...)
```

Arguments

`object`	fitted model object of class `rpart`. This is assumed to be the result of some function that produces an object with the same named components as that returned by the `rpart` function.
`newdata`	data frame containing the values at which predictions are required. The predictors referred to in the right side of `formula(object)` must be present by name in `newdata`. If missing, the fitted values are returned.
`type`	character string denoting the type of predicted value returned. If the `rpart` object is a classification tree, then the default is to return `prob` predictions, a matrix whose columns are the probability of the first, second, etc. class. (This agrees with the default behavior of `tree`). Otherwise, a vector result is returned.

na.action	a function to determine what should be done with missing values in newdata. The default is to pass them down the tree using surrogates in the way selected when the model was built. Other possibilities are na.omit and na.fail.
...	further arguments passed to or from other methods.

Details

This function is a method for the generic function predict for class rpart. It can be invoked by calling predict for an object of the appropriate class, or directly by calling predict.rpart regardless of the class of the object.

Value

A new object is obtained by dropping newdata down the object. For factor predictors, if an observation contains a level not used to grow the tree, it is left at the deepest possible node and frame$yval at the node is the prediction.

If type="vector":
vector of predicted responses. For regression trees this is the mean response at the node, for Poisson trees it is the estimated response rate, and for classification trees it is the predicted class (as a number).

If type="prob":
(for a classification tree) a matrix of class probabilities.

If type="matrix":
a matrix of the full responses (frame$yval2 if this exists, otherwise frame$yval). For regression trees, this is the mean response, for Poisson trees it is the response rate and the number of events at that node in the fitted tree, and for classification trees it is the concatonation of the predicted class, the class counts at that node in the fitted tree, and the class probabilities.

If type="class":
(for a classification tree) a factor of classifications based on the responses.

See Also

predict, rpart.object

Examples

```
z.auto <- rpart(Mileage ~ Weight, car.test.frame)
predict(z.auto)

fit <- rpart(Kyphosis ~ Age + Number + Start, data=kyphosis)
predict(fit, type="prob")   # class probabilities (default)
predict(fit, type="vector") # level numbers
predict(fit, type="class")  # factor
predict(fit, type="matrix") # level number, class frequencies, probabilities

sub <- c(sample(1:50, 25), sample(51:100, 25), sample(101:150, 25))
fit <- rpart(Species ~ ., data=iris, subset=sub)
fit
table(predict(fit, iris[-sub,], type="class"), iris[-sub, "Species"])
```

`print.rpart` *Print an Rpart Object*

Description

This function prints an `rpart` object. It is a method for the generic function `print` of class `rpart`.

Usage

```
## S3 method for class 'rpart':
print(x, minlength=0, spaces=2, cp, digits= getOption("digits"), ...)
```

Arguments

`x`	fitted model object of class `rpart`. This is assumed to be the result of some function that produces an object with the same named components as that returned by the `rpart` function.
`minlength`	Controls the abbreviation of labels: see `labels.rpart`.
`spaces`	the number of spaces to indent nodes of increasing depth.
`digits`	the number of digits of numbers to print.
`cp`	prune all nodes with a complexity less than `cp` from the printout. Ignored if unspecified.
`...`	arguments to be passed to or from other methods.

Details

This function is a method for the generic function `print` for class `"rpart"`. It can be invoked by calling print for an object of the appropriate class, or directly by calling `print.rpart` regardless of the class of the object.

Side Effects

A semi-graphical layout of the contents of `x$frame` is printed. Indentation is used to convey the tree topology. Information for each node includes the node number, split, size, deviance, and fitted value. For the `"class"` method, the class probabilties are also printed.

See Also

`print`, `rpart.object`, `summary.rpart`, `printcp`

Examples

```
z.auto <- rpart(Mileage ~ Weight, car.test.frame)
z.auto
## Not run: node), split, n, deviance, yval
      * denotes terminal node

 1) root 60 1354.58300 24.58333
   2) Weight>=2567.5 45  361.20000 22.46667
     4) Weight>=3087.5 22   61.31818 20.40909 *
```

```
      5) Weight<3087.5 23  117.65220 24.43478
       10) Weight>=2747.5 15   60.40000 23.80000 *
       11) Weight<2747.5 8   39.87500 25.62500 *
    3) Weight<2567.5 15  186.93330 30.93333 *
## End(Not run)
```

`printcp` *Displays CP table for Fitted Rpart Object*

Description

Displays the `cp` table for fitted `rpart` object.

Usage

```
printcp(x, digits=getOption("digits") - 2)
```

Arguments

`x`	fitted model object of class `rpart`. This is assumed to be the result of some function that produces an object with the same named components as that returned by the `rpart` function.
`digits`	the number of digits of numbers to print.

Details

Prints a table of optimal prunings based on a complexity parameter.

See Also

`summary.rpart`, `rpart.object`

Examples

```
z.auto <- rpart(Mileage ~ Weight, car.test.frame)
printcp(z.auto)
## Not run:
Regression tree:
rpart(formula = Mileage ~ Weight, data = car.test.frame)

Variables actually used in tree construction:
[1] Weight

Root node error: 1354.6/60 = 22.576

        CP nsplit rel error  xerror     xstd
1 0.595349      0   1.00000 1.03436 0.178526
2 0.134528      1   0.40465 0.60508 0.105217
3 0.012828      2   0.27012 0.45153 0.083330
4 0.010000      3   0.25729 0.44826 0.076998
## End(Not run)
```

prune.rpart *Cost-complexity Pruning of an Rpart Object*

Description

Determines a nested sequence of subtrees of the supplied `rpart` object by recursively `snipping` off the least important splits, based on the complexity parameter (`cp`).

Usage

```
prune(tree, ...)

## S3 method for class 'rpart':
prune(tree, cp, ...)
```

Arguments

`tree`	fitted model object of class `rpart`. This is assumed to be the result of some function that produces an object with the same named components as that returned by the `rpart` function.
`cp`	Complexity parameter to which the `rpart` object will be trimmed.
`...`	further arguments passed to or from other methods.

Value

A new `rpart` object that is trimmed to the value `cp`.

See Also

`rpart`

Examples

```
z.auto <- rpart(Mileage ~ Weight, car.test.frame)
zp <- prune(z.auto, cp=0.1)
plot(zp) #plot smaller rpart object
```

residuals.rpart *Residuals From a Fitted Rpart Object*

Description

Method for `residuals` for an `rpart` object.

Usage

```
## S3 method for class 'rpart':
residuals(object, type = c("usual", "pearson", "deviance"), ...)
```

Arguments

object	fitted model object of class "rpart".
type	Indicates the type of residual desired. For regression or anova trees all three residual definitions reduce to y - fitted. This is the residual returned for user method trees as well. For classification trees the usual residuals are the missclassification losses L(actual, predicted) where L is the loss matrix. With default losses this residual is 0/1 for correct/incorrect classification. The pearson residual is (1-fitted)/sqrt(fitted(1-fitted)) and the deviance residual is sqrt(minus twice logarithm of fitted). For poisson and exp (or survival) trees, the usual residual is the observed - expected number of events. The pearson and deviance residuals are as defined in McCullagh and Nelder.
...	further arguments passed to or from other methods.

Value

vector of residuals of type type from a fitted rpart object.

References

McCullagh P. and Nelder, J. A. (1989) *Generalized Linear Models*. London: Chapman and Hall.

Examples

```
fit <- rpart(skips ~ Opening + Solder + Mask + PadType + Panel,
       data=solder, method='anova')
summary(residuals(fit))
plot(predict(fit),residuals(fit))
```

---

rpart *Recursive Partitioning and Regression Trees*

Description

Fit a rpart model

Usage

```
rpart(formula, data, weights, subset, na.action = na.rpart, method,
      model = FALSE, x = FALSE, y = TRUE, parms, control, cost, ...)
```

Arguments

formula	a formula, as in the lm function.
data	an optional data frame in which to interpret the variables named in the formula
weights	optional case weights.
subset	optional expression saying that only a subset of the rows of the data should be used in the fit.

na.action	The default action deletes all observations for which y is missing, but keeps those in which one or more predictors are missing.
method	one of "anova", "poisson", "class" or "exp". If method is missing then the routine tries to make an intellegent guess. If y is a survival object, then method="exp" is assumed, if y has 2 columns then method="poisson" is assumed, if y is a factor then method="class" is assumed, otherwise method="anova" is assumed. It is wisest to specifiy the method directly, especially as more criteria are added to the function. Alternatively, method can be a list of functions named init, split and eval.
model	if logical: keep a copy of the model frame in the result? If the input value for model is a model frame (likely from an earlier call to the rpart function), then this frame is used rather than constructing new data.
x	keep a copy of the x matrix in the result.
y	keep a copy of the dependent variable in the result. If missing and model is supplied this defaults to FALSE.
parms	optional parameters for the splitting function. Anova splitting has no parameters. Poisson splitting has a single parameter, the coefficient of variation of the prior distribution on the rates. The default value is 1. Exponential splitting has the same parameter as Poisson. For classification splitting, the list can contain any of: the vector of prior probabilities (component prior), the loss matrix (component loss) or the splitting index (component split). The priors must be positive and sum to 1. The loss matrix must have zeros on the diagnoal and positive off-diagonal elements. The splitting index can be gini or information. The default priors are proportional to the data counts, the losses default to 1, and the split defaults to gini.
control	options that control details of the rpart algorithm.
cost	a vector of non-negative costs, one for each variable in the model. Defaults to one for all variables. These are scalings to be applied when considering splits, so the improvement on splitting on a variable is divided by its cost in deciding which split to choose.
...	arguments to rpart.control may also be specified in the call to rpart. They are checked against the list of valid arguments.

Details

This differs from the tree function mainly in its handling of surrogate variables. In most details it follows Breiman et. al. quite closely.

Value

an object of class rpart, a superset of class tree.

References

Breiman, Friedman, Olshen, and Stone. (1984) *Classification and Regression Trees.* Wadsworth.

See Also

rpart.control, rpart.object, summary.rpart, print.rpart

Examples

```
fit <- rpart(Kyphosis ~ Age + Number + Start, data=kyphosis)
fit2 <- rpart(Kyphosis ~ Age + Number + Start, data=kyphosis,
              parms=list(prior=c(.65,.35), split='information'))
fit3 <- rpart(Kyphosis ~ Age + Number + Start, data=kyphosis,
              control=rpart.control(cp=.05))
par(mfrow=c(1,2), xpd=NA) # otherwise on some devices the text is clipped
plot(fit)
text(fit, use.n=TRUE)
plot(fit2)
text(fit2, use.n=TRUE)
```

`rpart.control` *Control for Rpart Models*

Description

Various parameters that control aspects of the `rpart` fit.

Usage

```
rpart.control(minsplit=20, minbucket=round(minsplit/3), cp=0.01,
              maxcompete=4, maxsurrogate=5, usesurrogate=2, xval=10,
              surrogatestyle=0, maxdepth=30, ...)
```

Arguments

`minsplit`	the minimum number of observations that must exist in a node, in order for a split to be attempted.
`minbucket`	the minimum number of observations in any terminal `<leaf>` node. If only one of `minbucket` or `minsplit` is specified, the code either sets `minsplit` to `minbucket*3` or `minbucket` to `minsplit/3`, as appropriate.
`cp`	complexity parameter. Any split that does not decrease the overall lack of fit by a factor of `cp` is not attempted. For instance, with `anova` splitting, this means that the overall Rsquare must increase by `cp` at each step. The main role of this parameter is to save computing time by pruning off splits that are obviously not worthwhile. Essentially,the user informs the program that any split which does not improve the fit by `cp` will likely be pruned off by cross-validation, and that hence the program need not pursue it.
`maxcompete`	the number of competitor splits retained in the output. It is useful to know not just which split was chosen, but which variable came in second, third, etc.
`maxsurrogate`	the number of surrogate splits retained in the output. If this is set to zero the compute time will be shortened, since approximately half of the computational time (other than setup) is used in the search for surrogate splits.
`usesurrogate`	how to use surrogates in the splitting process. 0= display only; an observation with a missing value for the primary split rule is not sent further down the tree. 1= use surrogates, in order, to split subjects missing the primary variable; if all surrogates are missing the observation is not split. 2= if all surrogates are missing, then send the observation in the majority direction. A value of 0 corresponds to the action of `tree`, and 2 to the recommendations of Breiman, et.al.

`xval` number of cross-validations

`surrogatestyle`
controls the selection of a best surrogate. If set to 0 (default) the program uses the total number of correct classification for a potential surrogate variable, if set to 1 it uses the percent correct, calculated over the non-missing values of the surrogate. The first option more severely penalizes covariates with a large number of missing values.

`maxdepth` Set the maximum depth of any node of the final tree, with the root node counted as depth 0 (past 30 `rpart` will give nonsense results on 32-bit machines).

`...` mop up other arguments.

Value

a list containing the options.

See Also

`rpart`

`rpart.object` *Recursive Partitioning and Regression Trees Object*

Description

These are objects representing fitted `rpart` trees.

Value

`frame` data frame with one row for each node in the tree. The `row.names` of `frame` contain the (unique) node numbers that follow a binary ordering indexed by node depth. Elements of `frame` include `var`, the variable used in the split at each node (leaf nodes are denoted by the string `<leaf>`), `n`, the size of each node, `wt`, the sum of case weights for the node, `dev`, the deviance of each node, `yval`, the fitted value of the response at each node, and `splits`, a two column matrix of left and right split labels for each node. All of these are the same as for an `rpart` object.

Extra response information is in `yval2`, which contains the number of events at the node (poisson), or a matrix containing the fitted class, the class counts for each node and the class probabilities (classification). Also included in the frame are `complexity`, the complexity parameter at which this split will collapse, `ncompete`, the number of competitor splits retained, and `nsurrogate`, the number of surrogate splits retained.

`where` vector, the same length as the number of observations in the root node, containing the row number of `frame` corresponding to the leaf node that each observation falls into.

`splits` a matrix describing the splits. The row label is the name of the split variable, and columns are `count`, the number of observations sent left or right by the split (for competitor splits this is the number that would have been sent left or right had this split been used, for surrogate splits it is the number missing the primary split variable which were decided using this surrogate), `ncat`, the

number of categories or levels for the variable (+/−1 for a continuous variable), improve, which is the improvement in deviance given by this split, or, for surrogates, the concordance of the surrogate with the primary, and split, the numeric split point. The last column adj gives the adjusted concordance for surrogate splits. For a factor, the split column contains the row number of the csplit matrix. For a continuous variable, the sign of ncat determines whether the subset x < cutpoint or x > cutpoint is sent to the left.

csplit — this will be present only if one of the split variables is a factor. There is one row for each such split, and column i = 1 if this level of the factor goes to the left, 3 if it goes to the right, and 2 if that level is not present at this node of the tree. For an ordered categorical variable all levels are marked as R/L, including levels that are not present.

method — the method used to grow the tree.

cptable — the table of optimal prunings based on a complexity parameter.

terms — an object of mode expression and class term summarizing the formula. Used by various methods, but typically not of direct relevance to users.

call — an image of the call that produced the object, but with the arguments all named and with the actual formula included as the formula argument. To re-evaluate the call, say update(tree).

Optional components include the matrix of predictors (x) and the response variable (y) used to construct the rpart object.

Structure

The following components must be included in a legitimate rpart object. Of these, only the where component has the same length as the data used to fit the rpart object.

See Also

rpart.

---

rpconvert *Update an rpart object*

Description

Rpart objects changed (slightly) in their internal format in order to accomdate the changes for user-written split functions. This routine updates an old object to the new format.

Usage

```
rpconvert(x)
```

Arguments

x — an rpart object

Value

an updated object

See Also

rpart

---

rsq.rpart *Plots the Approximate R-Square for the Different Splits*

Description

Produces 2 plots. The first plots the r-square (apparent and apparent - from cross-validation) versus the number of splits. The second plots the Relative Error(cross-validation) +/- 1-SE from cross-validation versus the number of splits.

Usage

```
rsq.rpart(x)
```

Arguments

x	fitted model object of class rpart. This is assumed to be the result of some function that produces an object with the same named components as that returned by the rpart function.

Side Effects

Two plots are produced.

Note

The labels are only appropriate for the "anova" method.

Examples

```
z.auto <- rpart(Mileage ~ Weight, car.test.frame)
rsq.rpart(z.auto)
```

---

snip.rpart *Snip Subtrees of an Rpart Object*

Description

Creates a "snipped" rpart object, containing the nodes that remain after selected subtrees have been snipped off. The user can snip nodes using the toss arguement, or interactively by clicking the mouse button on specified nodes within the graphics window.

Usage

```
snip.rpart(x, toss)
```

Arguments

x	fitted model object of class rpart. This is assumed to be the result of some function that produces an object with the same named components as that returned by the rpart function.
toss	an integer vector containing indices (node numbers) of all subtrees to be snipped off. If missing, user selects branches to snip off as described below.

Details

A dendrogram of rpart is expected to be visible on the graphics device, and a graphics input device (e.g., a mouse) is required. Clicking (the selection button) on a node displays the node number, sample size, response yvalue, and Error (dev). Clicking a second time on the same node snips that subtree off and visually erases the subtree. This process may be repeated an number of times. Warnings result from selecting the root or leaf nodes. Clicking the exit button will stop the snipping process and return the resulting rpart object.

See the documentation for the specific graphics device for details on graphical input techniques.

Value

a rpart object containing the nodes that remain after specified or selected subtrees have been snipped off.

Warning

Visually erasing the plot is done by over-plotting with the background colour. This will do nothing if the background is transparent (often true for screen devices).

See Also

plot.rpart

Examples

```
## dataset not in R
## Not run:
z.survey <- rpart(market.survey) #grow the rpart object
plot(z.survey) #plot the tree
z.survey2 <- snip.rpart(z.survey,toss=2) #trim subtree at node 2
plot(z.survey2) #plot new tree

# can also interactively select the node using the mouse in the
# graphics window
## End(Not run)
```

---

solder *Soldering of Components on Printed-Circuit Boards*

Description

The solder data frame has 720 rows and 6 columns, representing a balanced subset of a designed experiment varying 5 factors on the soldering of components on printed-circuit boards.

Usage

```
solder
```

Format

This data frame contains the following columns:

Opening a factor with levels `L M S` indicating the amount of clearance around the mounting pad.

Solder a factor with levels `Thick Thin` giving the thickness of the solder used.

Mask a factor with levels `A1.5 A3 B3 B6` indicating the type and thickness of mask used.

PadType a factor with levels `D4 D6 D7 L4 L6 L7 L8 L9 W4 W9` giving the size and geometry of the mounting pad.

Panel `1:3` indicating the panel on a board being tested.

skips a numeric vector giving the number of visible solder skips.

Source

John M. Chambers and Trevor J. Hastie eds. (1992) *Statistical Models in S*, Wadsworth and Brooks/Cole, Pacific Grove, CA 1992.

Examples

```
fit <- rpart(skips ~ Opening + Solder + Mask + PadType + Panel,
       data=solder, method='anova')
summary(residuals(fit))
plot(predict(fit),residuals(fit))
```

---

summary.rpart *Summarize a Fitted Rpart Object*

Description

Returns a detailed listing of a fitted `rpart` object.

Usage

```
## S3 method for class 'rpart':
summary(object, cp=0, digits=getOption("digits"), file, ...)
```

Arguments

`object`	fitted model object of class `rpart`. This is assumed to be the result of some function that produces an object with the same named components as that returned by the `rpart` function.
`digits`	Number of significant digits to be used in the result.
`cp`	trim nodes with a complexity of less than `cp` from the listing.
`file`	write the output to a given file name. (Full listings of a tree are often quite long).
`...`	arguments to be passed to or from other methods.

Details

This function is a method for the generic function summary for class "rpart". It can be invoked by calling summary for an object of the appropriate class, or directly by calling summary.rpart regardless of the class of the object.

See Also

summary, rpart.object, printcp.

Examples

```
z.auto <- rpart(Mileage ~ Weight, car.test.frame)
summary(z.auto)
```

---

text.rpart *Place Text on a Dendrogram*

Description

Labels the current plot of the tree dendrogram with text.

Usage

```
## S3 method for class 'rpart':
text(x, splits=TRUE, label="yval", FUN=text, all=FALSE,
     pretty=NULL, digits=getOption("digits") - 3, use.n=FALSE,
     fancy=FALSE, fwidth=0.8, fheight=0.8, ...)
```

Arguments

x	fitted model object of class rpart. This is assumed to be the result of some function that produces an object with the same named components as that returned by the rpart function.
splits	logical flag. If TRUE (default), then the splits in the tree are labeled with the criterion for the split.
label	a column name of x$frame; values of this will label the nodes. For the "class" method, label="yval" results in the factor levels being used, "yprob" results in the probability of the winning factor level being used, and 'specific yval level' results in the probability of that factor level.
FUN	the name of a labeling function, e.g. text.
all	Logical. If TRUE, all nodes are labeled, otherwise just terminal nodes.
pretty	an integer denoting the extent to which factor levels in split labels will be abbreviated. A value of (0) signifies no abbreviation. A NULL, the default, signifies using elements of letters to represent the different factor levels.
digits	number of significant digits to include in numerical labels.
use.n	Logical. If TRUE, adds to label (#events level1/ #events level2/etc. for class, n for anova, and #events/n for poisson and exp).

fancy	Logical. If TRUE, nodes are represented by ellipses (interior nodes) and rectangles (leaves) and labeled by yval. The edges connecting the nodes are labeled by left and right splits.
fwidth	Relates to option fancy and the width of the ellipses and rectangles. If fwidth < 1 then it is a scaling factor (default = 0.8). If fwidth > 1 then it representes the number of character widths (for current graphical device) to use.
fheight	Relates to option fancy and the height of the ellipses and rectangles. If fheight <1 then it is a scaling factor (default = 0.8). If fheight > 1 then it represents the number of character heights (for current graphical device) to use.
...	Graphical parameters may also be supplied as arguments to this function (see par). As labels often extend outside the plot region it can be helpful to specify xpd = TRUE.

Side Effects

the current plot of a tree dendrogram is labeled.

See Also

text, plot.rpart, rpart, post.rpart, abbreviate

Examples

```
freen.tr <- rpart(y ~ ., freeny)
plot(freen.tr)
text(freen.tr, use.n=TRUE, all=TRUE)
```

---

xpred.rpart *Return Cross-Validated Predictions*

Description

Gives the predicted values for an rpart fit, under cross validation, for a set of complexity parameter values.

Usage

```
xpred.rpart(fit, xval=10, cp)
```

Arguments

fit	a rpart object.
xval	number of cross-validation groups. This may also be an explicit list of integers that define the cross-validation groups.
cp	the desired list of complexity values. By default it is taken from the cptable component of the fit.

Details

Complexity penalties are actually ranges, not values. If the cp values found in the table were .36, .28, and .13, for instance, this means that the first row of the table holds for all complexity penalties in the range $[.36, 1]$, the second row for cp in the range $[.28, .36)$ and the third row for $[.13, .28)$. By default, the geometric mean of each interval is used for cross validation.

Value

a matrix with one row for each observation and one column for each complexity value.

See Also

rpart

Examples

```
fit <- rpart(Mileage ~ Weight, car.test.frame)
xmat <- xpred.rpart(fit)
xerr <- (xmat - car.test.frame$Mileage)^2
apply(xerr, 2, sum)   # cross-validated error estimate

# approx same result as rel. error from printcp(fit)
apply(xerr, 2, sum)/var(car.test.frame$Mileage)
printcp(fit)
```

Chapter 21

The `spatial` package

`anova.trls` *Anova tables for fitted trend surface objects*

Description

Compute analysis of variance tables for one or more fitted trend surface model objects; where `anova.trls` is called with multiple objects, it passes on the arguments to `anovalist.trls`.

Usage

```
## S3 method for class 'trls':
anova(object, ...)
anovalist.trls(object, ...)
```

Arguments

`object`	A fitted trend surface model object from `surf.ls`
`...`	Further objects of the same kind

Value

`anova.trls` and `anovalist.trls` return objects corresponding to their printed tabular output.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`surf.ls`

Examples

```
library(stats)
data(topo, package="MASS")
topo0 <- surf.ls(0, topo)
topo1 <- surf.ls(1, topo)
topo2 <- surf.ls(2, topo)
topo3 <- surf.ls(3, topo)
topo4 <- surf.ls(4, topo)
anova(topo0, topo1, topo2, topo3, topo4)
summary(topo4)
```

---

correlogram *Compute Spatial Correlograms*

Description

Compute spatial correlograms of spatial data or residuals.

Usage

