The R Reference Index

The R Core Team

Version 1.4.1 (2002-01-30)

Contents

Chapter 1

The base package

`.Alias-deprecated` *Create Alias (Pointer) to R Object*

Description

`.Alias` creates an **alias** to another (part of) an R object which is more (memory-) efficient than usual assignment.

Usage

```
.Alias(expr)
```

Arguments

`expr`	an R expression; typically a name.

Details

Use as `new <- .Alias(expr)`, where `new` is a new name by which `expr` can be accessed.

Value

an identical copy of `expr`.

Warning

This has a **dangerous** semantic, and consequences can be unexpected (it can be used to defeat the call-by-value illusion). Know what you are doing *before* using `.Alias`!

See Also

`<-` for usual assignments.

Examples

```
mop <- options()
mop$browser <- "a browser"   # not set on all platforms
Op <- .Alias(mop)
## A change to mop is reflected in Op and vice versa
## -- ONLY if no new slots are created ...
mop$digits <- "Wow!"
Op$browser <- "another one"
mop$browser; Op$digits
all(names(mop) == names(Op) &
    sapply(seq(mop), function(i) all(Op[[i]] == mop[[i]])))
##> TRUE -- Op and mop ARE the same thing !

mop$newslot <- pi #--->> 'newslot' ==> (shallow) COPY of 'mop'
Op$newslot # R: still the old one, i.e. NULL
all(names(mop) == names(Op))# no longer TRUE

## Feel the power: 'call by reference', a function modifying its argument:
tst.Al <- function(x) {
   y <- .Alias(x) ; attributes(y) <- NULL ; invisible()
}
(x0 <- structure(1:5, my.att = "Y"))
tst.Al(x0) # *changes* x0 :
x0
stopifnot(is.null(attributes(x0)))
```

.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_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

```
.Script("perl", "maketitle.pl", file.path(.Library, "base", "TITLE"))
```

`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 vector of names to be abbreviated.
`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.

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

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.

See Also

`substr`.

Examples

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

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

---

abline *Add a Straight Line to a Plot*

Description

This function adds one or more straight lines through the current plot.

Usage

```
abline(a, b, untf = FALSE,\ dots)
abline(h=, untf = FALSE, ...)
abline(v=, untf = FALSE, ...)
abline(coef=, untf = FALSE, ...)
abline(reg=, untf = FALSE, ...)
```

Arguments

a,b	the intercept and slope.
untf	logical. See Details.
h	the y-value for a horizontal line.
v	the x-value for a vertical line.
coef	a vector of length two giving the intercept and slope.
reg	an object with a coef component. See Details.
...	graphical parameters.

Details

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 which contains reg$coef. If it is 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 alway refer to original coordinates.

The graphical parameters col and lty can be specified as arguments to abline; see par for details.

See Also

`lines` and `segments` for connected and arbitrary lines given by their *endpoints*. `par`.

Examples

```
data(cars)
z <- lm(dist ~ speed, data = cars)
plot(cars)
abline(z)
```

`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 vector

See Also

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

Examples

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

`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, ...)
add1.default(object, scope, scale = 0, test = c("none", "Chisq"),
             k = 2, trace = FALSE, ...)
add1.lm(object, scope, scale = 0, test = c("none", "Chisq", "F"),
        x = NULL, k = 2, ...)
add1.glm(object, scope, scale = 0, test = c("none", "Chisq", "F"),
         x = NULL, k = 2, ...)

drop1(object, scope, ...)
drop1.default(object, scope, scale = 0, test = c("none", "Chisq"),
              k = 2, trace = FALSE, ...)
drop1.lm(object, scope, scale = 0, all.cols = TRUE,
         test=c("none", "Chisq", "F"),k = 2, ...)
drop1.glm(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.
`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.

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`.

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)

B. D. Ripley

See Also

`step`, `aov`, `lm`, `extractAIC`.

Examples

```
example(step)#-> swiss
add1(lm1, ~ I(Education^2) + .^2)
drop1(lm1, test="F")

example(glm)
drop1(glm.D93, test="Chisq")
drop1(glm.D93, test="F")
```

---

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, ...)
aggregate.default(x, ...)
aggregate.data.frame(x, by, FUN, ...)
aggregate.ts(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.
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 functions 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

See Also

apply, lapply, tapply.

Examples

```
data(state)

## 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.)

data(presidents)
## 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 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.

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

logLik, AIC.logLik.

Examples

```
data(swiss)
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)))
```

AIC.logLik *AIC of a logLik Object*

Description

see *Description* in AIC.

Usage

```
AIC(object, ..., k = 2)
```

Arguments

object	an object inheriting from class "logLik", usually resulting from applying a logLik method to a fitted model object.
...	further arguments to be passed to or from methods.
k	numeric, the "penalty" per parameter to be used; the default k = 2 is the classical AIC.

Value

a numeric value with the corresponding 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

AIC, logLik.

---

airmiles *Commercial Airline Mileage*

Description

The revenue passenger miles flown by commercial airlines in the United States for each year from 1937 to 1960.

Usage

```
data(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

```
data(airmiles)
plot(airmiles, main = "airmiles data",
     xlab = "Passenger-miles flown by U.S. commercial airlines", col = 4)
```

---

airquality *New York Air Quality Measurements*

Description

Daily air quality measurements in New York, May to September 1973.

Usage

```
data(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

```
data(airquality)
pairs(airquality, panel = panel.smooth, main = "airquality data")
```

---

alias *Find Aliases (Dependencies) in a Model*

Description

Find aliases (linearly dependent terms) in a linear model specified by a formula.

Usage

```
alias(object, ...)
alias.formula(object, data, ...)
alias.lm(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; may be of class `"mtable"` which has its own `print` method.
`Partial`	The correlations of the estimable effects, with a zero diagonal.

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)

B.D. Ripley

Examples

```
## From Venables and Ripley (1997) p.210.
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)

## The next line is optional (for fractions package which gives neater
```

```
## results.)
has.VR <- require(MASS, quietly = TRUE)

op <- options(contrasts=c("contr.helmert", "contr.poly"))
npk.aov <- aov(yield ~ block + N*P*K, npk)
alias(npk.aov)
if(has.VR) detach(package:MASS)
options(op)# reset
```

`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.
`na.rm`	logical. If true `NA` values are removed before the result is computed.

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 the values in `x` are `TRUE`, and `FALSE` if any the values in `x` are `FALSE`.

If `x` consists of a mix of `TRUE` and `NA` values, then value is `NA`.

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 `identical` or combine the two, as shown in the documentation for `identical`.

Usage

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

all.equal.numeric(target, current,
                  tolerance= .Machine$double.eps ^ 0.5, scale=NULL, ...)
```

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.

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, comparisons are after scaling by `scale`.

For complex arguments, `Mod` of difference is used.

`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`.

Numerical differences are reported by relative error

See Also

`==`, and `all` for exact equality testing.

Examples

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

stopifnot(
all.equal(gamma(2:14),   cumprod(1:13))) # TRUE, but

all      (gamma(2:14) == cumprod(1:13)) # FALSE, since not exactly
all.equal(gamma(2:14),   cumprod(1:13), tol=0) # to see difference

all.equal(options(), .Options)
all.equal(options(), as.list(.Options))# TRUE
.Options $ myopt <- TRUE
all.equal(options(), as.list(.Options))
rm(.Options)
```

---

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)))
```

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.

See Also

`coefficients`, `effects`, `fitted.values`, `residuals`, `summary`.

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

```
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 `glm.obj`.
`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. 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).

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.

See Also

glm, anova.

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

```
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`.

See Also

The model fitting function `lm`.

Examples

```
## sequential table
data(LifeCycleSavings)
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
```

---

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

```
data(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

```
data(anscombe)
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)
```

---

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.
`na.rm`	logical. If true `NA` values are removed before the result is computed.

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 the values in `x` are `TRUE`, and `FALSE` if all the values in `x` are `FALSE`.

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

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")
```

---

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 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.

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.

Author(s)

B. D. Ripley

See Also

`lm`, `alias`, `proj`, `model.tables`

Examples

```
## From Venables and Ripley (1997) p.210.
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)

## as a test, not particularly sensible statistically
op <- options(contrasts=c("contr.helmert", "contr.treatment"))
npk.aovE <- aov(yield ~  N*P*K + Error(block), npk)
npk.aovE
summary(npk.aovE)
options(op)# reset to previous
```

---

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 `FALSE` then the returned object has the same dimensions as `a`, and the dimnames are dropped.

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.

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`.

Examples

```
stopifnot(
    append(1:5, 0:1, after=3)
 == append(1:3, c(0:1, 4:5)))
```

`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. In the case of functions like `+`, `%*%`, etc., the function name must be quoted.
`...`	optional arguments to `FUN`.

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 the calls to `FUN` return vectors of different lengths, `apply` returns a list of length `dim(X)[MARGIN]`.

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
```

`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, xout, method="linear", n=50,
          yleft, yright, rule = 1, f=0, ties = mean)
approxfun(x, y,       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 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 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. 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`.

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
```

---

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 who's `mode` equals `mode` are searched.
`simple.words`	logical; if `TRUE`, the `what` argument is only searched as whole only 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 to the same task as `apropos`. However, by default (`simple.words == TRUE`), only full words are searched.

Author(s)

Kurt Hornik and Martin Maechler (May 1997).

See Also

`objects` for listing objects from one place, `help.search` for searching the help system, `search` for the search path.

Examples

```
apropos("lm")
apropos(ls)
apropos("lq")

lm <- 1:pi
find(lm)        #> ".GlobalEnv"   "package:base"
find(lm, num=TRUE) # numbers with these names
find(lm, num=TRUE, mode="function")# only the second one
rm(lm)

apropos(".", mode="list")

# need a DOUBLE backslash '\\' (in case you don't see it anymore)
apropos("\\[")

# 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,}$")
table(nchar(apropos("^.{8,}$")))
```

`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.

See Also

`formals`, `help`.

Examples

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

`Arithmetic` *Arithmetic Operators*

Description

These binary operators perform arithmetic on vector objects.

Usage

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

Details

`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.

Value

They return numeric 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 )` unless `y == 0` where the result is `NA` or `NaN` (depending on the `typeof` of the arguments).

See Also

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

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 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.

`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.

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
```

`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 = NULL, lwd = par("lwd"), xpd = NULL)
```

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, xpd`	usual graphical parameters as in `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=2` an arrowhead is drawn at `(x0[i],y0[i])` and if `code=1` 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` and `lty` can be used to specify a color and line texture for the line segments which make up the arrows (`col` may be a vector).

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.

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')
```

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
as.environment("package:ctest")
```

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, ...)

as.function.default(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.

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 (to the nearest second).

Usage

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

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 take convert a wide variety of objects, including objects of the other class and of classes "date" (from package date), "chron" and "dates" (from package chron) to these classes. 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.)

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 date, 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 date, as class "POSIXlt"
unlist(unclass(z))            # a list shown as a named vector

as.POSIXlt(Sys.time(), "GMT") # the current time in GMT
```

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

x = value
```

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.

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. 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`.

Assignment Operators

There are three different assignment operators. The operators `<-` and `=` assign into the environment in which they are evaluated. The `<-` can be used anywhere, but the `=` is only allowed at the top level (that is, in the complete expression typed by the user) 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 then its value is redefined, otherwise assignment takes place globally. Note that their differs from that in the S language, but is useful in conjunction with the scoping rules of R.

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]]`). The name does not need to be quoted, though it can be.

See Also

`get`, `exists`, `environment`.

Examples

```
for(i in 1:6) { #-- Create objects  'r1', 'r2', ... 'r6' --
 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
```

`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 rectange.
`main`	overall title for the plot.
`xlab`	a label for the x axis. Defaults to the name of the row variable in `x` if non-`NULL`.
`ylab`	a label for the y axis. Defaults to the column names of the column variable in `x` if non-`NULL`.

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.

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://hotspur.psych.yorku.ca/SCS/sugi/sugi17-paper.html

See Also

`mosaicplot`; `chisq.test`.

Examples

```
data(HairEyeColor)
## Aggregate over sex:
x <- margin.table(HairEyeColor, c(1, 2))
x
assocplot(x, main = "Relation between hair and eye color")
```

---

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)))
```

Arguments

`what`	"database". This may currently be a `data.frame` or `list` or a R data file created with `save`.
`pos`	integer specifying position in `search()` where to attach.
`name`	alternative way to specify the database to be attached.

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 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 (eg 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`.

Note that by default assignment is not performed in an attached database. Attempting to modify a variable or function in an attached database will actually create a modified version in the user's workspace (the R global environment). For this reason `attach` can lead to confusion.

Value

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

See Also

`library`, `detach`, `search`, `objects`, `environment`.

Examples

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

`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

```
data(attenu)
```

Format

A dataframe 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

```
data(attenu)
## 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 *Attitudes Toward Supervisors*

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

```
data(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

```
data(attitude)
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)
```

---

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 character string specifying which attribute is to be accessed.

Value

This function provides access to a single object attribute. The simple form above returns the value of the named attribute. The assignment form causes the named attribute to take the value on the right of the assignment symbol.

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 attribute list. The first form above returns the an object's attribute list. The assignment forms make the list on the right-hand side of the assignment the object's attribute list (if appropriate).

Usage

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

Arguments

obj	an object

Details

The mostattributes assignment takes special care for the dim, names and dimnames attributes, and assigns them only when that is valid whereas as attributes assignment would give an error in that case.

See Also

attr.

Examples

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

Usage

```
autoload(name, package,...)
autoloader(name, package,...)
.AutoloadEnv
```

Arguments

`name`	string giving the name of an object
`package`	string giving the name of a package containing the object
`...`	other arguments to `library`

Value

This function is invoked for its side-effect.

See Also

`delay`, `library`

Examples

```
autoload("line","eda")
search()
ls("Autoloads")
all(ls("Autoloads") == ls(envir = .AutoloadEnv))
data(cars)
plot(cars)
z<-line(cars)
abline(coef(z))
search()
detach("package:eda")
search()
z<-line(cars)
search()
```

`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

data(warpbreaks)
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()
```

`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 = 0,
     pos = NA, outer = FALSE, font = NA, vfont = NULL, ...)
```

Arguments

`side`	an integer specifying which side of the plot the axis is to be drawn on.
`at`	the points at which tick-marks are to be drawn. Non-finite (infinite, `NaN` or `NA`) values are omitted.
`labels`	this can either be a logical value specifying whether (numerical) annotations are to be made at the tickmarks, or a vector of character strings to be placed at the tickpoints.
`tick`	a logical value specifying whether tickmarks should be drawn
`line`	the number of lines into the margin which the axis will be drawn. 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. this overrides the value 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.
`vfont`	vector font for text.
`...`	graphical parameters may also be passed as arguments to this function.

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.

Value

This function is invoked for its side effect, which is to add an axis to an already existing plot. The axis is placed as follows: 1=below, 2=left, 3=above and 4=right.

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), type = 's', xaxt='n', col = 'red')
axis(1, 1:7, LETTERS[1:7], col.axis = 'blue')
```

`axis.POSIXct` *Date-time Plotting Functions*

Description

Functions to manipulate objects of classes `"POSIXlt"` and `"POSIXct"` representing calendar dates and times (to the nearest second).

Usage

```
plot.POSIXct(x, y, xlab = "", ...)
plot.POSIXlt(x, y, xlab = "", ...)
axis.POSIXct(side, x, at, format, ...)
```

Arguments

`x, at`	A date-time 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`.
`...`	Further arguments to be passed from or to other methods, typically graphical parameters.

Details

The functions plot against an x-axis of date-times. `axis.POSIXct` works 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 for `axis.POSIXct` it specifies the locations of the ticks and labels: if `x` is specified a suitable grid of labels is chosen.

See Also

DateTimeClasses for details of the classes.

Examples

```
if(require(MASS, quietly = TRUE)) {
data(beav1)
attach(beav1)
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")
rm(time)
detach(beav1)
detach(package:MASS)
}

plot(.leap.seconds, 1:22, type="n", yaxt="n",
     xlab="leap seconds", ylab="", bty="n")
rug(.leap.seconds)
```

---

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

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)
( y2 <- backsolve(r, x, transpose = TRUE)) # 8 -12 -5
all(t(r) %*% y2 == x)# exactly on Linux (Pentium)
all(y  == backsolve(t(r), x, upper = FALSE, transpose = TRUE))
all(y2 == backsolve(t(r), x, upper = FALSE, transpose = FALSE))
```

---

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, upper)
bw.bcv(x, nb = 1000, lower, upper)
bw.SJ(x, nb=1000, lower, upper, 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.
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. (1999) *Modern Applied Statistics with S-PLUS.* Springer.

See Also

`density`.

`bandwith.nrd`, `ucv`, `bcv` and `width.SJ` in `MASS`, which are all scaled to the `width` argument of `density` and so give answers four times as large.

Examples

```
data(precip)
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)
```

---

barplot *Bar Plots*

Description

Creates a bar plot with vertical or horizontal bars.

Usage

```
barplot(height, width = 1, space = NULL,
        names.arg = NULL, legend.text = NULL, beside = FALSE,
        horiz = FALSE, density = NULL, angle = 45,
        col = heat.colors(NR), border = par("fg"),
        main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
        xlim = NULL, ylim = NULL,
        axes = TRUE, axisnames = TRUE,
        cex.axis = par("cex.axis"), cex.names = par("cex.axis"),
        inside = TRUE, plot = TRUE, ...)
```

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 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.
`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.
`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 names.
`inside`	logical. If `TRUE`, the lines which divide adjacent bars will be drawn.
`plot`	logical. If `FALSE`, nothing is plotted.
`...`	further graphical parameters (`par`) 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 `apply(mp, 2, mean)` for the midpoints of each *group* of bars, see example.

See Also

`plot(..., type="h")`, `dotchart`, `hist`.

Examples

```
tN <- table(Ni <- rpois(100, lambda=5))
r <- barplot(tN, col='gray')
#- 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)")

data(VADeaths, package = "base")
barplot(VADeaths, plot = FALSE)
barplot(VADeaths, plot = FALSE, beside = TRUE)

mp <- barplot(VADeaths) # default
tot <- apply(VADeaths, 2, sum)
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 = apply(mp, 2, mean), line = -2,
      text = paste("Mean", formatC(apply(hh, 2, mean))), 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))
```

BATCH *Batch Execution of R*

Description

Run R non-interactively with input from `infile` and optionally send output (stdout/stderr) to another file.

Usage

```
Rcmd 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`.
`outfile`	the name of a file to which to write output. If not given, the name used is the one of `infile`, with a possible '.R' extension stripped, and '.Rout' appended.

Details

By default, the input commands are printed along with the output. To suppress this behavior, add `options(echo = FALSE)` at the beginning of `infile`.

Files with an incomplete last line (no end of line mark) are now accepted.

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, as the `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, leg=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, leg=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, leg = 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, leg=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, leg=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, leg=paste("nu=",nus), col = nus+2, lwd=1)

## Check the Scaling :
for(nu in nus)
   print(all(abs(1- besselK(x,nu)*exp( x) / besselK(x,nu,expo=TRUE)) < 2e-15))
for(nu in nus)
   print(all(abs(1- besselI(x,nu)*exp(-x) / besselI(x,nu,expo=TRUE)) < 1e-15))

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, leg=paste("nu=",nus), col = nus+2, lwd=1)
```

`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,        lower.tail = TRUE, log.p = FALSE)
rbeta(n, shape1, shape2)
```

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 $P[X \leq x]$, 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 $0 < x < 1$.

Value

`dbeta` gives the density, `pbeta` the distribution function, `qbeta` the quantile function, and `rbeta` generates random deviates.

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)
```

bindenv *Binding and Environment Adjustments*

Description

These functions represent an experimental interface for adjustments to environments and bindings within environments. They allow for locking environments as well as individual bindings, and for linking a variable to a function.

Usage

```
lockEnvironment(env, bindings = FALSE)
environmentIsLocked(env)
lockBinding(sym, env)
bindingIsLocked(sym, env)
makeActiveBinding(sym, fun, env)
bindingIsActive(sym, env)
```

Arguments

`env`	an environment.
`bindings`	logical specifying whether bindings should be locked.
`sym`	a name object or character string
`fun`	a function taking zero or one arguments

Details

The function `lockEnvironment` locks its environment argument, which must be a proper environment, not NULL. Locking the NULL (base) environment may be supported later. Locking the environment prevents adding or removing variable bindings from the environment. Changing the value of a variable is still possible unless the binding has been locked.

`lockBinding` locks individual bindings in the specified environment. The value of a locked binding cannot be changed. Locked bindings may be removed from an environment unless the environment is locked.

`makeActiveBinding` installs `fun` so that getting the value of `sym` calls `fun` with no arguments, and assigning to `sym` calls `fun` with one argument, the value to be assigned. This allows things like C variables linked to R variables and variables linked to data bases to be implemented. It may also be useful for making thread-safe versions of some system globals.

Author(s)

Luke Tierney

Examples

```
# locking environments
e<-new.env()
assign("x",1, env=e)
get("x",env=e)
lockEnvironment(e)
get("x",env=e)
```

```
assign("x",2, env=e)
try(assign("y",2, env=e)) # error

# locking bindings
e<-new.env()
assign("x",1, env=e)
get("x",env=e)
lockBinding("x", e)
try(assign("x",2, env=e)) # error

# active bindings
f<-local({
    x <- 1
    function(v) {
       if (missing(v))
           cat("get\n")
       else {
           cat("set\n")
           x <<- v
       }
       x
    }
})
makeActiveBinding("fred", f, .GlobalEnv)
bindingIsActive("fred", .GlobalEnv)
fred
fred<-2
fred
```

`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.
`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.

References

Catherine Loader (2000). *Fast and Accurate Computation of Binomial Probabilities*; manuscript available from http://cm.bell-labs.com/cm/ms/departments/sia/catherine/dbinom

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
plot (0:n,     dbinom(0:n, n, pi/10, log=TRUE), type='l',
      main = "dbinom(*, log=TRUE) is better than  log(dbinom(*))")
lines(0:n, log(dbinom(0:n, n, pi/10)), col='red', lwd=2)
mtext("dbinom(k, log=TRUE)", adj=0)
mtext("extended range", adj=0, line = -1, font=4)
mtext("log(dbinom(k))", col="red", adj=1)
```

`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)
## 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)
```

`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) <- list
```

Arguments

`fun`	a function object or a character string naming the function to be manipulated. If not specified, the function calling `body` is used.
`list`	a list of R expressions.

Value

`body` returns the body of the function specified.

The assignment form sets the body of a function to the list on the right hand side.

See Also

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

Examples

```
body(body)
f <- function(x) x^5
body(f) <- expression(5^x)
f(3) # = 125
str(body(f))
```

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`.

See Also

`rect` for drawing of arbitrary rectangles.

Examples

```
plot(1:7,abs(rnorm(7)), type='h', axes = FALSE)
axis(1, labels = letters[1:7])
box(lty='137', col = 'red')
```

boxplot — *Box Plots*

Description

Produce box-and-whisker plot(s) of the given (grouped) values.

Usage

```
boxplot(x, ..., range = 1.5, width = NULL, varwidth = FALSE,
        notch = FALSE, outline = TRUE, names, boxwex = 0.8, plot = TRUE,
        border = par("fg"), col = NULL, log = "", pars = NULL,
        horizontal = FALSE, add = FALSE, at = NULL)
```

Arguments

`x, ...`	for specifying data from which the boxplots are to be produced as well as for giving graphical parameters. The named arguments in this (more precisely, in `list(x, ...)`) are treated as graphical parameters in addition to the ones given by argument `pars`. The other arguments specify the data, either as separate vectors, each corresponding to a component boxplot, or as a single list containing such vectors. `NA`s are allowed in the data.
`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 then the medians are significantly different at the 5 percent level.
`outline`	if `outline` is not true, the boxplot lines are not drawn.
`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.
`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 col the bodies of the box plots.
`log`	character indicating if x or y or both coordinates should be plotted in log scale.
`pars`	graphical parameters can also be passed as arguments to `boxplot`.
`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`).

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.
`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 which group the outlier belongs to
`names`	a vector of names for the groups

See Also

`boxplot.formula` for the formula interface; `boxplot.stats` which does the computation, `bxp` for the plotting; and `stripchart` for an alternative (with small data sets).

Examples

```
## boxplot on a formula:
data(InsectSprays)
boxplot(count ~ spray, data = InsectSprays, col = "lightgray")
# *add* notches (somewhat funny here):
boxplot(count ~ spray, data = InsectSprays,
        notch = TRUE, add = TRUE, col = "blue")

data(OrchardSprays)
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 :

data(ToothGrowth)
```

```
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))
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"))
```

`boxplot.formula` *Formula Notation for Box Plots*

Description

Produce box-and-whisker plot(s) of the given (grouped) values using formula notation.

Usage

```
boxplot(formula, data = NULL, ..., 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 the default boxplot method and graphical parameters may also be passed as arguments, see `par`.
`subset`	an optional vector specifying a subset of observations to be used for plotting.

Details

This is a method of the generic function `boxplot`. It operates by setting up the data from the formula specification, and then calling `boxplot.default`.

See Also

`boxplot.default`

Examples

```
data(OrchardSprays)
boxplot(decrease ~ treatment, data = OrchardSprays,
        log = "y", col="bisque")
```

`boxplot.stats` *Box Plot Statistics*

Description

This function is typically called by `boxplot` 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 (`NA`s and `NaN`s 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 interquartile coef 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 "hinge"s are versions of the first 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.

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 of non-`NA` observations in the sample.
`conf`	the lower and upper extremes of the "notch" (`if(do.conf)`).
`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.

See Also

`fivenum`, `boxplot`, `bxp`.

Examples

```
x <- c(1:100, 1000)
str(b1 <- boxplot.stats(x))
str(b2 <- boxplot.stats(x, do.conf=FALSE, do.out=FALSE))
stopifnot(b1 $ stats == b2 $ stats) # do.out=F is still robust
str(boxplot.stats(x, coef = 3, do.conf=FALSE))
## no outlier treatment:
str(boxplot.stats(x, coef = 0))

str(boxplot.stats(c(x, NA))) # slight change : n + 1
str(r <- boxplot.stats(c(x, -1:1/0)))
stopifnot(r$out == c(1000, -Inf, Inf))
```

---

bringToTop *Assign Focus to a Window*

Description

`bringToTop` brings the specified screen device's window to the front of the window stack (and gives it focus). With argument `-1`, it brings the console to the top.

Usage

```
bringToTop(which = dev.cur())
```

Arguments

`which`	a device number, or `-1`.

See Also

`windows`

---

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` causes a pause in the execution of the current expression and runs a copy of the R interpreter which has access to variables local to the environment where the call took place.

Local variables can be listed with `ls`, and manipulated with R expressions typed to this sub-interpreter. The interpreter copy is exited by typing `c`. Execution then resumes at the statement following the call to `browser`.

Typing `n` causes the step-through-debugger, to start and it is possible to step through the remainder of the function one line at a time.

Typing `Q` quits the current execution and returns you to the top-level prompt.

See Also

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

`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", wait = TRUE)
```

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).
`wait`	logical. Should R wait for the editor to return?

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` (which is the default 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.

On some systems 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`.

Author(s)

This help page is adapted from the Emacs manual

See Also

R FAQ

`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.

`bxp` *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, boxwex = 0.8, border = par("fg"), col = NULL,
    log = "", 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 boxplot lines are not drawn.
`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.
`notch.frac`	numeric in (0,1). When `notch=TRUE`, the fraction of the box width that the notches should use.
`border`	character, the color of the box borders. Is recycled for multiple boxes.
`col`	character; the color within the box. Is recycled for multiple boxes
`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 can be passed as arguments to this function, either as a list (`pars`) or normally(`...`).

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.

Examples

```
set.seed(753)
str(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)
bxp(bx.p, notch = TRUE, col= "lightblue", frame= FALSE, outl= FALSE,
    main = "bxp(*, frame= FALSE, outl= FALSE)")
bxp(bx.p, notch = TRUE, col= "lightblue", border="red", ylim = c(-4,4),
    pch = 22, bg = "green", log = "x", main = "... log='x', ylim=*")
par(op)
op <- par(mfrow= c(1,2))
data(PlantGrowth)
## single group -- no label
boxplot(weight~group,data=PlantGrowth,subset=group=="ctrl")
bx<-boxplot(weight~group,data=PlantGrowth,subset=group=="ctrl",plot=FALSE)
## with label
bxp(bx,show.names=TRUE)
par(op)
```

---

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(x)`.
`FUN`	a function to be applied to data frame subsets of `x`.
`...`	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 term.

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

```
data(warpbreaks)
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 *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.

Author(s)

B.D. Ripley

See Also

`contrasts`, `contr.sum`, etc.

Examples

```
## reset contrasts to defaults
options(contrasts=c("contr.treatment", "contr.poly"))
data(warpbreaks)
attach(warpbreaks)
tens <- C(tension, poly, 1)
attributes(tens)
detach()
## 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))

data(esoph) # following on from help(esoph)
model3 <- glm(cbind(ncases, ncontrols) ~ agegp + C(tobgp, , 1) +
     C(alcgp, , 1), data = esoph, family = binomial())
summary(model3)
```

---

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.

Usage

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

Arguments

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

See Also

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

Examples

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

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.

Value

`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).

`is.call` is used to determine whether `x` is a call (i.e., of mode `"call"`).

It is not possible to coerce objects to mode call (objects either are calls or they are not calls). `as.call` returns its argument if it is a call and otherwise terminates with an error message.

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
```

`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 `X11` and the data editor available?
`GNOME`	(Unix) Is the GNOME GUI in use and are `GTK` and `GNOME` graphics devices available?
`libz`	Is `gzfile` available?
`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?
`cledit`	Is command-line editing available in the current R session? This is false in non-interactive sessions.
IEEE754	Does this platform have IEEE 754 arithmetic. Note that this is more correctly known by the international standard IEC 60059.

See Also

`.Platform`

Examples

```
capabilities()

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

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

cars *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

```
data(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

```
data(cars)
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)
```

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, ...)
case.names.lm(object, full = FALSE, ...)

variable.names(object, ...)
variable.names.lm(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`

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
```

---

cat *Concatenate and Print*

Description

Prints the arguments, coercing them if necessary to character mode first.

Usage

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

Arguments

`...`	R objects which are coerced to character strings, concatenated, and printed, with the remaining arguments controlling the output.
`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`.
`sep`	character string to insert between the objects to print.
`fill`	a logical or 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.
`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` converts its arguments to character strings, concatenates them, separating them by the given `sep=` string, and then prints them.

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

`cat` is useful for producing output in user-defined functions.

Value

None (invisible `NULL`).

See Also

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

Examples

```
## 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=""))
```

`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.

See Also

`dt` for the t distribution which generalizes `dcauchy(*, l = 0, s = 1)`.

Examples

```
all.equal(dcauchy(-1:4), 1 / (pi*(1 + (-1:4)^2)))
```

`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. There may be methods for other R classes.

Usage

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

Arguments

`...`	vectors or matrices.
`deparse.level`	integer controlling the construction of labels; currently, `1` is the only possible value.

Details

The functions `cbind` and `rbind` are generic, with methods for data frames. The data frame method will be used if an 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.

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` when 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.

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.

See Also

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

Examples

```
cbind(1, 1:7) # the '1' (= shorter vector) is recycled
cbind(1:7, diag(3))# vector is subset -> warning

cbind(0, rbind(1, 1:3))

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

---

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.

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.

`is.character` returns `TRUE` or `FALSE` depending on whether its argument is of character type or not.

Note

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

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
```

`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.
`table`	the values to be matched against.
`nomatch`	the value 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 `NA` is returned.

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.
`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 repreated 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.

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)
```

`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")
str(L2 <- check.options(list(a=0:2), name.opt = "L1"))
str(check.options(NULL, reset = TRUE, name.opt = "L1"))
```

`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

```
data(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**, p.214.

References

McNeil, D. R. (1977) *Interactive Data Analysis.* New York: Wiley.

Examples

```
data(chickwts)
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)
```

`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.
`ncp`	non-centrality parameter. For `rnchisq`, `ncp=0` is the only possible value.
`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`$= n$ 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 $x \geq 0$. It 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.

Value

`dchisq` gives the density, `pchisq` gives the distribution function, `qchisq` gives the quantile function, and `rchisq` generates random deviates.

See Also

dgamma for the Gamma distribution which generalizes the chi-squared one.

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))
```

---

chol *The Choleski Decomposition*

Description

Compute the Choleski factorization of a symmetric (Hermitian), positive definite square matrix.

Usage

```
chol(x)
```

Arguments

x	a symmetric, positive definite matrix.

Details

Note that only the upper triangular part of x is used such that the above only holds when x *is* symmetric.

If the matrix is not positive-definite an error is generated. A non-negative definite but not strictly positive-definite matrix may or may not give an error depending on the numerical errors.

Value

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

References

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

See Also

chol2inv for its *inverse*, 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'
all(abs(m  -  t(cm) %*% cm) < 100* .Machine$double.eps) # TRUE
```

`chol2inv` *Inverse from Choleski Decomposition*

Description

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

Usage

```
chol2inv(x, size = ncol(x))
```

Arguments

`x`	a matrix. The first `nc` 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.

Value

The inverse of the decomposed matrix.

References

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

See Also

`chol`, `solve`.

Examples

```
cma <- chol(ma  <- cbind(1, 1:3, c(1,3,7)))
t(cma) %*% cma # = ma
all.equal(diag(3), ma %*% chol2inv(cma))
```

`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 components `x` and `y`

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.

Author(s)

B. D. Ripley

References

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)
plot(X, cex=0.5)
hpts <- chull(X)
hpts <- c(hpts, hpts[1])
lines(X[hpts, ])
```

`class` *Object Classes*

Description

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

Usage

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

Arguments

`x`	an objects
`what`	a character vector naming classes.
`which`	logical affecting return value: see Details.

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. 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.

The function `class` prints the vector of names of classes an object inherits from. Correspondingly, `class<-` sets the classes an object inherits from.

`unclass` returns (a copy of) its argument with its class information removed.

`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`.

See Also

`UseMethod`, `NextMethod`.

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
```

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

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

```
data(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

```
data(co2)
plot(co2, ylab = expression("Atmospheric concentration of CO"[2]),
     las = 1)
title(main = "co2 data set")
```

---

codes *Factor Codes*

Description

This (generic) function returns a numeric coding of a factor. It can also be used to assign to a factor using the coded form.

Usage

```
codes(x, ...)
codes(x) <- value
```

Arguments

`x`	an object from which to extract or set the codes.
`...`	further arguments passed to or from other methods.

Value

For an ordered factor, it returns the internal coding (1 for the lowest group, 2 for the second lowest, etc.).

For an unordered factor, an alphabetical ordering of the levels is assumed, i.e., the level that is coded 1 is the one whose name is sorted first according to the prevailing collating sequence. **Warning:** the sort order may well depend on the locale, and should not be assumed to be ASCII.

Note

Normally `codes` is not the appropriate function to use with an unordered factor. Use `unclass` or `as.numeric` to extract the codes used in the internal representation of the factor, as these do not assume that the codes are sorted.

The behaviour for unordered factors is dubious, but compatible with S version 3. To get the internal coding of a factor, use `as.integer`. Note in particular that the codes may not be the same in different language locales because of collating differences.

See Also

factor, levels, nlevels.

Examples

```
codes(rep(factor(c(20,10)),3))

x <- gl(3,5)
codes(x)[3] <- 2
x

data(esoph)
( ag <- esoph$alcgp[12:1] )
codes(ag)

codes(factor(1:10)) # BEWARE!
```

---

coefficients *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. (Note that the method is `coef` and not `coefficients`.)

Value

Coefficients extracted from the model object `object`.

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))
```

`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`.

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
```

`col2rgb` *Color to RGB Conversion*

Description

"Any R color" to RGB (red/green/blue) conversion.

Usage

```
col2rgb(col)
```

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]`.

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 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(1:7)# the ones from the palette()
stopifnot(col2rgb(0) ==  # col = 0 is the background color:
    print(col2rgb(par("bg"))))

col2rgb(paste("gold", 1:4, sep=""))

stopifnot(col2rgb("#08a0ff") == c(8, 160, 255))
## 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=""))
stopifnot(grC["red",] == grC["green",],
          grC["red",] == grC["blue",],
          grC["red", 1:4] == c(0,3,5,8))
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=","))
str(cl)

 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])
 }
```

`colors` *Color Names*

Description

Returns the built-in color names which R knows about.

Usage

```
colors()
```

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`, `gray`; `rainbow` for a nice example; and `heat.colors`, `topo.colors` for images.

`col2rgb` for translating to RGB numbers and extended examples.

Examples

```
str(colors())
```

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.

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.

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`).

Usage

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

Arguments

`x`	any R object
`value`	a `character` vector

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 vectors.

Usage

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

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

Don't 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 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.

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
```

`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

```
data(airquality)
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]
```

`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.)

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, `Mod`$(z) = \sqrt{x^2 + y^2}$, and for $\phi = Arg(z)$, $x = \cos(\phi)$ and $y = \sin(\phi)$.

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

Examples

```
( z <- 0i ^ (-3:3) )
stopifnot(Re(z) == 0 ^ (-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)

all(Mod ( 1 -  sin(z) / ( (exp(1i*z)-exp(-1i*z))/(2*1i) ))
    < 100*.Machine$double.eps)
## 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")
```

`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.

Author(s)

B.D. Ripley

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

open(con, open = "r", blocking = TRUE, ...)
close(con, type = "rw", ...)

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://`). `file` also accepts complete URLs.
`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 'Blocking' section below.
`encoding`	An integer vector of length 256.
`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.
`where`	integer. A file position (relative to the origin specified by `origin`), or `NA`.
`rw`	character. Empty or `"read"` or `"write"`, partial matches allowed.
`...`	arguments passed to or from other methods.

Details

The first six 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`: such connections can only be binary.

All platforms support file connections 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.

`close` closes and destroys a connection.

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.

`"r+"`, `"r+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.

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

The encoding vector is used to map the input from a file or pipe to the platform's native character set. Supplied examples are `native.enc` as well as `MacRoman`, `WinAnsi` and `ISOLatin1`, whose actual encoding is platform-dependent. Missing characters are mapped to a space in these encodings.

Value

`file`, `pipe`, `fifo`, `url`, `gzfile` 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.

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()`.

Note

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

References

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

See Also

`textConnection`, `seek`, `readLines`, `showConnections`, `pushBack`.

`capabilities` to see if `gzfile`, `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")

if(capabilities("libz")) {
  zz <- gzfile("ex.gz", "w")  # compressed file
```

```
  cat("TITLE extra line", "2 3 5 7", "", "11 13 17", file = zz, sep = "\n")
  close(zz)
  print(readLines(gzfile("ex.gz")))
  unlink("ex.gz")
}

## 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")
}

## 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
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)

## 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)
```

```
## 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)
```

`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.

See Also

`data`.

Examples

```
stopifnot(
 nchar(letters) == 1,
 month.abb == substr(month.name, 1, 3)
)
eps <- .Machine$double.eps
```

```
all.equal(pi, 4*atan(1), tol= 2*eps)

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

`contour` *Display Contours*

Description

Create a contour plot, or add contour lines to an existing plot.

Usage

```
contour(x, ...)
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 (`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.
`labels`	a vector giving the labels for the contour lines. If `NULL` then the levels are used as labels.
`labcex`	`cex` for contour labelling.
`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 graphical parameters (see `par`) and the arguments to `title` may also be supplied.

Details

`contour` is a generic function with only a default method in base R.

There is currently no documentation about the algorithm. The source code is in '$R_HOME/src/main/plot3d.c'.

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`.

See Also

`filled.contour` for "color-filled" contours, `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 = "/")
## Should not be necessary:
z[!is.finite(z)] <- NA
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)

data("volcano")
```

```
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)
par(opar)
```

`contrast` *Contrast Matrices*

Description

Return a matrix of contrasts.

Usage

```
contr.helmert(n, contrasts = TRUE)
contr.poly(n, contrasts = TRUE)
contr.sum(n, contrasts = TRUE)
contr.treatment(n, base = 1, 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.
`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` then a square indicator matrix is returned.

Note that as from R version 0.62.2, `contr.poly` returns contrasts based on orthogonal (rather than raw) polynomials.

Value

A matrix with `n` rows and `k` columns, with `k=n-1` if `contrasts` is `TRUE` and `k=n` if `contrasts` is `FALSE`.

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.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) <- ctr
```

Arguments

`x`	a factor.
`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`.
`ctr`	either a 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.

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.

Note

Prior to R version 1.2.0, `contrasts(, FALSE)` called a contrasts function with `contrasts = FALSE`. This normally gave the same result, but not for `contr.poly`, the default for ordered factors.

See Also

`C`, `contr.helmert`, `contr.poly`, `contr.sum`, `contr.treatment`; `glm`, `aov`, `lm`.

Examples

```
example(factor)
(fff <- factor(ff))
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)
```

---

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
```

See Also

ifelse, switch.

Examples

```
for(i in 1:5) print(1:i)
```

`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 'ts' 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")
```

`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).

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

Cleveland, W. S. (1993) *Visualizing Data.* New Jersey: Summit Press.

See Also

`pairs`, `panel.smooth`, `points`.

Examples

```
## Tonga Trench Earthquakes
data(quakes)
coplot(long ~ lat | depth, data = quakes)
given.depth <- co.intervals(quakes$depth, number=4, overlap=.1)
coplot(long ~ lat | depth, data = quakes, given.v=given.depth, rows=1)

## Conditioning on 2 variables:
ll.dm <- long ~ lat | 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

data(warpbreaks)
## given two factors
coplot(breaks ~ 1:54 | wool * tension, data = warpbreaks, show.given = 0:1)
coplot(breaks ~ 1:54 | wool * tension, data = warpbreaks,
       col = "red", bg = "pink", pch = 21,
       bar.bg = c(fac = "light blue"))

## Example with empty panels:
data(state)
attach(data.frame(state.x77))#> don't need 'data' arg. below
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)
detach() # data.frame(state.x77)
```

---

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.

`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.

Usage

```
var(x, y = NULL, na.rm = FALSE, use)
cor(x, y = NULL, use = "all.obs")
cov(x, y = NULL, use = "all.obs")
```

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).
`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"`.
`na.rm`	logical. Should missing values be removed?

Details

For `cov` and `cor` one must *either* give a matrix or data frame for `x` *or* give both `x` and `y`.

`var` 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.

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, and from R 1.2.3 fail if `x` has length zero.

Value

For `r <- cor(*, use = "all.obs")`, it is now guaranteed that `all(r <= 1)`.

See Also

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:
data(longley)
(Cl <- cor(longley))
## Graphical Correlation Matrix:
symnum(Cl) # highly correlated

##--- Missing value treatment:
data(swiss)
C1 <- cov(swiss)
range(eigen(C1, only=TRUE)$val) # 6.19  1921
swiss[1,2] <- swiss[7,3] <- swiss[25,5] <- NA # create 3 "missing"

 C2 <- cov(swiss) # Error: missing obs...

C2 <- cov(swiss, use = "complete")
range(eigen(C2, only=TRUE)$val) # 6.46  1930
C3 <- cov(swiss, use = "pairwise")
range(eigen(C3, only=TRUE)$val) # 6.19  1938
```

---

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")
```

`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)
```

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`.

Details

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.

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`.

`crossprod` *Matrix Crossproduct*

Description

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

Usage

```
crossprod(x, y=x)
```

Arguments

`x, y`	matrices.

See Also

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

Examples

```
(z <- crossprod(1:4))    # = sum(1 + 2^2 + 3^2 + 4^2)
drop(z)                  # scalar
```

`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 object.

Details

An `NA` value in `x` causes the corresponding and following elements of the return value to be `NA`.

Examples

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

`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, ...)
plot(x, from = 0, to = 1, xlim = NULL, ...)
```

Arguments

`expr`	an expression written as a function of `x`, or alternatively 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, ylab, log, ...`	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)
```

`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, ...)
cut.default(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 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 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 how many digits should be used in formatting the numbers `b1`, `b2`, ...

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.

See Also

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

Examples

```
Z <- rnorm(10000)
table(cut(Z, br = -6:6))
system.time(print(sum(table(cut(Z, br = -6:6, labels=FALSE)))))
system.time(print(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!
```

---

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

Description

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

Usage

```
cut(x, breaks, labels=NULL, start.on.monday=TRUE, ...)
```

Arguments

`x`	an object inheriting from class `"POSIXt"`.
`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 `"secs"`, `"mins"`, `"hours"`, `"days"`, `"weeks"`, `"months"` or `"years"`.
`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.
`start.on.monday`	logical. If `breaks = "weeks"`, should the week start on Mondays or Sundays?
`...`	arguments to be passed to or from other methods.

Value

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

See Also

`seq.POSIXt`, `cut`

Examples

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

`data` *Data Sets*

Description

Loads specified data sets, or list the available data sets.

Usage

```
data(..., list = character(0), package = .packages(),
     lib.loc = NULL, verbose = getOption("verbose"))
```

Arguments

`...`	a sequence of names or character strings.
`list`	a character vector.
`package`	a name or character vector giving the packages to look into for data sets. By default, all packages in the search path are used, then the 'data' directory (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. If the default is used, the loaded packages are searched before the libraries.
`verbose`	a logical. If `TRUE`, additional diagnostics are printed.

Details

Currently, four formats of data files are supported:

1. files ending '.RData' or '.rda' are `load()`ed.
2. files ending '.R' or '.r' are `source()`d in, with the R working directory changed temporarily to the directory containing the respective file.
3. files ending '.tab' or '.txt' are read using `read.table(..., header = TRUE)`, and hence result in a data frame.
4. files ending '.csv' are read using `read.table(..., header = TRUE, sep = ";")`, and also result in a data frame.

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.

If no data sets are specified, `data` lists the available data sets. It looks for a file '00Index' in a 'data' directory of each specified package, and uses these files to prepare a listing. If there is a 'data' area but no index 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. In earlier versions of R, an empty character vector was returned along with listing available data sets.

If `lib.loc` is not specified, the datasets are searched for amongst those packages already loaded, followed by the 'data' directory (if any) of the current working directory and then packages in the specified libraries. If `lib.loc` *is* specified, packages are searched for in the specified libraries, even if they are already loaded from another library.

To just look in the 'data' directory of the current working directory, set `package = NULL`.

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 pre-process 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.

Examples

```
data()                      # list all available data sets
data(package = base)        # list the data sets in the base package
data(USArrests, "VADeaths") # load the data sets 'USArrests' and 'VADeaths'
help(USArrests)             # give information on data set 'USArrests'
```

---

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.)

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)
```

`data.frame` *Data Frames*

Description

These functions create or manipulate 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)

as.data.frame(x, row.names = NULL, optional = FALSE)
is.data.frame(x)

row.names(x)
row.names(x) <- names
```

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`	a character 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. If necessary they are adjusted (by `make.names`) so that they are.
`optional`	logical. If `TRUE`, setting row names is optional.
`x`	object of class `data.frame`.

Details

Character variables passed to `data.frame` are converted to factor columns unless protected by `I`. It also applies to adding columns to a data frame.

If a list or data frame or matrix is passed to `data.frame` it is as if each column had been passed as a separate argument, with the exception of matrices of class `model.matrix`.

Value

For `data.frame(.)` a data frame, a matrix-like stucture whose columns may be of differing types (numeric, factor and character).

`as.data.frame` is generic function with many methods. It attempts to coerce its argument to be a data frame.

`is.data.frame` returns `TRUE` if its argument is a data frame and `FALSE` otherwise.

`row.names` can be used to set and retrieve the row names of a data frame, similarly to `rownames` for arrays (and it is a generic function that calls `rownames` for an array argument.

Note

In versions of R prior to 1.4.0 (and in S3) logical columns were converted to factors.

See Also

`print.data.frame`, `read.table`, `Math.data.frame` etc, about *Group* methods for `data.frame`s; `make.names`.

Examples

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

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

## do not convert to factor, using I() :
str(cbind(d, char = I(letters[1:10])), vec.len = 10)

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

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

Usage

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

Arguments

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

See Also

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

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.

Note

The details of interface to the data grid may differ by platform and GUI. The following description applies to the GraphApp-based implementation under Windows.

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.

The 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. The cursor changes to an I-beam to indicate that the cell is in enter mode. 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.)

Double-clicking on a cell selects the cell and makes it into an editable field (a cursor will appear at the end of the text and it will change to the text highlight colour). The edited text is entered by selecting another cell, for example by hitting Return. There is no way

to cancel the edits. The field will be expanded to the right if necessary to accommodate existing long strings, so it is preferable not to edit in the right-most displayed column. (The editable field is itself scrollable.)

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 `NA`s.) Enter the changes made in the popup window by clicking on its close box.

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.

There is a popup-menu accessed by right-clicking anywhere in the window that refers to the currently selected cell. This can copy the value to or paste from the clipboard, or paste in common values in that column. Copying and pasting can also be accessed by the usual keyboard shortcuts Control-C and Control-V.

Columns can be resized by selecting and dragging a line (the cursor will change) within limits: columns must be between 4 and 50 chars wide. The Autosize item on the popup menu will resize the currently selected column.

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 50 characters (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
data.entry(x,y)
```

`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' `ctime` ensuring the above fixed format. Timezone and Daylight Saving Time are taken account of, but *not* indicated in the result.

Examples

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

DateTimeClasses *Date-Time Classes*

Description

Description of the classes `"POSIXlt"` and `"POSIXct"` representing calendar dates and times (to the nearest second).

Usage

```
print.POSIXct(x, ...)
print.POSIXlt(x, ...)

summary.POSIXct(object, digits = 15, ...)
summary.POSIXlt(object, digits = 15, ...)

time + number
time - number
time1 lop time2
```

Arguments

`x, object`	An object to be printed or summarized.
`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, time1, time2`	date-time objects.
`number`	a numeric object.
`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.

Unfortunately, the conversion is complicated by the operation of time zones and leap seconds (22 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 2000, 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.

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 Also

`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 22 leapseconds in your timezone
```

---

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.
`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.
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, seperated 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 seperated 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.

See Also

`write.table`.

Examples

```
## Create a reduced version of the 'CONTENTS' file in package 'eda'
x <- read.dcf(file = system.file("CONTENTS", package = "eda"),
              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). Currently you can only debug functions that have bodies enclosed in braces. This is a bug and will be fixed soon. You take the next step by typing carriage return, `n` or `next`. You can see the values of variables by typing their names. Typing `c` or `cont` causes the debugger to continue to the end of the function. You can `debug` new functions before you step in to them from inside the debugger. Typing `Q` quits the current execution and returns you to the top–level prompt. Typing `where` causes the debugger to print out the current stack trace (all functions that are active). If you have variables with names that are identical to the controls (eg. `c` or `n`) then you need to use `print(c)` and `print(n)` to evaluate them.

See Also

`browser`, `traceback` to see the stack after an `Error: ...` message.

`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`.

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.

Author(s)

B. D. Ripley

See Also

`options` for setting `error` options.

Examples

```
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()}))
```

`Defunct` *Defunct Functions*

Description

The functions listed here are no longer part of R as they are not needed (any more).

Usage

```
.Defunct()

Version()
provide(package)
```

```
.Provided
category(...)
dnchisq(.)
pnchisq(.)
qnchisq(.)
rnchisq(.)
print.anova.glm(.)
print.anova.lm(.)
print.tabular(.)
print.plot(.)
save.plot(.)
system.test(.)
dotplot(...)
stripplot(...)
getenv(...)
read.table.url(url, method,...)
scan.url(url, file = tempfile(), method, ...)
source.url(url, file = tempfile(), method, ...)
httpclient(url, port=80, error.is.fatal=TRUE, check.MIME.type=TRUE,
           file=tempfile(), drop.ctrl.z=TRUE)
parse.dcf(text = NULL, file = "", fields = NULL, versionfix = FALSE)
```

Details

`.Defunct` is the function to which defunct functions are set.

`category` has been an old-S function before there were factors; should be replaced by `factor` throughout!

The `*chisq()` functions now take an optional non-centrality argument, so the `*nchisq()` functions are no longer needed.

The new function `dev.print()` should now be used for saving plots to a file or printing them.

`provide` and its object `.Provided` have been removed. They were never used for their intended purpose, to allow one package to subsume another.

`dotplot` and `stripplot` have been renamed to `dotchart` and `stripchart`, respectively.

`getenv` has been replaced by `Sys.getenv`.

`*.url` are replaced by calling `read.table`, `scan` or `source` on a `url` connection.

`httpclient` was used by the deprecated `"socket"` method of `download.file`.

`parse.dcf` has been replaced by `read.dcf`, which is much faster, but has a slightly different interface.

See Also

`Deprecated`

`delay` *Delay Evaluation*

Description

`delay` creates a *promise* to evaluate the given expression in the specified environment if its value is requested. This provides direct access to *lazy evaluation* mechanism used by R for the evaluation of (interpreted) functions.

Usage

```
delay(x, env=.GlobalEnv)
```

Arguments

`x`	an expression.
`env`	an evaluation environment

Details

This is an experimental feature and its addition is purely for evaluation purposes.

Value

A *promise* to evaluate the expression. The value which is returned by `delay` can be assigned without forcing its evaluation, but any further accesses will cause evaluation.

Examples

```
x <- delay({
    for(i in 1:7)
        cat("yippee!\n")
    10
})

x^2#- yippee
x^2#- simple number
```

`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.

`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)
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"))
         )
```

---

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, device = getOption("device"),
     package = .packages(), lib.loc = NULL,
     character.only = FALSE, verbose = getOption("verbose"))
```

Arguments

`topic`	the topic which should be demonstrated. If omitted, the list of available topics is displayed.
`device`	the graphics device to be used.
`package`	a name or character vector giving the packages to look into for data sets. 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 instead of name.
`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)


 ch <- "scoping"
 demo(ch, character = TRUE)
```

`density` *Kernel Density Estimation*

Description

The function `density` computes kernel density estimates with the given kernel and bandwidth.

Usage

```
density(x, bw = "nrd0", adjust = 1,
        kernel = c("gaussian", "epanechnikov", "rectangular", "triangular",
                   "biweight", "cosine", "optcosine"),
        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.
`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.