```
correlogram(krig, nint, plotit = TRUE,  ...)
```

Arguments

krig	trend-surface or kriging object with columns x, y, and z
nint	number of bins used
plotit	logical for plotting
...	parameters for the plot

Details

Divides range of data into nint bins, and computes the covariance for pairs with separation in each bin, then divides by the variance. Returns results for bins with 6 or more pairs.

Value

x and y coordinates of the correlogram, and cnt, the number of pairs averaged per bin.

Side Effects

Plots the correlogram if plotit = TRUE.

References

Ripley, B. D. (1981) *Spatial Statistics.* Wiley.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

variogram

Examples

```
data(topo, package="MASS")
topo.kr <- surf.ls(2, topo)
correlogram(topo.kr, 25)
d <- seq(0, 7, 0.1)
lines(d, expcov(d, 0.7))
```

`expcov` *Spatial Covariance Functions*

Description

Spatial covariance functions for use with `surf.gls`.

Usage

```
expcov(r, d, alpha = 0, se = 1)
gaucov(r, d, alpha = 0, se = 1)
sphercov(r, d, alpha = 0, se = 1, D = 2)
```

Arguments

`r`	vector of distances at which to evaluate the covariance
`d`	range parameter
`alpha`	proportion of nugget effect
`se`	standard deviation at distance zero
`D`	dimension of spheres.

Value

vector of covariance values.

References

Ripley, B. D. (1981) *Spatial Statistics.* Wiley.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`surf.gls`

Examples

```
data(topo, package="MASS")
topo.kr <- surf.ls(2, topo)
correlogram(topo.kr, 25)
d <- seq(0, 7, 0.1)
lines(d, expcov(d, 0.7))
```

`Kaver` *Average K-functions from Simulations*

Description

Forms the average of a series of (usually simulated) K-functions.

Usage

```
Kaver(fs, nsim, ...)
```

Arguments

`fs`	full scale for K-fn
`nsim`	number of simulations
`...`	arguments to simulate one point process object

Value

list with components `x` and `y` of the average K-fn on L-scale.

References

Ripley, B. D. (1981) *Spatial Statistics.* Wiley.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`Kfn`, `Kenvl`

Examples

```
towns <- ppinit("towns.dat")
par(pty="s")
plot(Kfn(towns, 40), type="b")
plot(Kfn(towns, 10), type="b", xlab="distance", ylab="L(t)")
for(i in 1:10) lines(Kfn(Psim(69), 10))
lims <- Kenvl(10,100,Psim(69))
lines(lims$x,lims$l, lty=2, col="green")
lines(lims$x,lims$u, lty=2, col="green")
lines(Kaver(10,25,Strauss(69,0.5,3.5)),  col="red")
```

`Kenvl` *Compute Envelope and Average of Simulations of K-fns*

Description

Computes envelope (upper and lower limits) and average of simulations of K-fns

Usage

```
Kenvl(fs, nsim, ...)
```

Arguments

`fs`	full scale for K-fn
`nsim`	number of simulations
`...`	arguments to produce one simulation

Value

list with components

`x`	distances
`lower`	min of K-fns
`upper`	max of K-fns
`aver`	average of K-fns

References

Ripley, B. D. (1981) *Spatial Statistics.* Wiley.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`Kfn`, `Kaver`

Examples

```
towns <- ppinit("towns.dat")
par(pty="s")
plot(Kfn(towns, 40), type="b")
plot(Kfn(towns, 10), type="b", xlab="distance", ylab="L(t)")
for(i in 1:10) lines(Kfn(Psim(69), 10))
lims <- Kenvl(10,100,Psim(69))
lines(lims$x,lims$l, lty=2, col="green")
lines(lims$x,lims$u, lty=2, col="green")
lines(Kaver(10,25,Strauss(69,0.5,3.5)), col="red")
```

`Kfn` *Compute K-fn of a Point Pattern*

Description

Actually computes L = sqrt(K/pi).

Usage

```
Kfn(pp, fs, k=100)
```

Arguments

`pp`	a list such as a pp object, including components `x` and `y`
`fs`	full scale of the plot
`k`	number of regularly spaced distances in (0, `fs`)

Details

relies on the domain D having been set by `ppinit` or `ppregion`.

Value

A list with components

`x`	vector of distances
`y`	vector of L-fn values
`k`	number of distances returned – may be less than `k` if `fs` is too large
`dmin`	minimum distance between pair of points
`lm`	maximum deviation from L(t) = t

References

Ripley, B. D. (1981) *Spatial Statistics.* Wiley.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`ppinit`, `ppregion`, `Kaver`, `Kenvl`

Examples

```
towns <- ppinit("towns.dat")
par(pty="s")
plot(Kfn(towns, 10), type="s", xlab="distance", ylab="L(t)")
```

ppgetregion *Get Domain for Spatial Point Pattern Analyses*

Description

Retrieves the rectangular domain `(xl, xu) x (yl, yu)` from the underlying `C` code.

Usage

```
ppgetregion()
```

Value

A vector of length four with names `c("xl", "xu", "yl", "yu")`.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`ppregion`

ppinit *Read a Point Process Object from a File*

Description

Read a file in standard format and create a point process object.

Usage

```
ppinit(file)
```

Arguments

`file`	string giving file name

Details

The file should contain

the number of points
a header (ignored)
xl xu yl yu scale
x y (repeated n times)

Value

class `"pp"` object with components `x`, `y`, `xl`, `xu`, `yl`, `yu`

Side Effects

Calls `ppregion` to set the domain.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`ppregion`

Examples

```
towns <- ppinit("towns.dat")
par(pty="s")
plot(Kfn(towns, 10), type="b", xlab="distance", ylab="L(t)")
```

---

pplik *Pseudo-likelihood Estimation of a Strauss Spatial Point Process*

Description

Pseudo-likelihood estimation of a Strauss spatial point process.

Usage

```
pplik(pp, R, ng=50, trace=FALSE)
```

Arguments

`pp`	a pp object
`R`	the fixed parameter `R`
`ng`	use a `ng` x `ng` grid with border `R` in the domain for numerical integration.
`trace`	logical? Should function evaluations be printed?

Value

estimate for `c` in the interval `[0, 1]`.

References

Ripley, B. D. (1988) *Statistical Inference for Spatial Processes.* Cambridge.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`Strauss`

Examples

```
pines <- ppinit("pines.dat")
pplik(pines, 0.7)
```

ppregion *Set Domain for Spatial Point Pattern Analyses*

Description

Sets the rectangular domain `(xl, xu)` x `(yl, yu)`.

Usage

```
ppregion(xl = 0, xu = 1, yl = 0, yu = 1)
```

Arguments

`xl`	Either `xl` or a list containing components `xl`, `xu`, `yl`, `yu` (such as a point-process object)
`xu`	
`yl`	
`yu`	

Value

none

Side Effects

initializes variables in the `C` subroutines.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`ppinit`, `ppgetregion`

predict.trls *Predict method for trend surface fits*

Description

Predicted values based on trend surface model object

Usage

```
## S3 method for class 'trls':
predict(object, x, y, ...)
```

Arguments

object	Fitted trend surface model object returned by surf.ls
x	Vector of prediction location eastings (x coordinates)
y	Vector of prediction location northings (y coordinates)
...	further arguments passed to or from other methods.

Value

predict.trls produces a vector of predictions corresponding to the prediction locations. To display the output with image or contour, use trmat or convert the returned vector to matrix form.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

surf.ls, trmat

Examples

```
data(topo, package="MASS")
topo2 <- surf.ls(2, topo)
topo4 <- surf.ls(4, topo)
x <- c(1.78, 2.21)
y <- c(6.15, 6.15)
z2 <- predict(topo2, x, y)
z4 <- predict(topo4, x, y)
cat("2nd order predictions:", z2, "\n4th order predictions:", z4, "\n")
```

---

prmat *Evaluate Kriging Surface over a Grid*

Description

Evaluate Kriging surface over a grid.

Usage

```
prmat(obj, xl, xu, yl, yu, n)
```

Arguments

obj	object returned by surf.gls
xl	limits of the rectangle for grid
xu	
yl	
yu	
n	use n x n grid within the rectangle

Value

list with components `x`, `y` and `z` suitable for `contour` and `image`.

References

Ripley, B. D. (1981) *Spatial Statistics.* Wiley.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`surf.gls`, `trmat`, `semat`

Examples

```
data(topo, package="MASS")
topo.kr <- surf.gls(2, expcov, topo, d=0.7)
prsurf <- prmat(topo.kr, 0, 6.5, 0, 6.5, 50)
contour(prsurf, levels=seq(700, 925, 25))
```

`Psim` *Simulate Binomial Spatial Point Process*

Description

Simulate Binomial spatial point process.

Usage

```
Psim(n)
```

Arguments

`n`	number of points

Details

relies on the region being set by `ppinit` or `ppregion`.

Value

list of vectors of `x` and `y` coordinates.

Side Effects

uses the random number generator.

References

Ripley, B. D. (1981) *Spatial Statistics.* Wiley.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

SSI, Strauss

Examples

```
towns <- ppinit("towns.dat")
par(pty="s")
plot(Kfn(towns, 10), type="s", xlab="distance", ylab="L(t)")
for(i in 1:10) lines(Kfn(Psim(69), 10))
```

---

semat *Evaluate Kriging Standard Error of Prediction over a Grid*

Description

Evaluate Kriging standard error of prediction over a grid.

Usage

```
semat(obj, xl, xu, yl, yu, n, se)
```

Arguments

obj	object returned by surf.gls
xl	limits of the rectangle for grid
xu	
yl	
yu	
n	use n x n grid within the rectangle
se	standard error at distance zero as a multiple of the supplied covariance. Otherwise estimated, and it assumed that a correlation function was supplied.

Value

list with components x, y and z suitable for contour and image.

References

Ripley, B. D. (1981) *Spatial Statistics.* Wiley.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

surf.gls, trmat, prmat

Examples

```
data(topo, package="MASS")
topo.kr <- surf.gls(2, expcov, topo, d=0.7)
prsurf <- prmat(topo.kr, 0, 6.5, 0, 6.5, 50)
contour(prsurf, levels=seq(700, 925, 25))
sesurf <- semat(topo.kr, 0, 6.5, 0, 6.5, 30)
contour(sesurf, levels=c(22,25))
```

SSI *Simulates Sequential Spatial Inhibition Point Process*

Description

Simulates SSI (sequential spatial inhibition) point process.

Usage

```
SSI(n, r)
```

Arguments

n	number of points
r	inhibition distance

Details

uses the region set by `ppinit` or `ppregion`.

Value

list of vectors of `x` and `y` coordinates

Side Effects

uses the random number generator.

Warnings

will never return if `r` is too large and it cannot place `n` points.

References

Ripley, B. D. (1981) *Spatial Statistics.* Wiley.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`Psim`, `Strauss`

Examples

```
towns <- ppinit("towns.dat")
par(pty = "s")
plot(Kfn(towns, 10), type = "b", xlab = "distance", ylab = "L(t)")
lines(Kaver(10, 25, SSI(69, 1.2)))
```

Strauss *Simulates Strauss Spatial Point Process*

Description

Simulates Strauss spatial point process.

Usage

```
Strauss(n, c=0, r)
```

Arguments

n	number of points
c	parameter c in [0, 1]. c = 0 corresponds to complete inhibition at distances up to r.
r	inhibition distance

Details

Uses spatial birth-and-death process for 4n steps, or for 40n steps starting from a binomial pattern on the first call from an other function. Uses the region set by ppinit or ppregion.

Value

list of vectors of x and y coordinates

Side Effects

uses the random number generator

References

Ripley, B. D. (1981) *Spatial Statistics.* Wiley.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

Psim, SSI

Examples

```
towns <- ppinit("towns.dat")
par(pty="s")
plot(Kfn(towns, 10), type="b", xlab="distance", ylab="L(t)")
lines(Kaver(10, 25, Strauss(69,0.5,3.5)))
```

`surf.gls` *Fits a Trend Surface by Generalized Least-squares*

Description

Fits a trend surface by generalized least-squares.

Usage

```
surf.gls(np, covmod, x, y, z, nx = 1000, ...)
```

Arguments

`np`	degree of polynomial surface
`covmod`	function to evaluate covariance or correlation function
`x`	x coordinates or a data frame with columns `x`, `y`, `z`
`y`	y coordinates
`z`	z coordinates. Will supersede `x$z`
`nx`	Number of bins for table of the covariance. Increasing adds accuracy, and increases size of the object.
`...`	parameters for `covmod`

Value

list with components

`beta`	the coefficients
`x`	
`y`	
`z`	and others for internal use only.

References

Ripley, B. D. (1981) *Spatial Statistics.* Wiley.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`trmat`, `surf.ls`, `prmat`, `semat`, `expcov`, `gaucov`, `sphercov`

Examples

```
library(MASS)  # for eqscplot
data(topo, package="MASS")
topo.kr <- surf.gls(2, expcov, topo, d=0.7)
trsurf <- trmat(topo.kr, 0, 6.5, 0, 6.5, 50)
eqscplot(trsurf, type = "n")
contour(trsurf, add = TRUE)

prsurf <- prmat(topo.kr, 0, 6.5, 0, 6.5, 50)
```

```
contour(prsurf, levels=seq(700, 925, 25))
sesurf <- semat(topo.kr, 0, 6.5, 0, 6.5, 30)
eqscplot(sesurf, type = "n")
contour(sesurf, levels = c(22, 25), add = TRUE)
```

`surf.ls` *Fits a Trend Surface by Least-squares*

Description

Fits a trend surface by least-squares.

Usage

```
surf.ls(np, x, y, z)
```

Arguments

`np`	degree of polynomial surface
`x`	x coordinates or a data frame with columns `x`, `y`, `z`
`y`	y coordinates
`z`	z coordinates. Will supersede `x$z`

Value

list with components

`beta`	the coefficients
`x`	
`y`	
`z`	and others for internal use only.

References

Ripley, B. D. (1981) *Spatial Statistics.* Wiley.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`trmat`, `surf.gls`

Examples

```
library(MASS)  # for eqscplot
data(topo, package="MASS")
topo.kr <- surf.ls(2, topo)
trsurf <- trmat(topo.kr, 0, 6.5, 0, 6.5, 50)
eqscplot(trsurf, type = "n")
contour(trsurf, add = TRUE)
points(topo)

eqscplot(trsurf, type = "n")
contour(trsurf, add = TRUE)
plot(topo.kr, add = TRUE)
title(xlab= "Circle radius proportional to Cook's influence statistic")
```

---

trls.influence *Regression diagnostics for trend surfaces*

Description

This function provides the basic quantities which are used in forming a variety of diagnostics for checking the quality of regression fits for trend surfaces calculated by `surf.ls`.

Usage

```
trls.influence(object)
## S3 method for class 'trls':
plot(x, border = "red", col = NA, pch = 4, cex = 0.6,
     add = FALSE, div = 8, ...)
```

Arguments

`object, x`	Fitted trend surface model from `surf.ls`
`div`	scaling factor for influence circle radii in `plot.trls`
`add`	add influence plot to existing graphics if `TRUE`
`border, col, pch, cex, ...`	additional graphical parameters

Value

`trls.influence` returns a list with components:

`r`	raw residuals as given by `residuals.trls`
`hii`	diagonal elements of the Hat matrix
`stresid`	standardised residuals
`Di`	Cook's statistic

References

Unwin, D. J., Wrigley, N. (1987) Towards a general-theory of control point distribution effects in trend surface models. *Computers and Geosciences,* **13**, 351–355.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

surf.ls, influence.measures, plot.lm

Examples

```
library(MASS)  # for eqscplot
data(topo, package = "MASS")
topo2 <- surf.ls(2, topo)
infl.topo2 <- trls.influence(topo2)
(cand <- as.data.frame(infl.topo2)[abs(infl.topo2$stresid) > 1.5, ])
cand.xy <- topo[as.integer(rownames(cand)), c("x", "y")]
trsurf <- trmat(topo2, 0, 6.5, 0, 6.5, 50)
eqscplot(trsurf, type = "n")
contour(trsurf, add = TRUE, col = "grey")
plot(topo2, add = TRUE, div = 3)
points(cand.xy, pch = 16, col = "orange")
text(cand.xy, labels = rownames(cand.xy), pos = 4, offset = 0.5)
```

---

trmat *Evaluate Trend Surface over a Grid*

Description

Evaluate trend surface over a grid.

Usage

```
trmat(obj, xl, xu, yl, yu, n)
```

Arguments

obj	object returned by surf.ls or surf.gls
xl	limits of the rectangle for grid
xu	
yl	
yu	
n	use n x n grid within the rectangle

Value

list with components x, y and z suitable for contour and image.

References

Ripley, B. D. (1981) *Spatial Statistics.* Wiley.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

surf.ls, surf.gls

Examples

```
data(topo, package="MASS")
topo.kr <- surf.ls(2, topo)
trsurf <- trmat(topo.kr, 0, 6.5, 0, 6.5, 50)
```

---

variogram *Compute Spatial Variogram*

Description

Compute spatial (semi-)variogram of spatial data or residuals.

Usage

```
variogram(krig, nint, plotit = TRUE, ...)
```

Arguments

`krig`	trend-surface or kriging object with columns `x`, `y`, and `z`
`nint`	number of bins used
`plotit`	logical for plotting
`...`	parameters for the plot

Details

Divides range of data into `nint` bins, and computes the average squared difference for pairs with separation in each bin. Returns results for bins with 6 or more pairs.

Value

`x` and `y` coordinates of the variogram and `cnt`, the number of pairs averaged per bin.

Side Effects

Plots the variogram if `plotit = TRUE`

References

Ripley, B. D. (1981) *Spatial Statistics.* Wiley.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

See Also

`correlogram`

Examples

```
data(topo, package="MASS")
topo.kr <- surf.ls(2, topo)
variogram(topo.kr, 25)
```

Chapter 22

The `splines` package

`splines-package` *Regression Spline Functions and Classes*

Description

Regression spline functions and classes.

Details

This package provides functions for working with regression splines using the B-spline basis, the natural cubic spline basis.

For a complete list of functions, use `library(help="splines")`.

Author(s)

Douglas M. Bates ⟨bates@stat.wisc.edu⟩ and William N. Venables ⟨Bill.Venables@cmis.csiro.au⟩

Maintainer: R Core Team ⟨R-core@r-project.org⟩

`asVector` *Coerce an Object to a Vector*

Description

This is a generic function. Methods for this function coerce objects of given classes to vectors.

Usage

```
asVector(object)
```

Arguments

`object`	An object.

Details

Methods for vector coercion in new classes must be created for the `asVector` generic instead of `as.vector`. The `as.vector` function is internal and not easily extended. Currently the only class with an `asVector` method is the `xyVector` class.

Value

a vector

Author(s)

Douglas Bates and Bill Venables

See Also

`xyVector`

Examples

```
ispl <- interpSpline( weight ~ height,  women )
pred <- predict(ispl)
class(pred)
utils::str(pred)
asVector(pred)
```

---

backSpline *Monotone Inverse Spline*

Description

Create a monotone inverse of a monotone natural spline.

Usage

```
backSpline(object)
```

Arguments

`object`	an object that inherits from class `nbSpline` or `npolySpline`. That is, the object must represent a natural interpolation spline but it can be either in the B-spline representation or the piecewise polynomial one. The spline is checked to see if it represents a monotone function.

Value

An object of class `polySpline` that contains the piecewise polynomial representation of a function that has the appropriate values and derivatives at the knot positions to be an inverse of the spline represented by `object`. Technically this object is not a spline because the second derivative is not constrained to be continuous at the knot positions. However, it is often a much better approximation to the inverse than fitting an interpolation spline to the y/x pairs.

Author(s)

Douglas Bates and Bill Venables

See Also

`interpSpline`

Examples

```
ispl <- interpSpline( women$height, women$weight )
bspl <- backSpline( ispl )
plot( bspl )                    # plots over the range of the knots
points( women$weight, women$height )
```

---

bs *B-Spline Basis for Polynomial Splines*

Description

Generate the B-spline basis matrix for a polynomial spline.

Usage

```
bs(x, df = NULL, knots = NULL, degree = 3, intercept = FALSE,
   Boundary.knots = range(x))
```

Arguments

`x`	the predictor variable. Missing values are allowed.
`df`	degrees of freedom; one can specify `df` rather than `knots`; `bs()` then chooses `df-degree-1` knots at suitable quantiles of `x` (which will ignore missing values).
`knots`	the *internal* breakpoints that define the spline. The default is `NULL`, which results in a basis for ordinary polynomial regression. Typical values are the mean or median for one knot, quantiles for more knots. See also `Boundary.knots`.
`degree`	degree of the piecewise polynomial—default is 3 for cubic splines.
`intercept`	if `TRUE`, an intercept is included in the basis; default is `FALSE`.
`Boundary.knots`	boundary points at which to anchor the B-spline basis (default the range of the data). If both `knots` and `Boundary.knots` are supplied, the basis parameters do not depend on `x`. Data can extend beyond `Boundary.knots`.

Value

A matrix of dimension `length(x) * df`, where either `df` was supplied or if `knots` were supplied, `df = length(knots) + 3 + intercept`. Attributes are returned that correspond to the arguments to `bs`, and explicitly give the `knots`, `Boundary.knots` etc for use by `predict.bs()`.

`bs()` is based on the function `spline.des()`. It generates a basis matrix for representing the family of piecewise polynomials with the specified interior knots and degree, evaluated at the values of `x`. A primary use is in modeling formulas to directly specify a piecewise polynomial term in a model.

References

Hastie, T. J. (1992) Generalized additive models. Chapter 7 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

ns, poly, smooth.spline, predict.bs, SafePrediction

Examples

```
require(stats)
bs(women$height, df = 5)
summary(fm1 <- lm(weight ~ bs(height, df = 5), data = women))

## example of safe prediction
plot(women, xlab = "Height (in)", ylab = "Weight (lb)")
ht <- seq(57, 73, len = 200)
lines(ht, predict(fm1, data.frame(height=ht)))

## Consistency:
x <- c(1:3,5:6)
stopifnot(identical(bs(x), bs(x, df = 3)),
          !is.null(kk <- attr(bs(x), "knots")),# not true till 1.5.1
          length(kk) == 0)
```

---

interpSpline *Create an Interpolation Spline*

Description

Create an interpolation spline, either from x and y vectors, or from a formula/data.frame combination.

Usage

```
interpSpline(obj1, obj2, bSpline = FALSE, period = NULL,
             na.action = na.fail)
```

Arguments

obj1	Either a numeric vector of x values or a formula.
obj2	If obj1 is numeric this should be a numeric vector of the same length. If obj1 is a formula this can be an optional data frame in which to evaluate the names in the formula.
bSpline	If TRUE the b-spline representation is returned, otherwise the piecewise polynomial representation is returned. Defaults to FALSE.
period	An optional positive numeric value giving a period for a periodic interpolation spline.
na.action	a optional function which indicates what should happen when the data contain NAs. The default action (na.omit) is to omit any incomplete observations. The alternative action na.fail causes interpSpline to print an error message and terminate if there are any incomplete observations.

Value

An object that inherits from class `spline`. The object can be in the B-spline representation, in which case it will be of class `nbSpline` for natural B-spline, or in the piecewise polynomial representation, in which case it will be of class `npolySpline`.

Author(s)

Douglas Bates and Bill Venables

See Also

`splineKnots`, `splineOrder`, `periodicSpline`.

Examples

```
ispl <- interpSpline( women$height, women$weight )
ispl2 <- interpSpline( weight ~ height,  women )
# ispl and ispl2 should be the same
plot( predict( ispl, seq( 55, 75, len = 51 ) ), type = "l" )
points( women$height, women$weight )
plot( ispl )    # plots over the range of the knots
points( women$height, women$weight )
splineKnots( ispl )
```

---

ns *Generate a Basis Matrix for Natural Cubic Splines*

Description

Generate the B-spline basis matrix for a natural cubic spline.

Usage