Details

The algorithm used in `density` 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

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. (1999) *Modern Applied Statistics with S-PLUS.* 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
data(faithful)
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))

(kernels <- eval(formals(density)$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)

(RKs <- cbind(sapply(kernels, function(k)density(kern = k, give.Rkern = TRUE))))
100*round(RKs["epanechnikov",]/RKs, 4) ## Efficiencies

data(precip)
bw <- bw.SJ(precip) ## sensible automatic choice
plot(density(precip, bw = bw, n = 2^13),
     main = "same sd bandwidths, 7 different kernels")
for(i in 2:length(kernels))
   lines(density(precip, bw = bw, kern =  kernels[i], n = 2^13), 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, n = 2^13),
     main = "equivalent bandwidths, 7 different kernels")
for(i in 2:length(kernels))
   lines(density(precip, bw = bw, adjust = h.f[i], kern =  kernels[i],
         n = 2^13), col = i)
legend(55, 0.035, legend = kernels, col = seq(kernels), lty = 1)
```

`deparse` *Expression Deparsing*

Description

Turn unevaluated expressions into character strings.

Usage

```
deparse(expr, width.cutoff = 60)
```

Arguments

`expr`	any R expression.
`width.cutoff`	integer in $[20, 500]$ determining the cutoff at which line-breaking is tried.

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`.

See Also

`substitute`, `parse`, `expression`.

Examples

```
deparse(args(lm))
deparse(args(lm), width = 100)
myplot <-
function(x, y)
    plot(x, y, xlab=deparse(substitute(x)),
        ylab=deparse(substitute(y)))
```

`Deprecated` *Deprecated Functions*

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

```
.Deprecated(new)
.Alias(expr)
reshapeWide(x, i = reshape.i,
             j = reshape.j,
             val = reshape.v,
             jnames = levels(j))

reshapeLong(x,jvars,  ilev = row.names(x),
             jlev = names(x)[jvars],
             iname = "reshape.i",
             jname = "reshape.j",
             vname = "reshape.v")
```

Details

`.Deprecated("<new name>")` is called from deprecated functions. The original help page for these functions is available at `help("oldName-deprecated")` (note the quotes).

`.Alias` provided an unreliable way to create duplicate references to the same object. There is no direct replacement. Where multiple references to a single object are required for semantic reasons consider using environments or external pointers. There are some notes on http://developer.r-project.org.

`reshape*`, which were experimental, are replace by `reshape`. This has a different syntax and allows multiple time-varying variables. See `reshapeLong-deprecated` and `reshapeWide-deprecated` for the earlier syntax.

See Also

`Defunct`, `.Alias-deprecated`,`reshapeWide-deprecated`, `reshapeLong-deprecated`

---

deriv *Symbolic and Algorithmic Derivatives of Simple Expressions*

Description

Compute derivatives of simple expressions, symbolically.

Usage

```
    D (expr, name)
 deriv(expr, namevec, function.arg, tag = ".expr", hessian = FALSE)
deriv3(expr, namevec, function.arg, tag = ".expr", hessian = TRUE)
```

Arguments

`expr`	`expression` or `call` to be differentiated.
`name,namevec`	character vector, giving the variable names (only one for `D()`) with respect to which derivatives will be computed.
`function.arg`	If specified, 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.

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`.

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 specified, `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.

See Also

`nlm` and `optim` for numeric minimization which could make use of derivatives, `nls` in package `nls`.

Examples

```
## formula argument :
dx2x <- deriv(~ x^2, "x") ; dx2x
expression({
         .value <- x^2
         .grad <- array(0, c(length(.value), 1), list(NULL, c("x")))
         .grad[, "x"] <- 2 * x
         attr(.value, "gradient") <- .grad
         .value
})
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)
stopifnot(eval(D.sc) ==
          attr(eval(dxy),"gradient")[,"x"])

## function returned:
deriv((ff <- y ~ sin(cos(x) * y)), c("x","y"), func = TRUE)
stopifnot(all.equal(deriv(ff, c("x","y"), func = TRUE ),
                    deriv(ff, c("x","y"), func = function(x,y){ } )))
## 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()" :

-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)
```

`det` *Calculate the Determinant of a Matrix*

Description

`det` calculates the determinant of a matrix either by QR decomposition or from the eigenvalues, see `qr` and `eigen`.

Usage

```
det(x, method = c("qr","eigenvalues"))
```

Arguments

`x`	numeric matrix.
`method`	`"qr"` (default) or `"eigenvalues"`

Note

Often, computing the determinant is *not* what you should be doing to solve a given problem.

The `"qr"` method is much faster for large matrices.

See Also

`eigen`, `qr`, `svd`

Examples

```
(x <- matrix(1:4, ncol=2))
det(x)
det(x, method="eigenvalues")

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 either a `data.frame` which has been `attach`ed or a package which was required previously.

Usage

```
detach(name, pos = 2)
```

Arguments

`name`	The object to detach. Defaults to `search()[pos]`. This can be a name or a character string but *not* a character vector.
`pos`	Index position in `search()` of database to detach. When `name` is `numeric`, `pos = name` is used.

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).

See Also

`attach`, `library`, `search`, `objects`.

Examples

```
require(eda)#package
detach(package:eda)
## could equally well use detach("package:eda")
## but NOT  pkg <- "package:eda"; detach(pkg)
## Instead, use
library(eda)
pkg <- "package:eda"
detach(pos = match(pkg, search()))

library(mva)
detach(2)# 'pos' used for 'name'
```

`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.

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

```
## 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
```

---

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`).

`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" then recording is turned off.

Usage

```
dev.copy(device, ..., which=dev.next())
dev.print(device=postscript, ...)
dev.copy2eps(...)
dev.control(displaylist)
```

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

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. This will not work unless options("printcmd") is set suitably and you have a PostScript printer: see postscript for how to set this up. Windows users may prefer to use dev.print(win.print).

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.

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

```
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()
```

`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 recent 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"`.

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 `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.

Note

Use of `bitmap` will leave a temporary file (with file name starting `Rbit`). Use of the `pdf`, `bmp`, `png` and `jpeg` devices is preferable to using these functions.

Author(s)

B. D. Ripley

See Also

`postscript`, `png` and `jpeg` and on Windows `bmp`.

`pdf` generate PDF directly.

`deviance` *Model Deviance*

Description

Returns the deviance of a fitted model object.

Usage

```
deviance(object, ...)
deviance.lm (object, ...)
deviance.glm(object, ...)
deviance.mlm(object, ...)
deviance.default(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.

There is no default method for this function.

Value

The value of the deviance extracted from the object `object`.

See Also

`df.residual`, `extractAIC`, `glm`, `lm`.

`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
- `windows` The graphics driver for Windows (on screen, to printer and to Windows metafile).
- `png` PNG bitmap device
- `jpeg` JPEG bitmap device
- `bmp` BMP bitmap device
- `xfig` Device for XFIG graphics file format
- `bitmap` bitmap pseudo-device via `GhostScript` (if available).

Usage

```
windows(...)
postscript(...)
pdf(...)
pictex(...)
win.metafile(...)
win.print(...)
png(...)
jpeg(...)
bmp(...)
xfig(...)
bitmap(...)

dev.interactive()
```

Value

`dev.interactive()` returns a logical, `TRUE` iff an interactive (screen) device is in use.

See Also

The individual help files for further information on any of the devices listed here;

`dev.cur`, `dev.print`, `graphics.off`, `image`, `dev2bitmap`.

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 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`.

diag *Matrix Diagonals*

Description

Extract or replace the diagonal of a matrix, or construct a diagonal matrix.

Usage

```
diag(x = 1, nrow, ncol)
diag(x) <- value
```

Arguments

`x`	a matrix, vector or 1D array.
`nrow, ncol`	Optional dimensions for the result.

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 assignment 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.

See Also

`matrix`.

Examples

```
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, ...)
diff.default(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 one for class `ts` objects. `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.

See Also

`diff.ts`.

Examples

```
diff(1:10, 2)
diff(1:10, 2, 2)
x <- cumsum(cumsum(1:10))
stopifnot(diff(x, lag = 2) == x[(1+2):10] - x[1:(10 - 2)],
          diff(x, lag = 2) == (3:10)^2,
          diff(diff(x))    == diff(x, differences = 2))
```

`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"))
round(x, digits = 0)
```

Arguments

`time1, time2`	date-time 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.
`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 (of either class) and returns an object of class `"difftime"` with an attribute indicating the units. There is a `round` method for objects of this class.

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 two date-time objects gives an object of this class, by calling `difftime` with `units="auto"`.

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)
```

`dim` *Dimensions of an Object*

Description

Retrieve or set the dimension of an object.

Usage

```
dim(x)
dim(x) <- values
```

Arguments

`x`	an R object, for example a matrix, array or data frame.

Details

The functions `dim` and `dim<-` are generic.

For an array (and hence in particular, for a matrix) they retrieve or set the `dim` attribute of the object. It is always `integer` or `NULL`.

`dim` has a method for `data.frame`s, which returns the length of the `row.names` attribute of `x` and the length of `x` (the numbers of "rows" and "columns").

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) <- nlist
```

Arguments

`x`	an R object, for example a matrix, array or data frame.
`nlist`	a list of the length `dim(x)` whose components are either null or character vectors the length of the appropriate dimension of `x`.

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. The list `nlist` can have names, and these will be used to label the dimensions of the array where appropriate.

Both have methods for data frames. The dimnames of a data frame are its `row.names` attribute and its `names`.

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]]
```

discoveries *Numbers of Important Discoveries*

Description

The numbers of "great" inventions and scientific discoveries in each year from 1860 to 1959.

Usage

```
data(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

```
data(discoveries)
plot(discoveries, ylab = "Number of important discoveries",
     las = 1)
title(main = "discoveries data set")
```

DLL.version *DLL Version Information*

Description

Return the version of the package and the version of R used to build the DLL, if available (usually only since R version 1.2.0).

Usage

```
DLL.version(path)
```

Arguments

path	character vector of length one giving the complete path to the DLL.

Value

If the DLL does not exist, NULL.

A character vector of two, giving the DLL version and the version of R used to build the DLL. If the information is not available, the corresponding string is empty.

Examples

```
DLL.version(file.path(R.home(), "bin/R.dll"))
DLL.version(file.path(R.home(), "library/lqs/libs/lqs.dll"))
```

---

do.call *Execute a Function Call*

Description

`do.call` executes a function call from the name of the function and a list of arguments to be passed to it.

Usage

```
do.call(what, args)
```

Arguments

`what`	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.

Value

The result of the (evaluated) function call.

See Also

`call` which creates an unevaluated call.

Examples

```
do.call("complex", list(imag = 1:3))
```

---

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",
         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.
`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 to be used.
`color`	the color(s) to be used for points an labels.
`gcolor`	the single color to be used for group labels and values.
`lcolor`	the color(s) to be used for the horizontal lines.
`main`	overall title for the plot.
`xlab`	title for the x axis.
`ylab`	title for the y axis.
`...`	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

Cleveland, W. S. (1985) *The Elements of Graphing Data.* Monterey, CA: Wadsworth.

Examples

```
data(VADeaths)
dotchart(VADeaths, main = "Death Rates in Virginia - 1940")
dotchart(t(VADeaths), main = "Death Rates in Virginia - 1940")
```

`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.

`is.double` returns `TRUE` or `FALSE` depending on whether its argument is of double type or not.

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.

See Also

`integer`.

Examples

```
is.double(1)
all(double(3) == 0)
```

`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. The default is to choose the first of these which `"internal"`. 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://"`.

`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 `CRAN.packages`.

The remaining details apply to method `"internal"` only.

The timeout for many parts of the transder 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, the full width of the bar is the known length. Otherwise the initial width represents 100Kbytes and is doubled whenever the current width is exceeded.

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"`.

Method `"wget"` can be used with proxy firewalls which require user/password authentication if proper values are stored in the configuration file for `wget`.

There is an alternative method if you have Internet Explorer 4 or later installed. You can use the flag `--internet2`, when the 'Internet Options' of the system are used to choose proxies and so on; these are set in the Control Panel and are those used for Internet Explorer. This version does not support `cacheOK = FALSE`.

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`, `CRAN.packages`, `download.packages` for applications

---

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 = "")
dget(file)
```

Arguments

`x`	an object.
`file`	either a character string naming a file or a connection. `""` indicates output to the console.

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.

Using `dget`, the object can be recreated (with the limitations mentioned above).

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")
```

---

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
```

`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, ...)
dummy.coef.lm(object, use.na = FALSE, ...)
dummy.coef.aovlist(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.

Author(s)

B.D. Ripley

See Also

`aov`, `model.tables`

Examples

```
options(contrasts=c("contr.helmert", "contr.poly"))
## From Venables and Ripley (1997) p.210.
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(1,0,0,1,0,1,1,0,0,1,0,1,0,1,1,0,0,0,0,1,1,1,0,1,0)
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)
dummy.coef(npk.aov)

npk.aovE <- aov(yield ~  N*P*K + Error(block), npk)
dummy.coef(npk.aovE)
```

`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 be `source`d into another R (or S) session.

Usage

```
dump(list, file="dumpdata.R", append=FALSE)
```

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`.

Details

At present the implementation of `dump` is very incomplete and it really only works for functions and simple vectors.

The function `save` is designed to be used for transporting R data between machines.

See Also

`dput`, `dget`,`write`.

Examples

```
x <- 1; y <- 1:10
dump(ls(patt='^[xyz]'), "xyz.Rdmped")
unlink("xyz.Rdmped")
```

duplicated *Determine Duplicate Elements*

Description

Determines which elements of a vector of 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)
```

Arguments

`x`	an atomic vector or a data fram.e
`incomparables`	a vector of values that cannot be compared. Currently, `FALSE` is the only possible value, meaning that all values can be compared.

Details

This is a generic function with methods for vectors and data frames. 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.

See Also

`unique`.

Examples

```
x <- c(9:20, 1:5, 3:7, 0:8)
## extract unique elements
(xu <- x[!duplicated(x)])
stopifnot(xu == unique(x), # but unique(x) is more efficient
          0:20 == sort(x[!duplicated(x)]))

data(iris)
stopifnot(duplicated(iris)[143] == TRUE)
```

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

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. It is ignored on Windows.
`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. It is ignored on Windows.
`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
`name`	a character string giving either the name of a C function or Fortran subroutine. Fortran names probably need to be given entirely in lower case (but this may be system-dependent).

Details

See 'See Also' and the *Writing R Extensions* manual for how to create 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 one should override them only if there is good reason and you understand the implications.

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`, `.Fortran` and `.External` can then be used to execute compiled C functions or Fortran subroutines contained in the library.

The function `dyn.unload` unlinks the shared library.

Functions `symbol.C` and `symbol.For` map function or subroutine names to the symbol name in the compiled code: `is.loaded` checks if the symbol name is loaded and hence available for use in `.C` or `.Fortran`.

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.

See Also

`library.dynam` to be used inside a package's `.First.lib` initialization.

`SHLIB` for how to create suitable DLLs.

`.C`, `.Fortran`, `.External`, `.Call`.

Examples

```
is.loaded(symbol.For("hcass2")) #-> probably  FALSE
library(mva)
is.loaded(symbol.For("hcass2")) #-> TRUE
```

---

edit *Invoke a Text Editor*

Description

Invoke a text editor on an R object.

Usage

```
edit(name = NULL, file = "", 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.
`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.

`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.

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

```
# 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")
```

`edit.data.frame` *Edit Data Frames and Matrices*

Description

Use data editor on data frame or matrix contents.

Usage

```
edit(name, factor.mode = c("character", "numeric"),
                edit.row.names = any(row.names(name) != 1:nrow(name)),
                ...)
edit(name, edit.row.names = any(rownames(name) != 1:nrow(name))), ...)
```

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 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.

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`.

Author(s)

Peter Dalgaard

See Also

`data.entry`, `edit`

Examples

```
data(InsectSprays)
edit(InsectSprays)
edit(InsectSprays, factor.mode="numeric")
```

`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 a `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 is the fraction of the maximum possible precision (inverse variance) obtainable by estimating in just that stratum.

This is used to pick strata in which to estimate terms in `model.tables.aovlist` and elsewhere.

Value

A matrix giving for each non-pure-error stratum (row) the efficiencies for each fixed-effect term in the model.

Author(s)

B.D. Ripley

See Also

`aov`, `model.tables.aovlist`, `se.contrast.aovlist`

Examples

```
## for balanced designs all efficiencies are zero or one.
## so as a statistically meaningless test:
options(contrasts=c("contr.helmert", "contr.poly"))
## From Venables and Ripley (1997) p.210.
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.aovE <- aov(yield ~  N*P*K + Error(block), npk)
eff.aovlist(npk.aovE)
```

`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 method for objects inheriting from class `"lm"`.

Usage

```
effects(object, ...)
effects.lm(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).

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.

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
```

`eigen` *Spectral Decomposition of a Matrix*

Description

Function `eigen` computes eigenvalues and eigenvectors by providing an interface to the EISPACK routines `RS`, `RG`, `CH` and `CG`.

Function `La.eigen` uses the LAPACK routines DSYEV/DSYEVR, DGEEV, ZHEEV and ZGEEV.

Usage

```
eigen(x, symmetric, only.values = FALSE)
La.eigen(x, symmetric, only.values = FALSE,
         method = c("dsyevr", "dsyev"))
```

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.
`method`	The LAPACK routine to use in the real symmetric case.

Details

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.

`La.eigen` is preferred to `eigen` 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.)

The LAPACK routine DSYEVR is usually substantially faster than DSYEV: see `http://www.cs.berkeley.edu/~demmel/DOE2000/Report0100.html`. Most benefits are seen with an optimized BLAS system.

Computing the eigenvectors is the slow part for large matrices.

Using `method="dsyevr"` requires IEEE 754 arithmetic. Should this not be supported on your platform, `method="dsyev"` is used, with a warning.

Value

The spectral decomposition of `x` is returned as components of a list.

`values`	a vector containing the p eigenvalues of `x`, sorted in *decreasing* order, according to `Mod(values)` if they are complex.

`vectors` a $p \times p$ matrix whose columns contain the eigenvectors of `x`, or `NULL` if `only.values` is `TRUE`.

For `eigen(, symmetric = FALSE)` the choice of length of the eigenvectors is not defined by LINPACK. 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

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.

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

Meps <- .Machine$double.eps
set.seed(123, kind="default")  # force a particular seed
m <- matrix(round(rnorm(25),3), 5,5)
sm <- m + t(m) #- symmetric matrix
em <- eigen(sm); V <- em$vect
print(lam <- em$values) # ordered DEcreasingly

stopifnot(
 abs(sm %*% V - V %*% diag(lam))          < 60*Meps,
 abs(sm       - V %*% diag(lam) %*% t(V)) < 60*Meps)

##------- Symmetric = FALSE:  -- different to above : ---

em <- eigen(sm, symmetric = FALSE); V2 <- em$vect
print(lam2 <- em$values) # ordered decreasingly in ABSolute value !
                         # and V2 is not normalized (where V is):
print(i <- rev(order(lam2)))
stopifnot(abs(lam - lam2[i]) < 60 * Meps)

zapsmall(Diag <- t(V2) %*% V2) # orthogonal, but not normalized
```

```
print(norm2V <- apply(V2 * V2, 2, sum))
stopifnot( abs(1- norm2V / diag(Diag)) < 60*Meps)

V2n <- sweep(V2,2, STATS= sqrt(norm2V), FUN="/")## V2n are now Normalized EV
apply(V2n * V2n, 2, sum)
##[1] 1 1 1 1 1

## Both are now TRUE:
stopifnot(abs(sm %*% V2n - V2n %*% diag(lam2))            < 60*Meps,
          abs(sm         - V2n %*% diag(lam2) %*% t(V2n)) < 60*Meps)

## Re-ordered as with symmetric:
sV <- V2n[,i]
slam <- lam2[i]
all(abs(sm %*% sV -  sV %*% diag(slam))             < 60*Meps)
all(abs(sm        -  sV %*% diag(slam) %*% t(sV)) < 60*Meps)
## sV  *is* now equal to V  -- up to sign (+-) and rounding errors
all(abs(c(1 - abs(sV / V)))       <     1000*Meps) # TRUE (P ~ 0.95)
```

`environment` *Environment Access*

Description

Get, set, test for and create environments.

Usage

```
environment(fun = NULL)
environment(fun) <- value
is.environment(obj)
.GlobalEnv
globalenv()
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 parent of the environment created.
`env`	an environment

Details

The global environment `.GlobalEnv` is the first item on the search path, more often known as the user's workspace. It can also be accessed by `globalenv()`.

The variable `.BaseNamespaceEnv` is part of some experimental support for name space management.

The assignment 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.

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 assignment 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 parent environment of its first argument.

See Also

The `envir` argument of `eval`.

Examples

```
##-- all three give the same:
environment()
environment(environment)
.GlobalEnv

ls(envir=environment(approxfun(1:2,1:2, method="const")))

is.environment(.GlobalEnv)# TRUE

e1 <- new.env(TRUE, NULL)
e2 <- new.env(FALSE, NULL)
assign("a", 3, env=e2)
parent.env(e1) <- e2
get("a", env=e1)
```

`esoph` *Smoking, Alcohol and (O)esophageal Cancer*

Description

Data from a case-control study of (o)esophageal cancer in Ile-et-Vilaine, France.

Usage

```
data(esoph)
```

Format

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

```
data(esoph)
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 + codes(tobgp)
                                       + codes(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` *Euro Conversion Rates*

Description

Conversion rates between the various Euro currencies.

Usage

```
data(euro)
```

Format

`euro` is a named vector of length 11, `euro.cross` a named matrix of size 11 by 11.

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

```
data(euro)
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"]

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 Cities in Europe*

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

```
data(eurodist)
```

Format

A `dist` object based on 21 objects. (You must have the mva package loaded to have the methods for this kind of object available).

Source

Crystal, D. Ed. (1990) *The Cambridge Encyclopaedia.* Cambridge: Cambridge University Press,

---

`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())
evalq(expr, envir, enclos)
eval.parent(expr, n = 1)
local(expr, envir = new.env())
```

Arguments

`expr`	object of mode `expression` or`call` or an "unevaluated expression".
`envir`	the `environment` in which `expr` is to be evaluated. May also be a list, a data frame, or an integer as in `sys.call`.
`enclos`	Relevant when `envir` is a list or a data frame. Specifies the enclosure, i.e., where R looks for objects not found in `envir`.
`n`	parent generations to go back

Details

`eval` evaluates the expression `expr` argument in the environment specified by `envir` and returns the computed value. If `envir` is not specified, then `sys.frame(sys.frame())`, the environment where the call to `eval` was made is used.

The `evalq` form is equivalent to `eval(quote(expr), ...)`.

As `eval` evaluates its first argument before passing it to the evaluator, it allows you to assign complicated expressions to symbols and then evaluate them. `evalq` avoids this.

`eval.parent(expr, n)` is a shorthand for `eval(expr, parent.frame(n))`.

`local` evaluates an expression in a local environment. It is equivalent to `evalq` except the 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 dataframes 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())`.

See Also

`expression`, `quote`, `sys.frame`, `parent.frame`, `environment`.

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

##
## 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))))
```

`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`.

Usage

```
example(topic, package = .packages(), lib.loc = NULL,
        echo = TRUE, verbose = getOption("verbose"),
        prompt.echo = paste(abbreviate(topic, 6),"> ", sep=""))
```

Arguments

`topic`	name or character: The online `help` topic the examples of which should be run.
`package`	a character vector with package names. 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.
`echo`	logical; if `TRUE`, show the R input when sourcing.
`verbose`	logical; if `TRUE`, show even more when running 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.

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(dbinom)
## force use of the standard package eda:
example("smooth", package="eda", lib.loc=.Library)
```

---

exists *Is an Object Defined?*

Description

Search for an R object of the given name on the search path.

Usage

```
exists(x, where = -1, envir = parent.frame(),
       frame = NULL, 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 in unquoted in an expression.
envir	an alternative way to specify an environment to look in, but it's 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 of object sought.
inherits	should the enclosing frames of the environment be inspected.

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. If `inherits` is `TRUE` and a value is not found for `x`, then the parent frames of the environment are searched until the name `x` is encountered. **Warning:** This is the default behaviour for R but not for S.

If `mode` is specified then only objects of that mode are sought. The function returns `TRUE` if the variable is encountered and `FALSE` if not.

The `mode` includes collections such as `"numeric"` and `"function"`: any member of the collection will suffice.

Value

Logical, true if and only if the object is found on the search path.

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(...)
```

Arguments

...	Vectors, factors or a list containing these.

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.

Author(s)

B.D. Ripley

Examples

```
expand.grid(height = seq(60, 80, 5), weight = seq(100, 300, 50),
            sex = c("Male","Female"))
```

---

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 e.g. `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

```
data(trees)
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)`.

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
r <- rexp(100)
all(abs(1 - dexp(1, r) / (r*exp(-r))) < 1e-14)
```

`expression` *Unevaluated Expressions*

Description

Creates or tests for objects of mode `"expression"`.

Usage

```
expression(...)

is.expression(x)
as.expression(x, ...)
```

Arguments

`...`	valid R expressions.
`x`	an arbitrary R object.

Value

`expression` returns a vector of mode `"expression"` containing its arguments as unevaluated "calls".

`is.expression` returns `TRUE` if `expr` is an expression object and `FALSE` otherwise.

`as.expression` attempts to coerce its argument into an expression object.

See Also

`call`, `eval`, `function`. Further, `text` and `legend` for plotting math 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 act on vectors, arrays, dataframes and lists to extract or replace subsets.

Usage

```
x[i]
x[i, j, ...]
x[i, j, ... , drop=TRUE]
x[[i]]
x[[i, j, ...]]
x$name
```

Arguments

`i,j,name`	elements to extract or replace
`drop`	For data frames, matrices, and arrays. If `TRUE` the result is coerced to the lowest possible dimension (see examples below).

Details

If one of these expressions appears on the left side of an assignment then that part of `x` is set to the value of the right hand side of the assignment.

These operators are generic. You can write methods to handle subsetting of specific classes of data.

The `[[` operator requires all relevant subscripts be supplied. With the `[` operator a comma separated blank indicates that all entries in that dimension are selected.

When `[.data.frame` is used for subsetting rows of a `data.frame`, it returns a dataframe with unique row names, using `make.names( * , unique = TRUE)`, see the `swiss` example below.

When operating on a list, the `[[` operator gives the specified element of the list while the `[` operator returns a list with the specified element(s) in it.

The operators `$` and `$<-` do not evaluate their second argument. It is translated to a string and that string is used to locate the correct component of the first argument.

See Also

`list`, `array`, `matrix`.

Examples

```
x <- 1:12; m <- matrix(1:6,nr=2); li <- list(pi=pi, e = exp(1))
x[10]                 # the tenth element of x
m[1,]                 # the first row of matrix m
m[1, , drop = FALSE]  # is a 1-row matrix
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

data(swiss)
swiss[ c(1, 1:2), ]   # duplicate row, unique row names
```

---

extractAIC *Extract AIC from a Fitted Model*

Description

Computes the (generalized) **A**kaike **I**nformation **C**riterion for a fitted parametric model.

Usage

```
extractAIC    (fit, scale,     k = 2, ...)
extractAIC.lm (fit, scale = 0, k = 2, ...)
extractAIC.glm(fit, scale = 0, k = 2, ...)
extractAIC.aov(fit, scale = 0, k = 2, ...)
extractAIC.coxph  (fit, scale, k = 2, ...)
extractAIC.negbin (fit, scale, k = 2, ...)
extractAIC.survreg(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`.
`k`	numeric specifying the "weight" of the *equivalent degrees of freedom* (≡`edf`) part in the AIC formula.
`...`	further arguments (currently unused in base R).