```
ns(x, df = NULL, knots = NULL, intercept = FALSE,
   Boundary.knots = range(x))
```

Arguments

`x`	the predictor variable. Missing values are allowed.
`df`	degrees of freedom. One can supply `df` rather than knots; `ns()` then chooses `df - 1 - intercept` knots at suitably chosen quantiles of `x` (which will ignore missing values).
`knots`	breakpoints that define the spline. The default is no knots; together with the natural boundary conditions this results in a basis for linear regression on `x`. Typical values are the mean or median for one knot, quantiles for more knots. See also `Boundary.knots`.
`intercept`	if `TRUE`, an intercept is included in the basis; default is `FALSE`.
`Boundary.knots`	boundary points at which to impose the natural boundary conditions and anchor the B-spline basis (default the range of the data). If both `knots` and `Boundary.knots` are supplied, the basis parameters do not depend on `x`. Data can extend beyond `Boundary.knots`

Value

A matrix of dimension `length(x) * df` where either `df` was supplied or if `knots` were supplied, `df = length(knots) + 1 + intercept`. Attributes are returned that correspond to the arguments to `ns`, and explicitly give the `knots`, `Boundary.knots` etc for use by `predict.ns()`.

`ns()` is based on the function `spline.des`. It generates a basis matrix for representing the family of piecewise-cubic splines with the specified sequence of interior knots, and the natural boundary conditions. These enforce the constraint that the function is linear beyond the boundary knots, which can either be supplied, else default to the extremes of the data. A primary use is in modeling formula to directly specify a natural spline term in a model.

References

Hastie, T. J. (1992) Generalized additive models. Chapter 7 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`bs`, `predict.ns`, `SafePrediction`

Examples

```
require(stats)
ns(women$height, df = 5)
summary(fm1 <- lm(weight ~ ns(height, df = 5), data = women))

## example of safe prediction
plot(women, xlab = "Height (in)", ylab = "Weight (lb)")
ht <- seq(57, 73, len = 200)
lines(ht, predict(fm1, data.frame(height=ht)))
```

`periodicSpline` *Create a Periodic Interpolation Spline*

Description

Create a periodic interpolation spline, either from `x` and `y` vectors, or from a formula/data.frame combination.

Usage

```
periodicSpline(obj1, obj2, knots, period = 2*pi, ord = 4)
```

Arguments

`obj1`	either a numeric vector of `x` values or a formula.
`obj2`	if `obj1` is numeric this should be a numeric vector of the same length. If `obj1` is a formula this can be an optional data frame in which to evaluate the names in the formula.
`knots`	optional numeric vector of knot positions.

period	positive numeric value giving the period for the periodic spline. Defaults to 2 * pi.
ord	integer giving the order of the spline, at least 2. Defaults to 4. See splineOrder for a definition of the order of a spline.

Value

An object that inherits from class spline. The object can be in the B-spline representation, in which case it will be a pbSpline object, or in the piecewise polynomial representation (a ppolySpline object).

Author(s)

Douglas Bates and Bill Venables

See Also

splineKnots, interpSpline

Examples

```
xx <- seq( -pi, pi, len = 16 )[-1]
yy <- sin( xx )
frm <- data.frame( xx, yy )
( pispl <- periodicSpline( xx, yy, period = 2 * pi ) )
pispl2 <- periodicSpline( yy ~ xx, frm, period = 2 * pi )
stopifnot(all.equal(pispl, pispl2))# pispl and pispl2 are the same

plot( pispl )           # displays over one period
points( yy ~ xx, col = "brown")
plot( predict( pispl, seq(-3*pi, 3*pi, len = 101) ), type = "l" )
```

---

polySpline *Piecewise Polynomial Spline Representation*

Description

Create the piecewise polynomial representation of a spline object.

Usage

```
polySpline(object, ...)
as.polySpline(object, ...)
```

Arguments

object	An object that inherits from class spline.
...	Optional additional arguments. At present no additional arguments are used.

Value

An object that inherits from class `polySpline`. This is the piecewise polynomial representation of a univariate spline function. It is defined by a set of distinct numeric values called knots. The spline function is a polynomial function between each successive pair of knots. At each interior knot the polynomial segments on each side are constrained to have the same value of the function and some of its derivatives.

Author(s)

Douglas Bates and Bill Venables

See Also

`interpSpline`, `periodicSpline`, `splineKnots`, `splineOrder`

Examples

```
ispl <- polySpline( interpSpline( weight ~ height,  women, bSpline = TRUE ) )
print( ispl )    # print the piecewise polynomial representation
plot( ispl )     # plots over the range of the knots
points( women$height, women$weight )
```

`predict.bs` *Evaluate a Spline Basis*

Description

Evaluate a predefined spline basis at given values.

Usage

```
## S3 method for class 'bs':
predict(object, newx, ...)

## S3 method for class 'ns':
predict(object, newx, ...)
```

Arguments

`object`	the result of a call to `bs` or `ns` having attributes describing `knots`, `degree`, etc.
`newx`	the `x` values at which evaluations are required.
`...`	Optional additional arguments. At present no additional arguments are used.

Value

An object just like `object`, except evaluated at the new values of `x`.

These are methods for the generic function `predict` for objects inheriting from classes `"bs"` or `"ns"`. See `predict` for the general behavior of this function.

See Also

bs, ns, poly.

Examples

```
basis <- ns(women$height, df = 5)
newX <- seq(58, 72, len = 51)
# evaluate the basis at the new data
predict(basis, newX)
```

`predict.bSpline` *Evaluate a Spline at New Values of x*

Description

The `predict` methods for the classes that inherit from the virtual classes `bSpline` and `polySpline` are used to evaluate the spline or its derivatives. The `plot` method for a spline object first evaluates `predict` with the `x` argument missing, then plots the resulting `xyVector` with `type = "l"`.

Usage

```
## S3 method for class 'bSpline':
predict(object, x, nseg=50, deriv=0, ...)
## S3 method for class 'nbSpline':
predict(object, x, nseg=50, deriv=0, ...)
## S3 method for class 'pbSpline':
predict(object, x, nseg=50, deriv=0, ...)
## S3 method for class 'npolySpline':
predict(object, x, nseg=50, deriv=0, ...)
## S3 method for class 'ppolySpline':
predict(object, x, nseg=50, deriv=0, ...)
```

Arguments

`object`	An object that inherits from the `bSpline` or the `polySpline` class.
`x`	A numeric vector of `x` values at which to evaluate the spline. If this argument is missing a suitable set of `x` values is generated as a sequence of `nseq` segments spanning the range of the knots.
`nseg`	A positive integer giving the number of segments in a set of equally-spaced `x` values spanning the range of the knots in `object`. This value is only used if `x` is missing.
`deriv`	An integer between 0 and `splineOrder(object) - 1` specifying the derivative to evaluate.
`...`	further arguments passed to or from other methods.

Value

an `xyVector` with components

`x`	the supplied or inferred numeric vector of `x` values
`y`	the value of the spline (or its `deriv`'th derivative) at the `x` vector

Author(s)

Douglas Bates and Bill Venables

See Also

xyVector, interpSpline, periodicSpline

Examples

```
ispl <- interpSpline( weight ~ height,  women )
opar <- par(mfrow = c(2, 2), las = 1)
plot(predict(ispl, nseg = 201),     # plots over the range of the knots
     main = "Original data with interpolating spline", type = "l",
     xlab = "height", ylab = "weight")
points(women$height, women$weight, col = 4)
plot(predict(ispl, nseg = 201, deriv = 1),
     main = "First derivative of interpolating spline", type = "l",
     xlab = "height", ylab = "weight")
plot(predict(ispl, nseg = 201, deriv = 2),
     main = "Second derivative of interpolating spline", type = "l",
     xlab = "height", ylab = "weight")
plot(predict(ispl, nseg = 401, deriv = 3),
     main = "Third derivative of interpolating spline", type = "l",
     xlab = "height", ylab = "weight")
par(opar)
```

---

splineDesign *Design Matrix for B-splines*

Description

Evaluate the design matrix for the B-splines defined by `knots` at the values in `x`.

Usage

```
splineDesign(knots, x, ord = 4, derivs, outer.ok = FALSE)
spline.des(knots, x, ord = 4, derivs, outer.ok = FALSE)
```

Arguments

`knots`	a numeric vector of knot positions with non-decreasing values.
`x`	a numeric vector of values at which to evaluate the B-spline functions or derivatives. Unless `outer.ok` is truee, the values in `x` must be between `knots[ord]` and `knots[ length(knots) + 1 - ord ]`.
`ord`	a positive integer giving the order of the spline function. This is the number of coefficients in each piecewise polynomial segment, thus a cubic spline has order 4. Defaults to 4.
`derivs`	an integer vector of the same length as `x` and with values between `0` and `ord - 1`. The derivative of the given order is evaluated at the `x` positions. Defaults to a vector of zeroes of the same length as `x`.
`outer.ok`	logical indicating if `x` should be allowed outside the *inner* knots, see the `x` argument.

Value

A matrix with `length( x )` rows and `length( knots ) - ord` columns. The i'th row of the matrix contains the coefficients of the B-splines (or the indicated derivative of the B-splines) defined by the `knot` vector and evaluated at the i'th value of `x`. Each B-spline is defined by a set of `ord` successive knots so the total number of B-splines is `length(knots)-ord`.

Note

The older `spline.des` function takes the same arguments but returns a list with several components including `knots`, `ord`, `derivs`, and `design`. The `design` component is the same as the value of the `splineDesign` function.

Author(s)

Douglas Bates and Bill Venables

Examples

```
splineDesign(knots = 1:10, x = 4:7)

knots <- c(1,1.8,3:5,6.5,7,8.1,9.2,10)# 10 => 10-4 = 6 Basis splines
x <- seq(min(knots)-1, max(knots)+1, length=501)
bb <- splineDesign(knots, x=x, outer.ok = TRUE)

plot(range(x), c(0,1), type="n", xlab="x", ylab="",
     main= "B-splines - sum to 1 inside inner knots")
mtext(expression(B[j](x) *"  and "* sum(B[j](x), j==1, 6)), adj=0)
abline(v=knots, lty=3, col="light gray")
abline(v=knots[c(4,length(knots)-3)], lty=3, col="gray10")
lines(x, rowSums(bb), col="gray", lwd=2)
matlines(x, bb, ylim = c(0,1), lty=1)
```

---

splineKnots *Knot Vector from a Spline*

Description

Return the knot vector corresponding to a spline object.

Usage

```
splineKnots(object)
```

Arguments

`object`	an object that inherits from class `"spline"`.

Value

A non-decreasing numeric vector of knot positions.

Author(s)

Douglas Bates and Bill Venables

Examples

```
ispl <- interpSpline( weight ~ height, women )
splineKnots( ispl )
```

---

splineOrder *Determine the Order of a Spline*

Description

Return the order of a spline object.

Usage

```
splineOrder(object)
```

Arguments

object	An object that inherits from class "spline".

Details

The order of a spline is the number of coefficients in each piece of the piecewise polynomial representation. Thus a cubic spline has order 4.

Value

A positive integer.

Author(s)

Douglas Bates and Bill Venables

See Also

splineKnots, interpSpline, periodicSpline

Examples

```
splineOrder( interpSpline( weight ~ height, women ) )
```

`xyVector` *Construct an xyVector Object*

Description

Create an object to represent a set of x-y pairs. The resulting object can be treated as a matrix or as a data frame or as a vector. When treated as a vector it reduces to the `y` component only.

The result of functions such as `predict.spline` is returned as an `xyVector` object so the x-values used to generate the y-positions are retained, say for purposes of generating plots.

Usage

```
xyVector(x, y)
```

Arguments

`x`	a numeric vector
`y`	a numeric vector of the same length as `x`

Value

An object of class `xyVector` with components

`x`	a numeric vector
`y`	a numeric vector of the same length as `x`

Author(s)

Douglas Bates and Bill Venables

Examples

```
ispl <- interpSpline( weight ~ height, women )
weights <- predict( ispl, seq( 55, 75, len = 51 ))
class( weights )
plot( weights, type = "l", xlab = "height", ylab = "weight" )
points( women$height, women$weight )
weights
```

Chapter 23

The `stats4` package

`stats4-package` *Statistical Functions using S4 Classes*

Description

Statistical Functions using S4 classes.

Details

This package contains functions and classes for statistics using the S version 4 class system.

The methods currently support maximum likelihood (function `mle()` returning class `"mle"`), including methods for `AIC` and `BIC` for model selection.

Author(s)

R Development Core Team and contributors worldwide

Maintainer: R Core Team <R-core@r-project.org>

`AIC-methods` *Methods for Function 'AIC' in Package 'stats4'*

Description

Calculate the Akaike information criterion for a fitted model object.

Methods

object = "ANY" Generic function: see `AIC`.

`BIC` *Bayesian Information Criterion*

Description

This generic function calculates the Bayesian information criterion, also known as Schwarz's Bayesian criterion (SBC), for one or several fitted model objects for which a log-likelihood value can be obtained, according to the formula -2log-likelihood$+n_{par}\log(n_{obs})$, where n_{par} represents the number of parameters and n_{obs} the number of observations in the fitted model.

Usage

```
BIC(object, ...)
```

Arguments

`object`	An object of a suitable class for the BIC to be calculated - usually a `"logLik"` object or an object for which a `logLik` method exists.
`...`	Some methods for this generic function may take additional, optional arguments. At present none do.

Value

Returns a numeric value with the corresponding BIC.

References

Schwarz, G. (1978) "Estimating the Dimension of a Model", *Annals of Statistics*, **6**, 461-464.

See Also

`logLik-methods`, `AIC-methods`

Examples

```
lm1 <- lm(Fertility ~ . , data = swiss)
AIC(lm1)
BIC(lm1)
```

coef-methods *Methods for Function 'coef' in Package 'stats4'*

Description

Extract the coefficient vector from "mle" objects.

Methods

object = "ANY" Generic function: see coef.

object = "mle" Extract the full coefficient vector (including any fixed coefficients) from the fit.

object = "summary.mle" Extract the coefficient vector and standard errors from the summary of the fit.

confint-methods *Methods for Function 'confint' in Package 'stats4'*

Description

Generate confidence intervals

Methods

object = "ANY" Generic function: see confint.

object = "mle" First generate profile and then confidence intervals from the profile.

object = "profile.mle" Generate confidence intervals based on likelihood profile.

logLik-methods *Methods for Function 'logLik' in Package 'stats4'*

Description

Extract the maximized log-likelihood from "mle" objects.

Methods

object = "ANY" Generic function: see logLik.

object = "mle" Extract log-likelihood from the fit.

mle *Maximum Likelihood Estimation*

Description

Estimate parameters by the method of maximum likelihood.

Usage

```
mle(minuslogl, start = formals(minuslogl), method = "BFGS",
    fixed = list(), ...)
```

Arguments

minuslogl	Function to calculate negative log-likelihood.
start	Named list. Initial values for optimizer.
method	Optimization method to use. See optim.
fixed	Named list. Parameter values to keep fixed during optimization.
...	Further arguments to pass to optim.

Details

The optim optimizer is used to find the minimum of the negative log-likelihood. An approximate covariance matrix for the parameters is obtained by inverting the Hessian matrix at the optimum.

Value

An object of class mle-class.

Note

Be careful to note that the argument is -log L (not -2 log L). It is for the user to ensure that the likelihood is correct, and that asymptotic likelihood inference is valid.

See Also

mle-class

Examples

```
x <- 0:10
y <- c(26, 17, 13, 12, 20, 5, 9, 8, 5, 4, 8)
ll <- function(ymax=15, xhalf=6)
    -sum(stats::dpois(y, lambda=ymax/(1+x/xhalf), log=TRUE))
(fit <- mle(ll))
mle(ll, fixed=list(xhalf=6))

summary(fit)
logLik(fit)
vcov(fit)
plot(profile(fit), absVal=FALSE)
confint(fit)
```

```
## use bounded optimization
## the lower bounds are really > 0, but we use >=0 to stress-test profiling
(fit1 <- mle(ll, method="L-BFGS-B", lower=c(0, 0)))
plot(profile(fit1), absVal=FALSE)

## a better parametrization:
ll2 <- function(lymax=log(15), lxhalf=log(6))
    -sum(stats::dpois(y, lambda=exp(lymax)/(1+x/exp(lxhalf)), log=TRUE))
(fit2 <- mle(ll2))
plot(profile(fit2), absVal=FALSE)
exp(confint(fit2))
```

`mle-class` *Class "mle" for Results of Maximum Likelihood Estimation*

Description

This class encapsulates results of a generic maximum likelihood procedure.

Objects from the Class

Objects can be created by calls of the form `new("mle", ...)`, but most often as the result of a call to `mle`.

Slots

`call`: Object of class `"language"`. The call to `mle`.

`coef`: Object of class `"numeric"`. Estimated parameters.

`fullcoef`: Object of class `"numeric"`. Fixed and estimated parameters.

`vcov`: Object of class `"matrix"`. Approximate variance-covariance matrix.

`min`: Object of class `"numeric"`. Minimum value of objective function.

`details`: Object of class `"list"`. List returned from `optim`.

`minuslogl`: Object of class `"function"`. The negative loglikelihood function.

`method`: Object of class `"character"`. The optimization method used.

Methods

confint `signature(object = "mle")`: Confidence intervals from likelihood profiles.

logLik `signature(object = "mle")`: Extract maximized log-likelihood.

profile `signature(fitted = "mle")`: Likelihood profile generation.

show `signature(object = "mle")`: Display object briefly.

summary `signature(object = "mle")`: Generate object summary.

update `signature(object = "mle")`: Update fit.

vcov `signature(object = "mle")`: Extract variance-covariance matrix.

plot-methods *Methods for Function 'plot' in Package 'stats4'*

Description

Plot profile likelihoods for "mle" objects.

Usage

```
## S4 method for signature 'profile.mle, missing':
plot(x, levels, conf = c(99, 95, 90, 80, 50)/100, nseg = 50,
     absVal = TRUE, ...)
```

Arguments

x	an object of class "profile.mle"
levels	levels, on the scale of the absolute value of a t statistic, at which to interpolate intervals. Usually conf is used instead of giving levels explicitly.
conf	a numeric vector of confidence levels for profile-based confidence intervals on the parameters.
nseg	an integer value giving the number of segments to use in the spline interpolation of the profile t curves.
absVal	a logical value indicating whether or not the plots should be on the scale of the absolute value of the profile t. Defaults to TRUE.
...	other arguments to the plot function can be passed here.

Methods

x = "ANY", y = "ANY" Generic function: see plot.

x = "profile.mle", y = "missing" Plot likelihood profiles for x.

profile-methods *Methods for Function profile in Package 'stats4'*

Description

Profile likelihood

Methods

fitted = "ANY" Generic function

fitted = "mle" Profile the likelihood in the vicinity of the optimum

profile.mle-class *Class "profile.mle"; Profiling information for "mle" object*

Description

Likelihood profiles along each parameter of likelihood function

Objects from the Class

Objects can be created by calls of the form `new("profile.mle", ...)`, but most often by invoking `profile` on an "mle" object.

Slots

profile: Object of class `"list"`. List of profiles, one for each requested parameter. Each profile is a data frame with the first column called `z` being the signed square root of the -2 log likelihood ratio, and the others being the parameters with names prefixed by `par.vals.`

summary: Object of class `"summary.mle"`. Summary of object being profiled.

Methods

confint `signature(object = "profile.mle")`: Use profile to generate approximate confidence intervals for parameters.

plot `signature(x = "profile.mle", y = "missing")`: Plot profiles for each parameter.

See Also

`mle`, `mle-class`, `summary.mle-class`

show-methods *Methods for Function show in Package 'stats4'*

Description

Show objects of classes `mle` and `summary.mle`

Methods

object = "mle" Print simple summary of `mle` object. Just the coefficients and the call.

object = "summary.mle" Shows call, table of coefficients and standard errors, and -2 log L

summary-methods *Methods for Function summary in Package 'stats4'*

Description

Summarize objects

Methods

object = "ANY" Generic function

object = "mle" Generate a summary as an object of class "summary.mle", containing estimates, asymptotic SE, and value of -2 log L

summary.mle-class *Class "summary.mle", summary of "mle" objects*

Description

Extract of "mle" object

Objects from the Class

Objects can be created by calls of the form `new("summary.mle", ...)`, but most often by invoking `summary` on an "mle" object. They contain values meant for printing by `show`.

Slots

call: Object of class `"language"` The call that generated the "mle" object.

coef: Object of class `"matrix"`. Estimated coefficients and standard errors

m2logL: Object of class `"numeric"`. Minus twice the log likelihood.

Methods

show `signature(object = "summary.mle")`: Pretty-prints `object`

coef `signature(object = "summary.mle")`: Extracts the contents of the `coef` slot

See Also

summary, mle, mle-class

update-methods *Methods for Function 'update' in Package 'stats4'*

Description

Update "mle" objects.

Usage

```
## S4 method for signature 'mle':
update(object, ..., evaluate = TRUE)
```

Arguments

object	An existing fit.
...	Additional arguments to the call, or arguments with changed values. Use name=NULL to remove the argument name.
evaluate	If true evaluate the new call else return the call.

Methods

object = "ANY" Generic function: see update.

object = "mle" Update a fit.

Examples

```
x <- 0:10
y <- c(26, 17, 13, 12, 20, 5, 9, 8, 5, 4, 8)
ll <- function(ymax=15, xhalf=6)
    -sum(stats::dpois(y, lambda=ymax/(1+x/xhalf), log=TRUE))
fit <- mle(ll)
## note the recorded call contains ..1, a problem with S4 dispatch
update(fit, fixed=list(xhalf=3))
```

vcov-methods *Methods for Function 'vcov' in Package 'stats4'*

Description

Extract the approximate variance-covariance matrix from "mle" objects.

Methods

object = "ANY" Generic function: see vcov.

object = "mle" Extract the estimated variance-covariance matrix for the estimated parameters (if any).

Chapter 24

The `survival` package

`aml` *Acute Myelogenous Leukemia survival data*

Description

Survival in patients with Acute Myelogenous Leukemia

Usage

```
data(aml)
data(leukemia)
```

Format

time:	survival or censoring time
status:	censoring status
x:	maintenance chemotherapy given? (factor)

Source

Miller "Survival Analysis"

`anova.coxph` *Analysis of Deviance for Cox model.*

Description

Compute an analysis of deviance table for one Cox model fit.

Usage

```
## S3 method for class 'coxph':
anova(object, ...,  test = NULL)
```

Arguments

object	An object of class coxph
...	Further coxph objects
test	a character string, (partially) matching one of "Chisq", "F" or "Cp". See stat.anova.

Details

Specifying a single object gives a sequential analysis of deviance table for that fit. That is, the reductions in the residual deviance as each term of the formula is added in turn are given in as the rows of a table, plus the residual deviances themselves.

If more than one object is specified, the table has a row for the residual degrees of freedom and deviance for each model. For all but the first model, the change in degrees of freedom and deviance is also given. (This only make statistical sense if the models are nested.) It is conventional to list the models from smallest to largest, but this is up to the user.

The table will optionally contain test statistics (and P values) comparing the reduction in deviance for the row to the residuals. Mallows' C_p statistic is the residual deviance plus twice the residual degrees of freedom, which is closely related to AIC. Don't use test=F, it doesn't make sense.

Value

An object of class "anova" inheriting from class "data.frame".

Warning

The comparison between two or more models by anova or will only be valid if they are fitted to the same dataset. This may be a problem if there are missing values.

See Also

coxph, anova.

Examples

```
fit <- coxph(Surv(futime, fustat) ~ resid.ds *rx + ecog.ps, data = ovarian)
anova(fit)
anova(fit,test="Chisq")
fit2 <- coxph(Surv(futime, fustat) ~ resid.ds +rx + ecog.ps, data
=ovarian)
anova(fit2,fit)
```

---

as.date *Coerce Data to Dates*

Description

Converts any of the following character forms to a Julian date: 8/31/56, 8-31-1956, 31 8 56, 083156, 31Aug56, or August 31 1956.

Usage

```
as.date(x, order = "mdy", ...)
```

Arguments

x	input data vector.
order	if x is character, defines the order in which the terms are assumed to appear in a xx/xx/xx date. The default is month/day/year; any permutation of mdy is legal.
...	if x is character, then any other arguments from mdy.date() can be used as well.

Details

If x is numeric, then floor(x) is returned, e.g., as.date(35) is the same as as.date(35.2) and gives Feb 2, 1960. If x is character, the program attempts to parse it.

Value

For each date, the number of days between it and January 1, 1960. The date will be missing if the string is not interpretable.

See Also

mdy.date, date.mmddyy, date.ddmmmyy

Examples

```
as.date(c("1jan1960", "2jan1960", "31mar1960", "30jul1960"))
```

---

attrassign *Create new-style "assign" attribute*

Description

The "assign" attribute on model matrices describes which columns come from which terms in the model formula. It has two version. R uses the original version, but the newer version is sometimes useful.

Usage