Details

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 parameters for usual parametric models) of `fit`.

For generalized linear models (i.e., for `lm`, `aov`, and `glm`), $-2 \log L$ is the *deviance*, as computed by `deviance(fit)`, plus a constant.

`k = 2` corresponds to the traditional AIC, using `k = log(n)` provides the BIC (Bayes IC) instead.

For further information, particularly about `scale`, see `step`.

Value

A numeric vector of length 2, giving

`edf`	the "**e**quivalent **d**egrees of **f**reedom" of the fitted model `fit`.
`AIC`	the (generalized) Akaike Information Criterion for `fit`.

Note

These functions are used in `add1`, `drop1` and `step` and that may be their main use.

Author(s)

B. D. Ripley

References

Venables, W. N. and Ripley, B. D. (1997) *Modern Applied Statistics with S-PLUS*. New York: Springer (2nd ed).

See Also

`deviance`, `add1`, `step`

Examples

```
example(glm)
extractAIC(glm.D93)#>>  5  15.129
```

`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.
`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)`. In R versions before 1.5, `min(integer(0)) == .Machine$integer.max`, and analogously for `max`, preserving argument *type*, whereas from R version 1.5.0, `max(x) == -Inf` and `min(x) == +Inf` whenever `length(x) == 0`.

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 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).

See Also

`range`.

Examples

```
min(5:1,pi)
pmin(5:1, pi)
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 names 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, levels = sort(unique(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)` 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>`.

Value

`factor` returns an object of class `"factor"` which has a set of numeric 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")`.

`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.

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.

See Also

`gl` for construction of "balanced" factors and `C` for factors with specified contrasts. `levels` and `nlevels` for accessing the levels, and `codes` to get integer codes.

Examples

```
ff <- factor(substring("statistics", 1:10, 1:10), levels=letters)
ff
codes(ff)
factor(ff)# drops the levels that do not occur
factor(factor(letters[7:10])[2:3]) # exercise indexing and reduction
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] <- T
x  # [1] 1    <NA> NA, <NA> used because NA is a level.
is.na(x)
# [1] FALSE  TRUE FALSE
```

`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.

Author(s)

B.D. Ripley

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"
```

`faithful` *Old Faithful Geyser Data*

Description

The 'faithful' data frame has 272 rows and 2 columns; the waiting time between eruptions and the duration of the eruption for the Old Faithful geyser in Yellowstone National Park, Wyoming, USA.

Usage

```
data(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

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

```
data(faithful)
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")
```

`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. The `binomial` family admits the links `"logit"`, `"probit"`, `"log"`, and `"cloglog"` (complementary log-log); the `Gamma` family the links `"identity"`, `"inverse"`, and `"log"`; the `poisson` family the links `"identity"`, `"log"`, and `"sqrt"`; the `quasi` family the links `"logit"`, `"probit"`, `"cloglog"`, `"identity"`, `"inverse"`, `"log"`, `"1/mu^2"` and `"sqrt"`. The function `power` can also be used to create a power link function for the `quasi` family. The other families have only one permissible link function: `"identity"` for the `gaussian` family, and `"1/mu^2"` for the `inverse.gaussian` family.
`variance`	for all families, other than `quasi`, the variance function is determined by the family. The `quasi` family will accept the specifications `"constant"`, `"mu(1-mu)"`, `"mu"`, `"mu^2"` and `"mu^3"` for the variance function.
`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

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-PLUS is closer to the quasi- variants.

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.

See Also

`glm`, `power`.

Examples

```
nf <- gaussian()# Normal family
nf
str(nf)# internal STRucture

gf <- Gamma()
gf
str(gf)
gf$linkinv
```

```
all(1:10 == gf$linkfun(gf$linkinv(1:10)))# is TRUE
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)

## 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"))
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, log = FALSE)
pf(q, df1, df2, ncp=0, lower.tail = TRUE, log.p = FALSE)
qf(p, df1, df2,        lower.tail = TRUE, log.p = FALSE)
rf(n, df1, df2)
```

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.
`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 $P[X \leq x]$, 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.

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 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

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)
all(fft(fft(x), inverse = TRUE)/(x*length(x)) == 1+0i)
eps <- 1e-11 ## In general, not exactly, but still:
for(N in 1:130) {
    cat("N=",formatC(N,wid=3),": ")
    x <- rnorm(N)
    if(N %% 5 == 0) {
        m5 <- matrix(x,ncol=5)
        cat("mvfft:",all(apply(m5,2,fft) == mvfft(m5)),"")
    }
    dd <- Mod(1 - (f2 <- fft(fft(x), inverse=TRUE)/(x*length(x))))
    cat(if(all(dd < eps))paste(" all < ", formatC(eps)) else
            paste("NO: range=",paste(formatC(range(dd)),collapse=",")),"\n")
}

plot(fft(c(9:0,0:13, numeric(301))), type = "l")
periodogram <- function(x, mean.x = mean(x)) { # simple periodogram
  n <- length(x)
  x <- unclass(x) - mean.x
  Mod(fft(x))[2:(n%/%2 + 1)]^2 / (2*pi*n) # drop I(0)
}
data(sunspots)
plot(10*log10(periodogram(sunspots)), type = "b", col = "blue")
```

`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.

Execute permission is ignored (always granted) under Windows, and all directories have both read and write permissions.

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**.

Author(s)

B. D. Ripley

See Also

`file.info`

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.

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.

File modes are probably only useful on Windows NT/2000 machines.

Value

A data frame with row names the file names and columns

size	integer: 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.

Entries for non-existent or non-readable files will be NA.

Note

This function will only be operational on systems with the `stat` system call, but that seems very widely available.

Author(s)

B. D. Ripley

See Also

`files`, `file.access`, `list.files`, and `DateTimeClasses` for the date formats.

Examples

```
ncol(finf <- file.info(dir()))# at least six
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 length one.

`file.show` *Display One or More Files*

Description

Display one or more files.

Usage

```
file.show(..., header, 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`. In particular, using an external pager on Windows does not, as there is no way to know when the external application has finished with the file.

Author(s)

Ross Ihaka, Brian Ripley.

See Also

`files`, `list.files`, `help`.

Examples

```
file.show(file.path(R.home(), "COPYRIGHTS"))
```

`files` *File 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)
dir.create(path)
basename(path)
dirname(path)
path.expand(path)
```

Arguments

`..., file1, file2, from, to, path`	character vectors, containing file names.
`overwrite`	logical; should the destination files be overwritten?

Details

The `...` arguments are concatenated to form one character string: you can specify the files separately or as one vector.

`file.create` creates files with the given names if they do not already exist and truncates them if they do. It returns a logical vector indicating the success or failure of the operation for each file.

`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. It returns a logical vector indicating whether or not it succeeded in removing each file.

`file.rename` attempts to rename a file. It returns a logical value indicating whether the operation succeeded. On Windows rename is not atomic, so it is possible that `to` will be deleted but `from` will not be renamed.

`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.

`dir.create` creates the last element of the path. It returns a logical, true for success.

`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. Tilde expansion is done: see the description for `path.expand` below.

In both `basename` and `dirname` trailing file separators are removed before dissecting the path, and for `dirname` any trailing file separators are removed from the result.

`path.expand` expands path(s) by replacing a leading tilde by the user's home directory (if defined on that platform).

Author(s)

Ross Ihaka, Brian Ripley

See Also

`file.info`, `file.access`, `file.path`, `file.show`, `list.files`, `unlink`.

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")
unlink("tmp", recursive=TRUE)
file.remove("A", "B", "C")

basename(file.path("","p1","p2","p3","filename"))
dirname(file.path("","p1","p2","p3","filename"))

path.expand("~/foo")
```

---

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,
               ...)
```

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.
`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 the main plot.
`plot.axes`	statements which draw axes 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, ...`	additional graphical parameters.

Note

This function currently uses the `layout` function and so is restricted to a full page display. In future it is likely to be replaced by a genuine `levelplot` function which will work in multipanel displays.

The ouput 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`.

Examples

```
data(volcano)
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) })
```

`fitted.values` *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, `lm` and `glm` 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`.

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 +/-`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))
```

`fix` *Fix an Object*

Description

`fix` invokes the editor specified in `options("editor")` on `x` and then assigns the new (edited) version of `x` in the user's workspace.

Usage

```
fix(x, ...)
```

Arguments

`x`	an R object.
`...`	arguments to pass to editor.

See Also

`edit`, `edit.data.frame`

Examples

```
## 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 ...
```

`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.
`...`	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]
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.

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.)

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 *`.

R functions can be invoked using `call_S` or `call_R` and can be passed lists or the simple types as arguments.

Header files for external code

Writing code for use with `.External` and `.Call` will 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.

Note

`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.

(Prior to R version 1.2.0 there has a third danger, that objects could be moved in memory by the garbage collector. The current garbage collector never moves objects.)

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.

See Also

`dyn.load`.

`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

```
data(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

```
data(Formaldehyde)
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)
```

`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) <- list
```

Arguments

`fun`	a function object or a character string naming the function to be manipulated. If not specified, the function calling `body` is used.
`list`	a list of R expressions.

Value

`formals` returns the formal argument list of the function specified.

The assignment form sets the formals of a function to the list on the right hand side.

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: `format.pval` is intended for formatting p-values.

Usage

```
format(x, ...)
format.AsIs(x, width = 12, ...)
format.data.frame(x, ..., justify = "none")
format.default(x, trim = FALSE, digits = getOption("digits"),
               nsmall = 0, justify = c("left", "right", "none"), ...)
format.factor(x, ...)
format.pval(pv, digits = max(1, getOption("digits") - 2),
            eps = .Machine$double.eps, na.form = "NA")
```

Arguments

`x`	any R object (conceptually); typically numeric.
`trim`	logical; if `TRUE`, leading blanks are trimmed off the strings.
`digits`	how many significant digits are to be used for `numeric` `x`. This is a suggestion: enough decimal places will be used so that the smallest (in magnitude) number has this many significant digits.
`nsmall`	number of digits which will always appear to the right of the decimal point in formatting real/complex numbers in non-scientific formats. Allowed values `0 <= nsmall <= 20`.
`justify`	should character vector be left-justified, right-justified or left alone. When justifying, the field width is that of the longest string.
`pv`	a numeric vector.
`eps`	a numerical tolerance: see Details.
`na.form`	character representation of `NA`s.
`width`	the returned vector has elements of at most `width`.
`...`	further arguments passed to or from other methods.

Details

These functions convert their first argument to a vector (or array) of character strings which have a common format (as is done by `print`), fulfilling `length(format*(x, *)) == length(x)`. The trimming with `trim = TRUE` is useful when the strings are to be used for plot `axis` annotation.

`format.AsIs` deals with columns of complicated objects that have been extracted from a data frame.

`format.pval` is mainly an auxiliary function for `print.summary.lm` etc., does separate formatting for fixed, floating point and very small values (those `< eps`).

The function `formatC` provides a rather more flexible formatting facility for numbers, but does *not* provide a common format for several numbers, nor it is platform-independent.

`format.data.frame` formats the data frame column by column, applying the appropriate method of `format` for each column.

Note

Currently `format` drops trailing zeroes, so `format(6.001, digits=2)` gives `"6"` and `format(c(6.0, 13.1), digits=2)` gives `c(" 6", "13")`.

Character(s) `"` in input strings `x` are escaped to `\"`.

See Also

`formatC`, `paste`, `as.character`.

Examples

```
format(1:10)

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)

## handling of quotes
zz <- data.frame(a=I("abc"), b=I("def\"gh"))
format(zz)

p <- c(47,13,2,.1,.023,.0045, 1e-100)/1000
format.pval(p)
format.pval(p / 0.9)
format.pval(p / 0.9, dig=3)
```

`format.info` *format(.) Information*

Description

Information is returned on how `format(x, digits = options("digits"))` would be formatted.

Usage

```
format.info(x)
```

Arguments

`x`	(numeric) vector; potential argument of `format(x,...)`.

Value

An `integer vector` of length 3, say `r`.

`r[1]`	width (number of characters) used for `format(x)`
`r[2]`	number of digits after decimal point.
`r[3]`	in `0:2`; if $\geq$`1`, *exponential* representation would be used, with exponent length of `r[3]+1`.

Note

The result **depends** on the value of `options("digits")`.

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))

# Reset old options:
options(dd)
```

---

formatC *Formatting Using C-style Formats*

Description

Formatting numbers individually and flexibly, using `C` style format specifications. `format.char` is a helper function for `formatC`.

Usage

```
formatC(x, digits = NULL, width = NULL,
        format = NULL, flag = "", mode = NULL)
format.char(x, width = NULL, flag = "-")
```

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 number of *significant* digits. Note that this can lead to quite long result strings, see examples below.
`flag`	format modifier as in Kernighan and Ritchie (1988, page 243). `"0"` pads leading zeros; `"-"` does left adjustment, others are `"+"`, `" "`, and `"#"`.

`mode` "double" (or "real"), "integer" or "character". Default: Determined from the storage mode of `x`.

Details

If you set `format` it over-rides 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`.

Value

A character object of same size and attributes as `x`. Unlike `format`, each number is formatted individually. Looping over each element of `x`, `sprintf(...)` is called (inside the C function `str_signif`).

`format.char(x)` and `formatC`, for character `x`, do simple (left or right) padding with white space.

Author(s)

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`.

Examples

```
xx  <- pi * 10^(-5:4)
dd <- options(digits = 4)   # only for format
cbind(format(xx), formatC(xx))
cbind(formatC(xx, wid = 9, flag = "-"))
cbind(formatC(xx, dig = 5, wid = 8, format = "f", flag = "0"))

format.char(c("a", "Abc", "no way"), wid = -7)  # <=> flag = "-"
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
options(dd) # reset
```

`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. Will be coerced to a character vector.
`y`	a vector of the same length as `x` with the corresponding descriptions. Will be coerced to a character vector.
`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

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 'eda' from its 'CONTENTS'
x <- read.dcf(file = system.file("CONTENTS", package = "eda"),
              fields = c("Entry", "Description"))
x <- as.data.frame(x)
writeLines(formatDL(x$Entry, x$Description))
## Same information in tagged description list style:
writeLines(formatDL(x$Entry, x$Description, style = "list"))
```

`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

```
y ~ model
formula(x, ...)
as.formula(object, env=parent.frame())
I(x)
```

Arguments

`x, object`	an 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 `0 + y~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`.

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 explicitly given.

See Also

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= "+"))))
```

`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://hotspur.psych.yorku.ca/ftp/sas/catdata/4fold.ps.gz

See Also

mosaicplot

Examples

```
data(UCBAdmissions)
## 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")
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()
```

See Also

`plot.window`, `plot.default`.

freeny *Freeny's Revenue Data*

Description

Freeny's data on quarterly revenue and explanatory variables.

Usage

```
data(freeny)
```

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.

Examples

```
data(freeny)
summary(freeny)
pairs(freeny, main = "freeny data")
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)
```

ftable *Flat Contingency Tables*

Description

Create "flat" contingency tables.

Usage

```
ftable(..., exclude = c(NA, NaN), row.vars = NULL, col.vars = NULL)
```

Arguments

`...`	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.

Function `ftable.formula` provides a formula method for creating flat contingency tables.

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.
data(Titanic)
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.
data(mtcars)
x <- ftable(mtcars[c("cyl", "vs", "am", "gear")])
x
ftable(x, row.vars = c(2, 4))
```

`ftable.formula` *Formula Notation for Flat Contingency Tables*

Description

Produce or manipulate a flat contingency table using formula notation.

Usage

```
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 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 `NA`s. 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

```
data(Titanic)
Titanic
x <- ftable(Survived ~ ., data = Titanic)
x
ftable(Sex ~ Class + Age, data = x)
```

`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, or a series of expressions separated by commas.

Details

In R (unlike S) the names in an argument list cannot be quoted non-standard names.

If `value` is a series of expressions, the value returned is a list of the evaluated expressions, with names set to the expressions where these are the names of R objects.

See Also

`args` and `body` for accessing the arguments and body of a function.

`debug` for debugging; `invisible` for `return(.)`ing *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)
```

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.
`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 `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 $x > 0$, $\alpha > 0$ and $\sigma > 0$. The mean and variance are $E(X) = \alpha\sigma$ and $Var(X) = \alpha\sigma^2$.

Value

`dgamma` gives the density, `pgamma` gives the distribution function `qgamma` gives the quantile function, and `rgamma` generates random deviates.

Note

The S parametrization is via `shape` and `rate`: S has no `scale` parameter. Prior to 1.4.0 R only had `scale`.

The cumulative hazard $H(t) = -\log(1 - F(t))$ is `-pgamma(t, ..., lower = FALSE, log = TRUE)`.

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))
```

---

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"))
gcinfo(verbose)
```

Arguments

verbose	logical; if TRUE, the garbage collection prints statistics about cons cells and the vector heap.

Details

A call of gc causes a garbage collection to take place. This takes 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.

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).

gcinfo returns the previous value of the flag.

See Also

Memory on R's memory management and gctorture if you are an R hacker.

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.time *Report Time Spent in Garbage Collection*

Description

This function reports the time spent in garbage collection so far in the R session.

Usage

```
gc.time()
```

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.

Note

CPU times will be returned as `NA` on Windows 9x/ME systems, but are genuine times on NT4 and 2000 systems. Times of child processes are not available and will always be given as `NA`.

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

`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.
`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 $x = 0, 1, 2, \ldots$

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.

See Also

`dnbinom` for the negative binomial which generalizes the geometric distribution.

Examples

```
pp <- sort(c((1:9)/10, 1 - .2^(2:8)))
print(qg <- qgeom(pp, prob = .2))
## test that qgeom is an inverse of pgeom
print(qg1 <- qgeom(pgeom(qg, prob=.2), prob =.2))
all(qg == qg1)
Ni <- rgeom(20, prob = 1/4); table(factor(Ni, 0:max(Ni)))
```

`get` *Return a Variable's Value*

Description

Search for an R object with a given name and return it if found.

Usage

```
get(x, pos=-1, envir=as.environment(pos), mode="any", inherits=TRUE)
```

Arguments

`x`	a variable name (given as a quoted 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 in unquoted in an expression.
`envir`	an alternative way to specify an environment to look in; see the details section.
`mode`	the mode of object sought.
`inherits`	should the enclosing frames of the environment be inspected?

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.

The `mode` includes collections such as `"numeric"` and `"function"`: any member of the collection will suffice.

Value

This function searches the specified environment for a bound variable whose name is given by the character string `x`. If the variable's value is not of the correct `mode`, it is ignored.

If `inherits` is `FALSE`, only the first frame of the specified environment is inspected. If `inherits` is `TRUE`, the search is continued up through the parent frames until a bound value of the right mode is found.

Using a `NULL` environment is equivalent to using the current environment.

See Also

`exists`.

Examples

```
get("%o%")
```

`getNativeSymbolInfo` *Obtain a description of a native (C/Fortran) symbol*

Description

This finds and returns as comprehensive a description of a dynamically loaded or "exported" built-in native symbol. 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.

Usage

```
getNativeSymbolInfo(name, PACKAGE)
```

Arguments

`name`	the name of the native symbol as used in a call to `is.loaded`, etc.
`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.

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

If the symbol is not found, an error is raised. Otherwise, the value is a list containing the following elements:

`name`	the name of the symbol, as given by the `name` argument.
`address`	the native memory address of the symbol which can be used to invoke the routine, and also compare with other symbol address. This is an external pointer object and of class `NativeSymbol`.
`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.

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 as directly 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

`is.loaded`, `.C`, `.Fortran`, `.External`, `.Call`, `dyn.load`.

Examples

```
getNativeSymbolInfo("dansari")

library(mva)
getNativeSymbolInfo(symbol.For("hcass2"))
```

`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 sis 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()

 old <- setCConverterStatus(1,FALSE)

 setCConverterStatus(1,old)


 removeCConverter(1)
 removeCConverter(getCConverterDescriptions()[1])
```

---

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.

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)
```

`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 = NULL, subset = NULL,
    na.action, start = NULL, offset = NULL,
    control = glm.control(...), model = TRUE, method = "glm.fit",
    x = FALSE, y = TRUE, contrasts = NULL, ...)

glm.fit(x, y, weights = rep(1, nrow(x)),
        start = NULL, etastart  =  NULL, mustart = NULL,
        offset = rep(0, nrow(x)),
        family = gaussian(), control = glm.control(),
        intercept = TRUE)
glm.fit.null(x, y, weights = rep(1, nrow(x)),
        start = NULL, etastart  =  NULL, mustart = NULL,
        offset = rep(0, nrow(x)),
        family = gaussian(), control = glm.control(),
        intercept = FALSE)
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. See `family` for details.
`data`	an optional data frame containing the variables in the model. By default 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.
`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`.
`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.

`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 (and presently only) method `glm.fit` uses iteratively reweighted least squares (IWLS).
`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?
`...`	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` models 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.

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` and `glm.fit.null` are the workhorse functions: the former calls the latter for a null model (with no intercept).

Value

`glm` returns an object of class `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.

`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

`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, in R 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-null 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.

References

McCullagh P. and Nelder, J. A. (1989) *Generalized Linear Models.* London: Chapman and Hall.

Dobson, A. J. (1990) *An Introduction to Generalized Linear Models.* London: Chapman and Hall.

Venables, W. N. and Ripley, B. D. (1999) *Modern Applied Statistics with S-PLUS.* 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 (1999, pp.217-8)


## Need the anorexia data from a recent version of the package MASS:
library(MASS)
data(anorexia)

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))
```

`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-04, maxit=10, trace=FALSE)
```

Arguments

`epsilon`	positive convergence tolerance *epsilon*; the iterations converge when $\lvert dev - devold\rvert/(\lvert dev\rvert + 0.1) < epsilon$.
`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.

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-15)
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

```
coefficients(x) ; coef(x)
family(object, ...)
fitted.values(x) ; fitted(x)
residuals(object, type = c("deviance", "pearson", "working",
                           "response", "partial"), ...)
```

Arguments

`object, x`	an object of class `glm`, typically the result of a call to `glm`.
`test`	a character string, matching one of `"Chisq"`, `"F"` or `"Cp"`. See `stat.anova`.
`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.

See Also

`glm` for computing `glm.obj`, `anova.glm`; the corresponding *generic* functions, `summary.glm`, `coefficients`, `deviance`, `df.residual`, `effects`, `fitted.values`, `residuals`.

`glm.summary` *Summarizing Generalized Linear Model Fits*

Description

These functions are all `methods` for class `glm` or `summary.glm` objects.

Usage

```
summary(object, dispersion = NULL, correlation = FALSE, ...)

print(x, digits = max(3, getOption("digits") - 3),
             na.print = "", symbolic.cor = p > 4,
             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 fitting family. By default it is obtained from `object`.
`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.
`na.print`	Unused.
`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`.

See Also

glm, summary.

Examples

```
## --- Continuing the Example from  ‘‘?glm’’:

summary(glm.D93)
```

`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, rgb.

Examples

```
gray(0:8 / 8)
```

`grep` *Pattern Matching and Replacement*

Description

`grep` searches for matches to `pattern` (its first argument) within the character vector `x` (second argument). `regexpr` does too, but returns 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, value=FALSE)
sub(pattern, replacement, x,
        ignore.case=FALSE, extended=TRUE)
gsub(pattern, replacement, x,
        ignore.case=FALSE, extended=TRUE)
regexpr(pattern, text,  extended=TRUE)
```

Arguments

`pattern`	character string containing a regular expression to be matched in the vector of character string `vec`.
`x, text`	a character vector where matches are sought.
`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.
`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.
`replacement`	a replacement for matched pattern in `sub` and `gsub`.

Details

The two `*sub` functions differ only in that `sub` replaces only the first occurrence of a `pattern` whereas `gsub` replaces all occurrences.

The regular expressions used are those specified by POSIX 1003.2, either extended or basic, depending on the value of the `extended` argument.

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.

For `sub` and `gsub` a character vector of the same length as the original.

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).

See Also

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) )
(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)
```

---

grid *Add Grid to a Plot*

Description

grid adds an nx by ny rectangular grid to an existing plot, using lines of type lty and color col.

If more fine tuning is required, use abline(h = ., v = .) directly.

Usage

```
grid(nx = NULL, ny = NULL, col = "lightgray", lty = "dotted")
```

Arguments

nx,ny	number of cells of the grid in x and y direction. Defaults to the number of tick marks on the corresponding axis.
col	character or (integer) numeric; color of the grid lines.
lty	character or (integer) numeric; line type of the grid lines.

See Also

plot, abline, lines, points.

Examples

```
data(iris)
## maybe change the desired number of tick marks:  par(lab=c(mx,my,7))
plot(iris$Sepal.L, iris$Sepal.W, col = rep(1:3, rep(50, 3)),
     xlim = c(4, 8), ylim = c(2, 4.5), panel.first = grid())
```

HairEyeColor *Hair and Eye Color of Statistics Students*

Description

Distribution of hair and eye color and sex in 592 statistics students.

Usage

```
data(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://hotspur.psych.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://hotspur.psych.yorku.ca/SCS/Papers/asa92.html

See Also

chisq.test, loglin, mosaicplot

Examples

```
data(HairEyeColor)
## Full mosaic
mosaicplot(HairEyeColor)
## Aggregate over sex:
x <- apply(HairEyeColor, c(1, 2), sum)
x
mosaicplot(x, main = "Relation between hair and eye color")
```

`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 = .packages(),
     lib.loc = NULL, verbose = getOption("verbose"),
     try.all.packages = getOption("help.try.all.packages"),
     chmhelp = getOption("chmhelp"),
     htmlhelp = getOption("htmlhelp"), winhelp = getOption("winhelp"),
     pager = getOption("pager"))
?topic
type?topic
```

Arguments

`topic`	a name or character string on which documentation is sought (but *not* a variable containing a character string!).
`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. 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 `Notes`.
`chmhelp`	logical (or `NULL`). If `TRUE` the Compiled HTML version of the help will be shown in a help viewer.
`htmlhelp`	logical (or `NULL`). If `TRUE`, the HTML version of the help will be shown in a browser.
`winhelp`	logical (or `NULL`). If `TRUE`, a Windows `.hlp` file will be used if one is available.
`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.

Details

In the case of unary and binary operators and control-flow special forms, the name may need to be quoted.

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). You need to customize the file 'R_HOME/bin/helpPRINT.bat' which contains an example. The appearance of the output can be customized through a file 'Rhelp.cfg' somewhere in your LaTeX search path.

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'.

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 = stepfun) # get help even when package is not loaded

data()                  # list all available data sets
?women                  # information about data set "women"

topi <- "women"
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'
```

`help.search` *Search the Help System*

Description

Allows for searching the help system for documentation matching a given regular expression in the (file) name, alias, title, or keyword entries (or any combination thereof). Topics and titles of the matched help entries are nicely displayed.