```
## Default S3 method:
attrassign(object, tt,...)
## S3 method for class 'lm':
attrassign(object,...)
```

Arguments

object	model matrix or linear model object
tt	terms object
...	ignored

Value

A list with names corresponding to the term names and elements that are vectors indicating which columns come from which terms

See Also

terms,model.matrix

Examples

```
formula<-Surv(time,status)~factor(edtrt)
tt<-terms(formula)
mf<-model.frame(tt,data=pbc)
mm<-model.matrix(tt,mf)
## a few rows of data
mm[1:3,]
## old-style assign attribute
attr(mm,"assign")
## new-style assign attribute
attrassign(mm,tt)
```

---

bladder *Bladder Cancer Recurrences*

Description

Data on recurrences of bladder cancer, used by many people to demonstrate methodology for recurrent event modelling.

Usage

```
data(bladder)
```

Format

bladder

id:	Patient id
rx:	Treatment 1=placebo 2=thiotepa
number:	Initial number of tumours (8=8 or more)
size:	size (cm) of largest initial tumour
stop:	recurrence or censoring time
enum:	which recurrence (up to 4)

bladder2

id:	Patient id
rx:	Treatment 1=placebo 2=thiotepa
number:	Initial number of tumours (8=8 or more)
size:	size (cm) of largest initial tumour
start:	start of interval (0 or previous recurrence time)
stop:	recurrence or censoring time

enum: which recurrence (up to 4)

Source

Wei, Lin, Weisfeld, JASA 1989

`cch` *Fits proportional hazards regression model to case-cohort data*

Description

Returns estimates and standard errors from relative risk regression fit to data from case-cohort studies. A choice is available among the Prentice, Self-Prentice and Lin-Ying methods for unstratified data. For stratified data the choice is between Borgan I, a generalization of the Self-Prentice estimator for unstratified case-cohort data, and Borgan II, a generalization of the Lin-Ying estimator.

Usage

```
cch(formula, data = sys.parent(), subcoh, id, stratum=NULL, cohort.size,
    method =c("Prentice","SelfPrentice","LinYing","I.Borgan","II.Borgan"),
    robust=FALSE)
```

Arguments

`formula`	A formula object that must have a `Surv` object as the response. The Surv object must be of type `"right"`, or of type `"counting"`.
`subcoh`	Vector of indicatorsfor subjects sampled as part of the sub-cohort. Code `1` or `TRUE` for members of the sub-cohort, `0` or `FALSE` for others. If `data` is a data frame then `subcoh` may be a one-sided formula.
`id`	Vector of unique identifiers, or formula specifying such a vector.
`stratum`	A vector of stratum indicators or a formula specifying such a vector
`cohort.size`	Vector with size of each stratum original cohort from which subcohort was sampled
`data`	An optional data frame in which to interpret the variables occurring in the formula.
`method`	Three procedures are available. The default method is "Prentice", with options for "SelfPrentice" or "LinYing".
`robust`	For `"LinYing"` only, if `robust=TRUE`, use design-based standard errors even for phase I

Details

Implements methods for case-cohort data analysis described by Therneau and Li (1999). The three methods differ in the choice of "risk sets" used to compare the covariate values of the failure with those of others at risk at the time of failure. "Prentice" uses the sub-cohort members "at risk" plus the failure if that occurs outside the sub-cohort and is score unbiased. "SelfPren" (Self-Prentice) uses just the sub-cohort members "at risk". These two have the same asymptotic variance-covariance matrix. "LinYing" (Lin-Ying) uses the all members of the sub-cohort and all failures outside the

sub-cohort who are "at risk". The methods also differ in the weights given to different score contributions.

The `data` argument must not have missing values for any variables in the model. There must not be any censored observations outside the subcohort.

Value

An object of class "cch" incorporating a list of estimated regression coefficients and two estimates of their asymptotic variance-covariance matrix.

`coef`	regression coefficients.
`naive.var`	Self-Prentice model based variance-covariance matrix.
`var`	Lin-Ying empirical variance-covariance matrix.

Author(s)

Norman Breslow, modified by Thomas Lumley

References

Prentice, RL (1986). A case-cohort design for epidemiologic cohort studies and disease prevention trials. Biometrika 73: 1–11.

Self, S and Prentice, RL (1988). Asymptotic distribution theory and efficiency results for case-cohort studies. Annals of Statistics 16: 64–81.

Lin, DY and Ying, Z (1993). Cox regression with incomplete covariate measurements. Journal of the American Statistical Association 88: 1341–1349.

Barlow, WE (1994). Robust variance estimation for the case-cohort design. Biometrics 50: 1064–1072

Therneau, TM and Li, H (1999). Computing the Cox model for case-cohort designs. Lifetime Data Analysis 5: 99–112.

Borgan, , Langholz, B, Samuelsen, SO, Goldstein, L and Pogoda, J (2000) Exposure stratified case-cohort designs. Lifetime Data Analysis 6, 39-58.

See Also

`twophase` and `svycoxph` in the "survey" package for more general two-phase designs. http://faculty.washington.edu/tlumley/survey/

Examples

```
## The complete Wilms Tumor Data
## (Breslow and Chatterjee, Applied Statistics, 1999)
## subcohort selected by simple random sampling.
##

data(nwtco)

subcoh <- nwtco$in.subcohort
selccoh <- with(nwtco, rel==1|subcoh==1)
ccoh.data <- nwtco[selccoh,]
ccoh.data$subcohort <- subcoh[selccoh]
## central-lab histology
```

```
ccoh.data$histol <- factor(ccoh.data$histol,labels=c("FH","UH"))
## tumour stage
ccoh.data$stage <- factor(ccoh.data$stage,labels=c("I","II","III","IV"))
ccoh.data$age <- ccoh.data$age/12 # Age in years

##
## Standard case-cohort analysis: simple random subcohort
##

fit.ccP <- cch(Surv(edrel, rel) ~ stage + histol + age, data =ccoh.data,
   subcoh = ~subcohort, id=~seqno, cohort.size=4028)

fit.ccP

fit.ccSP <- cch(Surv(edrel, rel) ~ stage + histol + age, data =ccoh.data,
   subcoh = ~subcohort, id=~seqno, cohort.size=4028, method="SelfPren")

summary(fit.ccSP)

##
## (post-)stratified on instit
##
stratsizes<-table(nwtco$instit)
fit.BI<- cch(Surv(edrel, rel) ~ stage + histol + age, data =ccoh.data,
   subcoh = ~subcohort, id=~seqno, stratum=~instit, cohort.size=stratsizes,
   method="I.Borgan")

summary(fit.BI)
```

`clogit` *Conditional logistic regression*

Description

Estimates a logistic regression model by maximising the conditional likelihood. Uses a model formula of the form `case.status~exposure+strata(matched.set)`. The default is to use the exact conditional likelihood, a commonly used approximate conditional likelihood is provided for compatibility with older software.

Usage

```
clogit(formula, data, method=c("exact", "approximate"), na.action=getOption("na.
```

Arguments

`formula`	Model formula
`data`	data frame
`method`	correct or approximate conditonal likelihood?
`na.action`	missing value handling
`subset`	subset of records to use
`control`	control values

Value

An object of class `"clogit"`, which is a wrapper for a `"coxph"` object.

Author(s)

Thomas Lumley

See Also

`strata`,`coxph`,`glm`

Examples

```
clogit(case~spontaneous+induced+strata(stratum),data=infert)
```

`cluster` *Identify Clusters*

Description

This is a special function used in the context of survival models. It identifies correlated groups of observations, and is used on the right hand side of a formula. Using `cluster()` in a formula implies that robust sandwich variance estimators are desired.

Usage

```
cluster(x)
```

Arguments

`x`	A character, factor, or numeric variable.

Value

`x`

See Also

`coxph`, `Surv`

Examples

```
data(kidney)
frailty.model <- coxph(Surv(time, status)~ age + sex + disease + frailty(id), kidney)
marginal.model <- coxph(Surv(time, status)~ age + sex + disease+cluster(id), kidney)
summary(frailty.model)
summary(marginal.model)

data(rats)
frailty.model <- survreg(Surv(time, status) ~ rx +  frailty(litter), rats )
marginal.model <- survreg(Surv(time, status) ~ rx +  cluster(litter), rats )
summary(frailty.model)
summary(marginal.model)
```

colon — *Chemotherapy for Stage B/C colon cancer*

Description

These are data from one of the first successful trials of adjuvant chemotherapy for colon cancer. Levamisole is a low-toxicity compound previously used to treat worm infestations in animals; 5-FU is a moderately toxic (as these things go) chemotherapy agent. There are two records per person, one for recurrence and one for death

Usage

```
data(colon)
```

Format

id:	id
study:	1 for all patients
rx:	Treatment - Obs(ervation), Lev(amisole), Lev(amisole)+5-FU
sex:	1=male
age:	in years
obstruct:	obstruction of colon by tumour
perfor:	perforation of colon
adhere:	adherence to nearby organs
nodes:	number of lymph nodes with detectable cancer
status:	censoring status
differ:	differentiation of tumour (1=well, 2=moderate, 3=poor)
extent:	Extent of local spread (1=submucosa, 2=muscle, 3=serosa, 4=contiguous structures)
surg:	time from surgery to registration (0=short, 1=long)
node4:	more than 4 positive lymph nodes
time:	days until death
etype:	1=recurrence,2=death

Source

Danyu Lin

cox.zph — *Test the Proportional Hazards Assumption of a Cox Regression*

Description

Test the proportional hazards assumption for a Cox regression model fit (`coxph`).

Usage

```
cox.zph(fit, transform="km", global=TRUE)
```

Arguments

`fit`	the result of fitting a Cox regression model, using the `coxph` function.
`transform`	a character string specifying how the survival times should be transformed before the test is performed. Possible values are `"km"`, `"rank"`, `"identity"` or a function of one argument. The default is `"km"` for right-censored data and `"identity"` for counting-processing data.
`global`	should a global chi-square test be done, in addition to the per-variable tests.

Value

an object of class `"cox.zph"`, with components:

`table`	a matrix with one row for each variable, and optionally a last row for the global test. Columns of the matrix contain the correlation coefficient between transformed survival time and the scaled Schoenfeld residuals, a chi-square, and the two-sided p-value. For the global test there is no appropriate correlation, so an NA is entered into the matrix as a placeholder.
`x`	the transformed time axis.
`y`	the matrix of scaled Schoenfeld residuals. There will be one column per variable and one row per event. The row labels contain the original event times (for the identity transform, these will be the same as `x`).
`call`	the calling sequence for the routine.
	The computations require the original `x` matrix of the Cox model fit. Thus it saves time if the `x=TRUE` option is used in `coxph`. This function would usually be followed by both a plot and a print of the result. The plot gives an estimate of the time-dependent coefficient `beta(t)`. If the proportional hazards assumption is true, `beta(t)` will be a horizontal line. The printout gives a test for `slope=0`.

References

P. Grambsch and T. Therneau (1994), Proportional hazards tests and diagnostics based on weighted residuals. *Biometrika,* **81**, 515-26.

See Also

`coxph`, `Surv`.

Examples

```
fit <- coxph( Surv(futime, fustat) ~ age + rx, ovarian)
temp<- cox.zph(fit)
print(temp)                  #display the results
plot(temp)                   #plot curves
```

coxph *Fit Proportional Hazards Regression Model*

Description

Fits a Cox proportional hazards regression model. Time dependent variables, time dependent strata, multiple events per subject, and other extensions are incorporated using the counting process formulation of Andersen and Gill.

Usage

```
coxph(formula, data=parent.frame(), weights, subset,
       na.action, init, control, method=c("efron","breslow","exact"),
       singular.ok=TRUE, robust=FALSE,
       model=FALSE, x=FALSE, y=TRUE,...
       )
```

Arguments

formula	a formula object, with the response on the left of a ~ operator, and the terms on the right. The response must be a survival object as returned by the Surv function.
data	a data.frame in which to interpret the variables named in the formula, or in the subset and the weights argument.
subset	expression saying that only a subset of the rows of the data should be used in the fit.
na.action	a missing-data filter function, applied to the model.frame, after any subset argument has been used. Default is options()$na.action.
weights	case weights.
init	vector of initial values of the iteration. Default initial value is zero for all variables.
control	Object of class coxph.control specifying iteration limit and other control options. Default is coxph.control(...).
method	a character string specifying the method for tie handling. If there are no tied death times all the methods are equivalent. Nearly all Cox regression programs use the Breslow method by default, but not this one. The Efron approximation is used as the default here, as it is much more accurate when dealing with tied death times, and is as efficient computationally. The exact method computes the exact partial likelihood, which is equivalent to a conditional logistic model. If there are a large number of ties the computational time will be excessive.
singular.ok	logical value indicating how to handle collinearity in the model matrix. If TRUE, the program will automatically skip over columns of the X matrix that are linear combinations of earlier columns. In this case the coefficients for such columns will be NA, and the variance matrix will contain zeros. For ancillary calculations, such as the linear predictor, the missing coefficients are treated as zeros.
robust	if TRUE a robust variance estimate is returned. Default is TRUE if the model includes a cluster() operative, FALSE otherwise.

`model`	flags to control what is returned. If these are true, then the model frame, the model matrix, and/or the response is returned as components of the fitted model, with the same names as the flag arguments.
`x`	Return the design matrix in the model object?
`y`	return the response in the model object?
`...`	Other arguments will be passed to `coxph.control`

Details

The proportional hazards model is usually expressed in terms of a single survival time value for each person, with possible censoring. Andersen and Gill reformulated the same problem as a counting process; as time marches onward we observe the events for a subject, rather like watching a Geiger counter. The data for a subject is presented as multiple rows or "observations", each of which applies to an interval of observation (start, stop].

Value

an object of class `"coxph"`. See `coxph.object` for details.

Side Effects

Depending on the call, the `predict`, `residuals`, and `survfit` routines may need to reconstruct the x matrix created by `coxph`. Differences in the environment, such as which data frames are attached or the value of `options()$contrasts`, may cause this computation to fail or worse, to be incorrect. See the survival overview document for details.

SPECIAL TERMS

There are two special terms that may be used in the model equation. A 'strata' term identifies a stratified Cox model; separate baseline hazard functions are fit for each strata. The `cluster` term is used to compute a robust variance for the model. The term `+ cluster(id)`, where `id == unique(id)`, is equivalent to specifying the `robust=T` argument, and produces an approximate jackknife estimate of the variance. If the `id` variable were not unique, but instead identifies clusters of correlated observations, then the variance estimate is based on a grouped jackknife.

CONVERGENCE

In certain data cases the actual MLE estimate of a coefficient is infinity, e.g., a dichotomous variable where one of the groups has no events. When this happens the associated coefficient grows at a steady pace and a race condition will exist in the fitting routine: either the log likelihood converges, the information matrix becomes effectively singular, an argument to exp becomes too large for the computer hardware, or the maximum number of interactions is exceeded. The routine attempts to detect when this has happened, not always successfully.

PENALISED REGRESSION

`coxph` can now maximise a penalised partial likelihood with arbitrary user-defined penalty. Supplied penalty functions include ridge regression (ridge), smoothing splines (pspline), and frailty models (frailty).

References

P. Andersen and R. Gill. "Cox's regression model for counting processes, a large sample study", *Annals of Statistics,* 10:1100-1120, 1982.

T. Therneau, P. Grambsch, and T. Fleming. "Martingale based residuals for survival models", *Biometrika,* March 1990.

See Also

cluster, survfit, Surv, strata,ridge, pspline,frailty.

Examples

```
# Create the simplest test data set
#
 test1 <- list(time=  c(4, 3,1,1,2,2,3),
                status=c(1,NA,1,0,1,1,0),
                x=     c(0, 2,1,1,1,0,0),
                sex=   c(0, 0,0,0,1,1,1))
 coxph( Surv(time, status) ~ x + strata(sex), test1)  #stratified model

#
# Create a simple data set for a time-dependent model
#
test2 <- list(start=c(1, 2, 5, 2, 1, 7, 3, 4, 8, 8),
                stop =c(2, 3, 6, 7, 8, 9, 9, 9,14,17),
                event=c(1, 1, 1, 1, 1, 1, 1, 0, 0, 0),
                x    =c(1, 0, 0, 1, 0, 1, 1, 1, 0, 0) )

summary( coxph( Surv(start, stop, event) ~ x, test2))
```

---

coxph.detail *Details of a Cox Model Fit*

Description

Details of a Cox model fit. Returns the individual contributions to the first and second derivative matrix, at each unique event time.

Usage

```
coxph.detail(object)
```

Arguments

object	a Cox model object, i.e., the result of coxph.

Details

This function may be useful for those who wish to investigate new methods or extensions to the Cox model. The example below shows one way to calculate the Schoenfeld residuals.

Value

a list with components

time	the vector of unique event times
nevent	the number of events at each of these time points.
means	a matrix with one row for each event time and one column for each variable in the Cox model, containing the weighted mean of the variable at that time, over all subjects still at risk at that time. The weights are the risk weights exp(x %*% fit$coef).
nrisk	number of subjects at risk.
hazard	the hazard increment.
score	the contribution to the score vector (first derivative of the log partial likelihood) at each time point.
imat	the contribution to the information matrix (second derivative of the log partial likelihood) at each time point.
varhaz	the variance of the hazard increment.
x,y	copies of the input data.
strata	only present for a stratified Cox model, this is a table giving the number of time points of component time that were contributed by each of the strata.

See Also

coxph, residuals.coxph

Examples

```
fit   <- coxph(Surv(futime,fustat) ~ age + rx + ecog.ps, ovarian, x=TRUE)
fitd  <- coxph.detail(fit)
events <- fit$y[,2]==1
etime  <- fit$y[events,1]   #the event times --- may have duplicates
indx   <- match(etime, fitd$time)
sresid <- fit$x[events,] - fitd$means[indx,]
```

---

coxph.object *Proportional Hazards Regression Object*

Description

This class of objects is returned by the coxph class of functions to represent a fitted proportional hazards model.

Objects of this class have methods for the functions print, summary, residuals, predict and survfit.

COMPONENTS

The following components must be included in a legitimate `coxph` object.

coefficients the coefficients of the linear predictor, which multiply the columns of the model matrix. If the model is over-determined there will be missing values in the vector corresponding to the redundant columns in the model matrix.

var the variance matrix of the coefficients. Rows and columns corresponding to any missing coefficients are set to zero.

naive.var this component will be present only if the `robust` option was true. If so, the `var` component will contain the robust estimate of variance, and this component will contain the ordinary estimate.

loglik a vector of length 2 containing the log-likelihood with the initial values and with the final values of the coefficients.

score value of the efficient score test, at the initial value of the coefficients.

rscore the robust log-rank statistic, if a robust variance was requested.

wald.test the Wald test of whether the final coefficients differ from the initial values.

iter number of iterations used.

linear.predictors the vector of linear predictors, one per subject.

residuals the martingale residuals.

means vector of column means of the X matrix. Subsequent survival curves are adjusted to this value.

n the number of observations used in the fit.

weights the vector of case weights, if one was used.

method the computation method used.

na.action the `na.action` attribute, if any, that was returned by the `na.action` routine.

The object will also contain the following, for documentation see the `lm` object: `terms`, `assign`, `formula`, `call`, and, optionally, `x`, `y`, and/or `frame`.

See Also

`coxph`, `coxph.detail`, `cox.zph`, `survfit`, `residuals.coxph`, `survreg`

`date.ddmmmyy` *Format a Julian date*

Description

Given a vector of Julian dates, this returns them in the form "10Nov89", "28Jul54", etc.

Usage

```
date.ddmmmyy(sdate)
```

Arguments

`sdate`	A vector of Julian dates, e.g., as returned by `mdy.date()`.

Value

A vector of character strings containing the formatted dates.

See Also

mdy.date, date.mdy

Examples

```
date.ddmmmyy(1:10)
```

---

date.mdy *Convert from Julian Dates to Month, Day, and Year*

Description

Convert a vector of Julian dates to a list of vectors with the corresponding values of month, day and year, and optionally weekday.

Usage

```
date.mdy(sdate, weekday = FALSE)
```

Arguments

sdate	a Julian date value, as returned by mdy.date(), number of days since 1/1/1960.
weekday	if TRUE, then the returned list also will contain the day of the week (Sunday=1, Saturday=7).

Value

A list with components month, day, and year.

References

Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (1992). *Numerical Recipes: The Art of Scientific Computing (Second Edition)*. Cambridge University Press.

Examples

```
day <- 7
temp <- date.mdy(mdy.date(month = 7, day = day, year = 1960))
## Check for illegal dates, such as 29 Feb in a non leap year
if (temp$day != day) {
  cat("Some illegal dates\n")
} else {
  cat("All days are legal\n")
}
```

date.mmddyy *Format a Julian date*

Description

Given a vector of Julian dates, this returns them in the form "10/11/89", "28/7/54", etc.

Usage

```
date.mmddyy(sdate, sep = "/")
```

Arguments

sdate	A vector of Julian dates, e.g., as returned by `mdy.date()`.
sep	Character string used to separate the month, day, and year portions of the returned string.

Value

A vector of character strings containing the formatted dates.

See Also

date.mdy, mdy.date, date.ddmmmyy

Examples

```
date.mmddyy(as.date(10))
```

date.mmddyyyy *Format a Julian date*

Description

Given a vector of Julian dates, this returns them in the form "10/11/1989", "28/7/1854", etc.

Usage

```
date.mmddyyyy(sdate, sep = "/")
```

Arguments

sdate	A vector of Julian dates, e.g., as returned by `mdy.date()`.
sep	Character string used to separate the month, day, and year portions of the returned string.

Value

A vector of character strings containing the formatted dates.

See Also

date.mdy, mdy.date, date.ddmmmyy

Examples

```
date.mmddyyyy(as.date(1:10))
```

---

date.object *Date Objects*

Description

Objects of class "date".

Usage

```
is.date(x)
```

Arguments

x	any R object.

Details

Dates are stored as the number of days since 1/1/1960, and are kept in integer format. (This is the same baseline value as is used by SAS). The numerical methods for dates treat date - date as a numeric, and date +- numeric as a date.

is.date returns TRUE if x has class "date", and FALSE otherwise. Its behavior is unaffected by any attributes of x; for example, x could be a date array (in contrast to the behavior of is.vector).

as.date returns x if x is a simple object of class "date", and otherwise a date vector of the same length as x and with data resulting from coercing the elements of x to class "date". See the manual page for as.date() for details.

Logical operations as well as the numeric functions exp(), log(), and so on are invalid.

Other methods exist for missing value, as.character(), printing, and summarizing.

See Also

date.mdy, mdy.date, date.ddmmmyy, as.date.

`frailty` *(Approximate) Frailty models*

Description

When included in a coxph or survreg, fits by penalised likelihood a random effects (frailty) model. `frailty` is generic, with methods for t, Gaussian and Gamma distributions.

Usage

```
frailty(x, distribution="gamma", ...)
frailty.gamma(x, sparse = (nclass > 5), theta, df, eps = 1e-05, method = c("em",
frailty.gaussian(x, sparse = (nclass > 5), theta, df, method = c("reml","aic", "
frailty.t(x, sparse = (nclass > 5), theta, df, eps = 1e-05, tdf = 5,method = c("
```

Arguments

`x`	group indicator
`distribution`	frailty distribution
`...`	Arguments for specific distribution, including (but not limited to)
`sparse`	Use sparse Newton-Raphson algorithm
`df`	Approximate degrees of freedom
`theta`	Penalty
`eps`	Accuracy of `df`
`method`	maximisation algorithm
`tdf`	df of t-distribution

Details

The penalised likelihood method is equivalent to maximum (partial) likelihood for the gamma frailty but not for the others.

The sparse algorithm uses the diagonal of the information matrix for the random effects, which saves a lot of space.

The frailty distributions are really the log-t and lognormal: t and Gaussian are random effects on the scale of the linear predictor.

Value

An object of class `coxph.penalty` containing a factor with attributes specifying the control functions.

References

Therneau TM, Grambsch PM, Pankratz VS (2003) "Penalized survival models and frailty" Journal of Computational and Graphical Statistics 12, 1: 156-175

See Also

`coxph`,`survreg`,`ridge`,`pspline`

Examples