Usage

```
help.search(pattern, fields = c("alias", "title"),
            apropos, keyword, whatis, ignore.case = TRUE,
            package = NULL, lib.loc = NULL,
            help.db = getOption("help.db"),
            verbose = getOption("verbose"),
            rebuild = FALSE)
```

Arguments

`pattern`	a character string containing a regular expression 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"`, `"alias"`, `"title"`, and `"keyword"`, corresponding to the help page's (file) name, the topics it provides documentation for, its title, and the keywords it can be classified to.
`apropos`	a character string containing a regular expression to be matched in the help page topics and title.
`keyword`	a character string containing a regular expression to be matched in the help page keywords.
`whatis`	a character string containing a regular expression 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 specified library trees `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.

Details

Upon installation of a package, the Perl script 'Rd2contents.pl' creates a 'CONTENTS' data base which contains the information on name, aliases, title and keywords (as well as the URL of the HTML version of the help file) in Debian Control Format. This is the data base searched by `help.search()`.

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 to the file 'help.db' in the '.R' subdirectory of the user's home directory or the current working directory.

Note that currently, the aliases in the matching help files are not displayed.

Value

The results are returned in an object of class `"hsearch"`, which has a print method for nicely displaying the results of the query. This mechanism is experimental, and may change in future versions of R.

See Also

`help`; `help.start` for starting the hypertext (currently HTML) version of R's online documentation, which offers a similar search mechanism.

`apropos` uses regexps and has nice examples.

Examples

```
help.search("linear models")    # In case you forgot how to fit linear
                                # models

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.

## Help pages with documented topics starting with 'try'.
help.search("\\btry", fields = "alias")
## Do not use '^' or '$' when matching aliases or keywords.
```

`help.start` *Hypertext Documentation*

Description

Start the hypertext (currently HTML) version of R's online documentation.

Usage

```
help.start(update = TRUE, gui = "irrelevant", browser = "irrelevant")
```

Arguments

`gui`	just for compatibility with S-PLUS.
`browser`	the name of the program to be used as hypertext browser.
`update`	logical: should this attempt to update the indices to reflect the currently installed packages.

Details

The Windows file association mechanism is used to send the HTML file to a browser, launching one if necessary.

Unlike Unix systems, running `help.start` does not send all future help requests to the browser: use `options(htmlhelp=TRUE)` to set that.

See Also

`help()` for on- and off-line help in ASCII/Editor or PostScript format.

Examples

```
help.start()
```

Hershey *Hershey Vector Fonts in R*

Description

If the `vfont` argument to one of the text-drawing functions (`text`, `mtext`, `title`, `axis`, and `contour`) is a character vector of length 2, hershey vector fonts are used to render the 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.
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.

Drawback:
You cannot use mathematical expressions (`plotmath`) with Hershey fonts.

Usage

```
Hershey
```

Details

The Hershey characters are organised into a set of fonts, which are 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 `example(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.

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 `example(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 (for example, if you only have a US keyboard). 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 `example(Hershey)` shows all of the ISO Latin-1 escape sequences.

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 `example(Hershey)` shows all of the special character escape sequences.

Cyrillic Characters: cyrillic characters are implemented according to the K018-R encoding. On a US keyboard, these can be produced using the Serif typeface and Cyrillic (or Oblique Cyrillic) fontindex and specifying 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 `example(Hershey)` shows the octal codes for the available cyrillic characters.

Japanese Characters: 83 Hiragana, 86 Katakana, and 603 Kanji characters are implemented according to the EUC (Extended Unix Code) encoding. Each character is idenitified 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`.

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 `example(Hershey)` shows all of the available raw glyphs.

References

http://www.gnu.org/software/plotutils/plotutils.html

See Also

text, contour, Japanese

Examples

```
str(Hershey)

######
# create tables of vector font functionality
######
make.table <- function(nr, nc) {
    savepar <- par(mar=rep(0, 4), pty="s")
    plot(c(0, nc*2 + 1), c(0, -(nr + 1)),
```

```
          type="n", xlab="", ylab="", axes=FALSE)
    savepar
}

get.r <- function(i, nr)     i %% nr + 1
get.c <- function(i, nr)     i %/% nr + 1

draw.title <- function(title, i = 0, nr, nc) {
    r <- get.r(i, nr)
    c <- get.c(i, nr)
    text((nc*2 + 1)/2, 0, title, font=2)
}

draw.sample.cell <- function(typeface, fontindex, string, i, nr) {
    r <- get.r(i, nr)
    c <- get.c(i, nr)
    text(2*(c - 1) + 1, -r, paste(typeface, fontindex))
    text(2*c, -r, string, vfont=c(typeface, fontindex), cex=1.5)
    rect(2*(c - 1) + .5, -(r - .5), 2*c + .5, -(r + .5), border="grey")
}

draw.vf.cell <- function(typeface, fontindex, string, i, nr, raw.string=NULL) {
    r <- get.r(i, nr)
    c <- get.c(i, nr)
    if (is.null(raw.string))
        raw.string <- paste("\\", string, sep="")
    text(2*(c - 1) + 1, -r, raw.string, col="grey")
    text(2*c, -r, string, vfont=c(typeface, fontindex))
    rect(2*(c - 1) + .5, -(r - .5), (2*c + .5), -(r + .5), border="grey")
}

nr <- 23
nc <- 1
oldpar <- make.table(nr, nc)
i <- 0
draw.title("Sample 'a' for each available font", i, nr, nc)
draw.sample.cell("serif", "plain", "a", i, nr); i <- i + 1
draw.sample.cell("serif", "italic", "a", i, nr); i <- i + 1
draw.sample.cell("serif", "bold", "a", i, nr); i <- i + 1
draw.sample.cell("serif", "bold italic", "a", i, nr); i <- i + 1
draw.sample.cell("serif", "cyrillic", "a", i, nr); i <- i + 1
draw.sample.cell("serif", "oblique cyrillic", "a", i, nr); i <- i + 1
draw.sample.cell("serif", "EUC", "a", i, nr); i <- i + 1
draw.sample.cell("sans serif", "plain", "a", i, nr); i <- i + 1
draw.sample.cell("sans serif", "italic", "a", i, nr); i <- i + 1
draw.sample.cell("sans serif", "bold", "a", i, nr); i <- i + 1
draw.sample.cell("sans serif", "bold italic", "a", i, nr); i <- i + 1
draw.sample.cell("script", "plain", "a", i, nr); i <- i + 1
draw.sample.cell("script", "italic", "a", i, nr); i <- i + 1
draw.sample.cell("script", "bold", "a", i, nr); i <- i + 1
draw.sample.cell("gothic english", "plain", "a", i, nr); i <- i + 1
draw.sample.cell("gothic german", "plain", "a", i, nr); i <- i + 1
draw.sample.cell("gothic italian", "plain", "a", i, nr); i <- i + 1
draw.sample.cell("serif symbol", "plain", "a", i, nr); i <- i + 1
draw.sample.cell("serif symbol", "italic", "a", i, nr); i <- i + 1
draw.sample.cell("serif symbol", "bold", "a", i, nr); i <- i + 1
draw.sample.cell("serif symbol", "bold italic", "a", i, nr); i <- i + 1
```

```
draw.sample.cell("sans serif symbol", "plain", "a", i, nr); i <- i + 1
draw.sample.cell("sans serif symbol", "italic", "a", i, nr); i <- i + 1

nr <- 25
nc <- 6
tf <- "serif"
fi <- "plain"
make.table(nr, nc)
i <- 0
draw.title("Symbol (incl. Greek) Escape Sequences", i, nr, nc)
## Greek alphabet in order
draw.vf.cell(tf, fi, "\\*A", i, nr); i<-i+1; { "Alpha"}
draw.vf.cell(tf, fi, "\\*B", i, nr); i<-i+1; { "Beta"}
draw.vf.cell(tf, fi, "\\*G", i, nr); i<-i+1; { "Gamma"}
draw.vf.cell(tf, fi, "\\*D", i, nr); i<-i+1; { "Delta"}
draw.vf.cell(tf, fi, "\\*E", i, nr); i<-i+1; { "Epsilon"}
draw.vf.cell(tf, fi, "\\*Z", i, nr); i<-i+1; { "Zeta"}
draw.vf.cell(tf, fi, "\\*Y", i, nr); i<-i+1; { "Eta"}
draw.vf.cell(tf, fi, "\\*H", i, nr); i<-i+1; { "Theta"}
draw.vf.cell(tf, fi, "\\*I", i, nr); i<-i+1; { "Iota"}
draw.vf.cell(tf, fi, "\\*K", i, nr); i<-i+1; { "Kappa"}
draw.vf.cell(tf, fi, "\\*L", i, nr); i<-i+1; { "Lambda"}
draw.vf.cell(tf, fi, "\\*M", i, nr); i<-i+1; { "Mu"}
draw.vf.cell(tf, fi, "\\*N", i, nr); i<-i+1; { "Nu"}
draw.vf.cell(tf, fi, "\\*C", i, nr); i<-i+1; { "Xi"}
draw.vf.cell(tf, fi, "\\*O", i, nr); i<-i+1; { "Omicron"}
draw.vf.cell(tf, fi, "\\*P", i, nr); i<-i+1; { "Pi"}
draw.vf.cell(tf, fi, "\\*R", i, nr); i<-i+1; { "Rho"}
draw.vf.cell(tf, fi, "\\*S", i, nr); i<-i+1; { "Sigma"}
draw.vf.cell(tf, fi, "\\*T", i, nr); i<-i+1; { "Tau"}
draw.vf.cell(tf, fi, "\\*U", i, nr); i<-i+1; { "Upsilon"}
draw.vf.cell(tf, fi, "\\+U", i, nr); i<-i+1; { "Upsilon1"}
draw.vf.cell(tf, fi, "\\*F", i, nr); i<-i+1; { "Phi"}
draw.vf.cell(tf, fi, "\\*X", i, nr); i<-i+1; { "Chi"}
draw.vf.cell(tf, fi, "\\*Q", i, nr); i<-i+1; { "Psi"}
draw.vf.cell(tf, fi, "\\*W", i, nr); i<-i+1; { "Omega"}
#
draw.vf.cell(tf, fi, "\\*a", i, nr); i<-i+1; { "alpha"}
draw.vf.cell(tf, fi, "\\*b", i, nr); i<-i+1; { "beta"}
draw.vf.cell(tf, fi, "\\*g", i, nr); i<-i+1; { "gamma"}
draw.vf.cell(tf, fi, "\\*d", i, nr); i<-i+1; { "delta"}
draw.vf.cell(tf, fi, "\\*e", i, nr); i<-i+1; { "epsilon"}
draw.vf.cell(tf, fi, "\\*z", i, nr); i<-i+1; { "zeta"}
draw.vf.cell(tf, fi, "\\*y", i, nr); i<-i+1; { "eta"}
draw.vf.cell(tf, fi, "\\*h", i, nr); i<-i+1; { "theta"}
draw.vf.cell(tf, fi, "\\+h", i, nr); i<-i+1; { "theta1"}
draw.vf.cell(tf, fi, "\\*i", i, nr); i<-i+1; { "iota"}
draw.vf.cell(tf, fi, "\\*k", i, nr); i<-i+1; { "kappa"}
draw.vf.cell(tf, fi, "\\*l", i, nr); i<-i+1; { "lambda"}
draw.vf.cell(tf, fi, "\\*m", i, nr); i<-i+1; { "mu"}
draw.vf.cell(tf, fi, "\\*n", i, nr); i<-i+1; { "nu"}
draw.vf.cell(tf, fi, "\\*c", i, nr); i<-i+1; { "xi"}
draw.vf.cell(tf, fi, "\\*o", i, nr); i<-i+1; { "omicron"}
draw.vf.cell(tf, fi, "\\*p", i, nr); i<-i+1; { "pi"}
draw.vf.cell(tf, fi, "\\*r", i, nr); i<-i+1; { "rho"}
draw.vf.cell(tf, fi, "\\*s", i, nr); i<-i+1; { "sigma"}
draw.vf.cell(tf, fi, "\\ts", i, nr); i<-i+1; { "sigma1"}
```

```
draw.vf.cell(tf, fi, "\\*t", i, nr); i<-i+1; { "tau"}
draw.vf.cell(tf, fi, "\\*u", i, nr); i<-i+1; { "upsilon"}
draw.vf.cell(tf, fi, "\\*f", i, nr); i<-i+1; { "phi"}
draw.vf.cell(tf, fi, "\\+f", i, nr); i<-i+1; { "phi1"}
draw.vf.cell(tf, fi, "\\*x", i, nr); i<-i+1; { "chi"}
draw.vf.cell(tf, fi, "\\*q", i, nr); i<-i+1; { "psi"}
draw.vf.cell(tf, fi, "\\*w", i, nr); i<-i+1; { "omega"}
draw.vf.cell(tf, fi, "\\+p", i, nr); i<-i+1; { "omega1"}
#
draw.vf.cell(tf, fi, "\\fa", i, nr); i<-i+1; { "universal"}
draw.vf.cell(tf, fi, "\\te", i, nr); i<-i+1; { "existential"}
draw.vf.cell(tf, fi, "\\st", i, nr); i<-i+1; { "suchthat"}
draw.vf.cell(tf, fi, "\\**", i, nr); i<-i+1; { "asteriskmath"}
draw.vf.cell(tf, fi, "\\=~", i, nr); i<-i+1; { "congruent"}
draw.vf.cell(tf, fi, "\\tf", i, nr); i<-i+1; { "therefore"}
draw.vf.cell(tf, fi, "\\pp", i, nr); i<-i+1; { "perpendicular"}
draw.vf.cell(tf, fi, "\\ul", i, nr); i<-i+1; { "underline"}
draw.vf.cell(tf, fi, "\\rx", i, nr); i<-i+1; { "radicalex"}

draw.vf.cell(tf, fi, "\\ap", i, nr); i<-i+1; { "similar"}
draw.vf.cell(tf, fi, "\\fm", i, nr); i<-i+1; { "minute"}
draw.vf.cell(tf, fi, "\\<=", i, nr); i<-i+1; { "lessequal"}
draw.vf.cell(tf, fi, "\\f/", i, nr); i<-i+1; { "fraction"}
draw.vf.cell(tf, fi, "\\if", i, nr); i<-i+1; { "infinity"}
draw.vf.cell(tf, fi, "\\Fn", i, nr); i<-i+1; { "florin"}
draw.vf.cell(tf, fi, "\\CL", i, nr); i<-i+1; { "club"}
draw.vf.cell(tf, fi, "\\DI", i, nr); i<-i+1; { "diamond"}
draw.vf.cell(tf, fi, "\\HE", i, nr); i<-i+1; { "heart"}
draw.vf.cell(tf, fi, "\\SP", i, nr); i<-i+1; { "spade"}
draw.vf.cell(tf, fi, "\\<>", i, nr); i<-i+1; { "arrowboth"}
draw.vf.cell(tf, fi, "\\<-", i, nr); i<-i+1; { "arrowleft"}
draw.vf.cell(tf, fi, "\\ua", i, nr); i<-i+1; { "arrowup"}
draw.vf.cell(tf, fi, "\\->", i, nr); i<-i+1; { "arrowright"}
draw.vf.cell(tf, fi, "\\da", i, nr); i<-i+1; { "arrowdown"}
draw.vf.cell(tf, fi, "\\de", i, nr); i<-i+1; { "degree"}
draw.vf.cell(tf, fi, "\\+-", i, nr); i<-i+1; { "plusminus"}
draw.vf.cell(tf, fi, "\\sd", i, nr); i<-i+1; { "second"}
draw.vf.cell(tf, fi, "\\>=", i, nr); i<-i+1; { "greaterequal"}
draw.vf.cell(tf, fi, "\\mu", i, nr); i<-i+1; { "multiply"}
draw.vf.cell(tf, fi, "\\pt", i, nr); i<-i+1; { "proportional"}
draw.vf.cell(tf, fi, "\\pd", i, nr); i<-i+1; { "partialdiff"}
draw.vf.cell(tf, fi, "\\bu", i, nr); i<-i+1; { "bullet"}
draw.vf.cell(tf, fi, "\\di", i, nr); i<-i+1; { "divide"}
draw.vf.cell(tf, fi, "\\!=", i, nr); i<-i+1; { "notequal"}
draw.vf.cell(tf, fi, "\\==", i, nr); i<-i+1; { "equivalence"}
draw.vf.cell(tf, fi, "\\~~", i, nr); i<-i+1; { "approxequal"}
draw.vf.cell(tf, fi, "\\..", i, nr); i<-i+1; { "ellipsis"}
draw.vf.cell(tf, fi, "\\an", i, nr); i<-i+1; { "arrowhorizex"}
draw.vf.cell(tf, fi, "\\CR", i, nr); i<-i+1; { "carriagereturn"}
draw.vf.cell(tf, fi, "\\Ah", i, nr); i<-i+1; { "aleph"}
draw.vf.cell(tf, fi, "\\Im", i, nr); i<-i+1; { "Ifraktur"}
draw.vf.cell(tf, fi, "\\Re", i, nr); i<-i+1; { "Rfraktur"}
draw.vf.cell(tf, fi, "\\wp", i, nr); i<-i+1; { "weierstrass"}
draw.vf.cell(tf, fi, "\\c*", i, nr); i<-i+1; { "circlemultiply"}
draw.vf.cell(tf, fi, "\\c+", i, nr); i<-i+1; { "circleplus"}
draw.vf.cell(tf, fi, "\\es", i, nr); i<-i+1; { "emptyset"}
draw.vf.cell(tf, fi, "\\ca", i, nr); i<-i+1; { "cap"}
```

```
draw.vf.cell(tf, fi, "\\cu", i, nr); i<-i+1; { "cup"}
draw.vf.cell(tf, fi, "\\SS", i, nr); i<-i+1; { "superset"}
draw.vf.cell(tf, fi, "\\ip", i, nr); i<-i+1; { "reflexsuperset"}
draw.vf.cell(tf, fi, "\\n<", i, nr); i<-i+1; { "notsubset"}
draw.vf.cell(tf, fi, "\\SB", i, nr); i<-i+1; { "subset"}
draw.vf.cell(tf, fi, "\\ib", i, nr); i<-i+1; { "reflexsubset"}
draw.vf.cell(tf, fi, "\\mo", i, nr); i<-i+1; { "element"}
draw.vf.cell(tf, fi, "\\nm", i, nr); i<-i+1; { "notelement"}
draw.vf.cell(tf, fi, "\\/_", i, nr); i<-i+1; { "angle"}
draw.vf.cell(tf, fi, "\\gr", i, nr); i<-i+1; { "nabla"}
draw.vf.cell(tf, fi, "\\rg", i, nr); i<-i+1; { "registerserif"}
draw.vf.cell(tf, fi, "\\co", i, nr); i<-i+1; { "copyrightserif"}
draw.vf.cell(tf, fi, "\\tm", i, nr); i<-i+1; { "trademarkserif"}
draw.vf.cell(tf, fi, "\\PR", i, nr); i<-i+1; { "product"}
draw.vf.cell(tf, fi, "\\sr", i, nr); i<-i+1; { "radical"}
draw.vf.cell(tf, fi, "\\md", i, nr); i<-i+1; { "dotmath"}
draw.vf.cell(tf, fi, "\\no", i, nr); i<-i+1; { "logicalnot"}
draw.vf.cell(tf, fi, "\\AN", i, nr); i<-i+1; { "logicaland"}
draw.vf.cell(tf, fi, "\\OR", i, nr); i<-i+1; { "logicalor"}
draw.vf.cell(tf, fi, "\\hA", i, nr); i<-i+1; { "arrowdblboth"}
draw.vf.cell(tf, fi, "\\lA", i, nr); i<-i+1; { "arrowdblleft"}
draw.vf.cell(tf, fi, "\\uA", i, nr); i<-i+1; { "arrowdblup"}
draw.vf.cell(tf, fi, "\\rA", i, nr); i<-i+1; { "arrowdblright"}
draw.vf.cell(tf, fi, "\\dA", i, nr); i<-i+1; { "arrowdbldown"}
draw.vf.cell(tf, fi, "\\lz", i, nr); i<-i+1; { "lozenge"}
draw.vf.cell(tf, fi, "\\la", i, nr); i<-i+1; { "angleleft"}
draw.vf.cell(tf, fi, "\\RG", i, nr); i<-i+1; { "registersans"}
draw.vf.cell(tf, fi, "\\CO", i, nr); i<-i+1; { "copyrightsans"}
draw.vf.cell(tf, fi, "\\TM", i, nr); i<-i+1; { "trademarksans"}
draw.vf.cell(tf, fi, "\\SU", i, nr); i<-i+1; { "summation"}
draw.vf.cell(tf, fi, "\\lc", i, nr); i<-i+1; { "bracketlefttp"}
draw.vf.cell(tf, fi, "\\lf", i, nr); i<-i+1; { "bracketleftbt"}
draw.vf.cell(tf, fi, "\\ra", i, nr); i<-i+1; { "angleright"}
draw.vf.cell(tf, fi, "\\is", i, nr); i<-i+1; { "integral"}
draw.vf.cell(tf, fi, "\\rc", i, nr); i<-i+1; { "bracketrighttp"}
draw.vf.cell(tf, fi, "\\rf", i, nr); i<-i+1; { "bracketrightbt"}
draw.vf.cell(tf, fi, "\\~=", i, nr); i<-i+1; { "congruent"}
draw.vf.cell(tf, fi, "\\pr", i, nr); i<-i+1; { "minute"}
draw.vf.cell(tf, fi, "\\in", i, nr); i<-i+1; { "infinity"}
draw.vf.cell(tf, fi, "\\n=", i, nr); i<-i+1; { "notequal"}
draw.vf.cell(tf, fi, "\\dl", i, nr); i<-i+1; { "nabla"}

nr <- 25
nc <- 4
make.table(nr, nc)
i <- 0
draw.title("ISO Latin-1 Escape Sequences", i, nr, nc)
draw.vf.cell(tf, fi, "\\r!", i, nr); i<-i+1; { "exclamdown"}
draw.vf.cell(tf, fi, "\\ct", i, nr); i<-i+1; { "cent"}
draw.vf.cell(tf, fi, "\\Po", i, nr); i<-i+1; { "sterling"}
draw.vf.cell(tf, fi, "\\Ye", i, nr); i<-i+1; { "yen"}
draw.vf.cell(tf, fi, "\\bb", i, nr); i<-i+1; { "brokenbar"}
draw.vf.cell(tf, fi, "\\sc", i, nr); i<-i+1; { "section"}
draw.vf.cell(tf, fi, "\\ad", i, nr); i<-i+1; { "dieresis"}
draw.vf.cell(tf, fi, "\\co", i, nr); i<-i+1; { "copyright"}
draw.vf.cell(tf, fi, "\\Of", i, nr); i<-i+1; { "ordfeminine"}
draw.vf.cell(tf, fi, "\\no", i, nr); i<-i+1; { "logicalnot"}
```

```
draw.vf.cell(tf, fi, "\\hy", i, nr); i<-i+1; { "hyphen"}
draw.vf.cell(tf, fi, "\\rg", i, nr); i<-i+1; { "registered"}
draw.vf.cell(tf, fi, "\\a-", i, nr); i<-i+1; { "macron"}
draw.vf.cell(tf, fi, "\\de", i, nr); i<-i+1; { "degree"}
draw.vf.cell(tf, fi, "\\+-", i, nr); i<-i+1; { "plusminus"}
draw.vf.cell(tf, fi, "\\S2", i, nr); i<-i+1; { "twosuperior"}
draw.vf.cell(tf, fi, "\\S3", i, nr); i<-i+1; { "threesuperior"}
draw.vf.cell(tf, fi, "\\aa", i, nr); i<-i+1; { "acute"}
draw.vf.cell(tf, fi, "\\*m", i, nr); i<-i+1; { "mu"}
draw.vf.cell(tf, fi, "\\md", i, nr); i<-i+1; { "periodcentered"}
draw.vf.cell(tf, fi, "\\S1", i, nr); i<-i+1; { "onesuperior"}
draw.vf.cell(tf, fi, "\\Om", i, nr); i<-i+1; { "ordmasculine"}
draw.vf.cell(tf, fi, "\\14", i, nr); i<-i+1; { "onequarter"}
draw.vf.cell(tf, fi, "\\12", i, nr); i<-i+1; { "onehalf"}
draw.vf.cell(tf, fi, "\\34", i, nr); i<-i+1; { "threequarters"}
draw.vf.cell(tf, fi, "\\r?", i, nr); i<-i+1; { "questiondown"}
draw.vf.cell(tf, fi, "\\`A", i, nr); i<-i+1; { "Agrave"}
draw.vf.cell(tf, fi, "\\'A", i, nr); i<-i+1; { "Aacute"}
draw.vf.cell(tf, fi, "\\^A", i, nr); i<-i+1; { "Acircumflex"}
draw.vf.cell(tf, fi, "\\~A", i, nr); i<-i+1; { "Atilde"}
draw.vf.cell(tf, fi, "\\:A", i, nr); i<-i+1; { "Adieresis"}
draw.vf.cell(tf, fi, "\\oA", i, nr); i<-i+1; { "Aring"}
draw.vf.cell(tf, fi, "\\AE", i, nr); i<-i+1; { "AE"}
draw.vf.cell(tf, fi, "\\,C", i, nr); i<-i+1; { "Ccedilla"}
draw.vf.cell(tf, fi, "\\`E", i, nr); i<-i+1; { "Egrave"}
draw.vf.cell(tf, fi, "\\'E", i, nr); i<-i+1; { "Eacute"}
draw.vf.cell(tf, fi, "\\^E", i, nr); i<-i+1; { "Ecircumflex"}
draw.vf.cell(tf, fi, "\\:E", i, nr); i<-i+1; { "Edieresis"}
draw.vf.cell(tf, fi, "\\`I", i, nr); i<-i+1; { "Igrave"}
draw.vf.cell(tf, fi, "\\'I", i, nr); i<-i+1; { "Iacute"}
draw.vf.cell(tf, fi, "\\^I", i, nr); i<-i+1; { "Icircumflex"}
draw.vf.cell(tf, fi, "\\:I", i, nr); i<-i+1; { "Idieresis"}
draw.vf.cell(tf, fi, "\\~N", i, nr); i<-i+1; { "Ntilde"}
draw.vf.cell(tf, fi, "\\`O", i, nr); i<-i+1; { "Ograve"}
draw.vf.cell(tf, fi, "\\'O", i, nr); i<-i+1; { "Oacute"}
draw.vf.cell(tf, fi, "\\^O", i, nr); i<-i+1; { "Ocircumflex"}
draw.vf.cell(tf, fi, "\\~O", i, nr); i<-i+1; { "Otilde"}
draw.vf.cell(tf, fi, "\\:O", i, nr); i<-i+1; { "Odieresis"}
draw.vf.cell(tf, fi, "\\mu", i, nr); i<-i+1; { "multiply"}
draw.vf.cell(tf, fi, "\\/O", i, nr); i<-i+1; { "Oslash"}
draw.vf.cell(tf, fi, "\\`U", i, nr); i<-i+1; { "Ugrave"}
draw.vf.cell(tf, fi, "\\'U", i, nr); i<-i+1; { "Uacute"}
draw.vf.cell(tf, fi, "\\^U", i, nr); i<-i+1; { "Ucircumflex"}
draw.vf.cell(tf, fi, "\\:U", i, nr); i<-i+1; { "Udieresis"}
draw.vf.cell(tf, fi, "\\'Y", i, nr); i<-i+1; { "Yacute"}
draw.vf.cell(tf, fi, "\\ss", i, nr); i<-i+1; { "germandbls"} # WRONG!
draw.vf.cell(tf, fi, "\\`a", i, nr); i<-i+1; { "agrave"}
draw.vf.cell(tf, fi, "\\'a", i, nr); i<-i+1; { "aacute"}
draw.vf.cell(tf, fi, "\\^a", i, nr); i<-i+1; { "acircumflex"}
draw.vf.cell(tf, fi, "\\~a", i, nr); i<-i+1; { "atilde"}
draw.vf.cell(tf, fi, "\\:a", i, nr); i<-i+1; { "adieresis"}
draw.vf.cell(tf, fi, "\\oa", i, nr); i<-i+1; { "aring"}
draw.vf.cell(tf, fi, "\\ae", i, nr); i<-i+1; { "ae"}
draw.vf.cell(tf, fi, "\\,c", i, nr); i<-i+1; { "ccedilla"}
draw.vf.cell(tf, fi, "\\`e", i, nr); i<-i+1; { "egrave"}
draw.vf.cell(tf, fi, "\\'e", i, nr); i<-i+1; { "eacute"}
draw.vf.cell(tf, fi, "\\^e", i, nr); i<-i+1; { "ecircumflex"}
```

```
draw.vf.cell(tf, fi, "\\:e", i, nr); i<-i+1; { "edieresis"}
draw.vf.cell(tf, fi, "\\`i", i, nr); i<-i+1; { "igrave"}
draw.vf.cell(tf, fi, "\\'i", i, nr); i<-i+1; { "iacute"}
draw.vf.cell(tf, fi, "\\^i", i, nr); i<-i+1; { "icircumflex"}
draw.vf.cell(tf, fi, "\\:i", i, nr); i<-i+1; { "idieresis"}
draw.vf.cell(tf, fi, "\\~n", i, nr); i<-i+1; { "ntilde"}
draw.vf.cell(tf, fi, "\\`o", i, nr); i<-i+1; { "ograve"}
draw.vf.cell(tf, fi, "\\'o", i, nr); i<-i+1; { "oacute"}
draw.vf.cell(tf, fi, "\\^o", i, nr); i<-i+1; { "ocircumflex"}
draw.vf.cell(tf, fi, "\\~o", i, nr); i<-i+1; { "otilde"}
draw.vf.cell(tf, fi, "\\:o", i, nr); i<-i+1; { "odieresis"}
draw.vf.cell(tf, fi, "\\di", i, nr); i<-i+1; { "divide"}
draw.vf.cell(tf, fi, "\\/o", i, nr); i<-i+1; { "oslash"}
draw.vf.cell(tf, fi, "\\`u", i, nr); i<-i+1; { "ugrave"}
draw.vf.cell(tf, fi, "\\'u", i, nr); i<-i+1; { "uacute"}
draw.vf.cell(tf, fi, "\\^u", i, nr); i<-i+1; { "ucircumflex"}
draw.vf.cell(tf, fi, "\\:u", i, nr); i<-i+1; { "udieresis"}
draw.vf.cell(tf, fi, "\\'y", i, nr); i<-i+1; { "yacute"}
draw.vf.cell(tf, fi, "\\:y", i, nr); i<-i+1; { "ydieresis"}

nr <- 25
nc <- 2
make.table(nr, nc)
i <- 0
draw.title("Special Escape Sequences", i, nr, nc)
draw.vf.cell(tf, fi, "\\AR", i, nr); i<-i+1; { "aries"}
draw.vf.cell(tf, fi, "\\TA", i, nr); i<-i+1; { "taurus"}
draw.vf.cell(tf, fi, "\\GE", i, nr); i<-i+1; { "gemini"}
draw.vf.cell(tf, fi, "\\CA", i, nr); i<-i+1; { "cancer"}
draw.vf.cell(tf, fi, "\\LE", i, nr); i<-i+1; { "leo"}
draw.vf.cell(tf, fi, "\\VI", i, nr); i<-i+1; { "virgo"}
draw.vf.cell(tf, fi, "\\LI", i, nr); i<-i+1; { "libra"}
draw.vf.cell(tf, fi, "\\SC", i, nr); i<-i+1; { "scorpio"}
draw.vf.cell(tf, fi, "\\SG", i, nr); i<-i+1; { "sagittarius"}
draw.vf.cell(tf, fi, "\\CP", i, nr); i<-i+1; { "capricornus"}
draw.vf.cell(tf, fi, "\\AQ", i, nr); i<-i+1; { "aquarius"}
draw.vf.cell(tf, fi, "\\PI", i, nr); i<-i+1; { "pisces"}
draw.vf.cell(tf, fi, "\\~-", i, nr); i<-i+1; { "modifiedcongruent"}
draw.vf.cell(tf, fi, "\\hb", i, nr); i<-i+1; { "hbar"}
draw.vf.cell(tf, fi, "\\IB", i, nr); i<-i+1; { "interbang"}
draw.vf.cell(tf, fi, "\\Lb", i, nr); i<-i+1; { "lambdabar"}
draw.vf.cell(tf, fi, "\\UD", i, nr); i<-i+1; { "undefined"}
draw.vf.cell(tf, fi, "\\SO", i, nr); i<-i+1; { "sun"}
draw.vf.cell(tf, fi, "\\ME", i, nr); i<-i+1; { "mercury"}
draw.vf.cell(tf, fi, "\\VE", i, nr); i<-i+1; { "venus"}
draw.vf.cell(tf, fi, "\\EA", i, nr); i<-i+1; { "earth"}
draw.vf.cell(tf, fi, "\\MA", i, nr); i<-i+1; { "mars"}
draw.vf.cell(tf, fi, "\\JU", i, nr); i<-i+1; { "jupiter"}
draw.vf.cell(tf, fi, "\\SA", i, nr); i<-i+1; { "saturn"}
draw.vf.cell(tf, fi, "\\UR", i, nr); i<-i+1; { "uranus"}
draw.vf.cell(tf, fi, "\\NE", i, nr); i<-i+1; { "neptune"}
draw.vf.cell(tf, fi, "\\PL", i, nr); i<-i+1; { "pluto"}
draw.vf.cell(tf, fi, "\\LU", i, nr); i<-i+1; { "moon"}
draw.vf.cell(tf, fi, "\\CT", i, nr); i<-i+1; { "comet"}
draw.vf.cell(tf, fi, "\\ST", i, nr); i<-i+1; { "star"}
draw.vf.cell(tf, fi, "\\AS", i, nr); i<-i+1; { "ascendingnode"}
draw.vf.cell(tf, fi, "\\DE", i, nr); i<-i+1; { "descendingnode"}
```

```
draw.vf.cell(tf, fi, "\\s-", i, nr); i<-i+1; { "s1"}
draw.vf.cell(tf, fi, "\\dg", i, nr); i<-i+1; { "dagger"}
draw.vf.cell(tf, fi, "\\dd", i, nr); i<-i+1; { "daggerdbl"}
draw.vf.cell(tf, fi, "\\li", i, nr); i<-i+1; { "line integral"}
draw.vf.cell(tf, fi, "\\-+", i, nr); i<-i+1; { "minusplus"}
draw.vf.cell(tf, fi, "\\||", i, nr); i<-i+1; { "parallel"}
draw.vf.cell(tf, fi, "\\rn", i, nr); i<-i+1; { "overscore"}
draw.vf.cell(tf, fi, "\\ul", i, nr); i<-i+1; { "underscore"}

nr <- 25
nc <- 3
make.table(nr, nc)
code <- c(300:307,310:317,320:327,330:337,340:347,350:357,360:367,370:377,
          243,263)
string <- c(
"\300","\301","\302","\303","\304","\305","\306","\307",
"\310","\311","\312","\313","\314","\315",
"\316","\317","\320","\321","\322","\323",
"\324","\325","\326","\327","\330","\331",
"\332","\333","\334","\335","\336","\337",
"\340","\341","\342","\343","\344","\345","\346","\347",
"\350","\351","\352","\353","\354","\355",
"\356","\357","\360","\361","\362","\363",
"\364","\365","\366","\367","\370","\371",
"\372","\373","\374","\375","\376","\377","\243","\263")
draw.title("Cyrillic Octal Codes", i = 0, nr ,nc)
for (i in 1:66)
    draw.vf.cell(tf, "cyrillic", string[i], i-1, nr,
                 raw.string=paste("\\", as.character(code[i]), sep=""))

nr <- 25
nc <- 3
make.table(nr, nc)
code <- c(252,254,256,262:269,275,278:281,284,745,746,750:768,796:802,
          804:807,809,814:828,830:834,840:844)
draw.title("Raw Hershey Escape Sequences", i=0, nr, nc)
for (i in 1:75)
    draw.vf.cell(tf, fi, paste("\\#H",formatC(code[i],wid=4,flag=0),sep=""),
                 i-1, nr)
make.table(nr, nc)
code <- c(845:847,850:856,860:874,899:909,2296:2299,2318:2332,2367:2382,
          4014,4109)
draw.title("More Raw Hershey Escape Sequences", i=0, nr, nc)
for (i in 1:73)
    draw.vf.cell(tf, fi, paste("\\#H",formatC(code[i],wid=4,flag=0),sep=""),
                 i-1, nr)

par(oldpar)
```