```
kfit <- coxph(Surv(time, status)~ age + sex + disease + frailty(id), kidney)
kfit0 <- coxph(Surv(time, status)~ age + sex + disease, kidney)
kfitm1 <- coxph(Surv(time,status) ~ age + sex + disease +
                frailty(id, dist='gauss'), kidney)
coxph(Surv(time, status) ~ age + sex + frailty(id, dist='gauss', method='aic',caic=TRUE),
# uncorrected aic
coxph(Surv(time, status) ~ age + sex + frailty(id, method='aic', caic=FALSE), kidney)

rfit2a <- survreg(Surv(time, status) ~ rx +
                  frailty.gaussian(litter, df=13, sparse=FALSE), rats )
rfit2b <- survreg(Surv(time, status) ~ rx +
                  frailty.gaussian(litter, df=13, sparse=TRUE), rats )
rfit2a
rfit2b
```

---

heart *Stanford Heart Transplant data*

Description

Survival of patients on the waiting list for the Stanford heart transplant program.

Usage

```
data(heart)
data(jasa)
data(jasa1)
```

Format

jasa: original data

birth.dt:	birth date
accept.dt:	acceptance into program
tx.date:	transplant date
fu.date:	end of followup
fustat:	dead or alive
surgery:	prior bypass surgery
age:	age (in days)
futime:	followup time
wait.time:	time before transplant
transplant:	transplant indicator
mismatch:	mismatch score
hla.a2:	particular type of mismatch
mscore:	another mismatch score
reject:	rejection occurred

jasa1, heart: processed data

start, stop, event:	Entry and exit time and status for this interval of time
age:	age-48 years

year:	year of acceptance (in years after 1 Nov 1967)
surgery:	prior bypass surgery 1=yes
transplant:	received transplant 1=yes
id:	patient id

Source

Crowley and Hu, JASA

See Also

stanford2

---

is.ratetable *Verify that an Object is of Class 'ratetable'*

Description

The function verifies not only the `class` attribute, but the structure of the object.

Usage

```
is.ratetable(x, verbose=FALSE)
```

Arguments

`x`	the object to be verified.
`verbose`	if `TRUE` and the object is not a ratetable, then return a character string describing the way(s) in which `x` fails to be a proper ratetable object.

Details

Rate tables are used by the `pyears` and `survexp` functions, and normally contain death rates for some population, categorized by age, sex, or other variables. They have a fairly rigid structure, and the `verbose` option can help in creating a new rate table.

Value

returns `TRUE` if `x` is a ratetable, and `FALSE` or a description if it is not.

See Also

pyears, survexp

Examples

```
is.ratetable(survexp.us)  ##Yes
is.ratetable(cancer)  ##No
```

kidney *Kidney data from survival5*

Description

Data on the recurrence times to infection, at the point of insertion of the catheter, for kidney patients using portable dialysis equipment. Catheters may be removed for reasons other than infection, in which case the observation is censored. Each patient has exactly 2 observations.

Format

patient:	id
time:	time
status:	event status
age:	in years
sex:	1=male, 2=female
disease:	disease type (0=GN, 1=AN, 2=PKD, 3=Other)
frail:	frailty estimate from original paper

Note

The original analysis had incorrect handling of ties and so is not exactly reproduced by survival.

Source

McGilchrist and Aisbett, Biometrics 47, 461-66, 1991

Examples

```
data(kidney)
kfit <- coxph(Surv(time, status)~ age + sex + disease + frailty(id), kidney)
kfit0 <- coxph(Surv(time, status)~ age + sex + disease, kidney)
kfitm1 <- coxph(Surv(time,status) ~ age + sex + disease +
                frailty(id, dist='gauss'), kidney)
```

lines.survfit *Add Lines or Points to a Survival Plot*

Description

Often used to add the expected survival curve(s) to a Kaplan-Meier plot generated with `plot.survfit`.

Usage

```
## S3 method for class 'survfit':
lines(x, type="s", mark=3, col=1, lty=1,
lwd=1, mark.time=TRUE , xscale=1,  firstx=0, firsty=1, xmax, fun, conf.int=FALSE
## S3 method for class 'survfit':
points(x, ...)
```

Arguments

x	a survival object, generated from the survfit or survexp functions.
type	the line type, as described in lines. The default is a step function for survfit objects, and a connected line for survexp objects.
mark	vectors giving the mark symbol, color, line type and line width for the added curves.
mark.time	controls the labeling of the curves. If FALSE, no labeling is done. If TRUE, then curves are marked at each censoring time. If mark.time is a numeric vector, then curves are marked at the specified time points.
xscale	a number used to divide the x values. If time was originally in days, a value of 365.24 would give a plotted scale in years.
firstx,firsty	the starting point for the survival curves. If either of these is set to NA or <blank> the plot will start at the first time point of the curve.
col,lty,lwd,...	passed to lines
xmax	the maximum horizontal plot coordinate. This shortens the curve before plotting it, so unlike using the xlim graphical parameter, warning messages about out of bounds points are not generated.
fun	an arbitrary function defining a transformation of the survival curve. For example fun=log is an alternative way to draw a log-survival curve (but with the axis labeled with log(S) values). Four often used transformations can be specified with a character argument instead: "log" is the same as using the log=T option, "event" plots cumulative events (f(y) =1-y), "cumhaz" plots the cumulative hazard function (f(y) = -log(y)) and "cloglog" creates a complementary log-log survival plot (f(y) = log(-log(y)) along with log scale for the x-axis).
conf.int	if TRUE, confidence bands for the curves are also plotted. If set to "only", then only the CI bands are plotted, and the curve itself is left off. This can be useful for fine control over the colors or line types of a plot.

Value

a list with components x and y, containing the coordinates of the last point on each of the curves (but not of the confidence limits). This may be useful for labeling.

Side Effects

one or more curves are added to the current plot.

See Also

lines, par, plot.survfit, survfit, survexp.

Examples

```
fit <- survfit(Surv(time, status) ~ sex, pbc,subset=1:312)
plot(fit, mark.time=FALSE, xscale=365.24,
        xlab='Years', ylab='Survival')
lines(fit[1], lwd=2, xscale=365.24)    #darken the first curve and add marks

# add expected survival curves for the two groups,
#   based on the US census data
tdata <- data.frame(age=pbc$age*365.24, sex=pbc$sex +1,
                    year= rep(mdy.date(1,1,1976), nrow(pbc)))
tdata<-tdata[1:312,] ## only the randomised people, with no missing data

efit <- survexp(~ sex+ratetable(sex=sex,age=age,year=year), data=tdata, ratetable=survexp
temp <- lines(efit, lty=2, xscale=365.24, lwd=2:1)
text(temp, c("Male", "Female"), adj= -.1) #labels just past the ends
title(main="Primary Biliary Cirrhosis, Observed and Expected")
```

---

lung *Mayo Clinic Lung Cancer Data*

Description

Survival in patients with lung cancer at Mayo Clinic. Performance scores rate how well the patient can perform usual daily activities.

Usage

```
data(lung)
data(cancer)
```

Format

inst:	Institution code
time:	Survival time in days
status:	censoring status 1=censored, 2=dead
age:	Age in years
sex:	Male=1 Female=2
ph.ecog:	ECOG performance score (0=good 5=dead)
ph.karno:	Karnofsky performance score (bad=0-good=100) rated by physician
pat.karno:	Karnofsky performance score rated by patient
meal.cal:	Calories consumed at meals
wt.loss:	Weight loss in last six months

Source

Terry Therneau

`mdy.date` *Convert to Julian Dates*

Description

Given a month, day, and year, returns the number of days since January 1, 1960.

Usage

```
mdy.date(month, day, year, nineteen = TRUE, fillday = FALSE,
         fillmonth = FALSE)
```

Arguments

`month`	vector of months.
`day`	vector of days.
`year`	vector of years.
`nineteen`	if `TRUE`, year values between 0 and 99 are assumed to be in the 20th century A.D.; otherwise, if `FALSE`, they are assumed to be in the 1st century A.D.
`fillday`	if `TRUE`, then missing days are replaced with 15.
`fillmonth`	if `TRUE`, then a missing month causes the month and day to be set to 7/1.

Details

The date functions are particularly useful in computing time spans, such as number of days on test, and similar functions can be found in other statistical packages. The baseline date of Jan 1, 1960 is, of course, completely arbitrary (it is the same one used by SAS).

The `fillday` and `fillmonth` options are perhaps useful only to the author and a very few others: we sometimes deal with patients whose birth date was in the 1800's, and only the month or even only the year is known. When the interval is greater than 80 years, a filler seems defensible.

Value

a vector of Julian dates.

References

Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (1992). *Numerical Recipes: The Art of Scientific Computing (Second Edition)*. Cambridge University Press.

See Also

`date.mmddyy`, `date.ddmmmyy`, `date.mmddyyyy`

Examples

```
mdy.date(3, 10, 53)
xzt <-1:10
xzy <- as.date(xzt)
test <- data.frame(x = xzt, date = xzy)
summary(test)
```

nwtco *Data from the National Wilm's Tumor Study*

Description

Missing data/masurement error example. Tumor histology predicts survival, but prediction is stronger with central lab histology than with the local institution determination.

Usage

```
data(nwtco)
```

Format

A data frame with 4028 observations on the following 9 variables.

seqno id number

instit Histology from local institution

histol Histology from central lab

stage Disease stage

study study

rel indicator for relapse

edrel time to relapse

age age in months

in.subcohort Included in the subcohort for the example in the paper

Source

http://faculty.washington.edu/norm/software.html

References

N.E. Breslow and N. Chatterjee, "Design and analysis of two-phase studies with binary outcome applied to Wilms tumour prognosis" Applied Statistics 48:457-68, 1999

Examples

```
data(nwtco)
with(nwtco, table(instit,histol))
anova(coxph(Surv(edrel,rel)~histol+instit,data=nwtco))
anova(coxph(Surv(edrel,rel)~instit+histol,data=nwtco))
```

ovarian *Ovarian cancer survival data*

Description

Survival in a randomised trial comparing two treatments for ovarian cancer (? from an Eastern Cooperative Oncology Group study)

Usage

```
data(ovarian)
```

Format

futime:	survival or censoring time
fustat:	censoring status
age:	in years
resid.ds:	residual disease present (1=no,2=yes)
rx:	treatment group
ecog.ps:	ECOG performance status (1 is better, see reference)

Source

Terry Therneau

References

for ECOG performance status: http://ecog.dfci.harvard.edu/~ecogdba/general/perf_stat.html

pbc *Mayo Clinic Primary Biliary Cirrhosis Data*

Description

Followup of 312 randomised and 108 unrandomised patients with primary biliary cirrhosis, a rare autoimmune liver disease, at Mayo Clinic. Most variables have some missing data, coded as -9.

Usage

```
data(pbc)
```

Format

age:	in years
alb:	serum albumin
alkphos:	alkaline phosphotase
ascites:	presence of ascites
bili:	serum bilirunbin
chol:	serum cholesterol
edema:	presence of edema
edtrt:	0 no edema, 0.5 untreated or successfully treated
	1 unsuccessfully treated edema
hepmeg:	enlarged liver
time:	survival time
platelet:	platelet count
protime:	standardised blood clotting time
sex:	1=male
sgot:	liver enzyme (now called AST)
spiders:	blood vessel malformations in the skin
stage:	histologic stage of disease (needs biopsy)
status:	censoring status
trt:	1/2/-9 for control, treatment, not randomised
trig:	triglycerides
copper:	urine copper

Source

Fleming TR, Harrington DP. "Counting Processes and Survival Analysis" Wiley:New York

---

plot.cox.zph *Graphical Test of Proportional Hazards*

Description

Displays a graph of the scaled Schoenfeld residuals, along with a smooth curve.

Usage

```
## S3 method for class 'cox.zph':
plot(x, resid=TRUE, se=TRUE, df=4, nsmo=40, var, ...)
```

Arguments

x	result of the cox.zph function.
resid	a logical value, if TRUE the residuals are included on the plot, as well as the smooth fit.
se	a logical value, if TRUE, confidence bands at two standard errors will be added.
df	the degrees of freedom for the fitted natural spline, df=2 leads to a linear fit.
nsmo	number of points used to plot the fitted spline.

var	the set of variables for which plots are desired. By default, plots are produced in turn for each variable of a model. Selection of a single variable allows other features to be added to the plot, e.g., a horizontal line at zero or a main title. This has been superseded by a subscripting method; see the example below.
...	additional arguments passed to the plot function.

Side Effects

a plot is produced on the current graphics device.

See Also

cox.zph, coxph

Examples

```
vfit <- coxph(Surv(time,status) ~ trt + factor(celltype) +
              karno + age, data=veteran, x=TRUE)
temp <- cox.zph(vfit)
plot(temp, var=5)      # Look at Karnofsy score, old way of doing plot
plot(temp[5])     # New way with subscripting
abline(0, 0, lty=3)
# Add the linear fit as well
abline(lm(temp$y[,5] ~ temp$x)$coefficients, lty=4, col=3)
title(main="VA Lung Study")
```

---

plot.survfit *Plot Method for 'survfit'*

Description

A plot of survival curves is produced, one curve for each strata. The log=T option does extra work to avoid log(0), and to try to create a pleasing result. If there are zeros, they are plotted by default at 0.8 times the smallest non-zero value on the curve(s).

If legend.text is supplied a legend is created.

Usage

```
## S3 method for class 'survfit':
plot(x, conf.int=, mark.time=TRUE,
 mark=3, col=1, lty=1, lwd=1, cex=1, log=FALSE, xscale=1, yscale=1,
 firstx=0, firsty=1, xmax, ymin=0, fun,
 xlab="", ylab="", xaxs="S", bty=NULL,
legend.text=NULL,legend.pos=0,legend.bty="n",main=NULL,...)
```

Arguments

x	an object of class survfit, usually returned by the survfit function.
conf.int	determines whether confidence intervals will be plotted. The default is to do so if there is only 1 curve, i.e., no strata.

mark.time — controls the labeling of the curves. If set to FALSE, no labeling is done. If TRUE, then curves are marked at each censoring time which is not also a death time. If mark.time is a numeric vector, then curves are marked at the specified time points.

mark — vector of mark parameters, which will be used to label the curves. The lines help file contains examples of the possible marks. The vector is reused cyclically if it is shorter than the number of curves.

col — a vector of integers specifying colors for each curve. The default value is 1.

lty — a vector of integers specifying line types for each curve. The default value is 1.

lwd — a vector of numeric values for line widths. The default value is 1.

cex — a numeric value specifying the size of the marks. Not a vector; all marks have the same size.

log — a logical value, if TRUE the y axis wll be on a log scale. Alternately, one of the standard character strings "x", "y", or "xy" can be given to specific logarithmic horizontal and/or vertical axes.

xscale — scale the x-axis values before plotting. If time were in days, then a value of 365.25 will give labels in years instead of the original days.

yscale — will be used to multiply the labels on the y axis. A value of 100, for instance, would be used to give a percent scale. Only the labels are changed, not the actual plot coordinates, so that adding a curve with "lines(surv.exp(...{}))", say, will perform as it did without the yscale argument.

firstx,firsty — the starting point for the survival curves. If either of these is set to NA the plot will start at the first time point of the curve.

xmax — the maximum horizontal plot coordinate. This can be used to shrink the range of a plot. It shortens the curve before plotting it, so that unlike using the xlim graphical parameter, warning messages about out of bounds points are not generated.

ymin — lower boundary for y values. Survival curves are most often drawn in the range of 0-1, even if none of the curves approach zero. The parameter is ignored if the fun argument is present, or if it has been set to NA.

fun — an arbitrary function defining a transformation of the survival curve. For example fun=log is an alternative way to draw a log-survival curve (but with the axis labeled with log(S) values), and fun=sqrt would generate a curve on square root scale. Four often used transformations can be specified with a character argument instead: "log" is the same as using the log=T option, "event" plots cumulative events (f(y) = 1-y), "cumhaz" plots the cumulative hazard function (f(y) = -log(y)), and "cloglog" creates a complimentary log-log survival plot (f(y) = log(-log(y)) along with log scale for the x-axis).

xlab — label given to the x-axis.

ylab — label given to the y-axis.

xaxs — either "S" for a survival curve or a standard x axis style as listed in par. Survival curves are usually displayed with the curve touching the y-axis, but not touching the bounding box of the plot on the other 3 sides. Type "S" accomplishes this by manipulating the plot range and then using the "i" style internally.

bty — see par

`legend.pos`	position for the legend: a vector of length 2, or 0 to put the legend in the lower left, 1 to put it in the upper right. These may not work well with transformed curves.
`legend.bty`	Box type, see `legend`
`legend.text`	Text for legend, see `legend`
`main`	Plot title
`...`	other graphical parameters

Value

a list with components `x` and `y`, containing the coordinates of the last point on each of the curves (but not the confidence limits). This may be useful for labeling.

BUGS

Survival curve objects created from a `coxph` model does not include the censoring times. Therefore, specifying `mark.time=T` does not work. If you want to mark censoring times on the curve(s) resulting from a `coxph` fit, provide a vector of times as the `mark.time` argument in the call to `plot.survfit` or `lines.survfit`.

See Also

`par`, `survfit`, `lines.survfit` `legend`.

Examples

```
leukemia.surv <- survfit(Surv(time, status) ~ x, data = aml)
plot(leukemia.surv, lty = 2:3)
legend(100, .9, c("Maintenance", "No Maintenance"), lty = 2:3)
title("Kaplan-Meier Curves\nfor AML Maintenance Study")

lsurv2 <- survfit(Surv(time, status) ~ x, aml, type='fleming')
plot(lsurv2, lty=2:3, fun="cumhaz",
        xlab="Months", ylab="Cumulative Hazard")

plot(leukemia.surv,lty=1:2,legend.pos=0,col=c("Red","Blue"),legend.text=c("Maintenance",
"No Maintenance"))
if (interactive()){
cat("Click on the graphics device to place a legend\n")
plot(leukemia.surv,lty=2:3,legend.pos=locator(1),legend.text=c("Maintenance",
"No Maintenance"))
}
```

`predict.coxph` *Predictions for Cox model*

Description

Compute fitted values and regression terms for a model fitted by `coxph`

Usage

```
## S3 method for class 'coxph':
predict(object, newdata, type=c("lp", "risk", "expected", "terms"), se.fit=FALSE
```

Arguments

object	A coxph object
newdata	Optional new data to predict at
type	Type of prediction wanted
se.fit	Return standard errors as well?
terms	If type="terms", which terms to return.
collapse	identifier for which rows correspond to different individuals
safe	Use a more intensive 'safe' prediction method
...	other unused arguments

Value

A vector or matrix of fitted values. If se.fit=TRUE a list whose first component is the fitted values and second is the standard errors.

See Also

predict,coxph,termplot

Examples

```
fit<-coxph(Surv(time,status)~x,data=aml)
predict(fit,type="lp")
predict(fit,type="risk")
predict(fit,type="expected")
predict(fit,type="terms")
predict(fit,type="lp",se.fit=TRUE)
predict(fit,type="risk",se.fit=TRUE)
predict(fit,type="expected",se.fit=TRUE)
predict(fit,type="terms",se.fit=TRUE)
```

---

predict.survreg *Predicted Values for a 'survreg' Object*

Description

Predicted values for a survreg object

Usage

```
## S3 method for class 'survreg':
predict(object, newdata,
type=c("response", "link", "lp", "linear",  "terms", "quantile",
        "uquantile"),
se.fit=FALSE, terms=NULL, p=c(0.1, 0.9),...)
```

Arguments

`object`	result of a model fit using the `survreg` function.
`newdata`	data for prediction. If absent, predictions are for the subjects used in the original fit.
`type`	the type of predicted value. This can be on the original scale of the data (response), the linear predictor (`"linear"`, with `"lp"` as an allowed abbreviation), a predicted quantile on the original scale of the data (`"quantile"`), a quantile on the linear predictor scale (`"uquantile"`), or the matrix of terms for the linear predictor (`"terms"`). At this time `"link"` and linear predictor (`"lp"`) are identical.
`se.fit`	if TRUE, include the standard errors of the prediction in the result.
`terms`	subset of terms. The default for residual type `"terms"` is a matrix with one column for every term (excluding the intercept) in the model.
`p`	vector of percentiles. This is used only for quantile predictions.
`...`	other arguments

Value

a vector or matrix of predicted values.

References

Escobar and Meeker (1992). Assessing influence in regression analysis with censored data. *Biometrics,* 48, 507-528.

See Also

`survreg`, `residuals.survreg`

Examples

```
# Draw figure 1 from Escobar and Meeker
fit <- survreg(Surv(time,status) ~ age + age^2, data=stanford2,
        dist='lognormal')
plot(stanford2$age, stanford2$time, xlab='Age', ylab='Days',
        xlim=c(0,65), ylim=c(.01, 10^6), log='y')
pred <- predict(fit, newdata=list(age=1:65), type='quantile',
                 p=c(.1, .5, .9))
matlines(1:65, pred, lty=c(2,1,2), col=1)
```

`print.survfit` *Print a Short Summary of a Survival Curve*

Description

Print number of observations, number of events, the restricted mean survival and its standard error, and the median survival with confidence limits for the median.

Usage

```
## S3 method for class 'survfit':
print(x, scale=1,digits = max(options()$digits - 4,
3), print.n=getOption("survfit.print.n"), show.rmean=getOption("survfit.print.me
```

Arguments

x	the result of a call to the survfit function.
print.n	What to use for number of subjects (see below)
digits	Number of digits to print
scale	a numeric value to rescale the survival time, e.g., if the input data to survfit were in days, scale=365 would scale the printout to years.
show.rmean	Show the restricted mean survival?
...	other unused arguments

Details

The restricted mean (rmean) and its standard error se(rmean) are based on a truncated estimator. If the last observation(s) is not a death, then the survival curve estimate does not go to zero and the mean survival time cannot be estimated. Instead, the quantity reported is the mean of survival restricted to the time before the last censoring. When the last censoring time is not random this quantity is occasionally of interest.

Any randomness in the last censoring time is not taken into account in computing the standard error of the restricted mean. The restricted mean is available mainly for compatibility with S, and is not shown by default.

The median and its confidence interval are defined by drawing a horizontal line at 0.5 on the plot of the survival curve and its confidence bands. The intersection of the line with the lower CI band defines the lower limit for the median's interval, and similarly for the upper band. If any of the intersections is not a point, then we use the smallest point of intersection, e.g., if the survival curve were exactly equal to 0.5 over an interval.

The "number of observations" is not well-defined for counting process data. Previous versions of this code used the number at risk at the first time point. This is misleading if many individuals enter late or change strata. The original S code for the current version uses the number of records, which is misleading when the counting process data actually represent a fixed cohort with time-dependent covariates.

Four possibilities are provided, controlled by print.n or by options(survfit.print.n): "none" prints NA, "records" prints the number of records, "start" prints the number at the first time point and "max" prints the maximum number at risk. The initial default is "start".

Value

x, with the invisible flag set.

Side Effects

The number of observations (see Details), the number of events, the median survival with its confidence interval, and optionally the restricted mean survival (rmean) and its standard error, are printed. If there are multiple curves, there is one line of output for each.

See Also

summary.survfit, survfit.object, survfit

Examples

```
##effect of print.n and show.rmean

a<-coxph(Surv(start,stop,event)~age+strata(transplant),data=heart)
b<-survfit(a)
print(b,print.n="none")
print(b,print.n="records")
print(b,print.n="start")
print(b,print.n="max")
print(b,show.rmean=TRUE)
```

---

pspline *Penalised smoothing splines*

Description

Specifies a penalised spline basis for the predictor. This is done by fitting a comparatively small set of splines and penalising the integrated second derivative. Results are similar to smoothing splines with a knot at each data point but computationally simpler.

Usage

```
pspline(x, df=4, theta, nterm=2.5 * df, degree=3, eps=0.1, method, ...)
```

Arguments

x	predictor
df	approximate degrees of freedom. df=0 means use AIC
theta	roughness penalty
nterm	number of splines in the basis
degree	degree of splines
eps	accuracy for df
method	Method for automatic choice of theta
...	I don't know what this does

Value

Object of class coxph.penalty containing the spline basis with attributes specifying control functions.

See Also

coxph,survreg,ridge,frailty

Examples

```
lfit6 <- survreg(Surv(time, status)~pspline(age, df=2), cancer)
plot(cancer$age, predict(lfit6), xlab='Age', ylab="Spline prediction")
title("Cancer Data")
fit0 <- coxph(Surv(time, status) ~ ph.ecog + age, cancer)
fit1 <- coxph(Surv(time, status) ~ ph.ecog + pspline(age,3), cancer)
fit3 <- coxph(Surv(time, status) ~ ph.ecog + pspline(age,8), cancer)
fit0
fit1
fit3
```

---

pyears *Person Years*

Description

This function computes the person-years of follow-up time contributed by a cohort of subjects, stratified into subgroups. It also computes the number of subjects who contribute to each cell of the output table, and optionally the number of events and/or expected number of events in each cell.

Usage

```
pyears(formula, data, weights, subset, na.action, ratetable=survexp.us,
scale=365.25, expect=c('event', 'pyears'), model=FALSE, x=FALSE, y=FALSE)
```

Arguments

formula	a formula object. The response variable will be a vector of follow-up times for each subject, or a Surv object containing the follow-up time and an event indicator. The predictors consist of optional grouping variables separated by + operators (exactly as in survfit), time-dependent grouping variables such as age (specified with tcut), and optionally a ratetable() term. This latter matches each subject to his/her expected cohort.
data	a data frame in which to interpret the variables named in the formula, or in the subset and the weights argument.
weights	case weights.
subset	expression saying that only a subset of the rows of the data should be used in the fit.
na.action	a missing-data filter function, applied to the model.frame, after any subset argument has been used. Default is options()$na.action.
ratetable	a table of event rates, such as survexp.uswhite.
scale	a scaling for the results. As most rate tables are in units/day, the default value of 365.25 causes the output to be reported in years.
expect	should the output table include the expected number of events, or the expected number of person-years of observation. This is only valid with a rate table.
model,x,y	If any of these is true, then the model frame, the model matrix, and/or the vector of response times will be returned as components of the final result.

Details

Because pyears may have several time variables, it is necessary that all of them be in the same units. For instance, in the call py <- pyears(futime rx + ratetable(age=age, sex=sex, year=entry.dt)) with a ratetable whose natural unit is days, it is important that futime, age and entry.dt all be in days. Given the wide range of possible inputs, it is difficult for the routine to do sanity checks of this aspect.

A special function tcut is needed to specify time-dependent cutpoints. For instance, assume that age is in years, and that the desired final arrays have as one of their margins the age groups 0-2, 2-10, 10-25, and 25+. A subject who enters the study at age 4 and remains under observation for 10 years will contribute follow-up time to both the 2-10 and 10-25 subsets. If cut(age, c(0,2,10,25,100)) were used in the formula, the subject would be classified according to his starting age only. The tcut function has the same arguments as cut, but produces a different output object which allows the pyears function to correctly track the subject.

The results of pyears are normally used as input to further calculations. The example below is from a study of hip fracture rates from 1930 - 1990 in Rochester, Minnesota. Survival post hip fracture has increased over that time, but so has the survival of elderly subjects in the population at large. A model of relative survival helps to clarify what has happened: Poisson regression is used, but replacing exposure time with expected exposure (for an age and sex matched control). Death rates change with age, of course, so the result is carved into 1 year increments of time. Males and females were done separately.

Value

a list with components

pyears	an array containing the person-years of exposure. (Or other units, depending on the rate table and the scale).
n	an array containing the number of subjects who contribute time to each cell of the pyears array.
event	an array containing the observed number of events. This will be present only if the response variable is a Surv object.
expected	an array containing the expected number of events (or person years). This will be present only if there was a ratetable term.
offtable	the number of person-years of exposure in the cohort that was not part of any cell in the pyears array. This is often useful as an error check; if there is a mismatch of units between two variables, nearly all the person years may be off table.
summary	a summary of the rate-table matching. This is also useful as an error check.
call	an image of the call to the function.
na.action	the na.action attribute contributed by an na.action routine, if any.

See Also

ratetable, survexp, Surv

Examples

```
#
# Simple case: a single male subject, born 6/6/36 and entered on study 6/6/55.
#
```

```
temp1 <- mdy.date(6,6,36)
temp2 <- mdy.date(6,6,55)
# Now compare the results from person-years
#
temp.age <- tcut(temp2-temp1, floor(c(-1, (18:31 * 365.24))),
        labels=c('0-18', paste(18:30, 19:31, sep='-')))
temp.yr  <- tcut(temp2, mdy.date(1,1,1954:1965), labels=1954:1964)
temp.time <- 3700   #total days of fu
py1 <- pyears(temp.time ~ temp.age + temp.yr, scale=1) #output in days

survexp.uswhite<-survexp.usr[,,"white",]
py2 <- pyears(temp.time ~ temp.age + temp.yr
                + ratetable(age=temp2-temp1, year=temp2, sex=1),
              scale=1, ratetable=survexp.uswhite ) #output in days
```

`ratetable` *Ratetable reference in formula*

Description

This function matches variable names in data to those in a ratetable for `survexp`

Usage

```
ratetable(...)
```

Arguments

`...`	tags matching dimensions of the ratetable and variables in the data frame (see example)

Value

A data frame

See Also

`survexp`,`survexp.us`,`is.ratetable`

Examples

```
fit <- survfit(Surv(time, status) ~ sex, pbc,subset=1:312)
if (require(date)){
tdata <- data.frame(agedays=pbc$age*365.24, sex=pbc$sex +1,
                    year= rep(mdy.date(1,1,1976), nrow(pbc)))
tdata<-tdata[1:312,] ## only the randomised people, with no missing data

efit <- survexp(~ sex+ratetable(sex=sex,age=agedays,year=year), data=tdata, ratetable=sur
}
```

`ratetables` *Census Data Sets for the Expected Survival and Person Years Functions*

Description

Census data sets for the expected survival and person years functions

Usage

```
survexp.uswhite<-survexp.usr[,,"white",]
```

Details

us total United States population, by age and sex, 1960 to 1980.

uswhite United States white population, by age and sex, 1950 to 1980. This is no longer included, but can be extracted from `survexp.usr` as shown above.

usr United States population, by age, sex and race, 1960 to 1980. Race is white, nonwhite, or black. For 1960 and 1970 the black population values were not reported separately, so the nonwhite values were used.

mn total Minnesota population, by age and sex, 1970 and 1980.

mnwhite Minnesota white population, by age and sex, 1960 to 1980.

fl total Florida population, by age and sex, 1970 and 1980.

flr Florida population, by age, sex and race, 1970-1980. Race is white, nonwhite, or black. For 1970 the black population values were not reported separately, so the nonwhite values were used.

az total Arizona population, by age and sex, 1970 and 1980.

azr Arizona population, by age, sex and race, 1970-1980. Race is white versus nonwhite. For 1970 the nonwhite population values were not reported separately. In order to make the rate table be a matrix, the 1980 values were repeated. (White and non-white values are quite different).

Each of these tables contains the daily hazard rate for a matched subject from the population, defined as -log(1-q)/365.24 where q is the 1 year probability of death as reported in the original tables. For age 25 in 1970, for instance, p = 1-q is is the probability that a subject who becomes 25 years of age in 1970 will achieve his/her 26th birthday. The tables are recast in terms of hazard per day entirely for computational convenience. (The fraction .24 in the denominator is based on 24 leap years per century.)

Each table is stored as an array, with additional attributes, and can be subset and manipulated as standard S arrays. Interpolation between calendar years is done "on the fly" by the survexp routine.

Some of the deficiencies, e.g. 1970 Arizona non-white, are a result of local (Mayo Clinic) conditions. The data probably exists, but we don't have a copy it in the library.

The tables have been augmented to contain extrapolated values for 1990 and 2000. The details can be found in Mayo Clinic Biostatistics technical report 63 at http://www.mayo.edu/hsr/techrpt.html

rats *Rat data from survival5*

Description

48 rats were injected with a carcinogen, and then randomized to either drug or placebo. The number of tumors ranges from 0 to 13; all rats were censored at 6 months after randomization.

Usage

```
data(rats)
```

Format

rat:	id
rx:	treatment,(1=drug, 0=control)
observation:	within rat
start:	entry time
stop:	exit time
status:	event status

Source

Gail, Sautner and Brown, Biometrics 36, 255-66, 1980

residuals.coxph *Calculate Residuals for a 'coxph' Fit*

Description

Calculates martingale, deviance, score or Schoenfeld residuals for a Cox proportional hazards model.

Usage

```
## S3 method for class 'coxph':
residuals(object,
       type=c("martingale", "deviance", "score", "schoenfeld",
              "dfbeta", "dfbetas", "scaledsch","partial"),
       collapse=FALSE, weighted=FALSE, ...)
## S3 method for class 'coxph.null':
residuals(object,
       type=c("martingale", "deviance","score","schoenfeld"),
       collapse=FALSE, weighted=FALSE, ...)
```

Arguments

object	an object inheriting from class coxph, representing a fitted Cox regression model. Typically this is the output from the coxph function.
type	character string indicating the type of residual desired. Possible values are "martingale", "deviance", "score", "schoenfeld", "dfbeta"', "dfbetas", and "scaledsch". Only enough of the string to determine a unique match is required.
collapse	vector indicating which rows to collapse (sum) over. In time-dependent models more than one row data can pertain to a single individual. If there were 4 individuals represented by 3, 1, 2 and 4 rows of data respectively, then collapse=c(1,1,1, 2, 3,3, 4,4,4,4) could be used to obtain per subject rather than per observation residuals.
weighted	if TRUE and the model was fit with case weights, then the weighted residuals are returned.
...	other unused arguments

Value

For martingale and deviance residuals, the returned object is a vector with one element for each subject (without collapse). For score residuals it is a matrix with one row per subject and one column per variable. The row order will match the input data for the original fit. For Schoenfeld residuals, the returned object is a matrix with one row for each event and one column per variable. The rows are ordered by time within strata, and an attribute strata is attached that contains the number of observations in each strata. The scaled Schoenfeld residuals are used in the cox.zph function.

The score residuals are each individual's contribution to the score vector. Two transformations of this are often more useful: dfbeta is the approximate change in the coefficient vector if that observation were dropped, and dfbetas is the approximate change in the coefficients, scaled by the standard error for the coefficients.

NOTE

For deviance residuals, the status variable may need to be reconstructed. For score and Schoenfeld residuals, the X matrix will need to be reconstructed.

References

T. Therneau, P. Grambsch, and T. Fleming. "Martingale based residuals for survival models", *Biometrika*, March 1990.

See Also

coxph

Examples

```
fit <- coxph(Surv(start, stop, event) ~ (age + surgery)* transplant,
             data=heart)
mresid <- resid(fit, collapse=heart$id)
```

`residuals.survreg` *Compute Residuals for 'survreg' Objects*

Description

This is a method for the function `residuals` for objects inheriting from class `survreg`.

Usage

```
## S3 method for class 'survreg':
residuals(object, type=c("response", "deviance","dfbeta","dfbetas",
"working","ldcase","ldresp","ldshape", "matrix"), rsigma=TRUE,
collapse=FALSE, weighted=FALSE, ...)
```

Arguments

`object`	an object inheriting from class `survreg`.
`type`	type of residuals, with choices of `"response"`, `"deviance"`, `"dfbeta"`, `"dfbetas"`, `"working"`, `"ldcase"`, `"lsresp"`, `"ldshape"`, and `"matrix"`. See the LaTeX documentation (`survival/doc/survival.ps.gz`) for more detail.
`rsigma`	include the scale parameters in the variance matrix, when doing computations. (I can think of no good reason not to).
`collapse`	optional vector of subject groups. If given, this must be of the same length as the residuals, and causes the result to be per group residuals.
`weighted`	give weighted residuals? Normally residuals are unweighted.
`...`	other unused arguments

Value

A vector or matrix of residuals is returned. Response residuals are on the scale of the original data, working residuals are on the scale of the linear predictor, and deviance residuals are on log-likelihood scale. The dfbeta residuals are a matrix, where the ith row gives the approximate change in the coefficients due to the addition of subject i. The dfbetas matrix contains the dfbeta residuals, with each column scaled by the standard deviation of that coefficient.

The matrix type produces a matrix based on derivatives of the log-likelihood function. Let L be the log-likelihood, p be the linear predictor $X\beta$, and s be $\log(\sigma)$. Then the 6 columns of the matrix are L, dL/dp,$\partial^2 L/\partial p^2$, dL/ds, $\partial^2 L/\partial s^2$ and $\partial^2 L/\partial p \partial s$. Diagnostics based on these quantities are discussed in an article by Escobar and Meeker. The main ones are the likelihood displacement residuals for perturbation of a case weight (`ldcase`), the response value (`ldresp`), and the `shape`.

References

Escobar, L. A. and Meeker, W. Q. (1992). Assessing influence in regression analysis with censored data. *Biometrics* **48**, 507-528.

See Also

`predict.survreg`

Examples

```
fit <- survreg(Surv(time,status) ~x, aml)
rr  <- residuals(fit, type='matrix')
```

`ridge` *Ridge regression*

Description

When used in a coxph or survreg model formula, specifies a ridge regression term. The likelihood is penalised by `theta`/2 time the sum of squared coefficients. If `scale=T` the penalty is calculated for coefficients based on rescaling the predictors to have unit variance. If `df` is specified then `theta` is chosen based on an approximate degrees of freedom.

Usage

```
ridge(..., theta, df=nvar/2, eps=0.1, scale=TRUE)
```

Arguments

`...`	predictors to be ridged
`theta`	penalty is `theta`/2 time sum of squared coefficients
`df`	Approximate degrees of freedom
`eps`	Accuracy required for `df`
`scale`	Scale variables before applying penalty?

Value

An object of class `coxph.penalty` containing the data and control functions.

References

Gray (1992) "Flexible methods of analysing survival data using splines, with applications to breast cancer prognosis" JASA 87:942–951

See Also

`coxph`,`survreg`,`pspline`,`frailty`

Examples

```
fit1 <- coxph(Surv(futime, fustat) ~ rx + ridge(age, ecog.ps, theta=1),
              ovarian)
fit1

lfit0 <- survreg(Surv(time, status) ~1, cancer)
lfit1 <- survreg(Surv(time, status) ~ age + ridge(ph.ecog, theta=5), cancer)
lfit2 <- survreg(Surv(time, status) ~ sex + ridge(age, ph.ecog, theta=1), cancer)
lfit3 <- survreg(Surv(time, status) ~ sex + age + ph.ecog, cancer)

lfit0
```

```
lfit1
lfit2
lfit3
```

`stanford2` *More Stanford Heart Transplant data*

Description

This contains the Stanford Heart Transplant data in a different format. The main data set is in `heart`

Usage

```
data(stanford2)
```

Format

id:	ID number
time:	survival or censoring time
status:	censoring status
age:	in years
t5:	T5 mismatch score

Source

Escobar & Meeker, Biometrics 1992 p519

See Also

`predict.survreg`,`heart`

`strata` *Identify Stratum Variables*

Description

This is a special function used in the context of the Cox survival model. It identifies stratification variables when they appear on the right hand side of a formula.

Usage

```
strata(..., na.group=FALSE, shortlabel=FALSE)
```

Arguments

`...`	any number of variables. All must be the same length.
`na.group`	a logical variable, if `TRUE`, then missing values are treated as a distinct level of each variable.
`shortlabel`	if `TRUE` omit variable names from resulting factor labels

Details

The result is identical to the `interaction` function, but for the labeling of the factors (`strata` is more verbose).

Value

a new factor, whose levels are all possible combinations of the factors supplied as arguments.

See Also

`coxph`,`interaction`

Examples

```
a<-factor(rep(1:3,4))
b<-factor(rep(1:4,3))
levels(strata(a))
levels(strata(a,b,shortlabel=TRUE))

coxph(Surv(futime, fustat) ~ age + strata(rx), data=ovarian)
```

`summary.survfit` *Summary of a Survival Curve*

Description

Returns a list containing the survival curve, confidence limits for the curve, and other information.

Usage

```
## S3 method for class 'survfit':
summary(object, times, censored=FALSE, scale=1, ...)
```

Arguments

`object`	output from a call to `survfit`.
`times`	vector of times; the returned matrix will contain 1 row for each time. This must be in increasing order and missing values are not allowed. If `censored=T`, the default `times` vector contains all the unique times in `fit`, otherwise the default `times` vector uses only the event (death) times.
`censored`	logical flag: should the censoring times be included in the output? This is ignored if the `times` argument is present.
`scale`	rescale the survival time, e.g., if the input data to `survfit` were in days, `scale=365.25` would scale the output to years.
`...`	other unused arguments

Value

a list with the following components

`time`	the timepoint on the curve.
`surv`	the value of the survival curve at time t+0.
`n.risk`	the number of subjects at risk at time t-0 (but see the comments on weights in the `survfit` help file).
`n.event`	if the `times` argument is missing, then this column is the number of events that occurred at time t. Otherwise, it is the cumulative number of events that have occurred since the last time listed until time t+0.
`std.err`	the standard error of the survival value.
`conf.int`	level of confidence for the confidence intervals of survival.
`lower`	lower confidence limits for the curve.
`upper`	upper confidence limits for the curve.
`strata`	indicates stratification of curve estimation. If `strata` is not `NULL`, there are multiple curves in the result and the `surv`, `time`, `n.risk`, etc. vectors will contain multiple curves, pasted end to end. The levels of `strata` (a factor) are the labels for the curves.
`call`	the statement used to create the `fit` object.
`na.action`	passed through from `fit`, if present.

See Also

`survfit`, `print.summary.survfit`.

Examples

```
summary( survfit( Surv(futime, fustat),data=ovarian))
summary( survfit( Surv(futime, fustat)~rx,data=ovarian))
```

`Surv` *Create a Survival Object*

Description

Create a survival object, usually used as a response variable in a model formula.

Usage

```
Surv(time, time2, event, type =, origin = 0)
is.Surv(x)
```

Arguments

time	for right censored data, this is the follow up time. For interval data, the first argument is the starting time for the interval.
x	any R object.
event	The status indicator, normally 0=alive, 1=dead. Other choices are TRUE/FALSE (TRUE = death) or 1/2 (2=death). For interval censored data, the status indicator is 0=right censored, 1=event at time, 2=left censored, 3=interval censored. Although unusual, the event indicator can be omitted, in which case all subjects are assumed to have an event.
time2	ending time of the interval for interval censored or counting process data only. Intervals are assumed to be open on the left and closed on the right, (start, end]. For counting process data, event indicates whether an event occurred at the end of the interval.
type	character string specifying the type of censoring. Possible values are "right", "left", "counting", "interval", or "interval2". The default is "right" or "counting" depending on whether the time2 argument is absent or present, respectively.
origin	for counting process data, the hazard function origin. This is most often used in conjunction with a model containing time dependent strata in order to align the subjects properly when they cross over from one strata to another.

Details

Typical usages are

```
Surv(time, event)
Surv(time, time2, event, type=, origin=0)
```

In theory it is possible to represent interval censored data without a third column containing the explicit status. Exact, right censored, left censored and interval censored observation would be represented as intervals of (a,a), (a, infinity), (-infinity,b), and (a,b) respectively; each specifying the interval within which the event is known to have occurred.

If type = "interval2" then the representation given above is assumed, with NA taking the place of infinity. If 'type="interval" event must be given. If event is 0, 1, or 2, the relevant information is assumed to be contained in time, the value in time2 is ignored, and the second column of the result will contain a placeholder.

Presently, the only methods allowing interval censored data are the parametric models computed by survreg, so the distinction between open and closed intervals is unimportant. The distinction is important for counting process data and the Cox model.

The function tries to distinguish between the use of 0/1 and 1/2 coding for left and right censored data using if (max(status)==2). If 1/2 coding is used and all the subjects are censored, it will guess wrong. Use 0/1 coding in this case.

Value

An object of class Surv. There are methods for print, is.na, and subscripting survival objects. Surv objects are implemented as a matrix of 2 or 3 columns.

In the case of is.Surv, a logical value TRUE if x inherits from class "Surv", otherwise an FALSE.

See Also

coxph, survfit, survreg.

Examples

```
with(aml, Surv(time, status))
with(heart, Surv(start,stop,event))
```

---

survdiff *Test Survival Curve Differences*

Description

Tests if there is a difference between two or more survival curves using the G^ρ family of tests, or for a single curve against a known alternative.

Usage

```
survdiff(formula, data, subset, na.action, rho=0)
```

Arguments

formula	a formula expression as for other survival models, of the form `Surv(time, status) ~ predictors`. For a one-sample test, the predictors must consist of a single `offset(sp)` term, where `sp` is a vector giving the survival probability of each subject. For a k-sample test, each unique combination of predictors defines a subgroup. A `strata` term may be used to produce a stratified test. To cause missing values in the predictors to be treated as a separate group, rather than being omitted, use the `strata` function with its `na.group=T` argument.
data	an optional data frame in which to interpret the variables occurring in the formula.
subset	expression indicating which subset of the rows of data should be used in the fit. This can be a logical vector (which is replicated to have length equal to the number of observations), a numeric vector indicating which observation numbers are to be included (or excluded if negative), or a character vector of row names to be included. All observations are included by default.
na.action	a missing-data filter function. This is applied to the `model.frame` after any subset argument has been used. Default is `options()$na.action`.
rho	a scalar parameter that controls the type of test.

Value

a list with components:

n	the number of subjects in each group.
obs	the weighted observed number of events in each group. If there are strata, this will be a matrix with one column per stratum.
exp	the weighted expected number of events in each group. If there are strata, this will be a matrix with one column per stratum.

`chisq`	the chisquare statistic for a test of equality.
`var`	the variance matrix of the test.
`strata`	optionally, the number of subjects contained in each stratum.

METHOD

This function implements the G-rho family of Harrington and Fleming (1982), with weights on each death of $S(t)^\rho$, where $S(t)$ is the Kaplan-Meier estimate of survival. With `rho = 0` this is the log-rank or Mantel-Haenszel test, and with `rho = 1` it is equivalent to the Peto & Peto modification of the Gehan-Wilcoxon test.

If the right hand side of the formula consists only of an offset term, then a one sample test is done. To cause missing values in the predictors to be treated as a separate group, rather than being omitted, use the `factor` function with its `exclude` argument.

References

Harrington, D. P. and Fleming, T. R. (1982). A class of rank test procedures for censored survival data. *Biometrika* **69**, 553-566.

Examples

```
## Two-sample test
survdiff(Surv(futime, fustat) ~ rx,data=ovarian)

## Stratified 7-sample test

survdiff(Surv(time, status) ~ pat.karno + strata(inst), data=lung)

## Expected survival for heart transplant patients based on
## US mortality tables
expect <- survexp(futime ~ ratetable(age=(accept.dt - birth.dt),
     sex=1,year=accept.dt,race="white"), jasa, cohort=FALSE,
     ratetable=survexp.usr)
## actual survival is much worse (no surprise)
survdiff(Surv(jasa$futime, jasa$fustat) ~ offset(expect))
```

---

survexp *Compute Expected Survival*

Description

Returns either the expected survival of a cohort of subjects, or the individual expected survival for each subject.

Usage