`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, ...)
hist.default(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), • an 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`, *relative* frequencies ("probabilities"), component`density`, are plotted. 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.
`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 foregound color.
`main, xlab, ylab`	these arguments to `title` have useful defaults here.
`xlim, ylim`	the range of x and y values with sensible defaults.
`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.
`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 to `title` and `axis`.

Details

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*".

The default for `breaks` is `"Sturges"`: see `nclass.Sturges`. Other names for which algorithms are supplied are `"Scott"` and `"FD"` / `"Friedman-Diaconis"`. 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.

References

Venables, W. N. and Ripley. B. D. (1999) *Modern Applied Statistics with S-PLUS*. Springer.

See Also

`nclass.Sturges`, `stem`, `density`.

Examples

```
data(islands)
op <- par(mfrow=c(2, 2))
hist(islands)
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)

str(hist(islands, plot= FALSE))        #-> 5  breaks
str(hist(islands, br=12, plot= FALSE)) #-> 10 (~= 12) breaks
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
```

`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)
```

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"

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`.

See Also

`rainbow`, `rgb`, `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)
```

`Hyperbolic` *Hyperbolic Functions*

Description

These functions give the obvious hyperbolic functions. They respectively compute the hyperbolic cosine, sine, tangent, arc-cosine, arc-sine, arc-tangent.

Usage

```
cosh(x)
sinh(x)
tanh(x)
acosh(x)
asinh(x)
atanh(x)
```

Arguments

`x`	a numeric vector

See Also

`cos`, `sin`, `tan`, `acos`, `asin`, `atan`.

Examples

```
Ceps <- .Machine$double.eps # ‘‘Computer epsilon’’
x <- seq(-3, 3, len=200)
stopifnot(
 abs(cosh(x) - (exp(x) + exp(-x))/2) < 20*Ceps,
 abs(sinh(x) - (exp(x) - exp(-x))/2) < 20*Ceps,
 Mod(cosh(x) - cos(1i*x))       < 20*Ceps,
 Mod(sinh(x) - sin(1i*x)/1i)    < 20*Ceps,
 abs(tanh(x)*cosh(x) - sinh(x)) < 20*Ceps
)

## Inverse:
all(abs(asinh(sinh(x)) - x) < 10*Ceps)
x[abs(acosh(cosh(x)) - abs(x)) > 100*Ceps] #- imprecise for small x
all(abs(atanh(tanh(x)) - x) < 100*Ceps)

all(abs(asinh(x) - log(x + sqrt(x^2 + 1))) < 10*Ceps)
cx <- cosh(x)
all(abs(acosh(cx) - log(cx + sqrt(cx^2 - 1))) < 1000*Ceps)
```

`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.

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)
```

---

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.

The function `all.equal` is also sometimes used to test equality this way, but it was intended for something different. First, it tries to allow for "reasonable" differences in numeric results. Second, it returns a descriptive character vector instead of `FALSE` when the objects do not match. Therefore, it is not the right function to use for reliable testing either. (If you *do* want to allow for numeric fuzziness in comparing objects, you can combine `all.equal` and `identical`, as shown in the examples below.)

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.

Value

A single logical value, `TRUE` or `FALSE`, never `NA` and never anything other than a single value.

Author(s)

John Chambers

See Also

`all.equal` for descriptions of how two objects differ; Comparison for operators that generate elementwise comparisons.

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

## how to test for object equality allowing for numeric fuzz
identical(all.equal(x, y), TRUE)
## If all.equal thinks the objects are different, it returns a
## character string, and this expression evaluates to FALSE
```

`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 to the pointer, its index will be returned as part of the value of the call.

Usage

```
identify(x, ...)
identify.default(x, y = NULL, labels = seq(along = x), pos = FALSE,
                 n = length(x), plot = TRUE, offset = 0.5, ...)
```

Arguments

`x,y`	coordinates of points in a scatter plot. Alternatively, any object which defines coordinates (a plotting structure, time series etc.) can be given as `x` and `y` left undefined.
`labels`	an optional vector, the same length as `x` and `y`, giving labels for the points.
`pos`	if `pos` is `TRUE`, a component is added to the return value which indicates where text was plotted relative to each identified point (1=below, 2=left, 3=above and 4=right).
`n`	the maximum number of points to be identified.
`plot`	if `plot` is `TRUE`, the labels are printed at the points and if `FALSE` they are omitted.
`offset`	the distance (in character widths) which separates the label from identified points.
`...`	further arguments to `par(.)`.

Details

If in addition, `plot` is `TRUE`, the point is labelled with the corresponding element of `text`.

The labels are placed either below, to the left, above or to the right of the identified point, depending on where the cursor was.

The identification process is terminated by clicking the second button and selecting 'Stop' from the menu, or from the 'Stop' menu on the graphics window.

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.

See Also

`locator`

`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`. If `yes` or `no` are too short, their elements are recycled.

Usage

```
ifelse(test, yes, no)
```

Arguments

`test`	a logical vector
`yes`	return values for true elements of `test`.
`no`	return values for false elements of `test`.

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)
```

`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, ...)
image.default(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 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 (`NA`s 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. 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 range of the finite values 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 where 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. (This was the default prior to R 1.1.0.) The current 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.

Details

The length of `x` should be equal to the `nrow(x)+1` or `nrow(x)`. 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.

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.

Note

Based on a function by Thomas Lumley ⟨tlumley@u.washington.edu⟩.

See Also

`contour`, `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)

data(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)
```

`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`

`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

```
data(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

```
data(infert)
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 equivalent to a Cox model :
if(require(survival)){
  faketime <- rep(42,nrow(infert))
  model3 <- coxph(Surv(faketime,case)~spontaneous+induced+strata(stratum),
                data=infert,method="exact")
  summary(model3)
  detach()# survival (conflicts)
}
```

`influence.measures` *Regression Diagnostics*

Description

This suite of functions can be used to compute some of the regression diagnostics discussed in Belsley, Kuh and Welsch (1980), and in Cook and Weisberg (1982).

Usage

```
influence.measures(lm.obj)

rstandard(lm.obj,
          infl = lm.influence(lm.obj),
          res = weighted.residuals(lm.obj),
          sd = sqrt(deviance(lm.obj)/df.residual(lm.obj)))
rstudent (lm.obj, infl = ..., res = ...)
dffits   (lm.obj, infl = ..., res = ...)
dfbetas  (lm.obj, infl = ...)
covratio (lm.obj, infl = ..., res = ...)
cooks.distance(lm.obj, infl = ..., res = ..., sd = ...)

hat(x, intercept = TRUE)
```

Arguments

`lm.obj`	the resulting object returned by `lm`.
`infl`	influence structure as returned by `lm.influence`.
`res`	(possibly weighted) residuals, with proper default.
`sd`	standard deviation to use, see default.
`x`	the X or design matrix.
`intercept`	should an intercept column be pre-prended to `x`?

Details

The primary 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.)

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`) 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 it for the cases excluded during fitting.

The function `hat()` exists mainly for S (version 2) compatibility.

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.

See Also

`lm.influence`.

Examples

```
## Analysis of the life-cycle savings data
## given in Belsley, Kuh and Welsch.
data(LifeCycleSavings)
lm.SR <- lm(sr ~ pop15 + pop75 + dpi + ddpi, data = LifeCycleSavings)
summary(inflm.SR <- influence.measures(lm.SR))
inflm.SR
which(apply(inflm.SR$is.inf, 1, any)) # which observations 'are' influential
dim(dfb <- dfbetas(lm.SR))            # the 1st columns of influence.measures
all(dfb == inflm.SR$infmat[, 1:5])
rstandard(lm.SR)
rstudent(lm.SR)
dffits(lm.SR)
covratio(lm.SR)

## 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))
influence.measures(lmH)
```

InsectSprays *Effectiveness of Insect Sprays*

Description

The counts of insects in agricultural experimental units treated with different insecticides.

Usage

```
data(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

```
data(InsectSprays)
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)
```

INSTALL *Install Add-on Packages from Sources*

Description

To install packages into the default library tree (which is rooted at '$R_HOME/library'), do `Rcmd INSTALL pkgs`.

To install into the library tree `lib` instead of the default one, use `Rcmd INSTALL -l lib pkgs`.

Usage

```
Rcmd INSTALL [options] [-l lib] pkgs
```

Arguments

`pkgs`	A list with the path names of the sources of the packages to be installed.
`lib`	the path name of the R library tree to install to.
`options`	a list of options through which in particular the build process for help files can be controlled. Use `Rcmd INSTALL --help` for the current list of options.

Details

Both `lib` and the elements of `pkgs` may be absolute or relative path names. `pkgs` can also contain name of package archive files of the form 'pkg_version.tar.gz' as obtained from CRAN (these are then extracted in a temporary directory 'R.INSTALL' of the current directory).

If the option `--save` is used, the installation procedure creates a binary image of the package code, which is then loaded when the library is attached, rather than evaluating the package source at that time. Having a file `install.R` in the package directory (its contents don't matter) makes this the default behavior for the package (option `--no-save` overrides). You may need `--save` if your package requires other packages to evaluate its own source.

For checking (via the examples in the help files 'man/*.Rd', use `Rcmd check [-l lib] <pkg>`: use `Rcmd check --help` for more information.

See Also

REMOVE, and library for information on using several library trees and creating packages; update.packages for automatic update of packages using the internet.

`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.

Value

`integer` creates a integer vector of the specified length. Each element of the vector is equal to `0`. Integer vectors exist so that data can be passed to C or Fortran code which expects them.

`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).

`is.integer` returns `TRUE` or `FALSE` depending on whether its argument is of integer type or not.

`integrate` *Integration of One-Dimensional Functions*

Description

Adaptive quadrature of functions of one variable over a finit 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 the same length. Returning a non-finite 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 passes to `f`.

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.

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 *Quadpack: a Subroutine Package for Automatic Integration.* Springer Verlag, 1983.

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)
integrate(integrand, lower = 0, upper = 10)
integrate(integrand, lower = 0, upper = 100000)
integrate(integrand, lower = 0, upper = 1000000, stop.on.error = FALSE)

try(integrate(function(x) 2, 0, 1))  ## no vectorizable function
integrate(function(x) rep(2, length(x)), 0, 1)  ## correct
```

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)
```

Arguments

`...`	The factors for which interaction is to be computed.
`drop`	If `drop` is `TRUE`, empty factor levels are dropped from the result. The default is to retain all factor levels.

Value

A factor which represents the interaction of the given factors.

See Also

`factor`.

Examples

```
a <- gl(2, 2, 8)
b <- gl(2, 4, 8)
interaction(a, b)
```

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"), legend = TRUE,
                 trace.label = deparse(substitute(trace.factor)),
                 fixed = FALSE, xlab, ylab, ylim, lty, col = 1,
                 pch = c(1:9, 0, letters), ...)
```

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`	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`	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.
`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.
`...`	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.

Examples

```
data(ToothGrowth)
attach(ToothGrowth)
interaction.plot(dose, supp, len, fixed=TRUE)
dose <- ordered(dose)
interaction.plot(dose, supp, len, fixed=TRUE)
detach()

data(OrchardSprays)
attach(OrchardSprays)
interaction.plot(treatment, rowpos, decrease)
interaction.plot(rowpos, treatment, decrease)
## order the rows by their mean effect
```

```
rowpos <- factor(rowpos, levels=sort.list(tapply(decrease, rowpos, mean)))
interaction.plot(rowpos, treatment, decrease)
detach()

data(esoph)
attach(esoph)
interaction.plot(agegp, alcgp, ncases/ncontrols)
interaction.plot(agegp, tobgp, ncases/ncontrols, trace.label="tobacco",
                 fixed=TRUE)
detach()
```

---

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.

Usage

```
.Internal(call)
```

Arguments

`call`	a call expression

See Also

`.Primitive`, `.C`, `.Fortran`.

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.

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
```

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

```
data(rivers)
IQR(rivers)
```

`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

```
data(iris)
data(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.

See Also

matplot some examples of which use `iris`.

Examples

```
data(iris3)
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]]))))
data(iris)
all.equal(ii, iris) # TRUE
```

`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))
```

`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 or not.

`Inf` and `-Inf` are positive and negative 'infinity' whereas `NaN` means "Not a Number".

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`).

`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`).

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*, see the many examples below.

References

ANSI/IEEE 754 Floating-Point Standard.

Currently (6/1999), Bill M.'s ⟨billm@melbpc.org.au⟩ tutorial and examples at
http://www.linuxsupportline.com/~billm/

See Also

`NA`, *'Not Available'* which is not a number as well, however usually used for missing values.

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
)
exp(-Inf) == 0
##              (actually, the last one seems to give NA on not-very-new
##               versions of Linux, which is a Linux bug and seems to be
##               corrected in newer 'libc6' based Linuxen).

stopifnot(
    is.na(0/0),
    !is.na(Inf),
    is.nan(0/0),

    !is.nan(NA)  &&  !is.infinite(NA)  && !is.finite(NA),
     is.nan(NaN) &&  !is.infinite(NaN) && !is.finite(NaN),
    !is.nan(c(1,NA)),
    c(FALSE,TRUE,FALSE) == is.nan(c   (1,NaN,NA)),
    c(FALSE,TRUE,FALSE) == is.nan(list(1,NaN,NA))#-> FALSE in older versions
)


lgamma(Inf) == Inf
Inf + Inf == Inf
Inf - Inf == NaN # NA --- should test with  'is.nan()

(1/0) * (1/0)# Inf
(1/0) / (1/0)# NaN

pm <- c(-1,1) # 'pm' = plus/minus

log(0) == - 1/0
exp(-Inf) == 0

sin(Inf)
cos(Inf)
tan(Inf)
all(atan(Inf*pm) == pm*pi/2) # TRUE

x <- c(100,-1e-13,Inf,-Inf, NaN, pi, NA)
x #  1.000000 -3.000000       Inf      -Inf        NA  3.141593        NA
names(x) <- formatC(x, dig=3)
is.finite(x)
##-   100 -1e-13 Inf -Inf NaN 3.14 NA
##-     T      T   .    .   .    T  .
is.na(x)
##-   100 -1e-13 Inf -Inf NaN 3.14 NA
##-     .      .   .    .   T    .  T
which(is.na(x) & !is.nan(x))# only 'NA': 7
```

```
is.na(x) | is.finite(x)
##-   100 -1e-13 Inf -Inf NaN 3.14 NA
##-     T      T   .    .   T    T  T
is.infinite(x)
##-   100 -1e-13 Inf -Inf NaN 3.14 NA
##-     .      .   T    T   .    .  .

##-- either  finite or infinite  or  NA:
all(is.na(x) != is.finite(x) | is.infinite(x)) # TRUE
all(is.nan(x) != is.finite(x) | is.infinite(x)) # FALSE: have 'real' NA

##--- Integer
(ix <- structure(as.integer(x),names= names(x)))
##-   100 -1e-13    Inf   -Inf    NaN   3.14     NA
##-   100      0     NA     NA     NA      3     NA
all(is.na(ix) != is.finite(ix) | is.infinite(ix)) # TRUE (still)

ix[3] == (iI <- as.integer(Inf))#> warning: NAs introduced by coercion
ix[4] == (imI<- as.integer(-Inf))
iI  == .Machine$integer.max # TRUE
imI == -.Machine$integer.max # TRUE

##--- Overflow in simple integer arithmetic:
as.integer(2)*iI #  -2
as.integer(3)*iI #  2147483645
as.integer(3)*iI == iI-2 # TRUE

storage.mode(ii <- -3:5)
storage.mode(zm <- outer(ii,ii, FUN="*"))# integer
storage.mode(zd <- outer(ii,ii, FUN="/"))# double
range(zd, na.rm=TRUE)# -Inf  Inf
zd[,ii==0]

(storage.mode(print(1:1 / 0:0)))# Inf "double"
(storage.mode(print(1:1 / 1:1)))# 1 "double"
(storage.mode(print(1:1 + 1:1)))# 2 "integer"
(storage.mode(print(2:2 * 2:2)))# 4 "integer"
```

`is.function` *Is an Object of Type Function?*

Description

Checks whether its argument is a function.

Usage

```
is.function(x)
```

Arguments

`x`	an R object.

Value

TRUE if x is a function, and FALSE otherwise.

---

is.language — *Is an Object a Language Object?*

Description

is.language returns TRUE if x is either a variable name, a call, or an expression.

Usage

```
is.language(x)
```

Arguments

x	object to be tested.

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.

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, either your the code must 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
# 'which()' only exists 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` does not have a list structure 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.

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)        #   "
is.a.r(y ~ x)     #   "
is.a.r(expression(x+1))# should be F-T (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.

---

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

```
data(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

```
data(islands)
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))")
```

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.

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 `example(Japanese)` shows tables of the escape sequences for the available Japanese characters.