```
survexp(formula, data, weights, subset, na.action, times, cohort=TRUE,
        conditional=FALSE, ratetable=survexp.us, scale=1, npoints,
        se.fit=, model=FALSE, x=FALSE, y=FALSE)
```

Arguments

formula	formula object. The response variable is a vector of follow-up times and is optional. The predictors consist of optional grouping variables separated by the + operator (as in survfit), along with a ratetable term. The ratetable term matches each subject to his/her expected cohort.
data	data frame in which to interpret the variables named in the formula, subset and weights arguments.
weights	case weights.
subset	expression indicating a subset of the rows of data to be used in the fit.
na.action	function to filter missing data. This is applied to the model frame after subset has been applied. Default is options()$na.action. A possible value for na.action is na.omit, which deletes observations that contain one or more missing values.
times	vector of follow-up times at which the resulting survival curve is evaluated. If absent, the result will be reported for each unique value of the vector of follow-up times supplied in formula.
cohort	logical value: if FALSE, each subject is treated as a subgroup of size 1. The default is TRUE.
conditional	logical value: if TRUE, the follow-up times supplied in formula are death times and conditional expected survival is computed. If FALSE, the follow-up times are potential censoring times. If follow-up times are missing in formula, this argument is ignored.
ratetable	a table of event rates, such as survexp.uswhite, or a fitted Cox model.
scale	numeric value to scale the results. If ratetable is in units/day, scale = 365.25 causes the output to be reported in years.
npoints	number of points at which to calculate intermediate results, evenly spaced over the range of the follow-up times. The usual (exact) calculation is done at each unique follow-up time. For very large data sets specifying npoints can reduce the amount of memory and computation required. For a prediction from a Cox model npoints is ignored.
se.fit	compute the standard error of the predicted survival. The default is to compute this whenever the routine can, which at this time is only for the Ederer method and a Cox model as the rate table.
model,x,y	flags to control what is returned. If any of these is true, then the model frame, the model matrix, and/or the vector of response times will be returned as components of the final result, with the same names as the flag arguments.

Details

Individual expected survival is usually used in models or testing, to 'correct' for the age and sex composition of a group of subjects. For instance, assume that birth date, entry date into the study, sex and actual survival time are all known for a group of subjects. The survexp.uswhite population tables contain expected death rates based on calendar year, sex and age. Then haz <- -log(survexp(death.time ratetable(sex=sex, year=entry.dt, age=(birth.dt-entry.dt)), cohort=F)) gives for each subject the total hazard experienced up to their observed death time or censoring time. This probability can be used as a rescaled time value in models: glm(status 1 + offset(log(haz)), family=poisson) glm(status x + offset(log(haz)), family=poisson) In the first model, a test for

intercept=0 is the one sample log-rank test of whether the observed group of subjects has equivalent survival to the baseline population. The second model tests for an effect of variable `x` after adjustment for age and sex.

Cohort survival is used to produce an overall survival curve. This is then added to the Kaplan-Meier plot of the study group for visual comparison between these subjects and the population at large. There are three common methods of computing cohort survival. In the "exact method" of Ederer the cohort is not censored; this corresponds to having no response variable in the formula. Hakulinen recommends censoring the cohort at the anticipated censoring time of each patient, and Verheul recommends censoring the cohort at the actual observation time of each patient. The last of these is the conditional method. These are obtained by using the respective time values as the follow-up time or response in the formula.

Value

if `cohort=T` an object of class `survexp`, otherwise a vector of per-subject expected survival values. The former contains the number of subjects at risk and the expected survival for the cohort at each requested time.

References

G. Berry. The analysis of mortality by the subject-years method. *Biometrics* 1983, 39:173-84. F Ederer, L Axtell, and S Cutler. The relative survival rate: a statistical methodology. *Natl Cancer Inst Monogr* 1961, 6:101-21. T. Hakulinen. Cancer survival corrected for heterogeneity in patient withdrawal. *Biometrics* 1982, 38:933. H. Verheul, E. Dekker, P. Bossuyt, A. Moulijn, and A. Dunning. Background mortality in clinical survival studies. *Lancet* 1993, 341:872-5.

See Also

`survfit`, `survexp.us`, `survexp.fit`, `pyears`, `date`

Examples

```
## compare survival to US population
cancer$year<-rep(as.date("1/1/1980"),nrow(cancer))
efit <- survexp( ~ ratetable(sex=sex, year=year, age=age*365), times=(1:4)*365,data=cance
plot(survfit(Surv(time, status) ~1,data=cancer))
lines(efit)

## compare data to Cox model
## fit to randomised patients in Mayo PBC data
m<-coxph(Surv(time,status)~edtrt+log(bili)+log(protime)+age+platelet,data=pbc,
      subset=trt>0)
##compare Kaplan-Meier to fitted model for 2 edema groups in
##unrandomised patients
plot(survfit(Surv(time,status)~edtrt,data=pbc,subset=trt==-9))
lines(survexp(~edtrt+ratetable(edtrt=edtrt,bili=bili,platelet=platelet,age=age,
  protime=protime),data=pbc,subset=trt==-9,ratetable=m,cohort=TRUE),col="purple")
```

`survexp.fit` *Work Function to Compute Expected Survival*

Description

Compute expected survival. This function is not to be called by the user.

Usage

```
survexp.fit(x, y, times, death, ratetable)
```

Arguments

`x`	a matrix. The first column contains the group, an integer value that divides the subjects into subsets. Remaining columns must match the dimensions of the `ratetable`, in the correct order.
`y`	the follow up time for each subject.
`times`	the vector of times at which a result will be computed.
`death`	death indicator
`ratetable`	a rate table, such as `survexp.uswhite`.

Details

For cohort survival it must be the potential censoring time for each subject, ignoring death.

For an exact estimate `times` should be a superset of `y`, so that each subject at risk is at risk for the entire sub-interval of time. For a large data set, however, this can use an inordinate amount of storage and/or compute time. If the `times` spacing is more coarse than this, an actuarial approximation is used which should, however, be extremely accurate as long as all of the returned values are > .99.

For a subgroup of size 1 and `times` > `y`, the conditional method reduces to exp(-h) where h is the expected cumulative hazard for the subject over his/her observation time. This is used to compute individual expected survival.

Value

A list containing the number of subjects and the expected survival(s) at each time point. If there are multiple groups, these will be matrices with one column per group.

WARNING

Most users will call the higher level routine `survexp`. Consequently, this function has very few error checks on its input arguments.

See Also

`survexp`, `survexp.us`

`survfit` *Compute a Survival Curve for Censored Data*

Description

Computes an estimate of a survival curve for censored data using either the Kaplan-Meier or the Fleming-Harrington method or computes the predicted survivor function for a Cox proportional hazards model.

Usage

```
survfit(formula, data, weights, subset, na.action,
        newdata, individual=F, conf.int=.95, se.fit=T,
        type=c("kaplan-meier","fleming-harrington", "fh2"),
        error=c("greenwood","tsiatis"),
        conf.type=c("log","log-log","plain","none"),
        conf.lower=c("usual", "peto", "modified"))
## S3 method for class 'survfit':
x[..., drop=FALSE]
basehaz(fit,centered=TRUE)
```

Arguments

`formula`	A formula object or a `coxph` object. If a formula object is supplied it must have a `Surv` object as the response on the left of the ~ operator and, if desired, terms separated by + operators on the right. One of the terms may be a `strata` object. For a single survival curve the `"~ 1"` part of the formula is not required.
`data`	a data frame in which to interpret the variables named in the formula, or in the `subset` and the `weights` argument.
`weights`	The weights must be nonnegative and it is strongly recommended that they be strictly positive, since zero weights are ambiguous, compared to use of the `subset` argument.
`subset`	expression saying that only a subset of the rows of the data should be used in the fit.
`na.action`	a missing-data filter function, applied to the model frame, after any `subset` argument has been used. Default is `options()$na.action`.
`newdata`	a data frame with the same variable names as those that appear in the `coxph` formula. Only applicable when `formula` is a `coxph` object. The curve(s) produced will be representative of a cohort who's covariates correspond to the values in `newdata`. Default is the mean of the covariates used in the `coxph` fit.
`individual`	a logical value indicating whether the data frame represents different time epochs for only one individual (T), or whether multiple rows indicate multiple individuals (F, the default). If the former only one curve will be produced; if the latter there will be one curve per row in `newdata`.
`conf.int`	the level for a two-sided confidence interval on the survival curve(s). Default is 0.95.
`se.fit`	a logical value indicating whether standard errors should be computed. Default is `TRUE`.

type	a character string specifying the type of survival curve. Possible values are "kaplan-meier", "fleming-harrington" or "fh2" if a formula is given and "aalen" or "kaplan-meier" if the first argument is a coxph object, (only the first two characters are necessary). The default is "aalen" when a coxph object is given, and it is "kaplan-meier" otherwise.
error	either the string "greenwood" for the Greenwood formula or "tsiatis" for the Tsiatis formula, (only the first character is necessary). The default is "tsiatis" when a coxph object is given, and it is "greenwood" otherwise.
conf.type	One of "none", "plain", "log" (the default), or "log-log". Only enough of the string to uniquely identify it is necessary. The first option causes confidence intervals not to be generated. The second causes the standard intervals curve +- k *se(curve), where k is determined from conf.int. The log option calculates intervals based on the cumulative hazard or log(survival). The last option bases intervals on the log hazard or log(-log(survival)). These last will never extend past 0 or 1.
conf.lower	controls modified lower limits to the curve, the upper limit remains unchanged. The modified lower limit is based on an 'effective n' argument. The confidence bands will agree with the usual calculation at each death time, but unlike the usual bands the confidence interval becomes wider at each censored observation. The extra width is obtained by multiplying the usual variance by a factor m/n, where n is the number currently at risk and m is the number at risk at the last death time. (The bands thus agree with the un-modified bands at each death time.) This is especially useful for survival curves with a long flat tail. The Peto lower limit is based on the same 'effective n' argument as the modified limit, but also replaces the usual Greenwood variance term with a simple approximation. It is known to be conservative.
x	a survfit object
fit	a coxph object
centered	Compute the baseline hazard at the covariate mean rather than at zero?
drop	Only FALSE is supported
...	Other arguments for future expansion

Details

Actually, the estimates used are the Kalbfleisch-Prentice (Kalbfleisch and Prentice, 1980, p.86) and the Tsiatis/Link/Breslow, which reduce to the Kaplan-Meier and Fleming-Harrington estimates, respectively, when the weights are unity. When curves are fit for a Cox model, subject weights of exp(sum(coef*(x-center))) are used, ignoring any value for weights input by the user. There is also an extra term in the variance of the curve, due to the variance ofthe coefficients and hence variance in the computed weights.

The Greenwood formula for the variance is a sum of terms d/(n*(n-m)), where d is the number of deaths at a given time point, n is the sum of weights for all individuals still at risk at that time, and m is the sum of weights for the deaths at that time. The justification is based on a binomial argument when weights are all equal to one; extension to the weighted case is ad hoc. Tsiatis (1981) proposes a sum of terms d/(n*n), based on a counting process argument which includes the weighted case.

The two variants of the F-H estimate have to do with how ties are handled. If there were 3 deaths out of 10 at risk, then the first would increment the hazard by 3/10 and the second by 1/10 + 1/9

+ 1/8. For curves created after a Cox model these correspond to the Breslow and Efron estimates, respectively, and the proper choice is made automatically. The `fh2` method will give results closer to the Kaplan-Meier.

Based on the work of Link (1984), the log transform is expected to produce the most accurate confidence intervals. If there is heavy censoring, then based on the work of Dorey and Korn (1987) the modified estimate will give a more reliable confidence band for the tails of the curve.

Value

a `survfit` object; see the help on `survfit.object` for details. Methods defined for `survfit` objects are provided for `print`, `plot`, `lines`, and `points`.

For `basehaz`, a dataframe with the baseline hazard, times, and strata.

The `"["` method returns a `survfit` object giving survival for the selected groups.

References

Dorey, F. J. and Korn, E. L. (1987). Effective sample sizes for confidence intervals for survival probabilities. *Statistics in Medicine* 6, 679-87.

Fleming, T. H. and Harrington, D.P. (1984). Nonparametric estimation of the survival distribution in censored data. *Comm. in Statistics* 13, 2469-86.

Kalbfleisch, J. D. and Prentice, R. L. (1980). *The Statistical Analysis of Failure Time Data.* Wiley, New York.

Link, C. L. (1984). Confidence intervals for the survival function using Cox's proportional hazards model with covariates. *Biometrics* 40, 601-610.

Tsiatis, A. (1981). A large sample study of the estimate for the integrated hazard function in Cox's regression model for survival data. *Annals of Statistics* 9, 93-108.

See Also

`print.survfit`, `plot.survfit`, `lines.survfit`, `summary.survfit`, `survfit.object` `coxph`, `Surv`, `strata`.

Examples

```
#fit a Kaplan-Meier and plot it
fit <- survfit(Surv(time, status) ~ x, data=aml)
plot(fit)
# plot only 1 of the 2 curves from above
plot(fit[2])

#fit a cox proportional hazards model and plot the
#predicted survival curve
fit <- coxph( Surv(futime,fustat)~resid.ds+rx+ecog.ps,data=ovarian)
plot( survfit( fit))
```

`survfit.object` *Survival Curve Object*

Description

This class of objects is returned by the `survfit` class of functions to represent a fitted survival curve.

Objects of this class have methods for the functions `print`, `summary`, `plot`, `points` and `lines`. The `print.survfit` method does more computation than is typical for a print method and is documented on a separate page.

COMPONENTS

The following components must be included in a legitimate `survfit` object.

time the time points at which the curve has a step.

n.risk the number of subjects at risk at t.

n.event the number of events that occur at time t.

surv the estimate of survival at time t+0. This may be a vector or a matrix.

strata if there are multiple curves, this component gives the number of elements of the `time` etc. vectors corresponding to the first curve, the second curve, and so on. The names of the elements are labels for the curves.

std.err the standard error of the cumulative hazard or -log(survival).

upper upper confidence limit for the survival curve.

lower lower confidence limit for the survival curve.

conf.type the approximation used to compute the confidence limits.

conf.int the level of the confidence limits, e.g. 90 or 95%.

na.action the returned value from the na.action function, if any. It will be used in the printout of the curve, e.g., the number of observations deleted due to missing values.

call the call that produced the object.

SUBSCRIPTS

Survfit objects that contain multiple survival curves can be subscripted. This is most often used to plot a subset of the curves. Usually a single subscript will be used. In one particular case – survival curves for multiple covariate values, from a Cox model that includes a `strata` statement – there is a matrix of curves and 2 subscripts may be used. (In this case `summary.survfit` will also print the data as a matrix).

See Also

`survfit`, `plot.survfit`, `summary.survfit`, `print.survfit`.

`survobrien` *O'Brien's Test for Association of a Single Variable with Survival*

Description

Peter O'Brien's test for association of a single variable with survival This test is proposed in Biometrics, June 1978.

Usage

```
survobrien(formula, data)
```

Arguments

`formula`	a valid formula for a cox model, without time dependent covariates.
`data`	a data frame.

Value

a new data frame. The original time and status variables are removed, and have been replaced with `start`, `stop`, and `event`. If a predictor variable is a factor or is protected with `I()`, it is retained as is. Other predictor variables have been replaced with time-dependent logit scores.

Because of the time dependent variables, the new data frame will have many more rows that the original data, approximately #rows * #deaths /2.

METHOD

A time-dependent cox model can now be fit to the new data. The univariate statistic, as originally proposed, is equivalent to single variable score tests from the time-dependent model. This equivalence is the rationale for using the time dependent model as a multivariate extension of the original paper.

In O'Brien's method, the x variables are re-ranked at each death time. A simpler method, proposed by Prentice, ranks the data only once at the start. The results are usually similar.

References

O'Brien, Peter, "A Nonparametric Test for Association with Censored Data", *Biometrics* 34: 243-250, 1978.

See Also

`survdiff`

Examples

```
xx <- survobrien(Surv(futime, fustat) ~ age + factor(rx) + I(ecog.ps),
                                 data=ovarian)
coxph(Surv(start, stop, event) ~ age, data=xx)
coxph(Surv(start, stop, event) ~ age + rx + ecog.ps, data=xx)
```

survreg *Regression for a Parametric Survival Model*

Description

Regression for a parametric survival model. These are all time-transformed location models, with the most useful case being the accelerated failure models that use a log transformation.

Usage

```
survreg(formula=formula(data), data=parent.frame(), weights,
subset,na.action,dist="weibull",  init=NULL, scale=0,
 control=survreg.control(),parms=NULL,model=FALSE, x=FALSE,
 y=TRUE, robust=FALSE, ...)
```

Arguments

formula	a formula expression as for other regression models. See the documentation for lm and formula for details.
data	optional data frame in which to interpret the variables occurring in the formula.
weights	Optional observation weights
subset	subset of the observations to be used in the fit.
na.action	function to be used to handle any NAs in the data.
dist	assumed distribution for y variable. If the argument is a character string, then it is assumed to name an element from survreg.distributions. These include "weibull", "exponential", "gaussian", "logistic", "lognormal" and "loglogistic". Otherwise, it is assumed to be a user defined list conforming to the format described in survreg.distributions.
parms	a list of fixed parameters. For the t-distribution for instance this is the degrees of freedom; most of the distributions have no parameters.
init	optional vector of initial values for the parameters.
scale	optional fixed value for the scale. If set to <=0 then the scale is estimated.
control	a list of control values, in the format producted by survreg.control.
model	if TRUE, the model frame is returned.
x	if TRUE, then the X matrix is returned.
y	if TRUE, then the y vector (or survival times) is returned.
robust	if TRUE, sandwich standard errors are computed. Defaults to TRUE when formula contains a cluster term.
...	other arguments which will be passed to survreg.control.

Value

an object of class survreg is returned.

Compatibility note

This routine underwent significant changes from survival4 to survival5. The survreg.old function gives a backwards-compatible interface. In S-PLUS the new function is called `survReg` and the old one `survreg`.

See Also

`survreg.object`, `survreg.distributions`, `pspline`, `frailty`, `ridge`, `survreg.old`

Examples

```
## These are all the same
survreg(Surv(futime, fustat) ~ ecog.ps + rx, ovarian, dist='weibull',scale=1)
survreg(Surv(futime, fustat) ~ ecog.ps + rx, ovarian,
dist="exponential")
survreg.old(Surv(futime, fustat) ~ ecog.ps + rx, ovarian, dist='extreme',fixed=list(scale
```

`survreg.control` *Package options for survreg and coxph*

Description

These functions check and package the fitting options for `survreg` and `coxph`

Usage

```
survreg.control(maxiter=30, rel.tolerance=1e-09, failure=1,
toler.chol=1e-10, iter.max, debug=0, outer.max=10)
coxph.control (eps = 1e-09, toler.chol = .Machine$double.eps^0.75,
    iter.max = 20, toler.inf = sqrt(eps), outer.max = 10)
```

Arguments

`maxiter`	maximum number of iterations
`rel.tolerance`	relative tolerance to declare convergence
`failure`	value of status variable indicating failure
`toler.chol`	Tolerance to declare Cholesky decomposition singular
`iter.max`	same as `maxiter`
`debug`	print debugging information
`outer.max`	maximum number of outer iterations for choosing penalty parameters
`eps`	Tolerance to declare convergence for Cox model
`toler.inf`	An undocumented parameter

Value

A list with the same elements as the input

See Also

survreg, coxph

Examples

`survreg.distributions`

Parametric Survival Distributions

Description

List of distributions for accelerated failure models. These are location-scale families for some transformation of time. The entry describes the cdf F and density f of a canonical member of the family.

Usage

```
survreg.distributions
```

Format

There are three basic formats; only the first two are used in the built-in distributions

name:	name of distribution
variance:	Variance
init(x,weights,...):	Function returning an initial mean and variance
deviance(y,scale,parms):	Function returning the deviance
density(x,parms):	Function returning F, $1-F$,f,f'/f,f''/f
quantile(p,parms):	Quantile function
scale:	Optional fixed value for scale parameter

and for transformations of the time variable

name:	name of distribution
dist:	name of transformed distribution
trans:	transformation (eg log)
dtrans:	derivative of transformation
itrans:	inverse of transformation
scale:	Optional fixed value for scale parameter

For transformations of user-defined families use

name:	name of distribution
dist:	transformed distribution in first format
trans:	transformation (eg log)
dtrans:	derivative of transformation
itrans:	inverse of transformation

scale: Optional fixed value for scale parameter

Details

There are four basic distributions:`extreme`, `gaussian`, `logistic` and `t`. The last three are parametrised in the same way as the distributions already present in R. The extreme value cdf is

$$F = 1 - e^{-e^t}.$$

When the logarithm of survival time has one of the first three distributions we obtain respectively `weibull`, `lognormal`, and `loglogistic`. The Weibull distribution is not parameterised the same way as in `rweibull`.

The other predefined distributions are defined in terms of these. The `exponential` and `rayleigh` distributions are Weibull distributions with fixed `scale` of 1 and 0.5 respectively, and `loggaussian` is a synonym for `lognormal`.

Parts of the built-in distributions are hardcoded in C, so the elements of `survreg.distributions` in the first format above must not be changed and new ones must not be added. The examples show how to specify user-defined distributions to `survreg`.

See Also

`survreg`, `pnorm`,`plogis`, `pt`

Examples

```
## not a good fit, but a useful example
survreg(Surv(futime,fustat)~ecog.ps+rx,data=ovarian,dist='extreme')
##
my.extreme<-survreg.distributions$extreme
my.extreme$name<-"Xtreme"
survreg(Surv(futime,fustat)~ecog.ps+rx,data=ovarian,dist=my.extreme)

## time transformation
survreg(Surv(futime,fustat)~ecog.ps+rx,data=ovarian,dist='weibull',scale=1)
my.weibull<-survreg.distributions$weibull
my.weibull$dist<-my.extreme
survreg(Surv(futime,fustat)~ecog.ps+rx,data=ovarian,dist=my.weibull,scale=1)

## change the transformation to work in years
## intercept changes by log(365), other coefficients stay the same
my.weibull$trans<-function(y) log(y/365)
my.weibull$itrans<-function(y) exp(365*y)
survreg(Surv(futime,fustat)~ecog.ps+rx,data=ovarian,dist=my.weibull,scale=1)

## Weibull parametrisation
y<-rweibull(1000, shape=2, scale=5)
survreg(Surv(y)~1, dist="weibull")
## survreg reports scale=1/2, intercept=log(5)
```

`survreg.object` *Parametric Survival Model Object*

Description

This class of objects is returned by the `survreg` function to represent a fitted parametric survival model. Objects of this class have methods for the functions `print`, `summary`, `predict`, and `residuals`.

COMPONENTS

The following components must be included in a legitimate `survreg` object.

coefficients the coefficients of the `linear.predictors`, which multiply the columns of the model matrix. It does not include the estimate of error (sigma). The names of the coefficients are the names of the single-degree-of-freedom effects (the columns of the model matrix). If the model is over-determined there will be missing values in the coefficients corresponding to non-estimable coefficients.

icoef coefficients of the baseline model, which will contain the intercept and log(scale), or mulitple scale factors for a stratified model.

var the variance-covariance matrix for the parameters, including the log(scale) parameter(s).

loglik a vector of length 2, containing the log-likelihood for the baseline and full models.

iter the number of iterations required

linear.predictors the linear predictor for each subject.

df the degrees of freedom for the final model. For a penalized model this will be a vector with one element per term.

scale the scale factor(s), with length equal to the number of strata.

idf degrees of freedom for the initial model.

means a vector of the column means of the coefficient matrix.

dist the distribution used in the fit.

The object will also have the following components found in other model results (some are optional): `linear predictors`, `weights`, `x`, `y`, `model`, `call`, `terms` and `formula`. See `lm`.

See Also

`survreg`, `lm`

survreg.old *Old (survival4) Regression for a parametric survival model*

Description

This routine is a backwards-compatible interface to the improved survreg function, which is better.

Usage

```
survreg.old(formula, data=sys.frame(sys.parent()), ..., link=c("log",
"identity"),dist=c("extreme", "logistic", "gaussian",
"exponential","rayleigh","weibull"), fixed=list())
```

Arguments

formula	a formula expression as for other regression models. See the documentation for lm and formula for details.
data	optional data frame in which to interpret the variables occuring in the formula.
...	other arguments to survreg
link	transformation to be used on the y variable.
dist	assumed distribution for the transformed y variable.
fixed	a list of fixed parameters, most often just the scale.

Value

an object of class survreg is returned, which inherits from class glm.

Examples

```
survreg.old(Surv(futime, fustat) ~ ecog.ps + rx, ovarian, dist='extreme',
               link='log', fixed=list(scale=1))   #Fit an exponential
```

survSplit *Split a survival data set at specified times*

Description

Given a survival data set and a set of specified cut times, split each record into multiple subrecords at each cut time. The new data set will be in 'counting process' format, with a start time, stop time, and event status for each record.

Usage

```
survSplit(data, cut, end, event, start, id = NULL, zero = 0,
  episode=NULL)
```

Arguments

data	data frame
cut	vector of timepoints to cut at
end	character string with name of event time variable
event	character string with name of censoring indicator
start	character string with name of start time variable (will be created if it does not exist)
id	character string with name of new id variable to create (optional)
zero	If start doesn't already exist, this is the time that the original records start. May be a vector or single value.
episode	character string with name of new episode variable (optional)

Details

The function also works when the original data are in counting-process format, but the id and episode options are of little use in this context.

Value

New, longer, data frame.

See Also

Surv, cut, reshape

Examples

```
aml3<-survSplit(aml,cut=c(5,10,50),end="time",start="start",
      event="status",episode="i")

summary(aml)
summary(aml3)

coxph(Surv(time,status)~x,data=aml)
## the same
coxph(Surv(start,time,status)~x,data=aml3)

aml4<-survSplit(aml3,cut=20,end="time",start="start", event="status")
coxph(Surv(start,time,status)~x,data=aml4)
```

---

tcut *Factors for person-year calculations*

Description

Attaches categories for person-year calculations to a variable without losing the underlying continuous representation

Usage