References

http://www.gnu.org/software/plotutils/plotutils.html

See Also

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")
```

```
######
# create tables of Japanese characters
######
make.table <- function(nr, nc) {
    opar <- par(mar=rep(0, 4), pty="s")
    plot(c(0, nc*(10%/%nc) + 1), c(0, -(nr + 1)),
         type="n", xlab="", ylab="", axes=FALSE)
    invisible(opar)
}

get.r <- function(i, nr)   i %% nr + 1
get.c <- function(i, nr)   i %/% nr + 1
Esc2 <- function(str)      paste("\\", str, sep="")

draw.title <- function(title, nc)
    text((nc*(10%/%nc) + 1)/2, 0, title, font=2)

draw.vf.cell <- function(typeface, fontindex, string, i, nr, raw.string=NULL) {
    r <- get.r(i, nr)
    c <- get.c(i, nr)
    x0 <- 2*(c - 1)
    if (is.null(raw.string)) raw.string <- Esc2(string)
    text(x0 + 1.1, -r, raw.string, col="grey")
    text(x0 + 2,   -r, string, vfont=c(typeface, fontindex))
    rect(x0 +  .5, -(r - .5), x0 + 2.5, -(r + .5), border="grey")
}

draw.vf.cell2 <- function(string, alt, i, nr) {
    r <- get.r(i, nr)
    c <- get.c(i, nr)
    x0 <- 3*(c - 1)
    text(x0 + 1.1, -r, Esc2(string <- Esc2(string)), col="grey")
    text(x0 + 2.2, -r, Esc2(Esc2(alt)), col="grey", cex=.6)
    text(x0 + 3,   -r, string, vfont=c("serif", "plain"))
    rect(x0 +  .5, -(r - .5), x0 + 3.5, -(r + .5), border="grey")
}

tf <- "serif"
fi <- "plain"
nr <- 25
nc <- 4
oldpar <- make.table(nr, nc)
index <- 0
digits <- c(0:9,"a","b","c","d","e","f")
draw.title("Hiragana : \\\\#J24nn", nc)
for (i in 2:7) {
    for (j in 1:16) {
        if (!((i == 2 && j == 1) || (i == 7 && j > 4))) {
            draw.vf.cell(tf, fi, paste("\\#J24", i, digits[j], sep=""),
                         index, nr)
            index <- index + 1
        }
    }
}

nr <- 25
nc <- 4
```

```
make.table(nr, nc)
index <- 0
digits <- c(0:9,"a","b","c","d","e","f")
draw.title("Katakana : \\\\#J25nn", nc)
for (i in 2:7) {
    for (j in 1:16) {
        if (!((i == 2 && j == 1) || (i == 7 && j > 7))) {
            draw.vf.cell(tf, fi, paste("\\#J25", i, digits[j], sep=""),
                         index, nr)
            index <- index + 1
        }
    }
}

nr <- 26
nc <- 3
make.table(nr, nc)
i <- 0
draw.title("Kanji (1)", nc)
draw.vf.cell2("#J3021", "#N0043", i, nr); i <- i + 1
draw.vf.cell2("#J3026", "#N2829", i, nr); i <- i + 1
draw.vf.cell2("#J302d", "#N0062", i, nr); i <- i + 1
draw.vf.cell2("#J3035", "#N0818", i, nr); i <- i + 1
draw.vf.cell2("#J303f", "#N1802", i, nr); i <- i + 1
draw.vf.cell2("#J3045", "#N2154", i, nr); i <- i + 1
draw.vf.cell2("#J304c", "#N0401", i, nr); i <- i + 1
draw.vf.cell2("#J3057", "#N2107", i, nr); i <- i + 1
draw.vf.cell2("#J3059", "#N0138", i, nr); i <- i + 1
draw.vf.cell2("#J305b", "#N3008", i, nr); i <- i + 1
draw.vf.cell2("#J305e", "#N3579", i, nr); i <- i + 1
draw.vf.cell2("#J3061", "#N4214", i, nr); i <- i + 1
draw.vf.cell2("#J306c", "#N0001", i, nr); i <- i + 1
draw.vf.cell2("#J3070", "#N3294", i, nr); i <- i + 1
draw.vf.cell2("#J3078", "#N1026", i, nr); i <- i + 1
draw.vf.cell2("#J307a", "#N1562", i, nr); i <- i + 1
draw.vf.cell2("#J3122", "#N5006", i, nr); i <- i + 1
draw.vf.cell2("#J3126", "#N0878", i, nr); i <- i + 1
draw.vf.cell2("#J3127", "#N1280", i, nr); i <- i + 1
draw.vf.cell2("#J3129", "#N3673", i, nr); i <- i + 1
draw.vf.cell2("#J312b", "#N5042", i, nr); i <- i + 1
draw.vf.cell2("#J3132", "#N2629", i, nr); i <- i + 1
draw.vf.cell2("#J313b", "#N2973", i, nr); i <- i + 1
draw.vf.cell2("#J313f", "#N4725", i, nr); i <- i + 1
draw.vf.cell2("#J3140", "#N5046", i, nr); i <- i + 1
draw.vf.cell2("#J314a", "#N0130", i, nr); i <- i + 1
draw.vf.cell2("#J3155", "#N2599", i, nr); i <- i + 1
draw.vf.cell2("#J315f", "#N0617", i, nr); i <- i + 1
draw.vf.cell2("#J3173", "#N4733", i, nr); i <- i + 1
draw.vf.cell2("#J3176", "#N1125", i, nr); i <- i + 1
draw.vf.cell2("#J3177", "#N2083", i, nr); i <- i + 1
draw.vf.cell2("#J317e", "#N1504", i, nr); i <- i + 1
draw.vf.cell2("#J3221", "#N1885", i, nr); i <- i + 1
draw.vf.cell2("#J3223", "#N2361", i, nr); i <- i + 1
draw.vf.cell2("#J3226", "#N2922", i, nr); i <- i + 1
draw.vf.cell2("#J322b", "#N5399", i, nr); i <- i + 1
draw.vf.cell2("#J322f", "#N0551", i, nr); i <- i + 1
draw.vf.cell2("#J3235", "#N0260", i, nr); i <- i + 1
```

```
draw.vf.cell2("#J3239", "#N2634", i, nr); i <- i + 1
draw.vf.cell2("#J323b", "#N5110", i, nr); i <- i + 1
draw.vf.cell2("#J323c", "#N0009", i, nr); i <- i + 1
draw.vf.cell2("#J323d", "#N0350", i, nr); i <- i + 1
draw.vf.cell2("#J323f", "#N0409", i, nr); i <- i + 1
draw.vf.cell2("#J3241", "#N0422", i, nr); i <- i + 1
draw.vf.cell2("#J3243", "#N0716", i, nr); i <- i + 1
draw.vf.cell2("#J3244", "#N0024", i, nr); i <- i + 1
draw.vf.cell2("#J3246", "#N0058", i, nr); i <- i + 1
draw.vf.cell2("#J3248", "#N1311", i, nr); i <- i + 1
draw.vf.cell2("#J324a", "#N3272", i, nr); i <- i + 1
draw.vf.cell2("#J324c", "#N0107", i, nr); i <- i + 1
draw.vf.cell2("#J324f", "#N2530", i, nr); i <- i + 1
draw.vf.cell2("#J3250", "#N2743", i, nr); i <- i + 1
draw.vf.cell2("#J3256", "#N3909", i, nr); i <- i + 1
draw.vf.cell2("#J3259", "#N3956", i, nr); i <- i + 1
draw.vf.cell2("#J3261", "#N4723", i, nr); i <- i + 1
draw.vf.cell2("#J3267", "#N2848", i, nr); i <- i + 1
draw.vf.cell2("#J3268", "#N0050", i, nr); i <- i + 1
draw.vf.cell2("#J3272", "#N4306", i, nr); i <- i + 1
draw.vf.cell2("#J3273", "#N1028", i, nr); i <- i + 1
draw.vf.cell2("#J3323", "#N2264", i, nr); i <- i + 1
draw.vf.cell2("#J3324", "#N2553", i, nr); i <- i + 1
draw.vf.cell2("#J3326", "#N2998", i, nr); i <- i + 1
draw.vf.cell2("#J3328", "#N3537", i, nr); i <- i + 1
draw.vf.cell2("#J332b", "#N4950", i, nr); i <- i + 1
draw.vf.cell2("#J332d", "#N4486", i, nr); i <- i + 1
draw.vf.cell2("#J3330", "#N1168", i, nr); i <- i + 1
draw.vf.cell2("#J3346", "#N1163", i, nr); i <- i + 1
draw.vf.cell2("#J334b", "#N2254", i, nr); i <- i + 1
draw.vf.cell2("#J3351", "#N4301", i, nr); i <- i + 1
draw.vf.cell2("#J3353", "#N4623", i, nr); i <- i + 1
draw.vf.cell2("#J3357", "#N5088", i, nr); i <- i + 1
draw.vf.cell2("#J3358", "#N1271", i, nr); i <- i + 1
draw.vf.cell2("#J335a", "#N2324", i, nr); i <- i + 1
draw.vf.cell2("#J3364", "#N0703", i, nr); i <- i + 1
draw.vf.cell2("#J3424", "#N2977", i, nr); i <- i + 1
draw.vf.cell2("#J3428", "#N1322", i, nr); i <- i + 1

make.table(nr, nc)
i <- 0
draw.title("Kanji (2)", nc)
draw.vf.cell2("#J342c", "#N1466", i, nr); i <- i + 1
draw.vf.cell2("#J3433", "#N1492", i, nr); i <- i + 1
draw.vf.cell2("#J3434", "#N0790", i, nr); i <- i + 1
draw.vf.cell2("#J3436", "#N1731", i, nr); i <- i + 1
draw.vf.cell2("#J3437", "#N1756", i, nr); i <- i + 1
draw.vf.cell2("#J3445", "#N2988", i, nr); i <- i + 1
draw.vf.cell2("#J3449", "#N3416", i, nr); i <- i + 1
draw.vf.cell2("#J3454", "#N4750", i, nr); i <- i + 1
draw.vf.cell2("#J3456", "#N4949", i, nr); i <- i + 1
draw.vf.cell2("#J3458", "#N4958", i, nr); i <- i + 1
draw.vf.cell2("#J346f", "#N0994", i, nr); i <- i + 1
draw.vf.cell2("#J3470", "#N1098", i, nr); i <- i + 1
draw.vf.cell2("#J3476", "#N1496", i, nr); i <- i + 1
draw.vf.cell2("#J347c", "#N3785", i, nr); i <- i + 1
draw.vf.cell2("#J3521", "#N2379", i, nr); i <- i + 1
```

```
draw.vf.cell2("#J3522", "#N1582", i, nr); i <- i + 1
draw.vf.cell2("#J3524", "#N2480", i, nr); i <- i + 1
draw.vf.cell2("#J3525", "#N2507", i, nr); i <- i + 1
draw.vf.cell2("#J352d", "#N4318", i, nr); i <- i + 1
draw.vf.cell2("#J3530", "#N4610", i, nr); i <- i + 1
draw.vf.cell2("#J3534", "#N5276", i, nr); i <- i + 1
draw.vf.cell2("#J3535", "#N5445", i, nr); i <- i + 1
draw.vf.cell2("#J3546", "#N3981", i, nr); i <- i + 1
draw.vf.cell2("#J3555", "#N4685", i, nr); i <- i + 1
draw.vf.cell2("#J355a", "#N0154", i, nr); i <- i + 1
draw.vf.cell2("#J355b", "#N0885", i, nr); i <- i + 1
draw.vf.cell2("#J355d", "#N1560", i, nr); i <- i + 1
draw.vf.cell2("#J3565", "#N2941", i, nr); i <- i + 1
draw.vf.cell2("#J3566", "#N3314", i, nr); i <- i + 1
draw.vf.cell2("#J3569", "#N3496", i, nr); i <- i + 1
draw.vf.cell2("#J356d", "#N2852", i, nr); i <- i + 1
draw.vf.cell2("#J356e", "#N1051", i, nr); i <- i + 1
draw.vf.cell2("#J356f", "#N1387", i, nr); i <- i + 1
draw.vf.cell2("#J3575", "#N4109", i, nr); i <- i + 1
draw.vf.cell2("#J3577", "#N4548", i, nr); i <- i + 1
draw.vf.cell2("#J357b", "#N5281", i, nr); i <- i + 1
draw.vf.cell2("#J357e", "#N0295", i, nr); i <- i + 1
draw.vf.cell2("#J3621", "#N0431", i, nr); i <- i + 1
draw.vf.cell2("#J3626", "#N0581", i, nr); i <- i + 1
draw.vf.cell2("#J362d", "#N1135", i, nr); i <- i + 1
draw.vf.cell2("#J362f", "#N1571", i, nr); i <- i + 1
draw.vf.cell2("#J3635", "#N2052", i, nr); i <- i + 1
draw.vf.cell2("#J3636", "#N2378", i, nr); i <- i + 1
draw.vf.cell2("#J364a", "#N0103", i, nr); i <- i + 1
draw.vf.cell2("#J364b", "#N2305", i, nr); i <- i + 1
draw.vf.cell2("#J364c", "#N2923", i, nr); i <- i + 1
draw.vf.cell2("#J3651", "#N1065", i, nr); i <- i + 1
draw.vf.cell2("#J3661", "#N4671", i, nr); i <- i + 1
draw.vf.cell2("#J3662", "#N4815", i, nr); i <- i + 1
draw.vf.cell2("#J3664", "#N4855", i, nr); i <- i + 1
draw.vf.cell2("#J3665", "#N0146", i, nr); i <- i + 1
draw.vf.cell2("#J3671", "#N3128", i, nr); i <- i + 1
draw.vf.cell2("#J3675", "#N3317", i, nr); i <- i + 1
draw.vf.cell2("#J367e", "#N1386", i, nr); i <- i + 1
draw.vf.cell2("#J3738", "#N0449", i, nr); i <- i + 1
draw.vf.cell2("#J3739", "#N0534", i, nr); i <- i + 1
draw.vf.cell2("#J373e", "#N2937", i, nr); i <- i + 1
draw.vf.cell2("#J373f", "#N1077", i, nr); i <- i + 1
draw.vf.cell2("#J3741", "#N1589", i, nr); i <- i + 1
draw.vf.cell2("#J3742", "#N1602", i, nr); i <- i + 1
draw.vf.cell2("#J374f", "#N0195", i, nr); i <- i + 1
draw.vf.cell2("#J3750", "#N3523", i, nr); i <- i + 1
draw.vf.cell2("#J3757", "#N4312", i, nr); i <- i + 1
draw.vf.cell2("#J375a", "#N4620", i, nr); i <- i + 1
draw.vf.cell2("#J3767", "#N2412", i, nr); i <- i + 1
draw.vf.cell2("#J3768", "#N2509", i, nr); i <- i + 1
draw.vf.cell2("#J376a", "#N3313", i, nr); i <- i + 1
draw.vf.cell2("#J376b", "#N3540", i, nr); i <- i + 1
draw.vf.cell2("#J376c", "#N4205", i, nr); i <- i + 1
draw.vf.cell2("#J376e", "#N2169", i, nr); i <- i + 1
draw.vf.cell2("#J3777", "#N1045", i, nr); i <- i + 1
draw.vf.cell2("#J3824", "#N2868", i, nr); i <- i + 1
```

```
draw.vf.cell2("#J3826", "#N3180", i, nr); i <- i + 1
draw.vf.cell2("#J3828", "#N3543", i, nr); i <- i + 1
draw.vf.cell2("#J382b", "#N4284", i, nr); i <- i + 1
draw.vf.cell2("#J3833", "#N5220", i, nr); i <- i + 1

make.table(nr, nc)
i <- 0
draw.title("Kanji (3)", nc)
draw.vf.cell2("#J3835", "#N0275", i, nr); i <- i + 1
draw.vf.cell2("#J3836", "#N0825", i, nr); i <- i + 1
draw.vf.cell2("#J3839", "#N1568", i, nr); i <- i + 1
draw.vf.cell2("#J383a", "#N2637", i, nr); i <- i + 1
draw.vf.cell2("#J383b", "#N2656", i, nr); i <- i + 1
draw.vf.cell2("#J383d", "#N2943", i, nr); i <- i + 1
draw.vf.cell2("#J3840", "#N4309", i, nr); i <- i + 1
draw.vf.cell2("#J3842", "#N4987", i, nr); i <- i + 1
draw.vf.cell2("#J3845", "#N0770", i, nr); i <- i + 1
draw.vf.cell2("#J3847", "#N1036", i, nr); i <- i + 1
draw.vf.cell2("#J384c", "#N1567", i, nr); i <- i + 1
draw.vf.cell2("#J384d", "#N1817", i, nr); i <- i + 1
draw.vf.cell2("#J384e", "#N2044", i, nr); i <- i + 1
draw.vf.cell2("#J385d", "#N5415", i, nr); i <- i + 1
draw.vf.cell2("#J385e", "#N0015", i, nr); i <- i + 1
draw.vf.cell2("#J3861", "#N0162", i, nr); i <- i + 1
draw.vf.cell2("#J3865", "#N1610", i, nr); i <- i + 1
draw.vf.cell2("#J3866", "#N1628", i, nr); i <- i + 1
draw.vf.cell2("#J386c", "#N4374", i, nr); i <- i + 1
draw.vf.cell2("#J3872", "#N0290", i, nr); i <- i + 1
draw.vf.cell2("#J3877", "#N1358", i, nr); i <- i + 1
draw.vf.cell2("#J3878", "#N0579", i, nr); i <- i + 1
draw.vf.cell2("#J387d", "#N0868", i, nr); i <- i + 1
draw.vf.cell2("#J387e", "#N0101", i, nr); i <- i + 1
draw.vf.cell2("#J3929", "#N1451", i, nr); i <- i + 1
draw.vf.cell2("#J3931", "#N1683", i, nr); i <- i + 1
draw.vf.cell2("#J393d", "#N2343", i, nr); i <- i + 1
draw.vf.cell2("#J3943", "#N0092", i, nr); i <- i + 1
draw.vf.cell2("#J394d", "#N3684", i, nr); i <- i + 1
draw.vf.cell2("#J3954", "#N4213", i, nr); i <- i + 1
draw.vf.cell2("#J3955", "#N1641", i, nr); i <- i + 1
draw.vf.cell2("#J395b", "#N4843", i, nr); i <- i + 1
draw.vf.cell2("#J395d", "#N4883", i, nr); i <- i + 1
draw.vf.cell2("#J395f", "#N4994", i, nr); i <- i + 1
draw.vf.cell2("#J3960", "#N1459", i, nr); i <- i + 1
draw.vf.cell2("#J3961", "#N5188", i, nr); i <- i + 1
draw.vf.cell2("#J3962", "#N5248", i, nr); i <- i + 1
draw.vf.cell2("#J3966", "#N0882", i, nr); i <- i + 1
draw.vf.cell2("#J3967", "#N0383", i, nr); i <- i + 1
draw.vf.cell2("#J3971", "#N1037", i, nr); i <- i + 1
draw.vf.cell2("#J3975", "#N5403", i, nr); i <- i + 1
draw.vf.cell2("#J397c", "#N5236", i, nr); i <- i + 1
draw.vf.cell2("#J397e", "#N4660", i, nr); i <- i + 1
draw.vf.cell2("#J3a21", "#N2430", i, nr); i <- i + 1
draw.vf.cell2("#J3a23", "#N0352", i, nr); i <- i + 1
draw.vf.cell2("#J3a2c", "#N2261", i, nr); i <- i + 1
draw.vf.cell2("#J3a38", "#N1455", i, nr); i <- i + 1
draw.vf.cell2("#J3a39", "#N3662", i, nr); i <- i + 1
draw.vf.cell2("#J3a42", "#N1515", i, nr); i <- i + 1
```

```
draw.vf.cell2("#J3a46", "#N0035", i, nr); i <- i + 1
draw.vf.cell2("#J3a47", "#N2146", i, nr); i <- i + 1
draw.vf.cell2("#J3a59", "#N3522", i, nr); i <- i + 1
draw.vf.cell2("#J3a5f", "#N1055", i, nr); i <- i + 1
draw.vf.cell2("#J3a6e", "#N0407", i, nr); i <- i + 1
draw.vf.cell2("#J3a72", "#N2119", i, nr); i <- i + 1
draw.vf.cell2("#J3a79", "#N2256", i, nr); i <- i + 1
draw.vf.cell2("#J3b2e", "#N3113", i, nr); i <- i + 1
draw.vf.cell2("#J3b30", "#N0008", i, nr); i <- i + 1
draw.vf.cell2("#J3b33", "#N1407", i, nr); i <- i + 1
draw.vf.cell2("#J3b36", "#N2056", i, nr); i <- i + 1
draw.vf.cell2("#J3b3b", "#N3415", i, nr); i <- i + 1
draw.vf.cell2("#J3b40", "#N4789", i, nr); i <- i + 1
draw.vf.cell2("#J3b45", "#N0362", i, nr); i <- i + 1
draw.vf.cell2("#J3b4d", "#N1025", i, nr); i <- i + 1
draw.vf.cell2("#J3b4e", "#N1160", i, nr); i <- i + 1
draw.vf.cell2("#J3b4f", "#N1208", i, nr); i <- i + 1
draw.vf.cell2("#J3b52", "#N1264", i, nr); i <- i + 1
draw.vf.cell2("#J3b54", "#N0284", i, nr); i <- i + 1
draw.vf.cell2("#J3b57", "#N3001", i, nr); i <- i + 1
draw.vf.cell2("#J3b58", "#N1904", i, nr); i <- i + 1
draw.vf.cell2("#J3b59", "#N2039", i, nr); i <- i + 1
draw.vf.cell2("#J3b5e", "#N2211", i, nr); i <- i + 1
draw.vf.cell2("#J3b5f", "#N2429", i, nr); i <- i + 1
draw.vf.cell2("#J3b60", "#N2439", i, nr); i <- i + 1
draw.vf.cell2("#J3b61", "#N2478", i, nr); i <- i + 1
draw.vf.cell2("#J3b64", "#N3265", i, nr); i <- i + 1

make.table(nr, nc)
i <- 0
draw.title("Kanji (4)", nc)
draw.vf.cell2("#J3b65", "#N3492", i, nr); i <- i + 1
draw.vf.cell2("#J3b66", "#N3510", i, nr); i <- i + 1
draw.vf.cell2("#J3b6a", "#N3845", i, nr); i <- i + 1
draw.vf.cell2("#J3b73", "#N2435", i, nr); i <- i + 1
draw.vf.cell2("#J3b75", "#N5428", i, nr); i <- i + 1
draw.vf.cell2("#J3b76", "#N0272", i, nr); i <- i + 1
draw.vf.cell2("#J3b7a", "#N1281", i, nr); i <- i + 1
draw.vf.cell2("#J3b7d", "#N1903", i, nr); i <- i + 1
draw.vf.cell2("#J3b7e", "#N2126", i, nr); i <- i + 1
draw.vf.cell2("#J3c21", "#N0638", i, nr); i <- i + 1
draw.vf.cell2("#J3c27", "#N3209", i, nr); i <- i + 1
draw.vf.cell2("#J3c28", "#N3228", i, nr); i <- i + 1
draw.vf.cell2("#J3c2a", "#N3697", i, nr); i <- i + 1
draw.vf.cell2("#J3c2b", "#N3841", i, nr); i <- i + 1
draw.vf.cell2("#J3c2d", "#N3860", i, nr); i <- i + 1
draw.vf.cell2("#J3c2f", "#N5375", i, nr); i <- i + 1
draw.vf.cell2("#J3c30", "#N1556", i, nr); i <- i + 1
draw.vf.cell2("#J3c34", "#N4619", i, nr); i <- i + 1
draw.vf.cell2("#J3c37", "#N0261", i, nr); i <- i + 1
draw.vf.cell2("#J3c3c", "#N1300", i, nr); i <- i + 1
draw.vf.cell2("#J3c3e", "#N2631", i, nr); i <- i + 1
draw.vf.cell2("#J3c41", "#N4518", i, nr); i <- i + 1
draw.vf.cell2("#J3c42", "#N1297", i, nr); i <- i + 1
draw.vf.cell2("#J3c4d", "#N4603", i, nr); i <- i + 1
draw.vf.cell2("#J3c50", "#N2074", i, nr); i <- i + 1
draw.vf.cell2("#J3c54", "#N3685", i, nr); i <- i + 1
```

```
draw.vf.cell2("#J3c56", "#N4608", i, nr); i <- i + 1
draw.vf.cell2("#J3c5c", "#N1377", i, nr); i <- i + 1
draw.vf.cell2("#J3c61", "#N4809", i, nr); i <- i + 1
draw.vf.cell2("#J3c63", "#N3926", i, nr); i <- i + 1
draw.vf.cell2("#J3c67", "#N0285", i, nr); i <- i + 1
draw.vf.cell2("#J3c68", "#N3699", i, nr); i <- i + 1
draw.vf.cell2("#J3c6a", "#N1827", i, nr); i <- i + 1
draw.vf.cell2("#J3c6f", "#N3295", i, nr); i <- i + 1
draw.vf.cell2("#J3c72", "#N2573", i, nr); i <- i + 1
draw.vf.cell2("#J3c73", "#N5186", i, nr); i <- i + 1
draw.vf.cell2("#J3c7e", "#N0622", i, nr); i <- i + 1
draw.vf.cell2("#J3d29", "#N3273", i, nr); i <- i + 1
draw.vf.cell2("#J3d2a", "#N3521", i, nr); i <- i + 1
draw.vf.cell2("#J3d2e", "#N3863", i, nr); i <- i + 1
draw.vf.cell2("#J3d39", "#N4798", i, nr); i <- i + 1
draw.vf.cell2("#J3d3d", "#N0768", i, nr); i <- i + 1
draw.vf.cell2("#J3d3e", "#N1613", i, nr); i <- i + 1
draw.vf.cell2("#J3d44", "#N3597", i, nr); i <- i + 1
draw.vf.cell2("#J3d45", "#N0224", i, nr); i <- i + 1
draw.vf.cell2("#J3d50", "#N0097", i, nr); i <- i + 1
draw.vf.cell2("#J3d51", "#N1621", i, nr); i <- i + 1
draw.vf.cell2("#J3d55", "#N2122", i, nr); i <- i + 1
draw.vf.cell2("#J3d60", "#N0791", i, nr); i <- i + 1
draw.vf.cell2("#J3d63", "#N3509", i, nr); i <- i + 1
draw.vf.cell2("#J3d68", "#N1162", i, nr); i <- i + 1
draw.vf.cell2("#J3d6b", "#N2138", i, nr); i <- i + 1
draw.vf.cell2("#J3d71", "#N3719", i, nr); i <- i + 1
draw.vf.cell2("#J3d77", "#N1185", i, nr); i <- i + 1
draw.vf.cell2("#J3d7c", "#N4993", i, nr); i <- i + 1
draw.vf.cell2("#J3e26", "#N0321", i, nr); i <- i + 1
draw.vf.cell2("#J3e2e", "#N1355", i, nr); i <- i + 1
draw.vf.cell2("#J3e2f", "#N0166", i, nr); i <- i + 1
draw.vf.cell2("#J3e3d", "#N2137", i, nr); i <- i + 1
draw.vf.cell2("#J3e3e", "#N2212", i, nr); i <- i + 1
draw.vf.cell2("#J3e46", "#N2772", i, nr); i <- i + 1
draw.vf.cell2("#J3e4b", "#N3192", i, nr); i <- i + 1
draw.vf.cell2("#J3e4e", "#N3280", i, nr); i <- i + 1
draw.vf.cell2("#J3e57", "#N1638", i, nr); i <- i + 1
draw.vf.cell2("#J3e5a", "#N4341", i, nr); i <- i + 1
draw.vf.cell2("#J3e5d", "#N4472", i, nr); i <- i + 1
draw.vf.cell2("#J3e65", "#N0798", i, nr); i <- i + 1
draw.vf.cell2("#J3e68", "#N0223", i, nr); i <- i + 1
draw.vf.cell2("#J3e6c", "#N1113", i, nr); i <- i + 1
draw.vf.cell2("#J3e6f", "#N1364", i, nr); i <- i + 1
draw.vf.cell2("#J3e75", "#N2839", i, nr); i <- i + 1
draw.vf.cell2("#J3e78", "#N4002", i, nr); i <- i + 1
draw.vf.cell2("#J3f22", "#N2303", i, nr); i <- i + 1
draw.vf.cell2("#J3f27", "#N3889", i, nr); i <- i + 1
draw.vf.cell2("#J3f29", "#N5154", i, nr); i <- i + 1
draw.vf.cell2("#J3f2d", "#N0403", i, nr); i <- i + 1

make.table(nr, nc)
i <- 0
draw.title("Kanji (5)", nc)
draw.vf.cell2("#J3f34", "#N1645", i, nr); i <- i + 1
draw.vf.cell2("#J3f36", "#N1920", i, nr); i <- i + 1
draw.vf.cell2("#J3f37", "#N2080", i, nr); i <- i + 1
```