```
tcut(x, breaks, labels, scale=1)
```

Arguments

x	numeric/date variable
breaks	breaks between categories, which are right-continuous
labels	labels for categories
scale	Multiply x and breaks by this.

Value

An object of class tcut

See Also

cut, pyears

Examples

```
temp1 <- mdy.date(6,6,36)
temp2 <- mdy.date(6,6,55)# Now compare the results from person-years
#
temp.age <- tcut(temp2-temp1, floor(c(-1, (18:31 * 365.24))),
        labels=c('0-18', paste(18:30, 19:31, sep='-')))
temp.yr  <- tcut(temp2, mdy.date(1,1,1954:1965), labels=1954:1964)
temp.time <- 3700   #total days of fu
py1 <- pyears(temp.time ~ temp.age + temp.yr, scale=1) #output in days
py1
```

---

tobin *Tobin's Tobit data*

Description

Economists fit a parametric censored data model called the 'tobit'. These data are from Tobin's original paper.

Usage

```
data(tobin)
```

Format

A data frame with 20 observations on the following 3 variables.

durable Durable goods purchase

age Age in years

quant Liquidity ratio (x 1000)

Source

J. Tobin, Estimation of relationships for limited dependent variables, Econometrica, v26, 24-36, 1958.

Examples

```
tfit <- survreg(Surv(durable, durable>0, type='left') ~age + quant,
                data=tobin, dist='gaussian')

predict(tfit,type="response")
```

---

untangle.specials *Help Process the 'specials' Argument of the 'terms' Function.*

Description

Given a `terms` structure and a desired special name, this returns an index appropriate for subscripting the `terms` structure and another appropriate for the data frame.

Usage

```
untangle.specials(tt, special, order=1)
```

Arguments

`tt`	a `terms` object.
`special`	the name of a special function, presumably used in the terms object.
`order`	the order of the desired terms. If set to 2, interactions with the special function will be included.

Value

a list with two components:

`vars`	a vector of variable names, as would be found in the data frame, of the specials.
`terms`	a numeric vector, suitable for subscripting the terms structure, that indexes the terms in the expanded model formula which involve the special.

Examples

```
formula<-Surv(tt,ss)~x+z*strata(id)
tms<-terms(formula,specials="strata")
## the specials attribute
attr(tms,"specials")
## main effects
untangle.specials(tms,"strata")
## and interactions
untangle.specials(tms,"strata",order=1:2)
```

`veteran` *Veterans' Administration Lung Cancer study*

Description

Randomised trial of two treatment regimens for lung cancer. This is a standard survival analysis data set

Usage

```
data(veteran)
```

Format

trt:	1=standard 2=test
celltype:	1=squamous, 2=smallcell, 3=adeno, 4=large
time:	survival time
status:	censoring status
karno:	Karnofsky performance score (100=good)
diagtime:	months from diagnosis to randomisation
age:	in years
prior:	prior therapy 0=no, 1=yes

Source

Kalbfleisch and Prentice "The Statistical Analysis of Failure Time Data"

Chapter 25

The `tcltk` package

`tcltk-package` *Tcl/Tk Interface*

Description

Interface and language bindings to Tcl/Tk GUI elements.

Details

This package provides access to the platform-independent Tcl scripting language and Tk GUI elements. See TkWidgets for a list of supported widgets, TkWidgetcmds for commands to work with them, and references in those files for more.

The Tcl/Tk documentation is in the system man pages.

For a complete list of functions, use `ls("package:tcltk")`.

Note that Tk will not be initialized if there is no `DISPLAY` variable set, but Tcl can still be used. This is most useful to allow the loading of a package which depends on **tcltk** in a session that does not actually use it (e.g. during installation).

Author(s)

R Development Core Team

Maintainer: R Core Team ⟨R-core@r-project.org⟩

`TclInterface` *Low-level Tcl/Tk Interface*

Description

These functions and variables provide the basic glue between R and the Tcl interpreter and Tk GUI toolkit. Tk windows may be represented via R objects. Tcl variables can be accessed via objects of class `tclVar` and the C level interface to Tcl objects is accessed via objects of class `tclObj`.

Usage

```
.Tcl(...)
.Tcl.objv(objv)
.Tcl.args(...)
.Tcl.args.objv(...)
.Tcl.callback(...)
.Tk.ID(win)
.Tk.newwin(ID)
.Tk.subwin(parent)
.TkRoot

tkdestroy(win)
is.tkwin(x)

tclvalue(x)
tclvalue(x) <- value

tclVar(init="")
## S3 method for class 'tclVar':
as.character(x)
## S3 method for class 'tclVar':
tclvalue(x)
## S3 method for class 'tclVar':
tclvalue(x) <- value

tclArray()
## S3 method for class 'tclArray':
x[[...]]
## S3 method for class 'tclArray':
x[[...]] <- value
## S3 method for class 'tclArray':
x$i
## S3 method for class 'tclArray':
x$i <- value

## S3 method for class 'tclArray':
names(x)
## S3 method for class 'tclArray':
length(x)

tclObj(x)
tclObj(x) <- value
## S3 method for class 'tclVar':
tclObj(x)
## S3 method for class 'tclVar':
tclObj(x) <- value

as.tclObj(x, drop=FALSE)
is.tclObj(x)

## S3 method for class 'tclObj':
as.character(x)
```

```
## S3 method for class 'tclObj':
as.integer(x, ...)
## S3 method for class 'tclObj':
as.double(x, ...)
## S3 method for class 'tclObj':
tclvalue(x)

## Default S3 method:
tclvalue(x)
## Default S3 method:
tclvalue(x) <- value

addTclPath(path = ".")
tclRequire(package, warn = TRUE)
```

Arguments

objv	a named vector of Tcl objects
win	a window structure
x	an object
i	character or (unquoted) name
drop	logical. Indicates whether a single-element vector should be made into a simple Tcl object or a list of length one
value	For tclvalue assignments, a character string. For tclObj assignments, an object of class tclObj
ID	a window ID
parent	a window which becomes the parent of the resulting window
path	path to a directory containing Tcl packages
package	a Tcl package name
warn	logical. Warn if not found?
...	Additional arguments. See below.
init	initialization value

Details

Many of these functions are not intended for general use but are used internally by the commands that create and manipulate Tk widgets and Tcl objects. At the lowest level .Tcl sends a command as a text string to the Tcl interpreter and returns the result as an object of class tclObj (see below). A newer variant .Tcl.objv accepts arguments in the form of a named list of tclObj objects.

.Tcl.args converts an R argument list of tag=value pairs to the Tcl -option value style, thus enabling a simple translation between the two languages. To send a value with no preceding option flag to Tcl, just use an untagged argument. In the rare case one needs an option with no subsequent value tag=NULL can be used. Most values are just converted to character mode and inserted in the command string, but window objects are passed using their ID string, and callbacks are passed via the result of .Tcl.callback. Tags are converted to option flags simply by prepending a -

.Tcl.args.objv serves a similar purpose as .Tcl.args but produces a list of tclObj objects suitable for passing to .Tcl.objv. The names of the list are converted to Tcl option style internally by .Tcl.objv.

Callbacks can be either *atomic callbacks* handled by `.Tcl.callback` or expressions. An expression is treated as a list of atomic callbacks, with the following exceptions: if an element is a name, it is first evaluated in the callers frame, and likewise if it is an explicit function definition; the `break` expression is translated directly to the Tcl counterpart. `.Tcl.callback` converts R functions and unevaluated calls to Tcl command strings. The argument must be either a function closure or an object of mode `"call"` followed by an environment. The return value in the first case is of the form `R_call 0x408b94d4` in which the hexadecimal number is the memory address of the function. In the second case it will be of the form `R_call_lang 0x8a95904 0x819bfd0`. For expressions, a sequence of similar items is generated, separated by semicolons. `.Tcl.args` takes special precautions to ensure that functions or calls will continue to exist at the specified address by assigning the callback into the relevant window environment (see below).

Tk windows are represented as objects of class `tkwin` which are lists containing a `ID` field and an `env` field which is an R environments, enclosed in the global environment. The value of the `ID` field is identical to the Tk window name. The `env` environment contains a `parent` variable and a `num.subwin` variable. If the window obtains subwindows and callbacks, they are added as variables to the environment. `.TkRoot` is the top window with ID "."; this window is not displayed in order to avoid ill effects of closing it via window manager controls. The `parent` variable is undefined for `.TkRoot`.

`.Tk.ID` extracts the `ID` of a window, `.Tk.newwin` creates a new window environment with a given ID and `.Tk.subwin` creates a new window which is a subwindow of a given parent window.

`tkdestroy` destroys a window and also removes the reference to a window from its parent.

`is.tkwin` can be used to test whether a given object is a window environment.

`tclVar` creates a new Tcl variable and initializes it to `init`. An R object of class `tclVar` is created to represent it. Using `as.character` on the object returns the Tcl variable name. Accessing the Tcl variable from R is done using the `tclvalue` function, which can also occur on the left sie of assignments. If `tclvalue` is passed an argument which is not a `tclVar` object, then it will assume that it is a character string explicitly naming global Tcl variable. Tcl variables created by `tclVar` are uniquely named and automatically unset by the garbage collector when the representing object is no longer in use.

`tclArray` creates a new Tcl array and initalizes it to the empty array. An R object of class `tclArray` and inheriting from class `tclVar` is created to represent it. You can access elements of the Tcl array using indexing with `[[` or `$`, which also allow replacement forms. Notice that Tcl arrays are associative by nature and hence unordered; indexing with a numeric index `i` refers to the element with the *name* `as.character(i)`. Multiple indices are pasted together separated by commas to form a single name. You can query the length and the set of names in an array using methods for `length` and `names`, respectively; these cannot meaningfully be set so assignment forms exist only to print an error message.

It is possible to access Tcl's 'dual-ported' objects directly, thus avoiding parsing and deparsing of their string representation. This works by using objects of class `tclObj`. The string representation of such objects can be extracted (but not set) using `tclvalue` and conversion to vectors of mode `"character"`, `"double"`, or `"integer"`. Conversely, such vectors can be converted using `as.tclObj`. There is an ambiguity as to what should happen for length one vectors, controlled by the `drop` argument; there are cases where the distiction matters to Tcl, although mostly it treats them equivalently. Notice that `tclvalue` and `as.character` differ on an object whose string representation has embedded spaces, the former is sometimes to be preferred, in particular when applied to the result of `tclread`, `tkgetOpenFile`, and similar functions.

The object behind a `tclVar` object is extracted using `tclObj(x)` which also allows an assignment form, in which the right hand side of the assignment is automatically converted using `as.tclObj`. There is a print method for `tclObj` objects; it prints `<Tcl>` followed by the string

representation of the object. Notice that `as.character` on a `tclVar` object is the *name* of the corresponding Tcl variable and not the value.

Tcl packages can be loaded with `tclRequire`; it may be necessary to add the directory where they are found to the Tcl search path with `addTclPath`.

Note

Strings containing unbalanced braces are currently not handled well in many circumstances.

See Also

`TkWidgets`, `TkCommands`, `TkWidgetcmds`.

`capabilities("tcltk")` to see if Tcl/Tk support was compiled into this build of R.

Examples

```
## Not run:
## These cannot be run by example() but should be OK when pasted
## into an interactive R session with the tcltk package loaded
.Tcl("format \"%s\n\" \"Hello, World!\"")
f <- function()cat("HI!\n")
.Tcl.callback(f)
.Tcl.args(text="Push!", command=f) # NB: Different address

xyzzy <- tclVar(7913)
tclvalue(xyzzy)
tclvalue(xyzzy) <- "foo"
as.character(xyzzy)
tcl("set", as.character(xyzzy))

top <- tktoplevel() # a Tk widget, see Tk-widgets
ls(envir=top$env, all=TRUE)
ls(envir=.TkRoot$env, all=TRUE)# .Tcl.args put a callback ref in here
## End(Not run)
```

`tclServiceMode` *Allow Tcl events to be serviced or not*

Description

This function controls or reports on the Tcl service mode, i.e. whether Tcl will respond to events.

Usage

```
tclServiceMode(on = NULL)
```

Arguments

`on`	(logical) Whether event servicing is turned on.

Details

If called with `on == NULL` (the default), no change is made.

Value

The value of the Tcl service mode before the call.

Examples

```
## Not run:

    oldmode <- tclServiceMode(FALSE)
    # Do some work to create a nice picture.  Nothing will be displayed until...
    tclServiceMode(oldmode)
## End(Not run)
```

TkCommands *Tk non-widget commands*

Description

These functions interface to Tk non-widget commands, such as the window manager interface commands and the geometry managers.

Usage

```
tcl(...)
tktitle(x)

tktitle(x) <- value

tkbell(...)
tkbind(...)
tkbindtags(...)
tkfocus(...)
tklower(...)
tkraise(...)

tkclipboard.append(...)
tkclipboard.clear(...)

tkevent.add(...)
tkevent.delete(...)
tkevent.generate(...)
tkevent.info(...)

tkfont.actual(...)
tkfont.configure(...)
tkfont.create(...)
tkfont.delete(...)
tkfont.families(...)
tkfont.measure(...)
tkfont.metrics(...)
tkfont.names(...)
```

```
tkgrab(...)
tkgrab.current(...)
tkgrab.release(...)
tkgrab.set(...)
tkgrab.status(...)

tkimage.cget(...)
tkimage.configure(...)
tkimage.create(...)
tkimage.names(...)

## NB: some widgets also have a selection.clear command, hence the "X".

tkXselection.clear(...)
tkXselection.get(...)
tkXselection.handle(...)
tkXselection.own(...)

tkwait.variable(...)
tkwait.visibility(...)
tkwait.window(...)

## winfo actually has a large number of subcommands, but it's rarely
## used, so use tkwinfo("atom", ...) etc. instead.

tkwinfo(...)

# Window manager interface

tkwm.aspect(...)
tkwm.client(...)
tkwm.colormapwindows(...)
tkwm.command(...)
tkwm.deiconify(...)
tkwm.focusmodel(...)
tkwm.frame(...)
tkwm.geometry(...)
tkwm.grid(...)
tkwm.group(...)
tkwm.iconbitmap(...)
tkwm.iconify(...)
tkwm.iconmask(...)
tkwm.iconname(...)
tkwm.iconposition(...)
tkwm.iconwindow(...)
tkwm.maxsize(...)
tkwm.minsize(...)
tkwm.overrideredirect(...)
tkwm.positionfrom(...)
tkwm.protocol(...)
tkwm.resizable(...)
tkwm.sizefrom(...)
```

```
tkwm.state(...)
tkwm.title(...)
tkwm.transient(...)
tkwm.withdraw(...)

### Geometry managers

tkgrid(...)
tkgrid.bbox(...)
tkgrid.columnconfigure(...)
tkgrid.configure(...)
tkgrid.forget(...)
tkgrid.info(...)
tkgrid.location(...)
tkgrid.propagate(...)
tkgrid.rowconfigure(...)
tkgrid.remove(...)
tkgrid.size(...)
tkgrid.slaves(...)

tkpack(...)
tkpack.configure(...)
tkpack.forget(...)
tkpack.info(...)
tkpack.propagate(...)
tkpack.slaves(...)

tkplace(...)
tkplace.configure(...)
tkplace.forget(...)
tkplace.info(...)
tkplace.slaves(...)

## Standard dialogs
tkgetOpenFile(...)
tkgetSaveFile(...)
tkchooseDirectory(...)
tkmessageBox(...)
tkdialog(...)
tkpopup(...)

## File handling functions
tclfile.tail(...)
tclfile.dir(...)
tclopen(...)
tclclose(...)
tclputs(...)
tclread(...)
```

Arguments

x	A window object

value	For tktitle assignments, a character string.
...	Handled via .Tcl.args

Details

tcl provides a generic interface to calling any Tk or Tcl command by simply running .Tcl.args.objv on the argument list and passing the result to .Tcl.objv. Most of the other commands simply call tcl with a particular first argument and sometimes also a second argument giving the subcommand.

tktitle and its assignment form provides an alternate interface to Tk's wm title

There are far too many of these commands to describe them and their arguments in full. Please refer to the Tcl/Tk documentation for details. With a few exceptions, the pattern is that Tk subcommands like pack configure are converted to function names like tkpack.configure, and Tcl subcommands are like tclfile.dir.

See Also

TclInterface, TkWidgets, TkWidgetcmds

Examples

```
## Not run:
## These cannot be run by examples() but should be OK when pasted
## into an interactive R session with the tcltk package loaded

tt <- tktoplevel()
tkpack(l1<-tklabel(tt,text="Heave"), l2<-tklabel(tt,text="Ho"))
tkpack.configure(l1, side="left")

## Try stretching the window and then

tkdestroy(tt)
## End(Not run)
```

---

tkpager *Page file using Tk text widget*

Description

This plugs into file.show, showing files in separate windows.

Usage

```
tkpager(file, header, title, delete.file)
```

Arguments

file	character vector containing the names of the files to be displayed
header	headers to use for each file
title	common title to use for the window(s). Pasted together with the header to form actual window title.
delete.file	logical. Should file(s) be deleted after display?

Note

The "
b_" string used for underlining is currently quietly removed. The font and background colour are currently hardcoded to Courier and gray90.

See Also

file.show

---

tkStartGUI *Tcl/Tk GUI startup*

Description

Starts up the Tcl/Tk GUI

Usage

```
tkStartGUI()
```

Details

Starts a GUI console implemented via a Tk text widget. This should probably be called at most once per session. Also redefines the file pager (as used by help()) to be the Tk pager.

Note

tkStartGUI() saves its evaluation environment as .GUIenv. This means that the user interface elements can be accessed in order to extend the interface. The three main objects are named Term, Menu, and Toolbar, and the various submenus and callback functions can be seen with ls(envir=.GUIenv).

Author(s)

Peter Dalgaard

TkWidgetcmds *Tk widget commands*

Description

These functions interface to Tk widget commands.

Usage

```
tkactivate(widget, ...)
tkadd(widget, ...)
tkaddtag(widget, ...)
tkbbox(widget, ...)
tkcanvasx(widget, ...)
tkcanvasy(widget, ...)
tkcget(widget, ...)
tkcompare(widget, ...)
tkconfigure(widget, ...)
tkcoords(widget, ...)
tkcreate(widget, ...)
tkcurselection(widget,...)
tkdchars(widget, ...)
tkdebug(widget, ...)
tkdelete(widget, ...)
tkdelta(widget, ...)
tkdeselect(widget, ...)
tkdlineinfo(widget, ...)
tkdtag(widget, ...)
tkdump(widget, ...)
tkentrycget(widget, ...)
tkentryconfigure(widget, ...)
tkfind(widget, ...)
tkflash(widget, ...)
tkfraction(widget, ...)
tkget(widget, ...)
tkgettags(widget, ...)
tkicursor(widget, ...)
tkidentify(widget, ...)
tkindex(widget, ...)
tkinsert(widget, ...)
tkinvoke(widget, ...)
tkitembind(widget, ...)
tkitemcget(widget, ...)
tkitemconfigure(widget, ...)
tkitemfocus(widget, ...)
tkitemlower(widget, ...)
tkitemraise(widget, ...)
tkitemscale(widget, ...)
tkmark.gravity(widget, ...)
tkmark.names(widget, ...)
tkmark.next(widget, ...)
tkmark.previous(widget, ...)
tkmark.set(widget, ...)
tkmark.unset(widget, ...)
tkmove(widget, ...)
tknearest(widget, ...)
tkpost(widget, ...)
tkpostcascade(widget, ...)
tkpostscript(widget, ...)
tkscan.mark(widget, ...)
```

```
tkscan.dragto(widget, ...)
tksearch(widget, ...)
tksee(widget, ...)
tkselect(widget, ...)
tkselection.adjust(widget, ...)
tkselection.anchor(widget, ...)
tkselection.clear(widget, ...)
tkselection.from(widget, ...)
tkselection.includes(widget, ...)
tkselection.present(widget, ...)
tkselection.range(widget, ...)
tkselection.set(widget, ...)
tkselection.to(widget,...)
tkset(widget, ...)
tksize(widget, ...)
tktoggle(widget, ...)
tktag.add(widget, ...)
tktag.bind(widget, ...)
tktag.cget(widget, ...)
tktag.configure(widget, ...)
tktag.delete(widget, ...)
tktag.lower(widget, ...)
tktag.names(widget, ...)
tktag.nextrange(widget, ...)
tktag.prevrange(widget, ...)
tktag.raise(widget, ...)
tktag.ranges(widget, ...)
tktag.remove(widget, ...)
tktype(widget, ...)
tkunpost(widget, ...)
tkwindow.cget(widget, ...)
tkwindow.configure(widget, ...)
tkwindow.create(widget, ...)
tkwindow.names(widget, ...)
tkxview(widget, ...)
tkxview.moveto(widget, ...)
tkxview.scroll(widget, ...)
tkyposition(widget, ...)
tkyview(widget, ...)
tkyview.moveto(widget, ...)
tkyview.scroll(widget, ...)
```

Arguments

`widget`	The widget this applies to
`...`	Handled via `.Tcl.args`

Details

There are far too many of these commands to describe them and their arguments in full. Please refer to the Tcl/Tk documentation for details. Except for a few exceptions, the pattern is that Tcl widget commands possibly with subcommands like `.a.b selection clear` are converted to function names like `tkselection.clear` and the widget is given as the first argument.

See Also

TclInterface, TkWidgets, TkCommands

Examples

```
## Not run:
## These cannot be run by examples() but should be OK when pasted
## into an interactive R session with the tcltk package loaded

tt <- tktoplevel()
tkpack(txt.w <- tktext(tt))
tkinsert(txt.w, "0.0", "plot(1:10)")

# callback function
eval.txt <- function()
   eval(parse(text=tclvalue(tkget(txt.w, "0.0", "end"))))
tkpack(but.w <- tkbutton(tt,text="Submit", command=eval.txt))

## Try pressing the button, edit the text and when finished:

tkdestroy(tt)
## End(Not run)
```

TkWidgets *Tk widgets*

Description

Create Tk widgets and associated R objects.

Usage

```
tkwidget(parent, type, ...)
tkbutton(parent, ...)
tkcanvas(parent, ...)
tkcheckbutton(parent, ...)
tkentry(parent, ...)
tkframe(parent, ...)
tklabel(parent, ...)
tklistbox(parent, ...)
tkmenu(parent, ...)
tkmenubutton(parent, ...)
tkmessage(parent, ...)
tkradiobutton(parent, ...)
tkscale(parent, ...)
tkscrollbar(parent, ...)
tktext(parent, ...)
tktoplevel(parent=.TkRoot, ...)
```

Arguments

parent	Parent of widget window
type	string describing the type of widget desired
...	handled via .Tcl.args

Details

These functions create Tk widgets. tkwidget creates a widget of a given type, the others simply call tkwidget with the respective type argument.

It is not possible to describe the widgets and their arguments in full. Please refer to the Tcl/Tk documentation.

See Also

TclInterface, TkCommands, TkWidgetcmds

Examples

```
## Not run:
## These cannot be run by examples() but should be OK when pasted
## into an interactive R session with the tcltk package loaded

tt <- tktoplevel()
label.widget <- tklabel(tt, text="Hello, World!")
button.widget <- tkbutton(tt, text="Push",
                          command=function()cat("OW!\n"))
tkpack(label.widget, button.widget) # geometry manager
                                    # see Tk-commands

## Push the button and then...

tkdestroy(tt)
## End(Not run)
```

---

tk_select.list *Select Items from a List*

Description

Select item(s) from a character vector using a Tk listbox.

Usage

```
tk_select.list(list, preselect = NULL, multiple = FALSE, title = NULL)
```

Arguments

list	character. A list of items.
preselect	a character vector, or NULL. If non-null and if the string(s) appear in the list, the item(s) are selected initially.
multiple	logical: can more than one item be selected?
title	optional character string for window title.

Details

This is a version of `select.list` implemented as a Tk list box plus `OK` and `Cancel` buttons. There will be a scrollbar if the list is too long to fit comfortably on the screen.

The dialog box is *modal*, so a selection must be made or cancelled before the R session can proceed.

Value

A character vector of selected items. If `multiple` is false and no item was selected (or `Cancel` was used), `""` is returned. If `multiple` is true and no item was selected (or `Cancel` was used) then a character vector of length 0 is returned.

See Also

`select.list` (a text version except on Windows), `menu` (whose `graphics=TRUE` mode uses this on most Unix-alikes.

Index