```
draw.vf.cell2("#J3f39", "#N2301", i, nr); i <- i + 1
draw.vf.cell2("#J3f3f", "#N0783", i, nr); i <- i + 1
draw.vf.cell2("#J3f43", "#N3837", i, nr); i <- i + 1
draw.vf.cell2("#J3f48", "#N4601", i, nr); i <- i + 1
draw.vf.cell2("#J3f49", "#N4646", i, nr); i <- i + 1
draw.vf.cell2("#J3f4a", "#N4709", i, nr); i <- i + 1
draw.vf.cell2("#J3f4c", "#N5055", i, nr); i <- i + 1
draw.vf.cell2("#J3f4d", "#N0339", i, nr); i <- i + 1
draw.vf.cell2("#J3f5e", "#N1034", i, nr); i <- i + 1
draw.vf.cell2("#J3f62", "#N0211", i, nr); i <- i + 1
draw.vf.cell2("#J3f65", "#N2482", i, nr); i <- i + 1
draw.vf.cell2("#J3f69", "#N3676", i, nr); i <- i + 1
draw.vf.cell2("#J3f74", "#N2057", i, nr); i <- i + 1
draw.vf.cell2("#J402d", "#N1666", i, nr); i <- i + 1
draw.vf.cell2("#J402e", "#N1799", i, nr); i <- i + 1
draw.vf.cell2("#J4030", "#N2436", i, nr); i <- i + 1
draw.vf.cell2("#J4031", "#N2121", i, nr); i <- i + 1
draw.vf.cell2("#J4032", "#N2143", i, nr); i <- i + 1
draw.vf.cell2("#J4035", "#N0027", i, nr); i <- i + 1
draw.vf.cell2("#J4038", "#N2991", i, nr); i <- i + 1
draw.vf.cell2("#J403e", "#N4273", i, nr); i <- i + 1
draw.vf.cell2("#J4044", "#N5076", i, nr); i <- i + 1
draw.vf.cell2("#J4045", "#N5077", i, nr); i <- i + 1
draw.vf.cell2("#J404e", "#N2108", i, nr); i <- i + 1
draw.vf.cell2("#J404f", "#N2194", i, nr); i <- i + 1
draw.vf.cell2("#J4050", "#N3176", i, nr); i <- i + 1
draw.vf.cell2("#J4051", "#N3306", i, nr); i <- i + 1
draw.vf.cell2("#J4056", "#N4534", i, nr); i <- i + 1
draw.vf.cell2("#J405a", "#N0667", i, nr); i <- i + 1
draw.vf.cell2("#J405c", "#N1951", i, nr); i <- i + 1
draw.vf.cell2("#J405e", "#N1855", i, nr); i <- i + 1
draw.vf.cell2("#J4063", "#N5044", i, nr); i <- i + 1
draw.vf.cell2("#J4064", "#N3539", i, nr); i <- i + 1
draw.vf.cell2("#J4065", "#N3855", i, nr); i <- i + 1
draw.vf.cell2("#J4068", "#N0571", i, nr); i <- i + 1
draw.vf.cell2("#J4069", "#N0156", i, nr); i <- i + 1
draw.vf.cell2("#J406e", "#N1447", i, nr); i <- i + 1
draw.vf.cell2("#J4070", "#N1823", i, nr); i <- i + 1
draw.vf.cell2("#J407e", "#N3580", i, nr); i <- i + 1
draw.vf.cell2("#J4125", "#N3873", i, nr); i <- i + 1
draw.vf.cell2("#J4130", "#N0595", i, nr); i <- i + 1
draw.vf.cell2("#J4133", "#N2770", i, nr); i <- i + 1
draw.vf.cell2("#J4134", "#N0384", i, nr); i <- i + 1
draw.vf.cell2("#J4147", "#N3511", i, nr); i <- i + 1
draw.vf.cell2("#J4148", "#N3520", i, nr); i <- i + 1
draw.vf.cell2("#J4150", "#N0859", i, nr); i <- i + 1
draw.vf.cell2("#J4158", "#N1402", i, nr); i <- i + 1
draw.vf.cell2("#J415b", "#N1728", i, nr); i <- i + 1
draw.vf.cell2("#J4161", "#N2100", i, nr); i <- i + 1
draw.vf.cell2("#J416a", "#N2241", i, nr); i <- i + 1
draw.vf.cell2("#J416d", "#N3567", i, nr); i <- i + 1
draw.vf.cell2("#J4170", "#N3939", i, nr); i <- i + 1
draw.vf.cell2("#J4175", "#N4234", i, nr); i <- i + 1
draw.vf.cell2("#J4176", "#N4539", i, nr); i <- i + 1
draw.vf.cell2("#J417c", "#N0540", i, nr); i <- i + 1
draw.vf.cell2("#J417d", "#N1137", i, nr); i <- i + 1
draw.vf.cell2("#J4224", "#N4701", i, nr); i <- i + 1
```

```
draw.vf.cell2("#J4226", "#N0509", i, nr); i <- i + 1
draw.vf.cell2("#J422b", "#N0196", i, nr); i <- i + 1
draw.vf.cell2("#J422c", "#N2632", i, nr); i <- i + 1
draw.vf.cell2("#J422d", "#N4546", i, nr); i <- i + 1
draw.vf.cell2("#J422e", "#N4700", i, nr); i <- i + 1
draw.vf.cell2("#J4233", "#N3544", i, nr); i <- i + 1
draw.vf.cell2("#J4236", "#N0590", i, nr); i <- i + 1
draw.vf.cell2("#J4238", "#N1267", i, nr); i <- i + 1
draw.vf.cell2("#J423e", "#N0361", i, nr); i <- i + 1
draw.vf.cell2("#J423f", "#N1169", i, nr); i <- i + 1
draw.vf.cell2("#J4240", "#N1172", i, nr); i <- i + 1
draw.vf.cell2("#J424a", "#N2313", i, nr); i <- i + 1
draw.vf.cell2("#J424e", "#N0405", i, nr); i <- i + 1
draw.vf.cell2("#J4250", "#N2067", i, nr); i <- i + 1
draw.vf.cell2("#J4256", "#N1743", i, nr); i <- i + 1
draw.vf.cell2("#J4265", "#N0364", i, nr); i <- i + 1

make.table(nr, nc)
i <- 0
draw.title("Kanji (6)", nc)
draw.vf.cell2("#J4267", "#N1171", i, nr); i <- i + 1
draw.vf.cell2("#J4268", "#N3385", i, nr); i <- i + 1
draw.vf.cell2("#J426a", "#N2164", i, nr); i <- i + 1
draw.vf.cell2("#J426c", "#N2655", i, nr); i <- i + 1
draw.vf.cell2("#J4274", "#N2503", i, nr); i <- i + 1
draw.vf.cell2("#J4323", "#N4721", i, nr); i <- i + 1
draw.vf.cell2("#J432b", "#N4458", i, nr); i <- i + 1
draw.vf.cell2("#J432f", "#N4384", i, nr); i <- i + 1
draw.vf.cell2("#J4331", "#N0139", i, nr); i <- i + 1
draw.vf.cell2("#J433a", "#N1418", i, nr); i <- i + 1
draw.vf.cell2("#J433b", "#N3172", i, nr); i <- i + 1
draw.vf.cell2("#J4346", "#N1575", i, nr); i <- i + 1
draw.vf.cell2("#J434b", "#N2996", i, nr); i <- i + 1
draw.vf.cell2("#J434d", "#N0488", i, nr); i <- i + 1
draw.vf.cell2("#J434e", "#N3169", i, nr); i <- i + 1
draw.vf.cell2("#J434f", "#N1056", i, nr); i <- i + 1
draw.vf.cell2("#J4356", "#N3644", i, nr); i <- i + 1
draw.vf.cell2("#J4359", "#N4722", i, nr); i <- i + 1
draw.vf.cell2("#J435d", "#N3366", i, nr); i <- i + 1
draw.vf.cell2("#J4362", "#N3325", i, nr); i <- i + 1
draw.vf.cell2("#J4363", "#N3940", i, nr); i <- i + 1
draw.vf.cell2("#J4365", "#N3665", i, nr); i <- i + 1
draw.vf.cell2("#J4366", "#N0081", i, nr); i <- i + 1
draw.vf.cell2("#J4368", "#N1291", i, nr); i <- i + 1
draw.vf.cell2("#J436b", "#N0053", i, nr); i <- i + 1
draw.vf.cell2("#J436c", "#N2236", i, nr); i <- i + 1
draw.vf.cell2("#J436e", "#N4115", i, nr); i <- i + 1
draw.vf.cell2("#J442b", "#N3788", i, nr); i <- i + 1
draw.vf.cell2("#J442c", "#N2702", i, nr); i <- i + 1
draw.vf.cell2("#J4436", "#N4543", i, nr); i <- i + 1
draw.vf.cell2("#J4439", "#N4938", i, nr); i <- i + 1
draw.vf.cell2("#J443b", "#N5340", i, nr); i <- i + 1
draw.vf.cell2("#J443e", "#N0775", i, nr); i <- i + 1
draw.vf.cell2("#J444c", "#N4703", i, nr); i <- i + 1
draw.vf.cell2("#J4463", "#N0406", i, nr); i <- i + 1
draw.vf.cell2("#J446a", "#N1296", i, nr); i <- i + 1
draw.vf.cell2("#J446c", "#N1508", i, nr); i <- i + 1
```

```
draw.vf.cell2("#J446d", "#N1514", i, nr); i <- i + 1
draw.vf.cell2("#J4472", "#N1914", i, nr); i <- i + 1
draw.vf.cell2("#J4478", "#N3285", i, nr); i <- i + 1
draw.vf.cell2("#J4479", "#N3581", i, nr); i <- i + 1
draw.vf.cell2("#J4526", "#N1987", i, nr); i <- i + 1
draw.vf.cell2("#J452a", "#N3097", i, nr); i <- i + 1
draw.vf.cell2("#J452f", "#N0931", i, nr); i <- i + 1
draw.vf.cell2("#J4534", "#N4844", i, nr); i <- i + 1
draw.vf.cell2("#J4535", "#N0588", i, nr); i <- i + 1
draw.vf.cell2("#J4537", "#N0016", i, nr); i <- i + 1
draw.vf.cell2("#J453e", "#N4615", i, nr); i <- i + 1
draw.vf.cell2("#J4540", "#N0804", i, nr); i <- i + 1
draw.vf.cell2("#J4544", "#N2994", i, nr); i <- i + 1
draw.vf.cell2("#J4545", "#N5050", i, nr); i <- i + 1
draw.vf.cell2("#J454c", "#N1614", i, nr); i <- i + 1
draw.vf.cell2("#J4559", "#N1511", i, nr); i <- i + 1
draw.vf.cell2("#J455a", "#N1050", i, nr); i <- i + 1
draw.vf.cell2("#J455f", "#N1161", i, nr); i <- i + 1
draw.vf.cell2("#J4561", "#N0665", i, nr); i <- i + 1
draw.vf.cell2("#J4563", "#N1109", i, nr); i <- i + 1
draw.vf.cell2("#J4567", "#N0230", i, nr); i <- i + 1
draw.vf.cell2("#J456c", "#N0213", i, nr); i <- i + 1
draw.vf.cell2("#J4574", "#N2745", i, nr); i <- i + 1
draw.vf.cell2("#J4576", "#N1359", i, nr); i <- i + 1
draw.vf.cell2("#J4579", "#N3396", i, nr); i <- i + 1
draw.vf.cell2("#J4626", "#N4465", i, nr); i <- i + 1
draw.vf.cell2("#J4630", "#N0730", i, nr); i <- i + 1
draw.vf.cell2("#J4631", "#N0619", i, nr); i <- i + 1
draw.vf.cell2("#J4633", "#N1354", i, nr); i <- i + 1
draw.vf.cell2("#J463b", "#N4724", i, nr); i <- i + 1
draw.vf.cell2("#J463c", "#N4853", i, nr); i <- i + 1
draw.vf.cell2("#J4643", "#N2860", i, nr); i <- i + 1
draw.vf.cell2("#J4649", "#N4375", i, nr); i <- i + 1
draw.vf.cell2("#J465e", "#N2160", i, nr); i <- i + 1
draw.vf.cell2("#J4662", "#N0082", i, nr); i <- i + 1
draw.vf.cell2("#J466e", "#N0778", i, nr); i <- i + 1
draw.vf.cell2("#J4671", "#N5038", i, nr); i <- i + 1
draw.vf.cell2("#J4673", "#N0273", i, nr); i <- i + 1
draw.vf.cell2("#J4679", "#N3724", i, nr); i <- i + 1

make.table(nr, nc)
i <- 0
draw.title("Kanji (7)", nc)
draw.vf.cell2("#J467c", "#N2097", i, nr); i <- i + 1
draw.vf.cell2("#J467e", "#N0574", i, nr); i <- i + 1
draw.vf.cell2("#J4721", "#N1189", i, nr); i <- i + 1
draw.vf.cell2("#J472e", "#N2797", i, nr); i <- i + 1
draw.vf.cell2("#J472f", "#N0188", i, nr); i <- i + 1
draw.vf.cell2("#J4733", "#N2808", i, nr); i <- i + 1
draw.vf.cell2("#J4734", "#N3472", i, nr); i <- i + 1
draw.vf.cell2("#J4748", "#N2529", i, nr); i <- i + 1
draw.vf.cell2("#J474f", "#N5191", i, nr); i <- i + 1
draw.vf.cell2("#J4769", "#N3275", i, nr); i <- i + 1
draw.vf.cell2("#J4772", "#N3095", i, nr); i <- i + 1
draw.vf.cell2("#J477e", "#N5385", i, nr); i <- i + 1
draw.vf.cell2("#J4821", "#N0049", i, nr); i <- i + 1
draw.vf.cell2("#J482c", "#N0577", i, nr); i <- i + 1
```

```
draw.vf.cell2("#J482f", "#N3092", i, nr); i <- i + 1
draw.vf.cell2("#J483e", "#N0132", i, nr); i <- i + 1
draw.vf.cell2("#J483f", "#N0817", i, nr); i <- i + 1
draw.vf.cell2("#J4841", "#N1469", i, nr); i <- i + 1
draw.vf.cell2("#J484c", "#N3865", i, nr); i <- i + 1
draw.vf.cell2("#J4856", "#N4811", i, nr); i <- i + 1
draw.vf.cell2("#J4860", "#N1604", i, nr); i <- i + 1
draw.vf.cell2("#J4866", "#N2470", i, nr); i <- i + 1
draw.vf.cell2("#J4869", "#N3109", i, nr); i <- i + 1
draw.vf.cell2("#J4873", "#N5080", i, nr); i <- i + 1
draw.vf.cell2("#J4874", "#N5152", i, nr); i <- i + 1
draw.vf.cell2("#J4878", "#N1383", i, nr); i <- i + 1
draw.vf.cell2("#J4879", "#N1631", i, nr); i <- i + 1
draw.vf.cell2("#J487e", "#N3658", i, nr); i <- i + 1
draw.vf.cell2("#J4921", "#N5421", i, nr); i <- i + 1
draw.vf.cell2("#J492e", "#N3397", i, nr); i <- i + 1
draw.vf.cell2("#J4934", "#N0033", i, nr); i <- i + 1
draw.vf.cell2("#J4938", "#N2359", i, nr); i <- i + 1
draw.vf.cell2("#J4939", "#N0131", i, nr); i <- i + 1
draw.vf.cell2("#J493d", "#N0108", i, nr); i <- i + 1
draw.vf.cell2("#J4942", "#N3042", i, nr); i <- i + 1
draw.vf.cell2("#J4943", "#N3271", i, nr); i <- i + 1
draw.vf.cell2("#J494a", "#N0923", i, nr); i <- i + 1
draw.vf.cell2("#J4954", "#N0017", i, nr); i <- i + 1
draw.vf.cell2("#J495b", "#N1468", i, nr); i <- i + 1
draw.vf.cell2("#J4963", "#N2832", i, nr); i <- i + 1
draw.vf.cell2("#J4969", "#N4488", i, nr); i <- i + 1
draw.vf.cell2("#J4977", "#N5148", i, nr); i <- i + 1
draw.vf.cell2("#J497d", "#N1484", i, nr); i <- i + 1
draw.vf.cell2("#J4a23", "#N4255", i, nr); i <- i + 1
draw.vf.cell2("#J4a26", "#N0173", i, nr); i <- i + 1
draw.vf.cell2("#J4a2a", "#N2857", i, nr); i <- i + 1
draw.vf.cell2("#J4a2c", "#N0578", i, nr); i <- i + 1
draw.vf.cell2("#J4a38", "#N2064", i, nr); i <- i + 1
draw.vf.cell2("#J4a39", "#N4959", i, nr); i <- i + 1
draw.vf.cell2("#J4a3f", "#N0026", i, nr); i <- i + 1
draw.vf.cell2("#J4a42", "#N0589", i, nr); i <- i + 1
draw.vf.cell2("#J4a44", "#N4945", i, nr); i <- i + 1
draw.vf.cell2("#J4a46", "#N3461", i, nr); i <- i + 1
draw.vf.cell2("#J4a50", "#N0511", i, nr); i <- i + 1
draw.vf.cell2("#J4a51", "#N0306", i, nr); i <- i + 1
draw.vf.cell2("#J4a52", "#N2842", i, nr); i <- i + 1
draw.vf.cell2("#J4a55", "#N4661", i, nr); i <- i + 1
draw.vf.cell2("#J4a6c", "#N2466", i, nr); i <- i + 1
draw.vf.cell2("#J4a7c", "#N2084", i, nr); i <- i + 1
draw.vf.cell2("#J4a7d", "#N2082", i, nr); i <- i + 1
draw.vf.cell2("#J4b21", "#N2535", i, nr); i <- i + 1
draw.vf.cell2("#J4b26", "#N3749", i, nr); i <- i + 1
draw.vf.cell2("#J4b4c", "#N0751", i, nr); i <- i + 1
draw.vf.cell2("#J4b4f", "#N5404", i, nr); i <- i + 1
draw.vf.cell2("#J4b5c", "#N0096", i, nr); i <- i + 1
draw.vf.cell2("#J4b63", "#N5390", i, nr); i <- i + 1
draw.vf.cell2("#J4b68", "#N2467", i, nr); i <- i + 1
draw.vf.cell2("#J4b74", "#N0855", i, nr); i <- i + 1
draw.vf.cell2("#J4b7c", "#N0007", i, nr); i <- i + 1
draw.vf.cell2("#J4c23", "#N0913", i, nr); i <- i + 1
draw.vf.cell2("#J4c24", "#N0179", i, nr); i <- i + 1
```

```
draw.vf.cell2("#J4c29", "#N1316", i, nr); i <- i + 1
draw.vf.cell2("#J4c35", "#N2773", i, nr); i <- i + 1
draw.vf.cell2("#J4c37", "#N3164", i, nr); i <- i + 1
draw.vf.cell2("#J4c3e", "#N1170", i, nr); i <- i + 1
draw.vf.cell2("#J4c40", "#N2110", i, nr); i <- i + 1

make.table(nr, nc)
i <- 0
draw.title("Kanji (8)", nc)
draw.vf.cell2("#J4c4c", "#N5087", i, nr); i <- i + 1
draw.vf.cell2("#J4c53", "#N2473", i, nr); i <- i + 1
draw.vf.cell2("#J4c5a", "#N2170", i, nr); i <- i + 1
draw.vf.cell2("#J4c5c", "#N3127", i, nr); i <- i + 1
draw.vf.cell2("#J4c64", "#N4944", i, nr); i <- i + 1
draw.vf.cell2("#J4c67", "#N4940", i, nr); i <- i + 1
draw.vf.cell2("#J4c6b", "#N0298", i, nr); i <- i + 1
draw.vf.cell2("#J4c70", "#N3168", i, nr); i <- i + 1
draw.vf.cell2("#J4c72", "#N1598", i, nr); i <- i + 1
draw.vf.cell2("#J4c74", "#N4074", i, nr); i <- i + 1
draw.vf.cell2("#J4c78", "#N2233", i, nr); i <- i + 1
draw.vf.cell2("#J4c7d", "#N2534", i, nr); i <- i + 1
draw.vf.cell2("#J4d2d", "#N3727", i, nr); i <- i + 1
draw.vf.cell2("#J4d30", "#N2565", i, nr); i <- i + 1
draw.vf.cell2("#J4d3a", "#N5030", i, nr); i <- i + 1
draw.vf.cell2("#J4d3c", "#N1167", i, nr); i <- i + 1
draw.vf.cell2("#J4d3e", "#N0408", i, nr); i <- i + 1
draw.vf.cell2("#J4d4f", "#N2659", i, nr); i <- i + 1
draw.vf.cell2("#J4d51", "#N2993", i, nr); i <- i + 1
draw.vf.cell2("#J4d53", "#N3656", i, nr); i <- i + 1
draw.vf.cell2("#J4d55", "#N4001", i, nr); i <- i + 1
draw.vf.cell2("#J4d57", "#N4274", i, nr); i <- i + 1
draw.vf.cell2("#J4d5b", "#N5012", i, nr); i <- i + 1
draw.vf.cell2("#J4d63", "#N3680", i, nr); i <- i + 1
draw.vf.cell2("#J4d68", "#N0202", i, nr); i <- i + 1
draw.vf.cell2("#J4d6b", "#N5049", i, nr); i <- i + 1
draw.vf.cell2("#J4d70", "#N3856", i, nr); i <- i + 1
draw.vf.cell2("#J4d71", "#N0199", i, nr); i <- i + 1
draw.vf.cell2("#J4d72", "#N1431", i, nr); i <- i + 1
draw.vf.cell2("#J4d78", "#N3264", i, nr); i <- i + 1
draw.vf.cell2("#J4d7d", "#N2942", i, nr); i <- i + 1
draw.vf.cell2("#J4e24", "#N4813", i, nr); i <- i + 1
draw.vf.cell2("#J4e25", "#N5040", i, nr); i <- i + 1
draw.vf.cell2("#J4e26", "#N5005", i, nr); i <- i + 1
draw.vf.cell2("#J4e28", "#N0319", i, nr); i <- i + 1
draw.vf.cell2("#J4e29", "#N3343", i, nr); i <- i + 1
draw.vf.cell2("#J4e2e", "#N2576", i, nr); i <- i + 1
draw.vf.cell2("#J4e32", "#N3191", i, nr); i <- i + 1
draw.vf.cell2("#J4e33", "#N3471", i, nr); i <- i + 1
draw.vf.cell2("#J4e35", "#N5440", i, nr); i <- i + 1
draw.vf.cell2("#J4e3e", "#N0034", i, nr); i <- i + 1
draw.vf.cell2("#J4e41", "#N3468", i, nr); i <- i + 1
draw.vf.cell2("#J4e49", "#N3885", i, nr); i <- i + 1
draw.vf.cell2("#J4e4c", "#N2141", i, nr); i <- i + 1
draw.vf.cell2("#J4e4f", "#N0715", i, nr); i <- i + 1
draw.vf.cell2("#J4e53", "#N2210", i, nr); i <- i + 1
draw.vf.cell2("#J4e55", "#N2807", i, nr); i <- i + 1
draw.vf.cell2("#J4e58", "#N4630", i, nr); i <- i + 1
```

```
draw.vf.cell2("#J4e60", "#N5138", i, nr); i <- i + 1
draw.vf.cell2("#J4e63", "#N0428", i, nr); i <- i + 1
draw.vf.cell2("#J4e64", "#N0642", i, nr); i <- i + 1
draw.vf.cell2("#J4e6d", "#N5048", i, nr); i <- i + 1
draw.vf.cell2("#J4e6e", "#N5056", i, nr); i <- i + 1
draw.vf.cell2("#J4e73", "#N2438", i, nr); i <- i + 1
draw.vf.cell2("#J4f22", "#N4702", i, nr); i <- i + 1
draw.vf.cell2("#J4f27", "#N2750", i, nr); i <- i + 1
draw.vf.cell2("#J4f29", "#N4561", i, nr); i <- i + 1
draw.vf.cell2("#J4f37", "#N3683", i, nr); i <- i + 1
draw.vf.cell2("#J4f3b", "#N0283", i, nr); i <- i + 1
draw.vf.cell2("#J4f40", "#N4391", i, nr); i <- i + 1
draw.vf.cell2("#J4f42", "#N3268", i, nr); i <- i + 1
draw.vf.cell2("#J4f43", "#N4358", i, nr); i <- i + 1
draw.vf.cell2("#J4f44", "#N0054", i, nr); i <- i + 1
draw.vf.cell2("#J4f47", "#N1710", i, nr); i <- i + 1

draw.vf.cell2("#J534c", "#N0973", i, nr); i <- i + 1
draw.vf.cell2("#J5879", "#N1794", i, nr); i <- i + 1
draw.vf.cell2("#J5960", "#N1942", i, nr); i <- i + 1
draw.vf.cell2("#J626f", "#N3200", i, nr); i <- i + 1
draw.vf.cell2("#J6446", "#N3458", i, nr); i <- i + 1
draw.vf.cell2("#J6647", "#N5083", i, nr); i <- i + 1
draw.vf.cell2("#J6d55", "#N4633", i, nr); i <- i + 1

par(oldpar)
```

`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 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.

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, ...)
kappa.lm     (z, ...)
kappa.default(z, exact = FALSE, ...)
kappa.qr     (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.

Value

The condition number, *kappa*, or an approximation if `exact=FALSE`.

Author(s)

B.D. Ripley

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

Matrix Algebra Useful for Statistics, Shayle R. Searle, John Wiley and Sons, 1982.

See Also

`outer`, on which `kronecker` is built and `matmult` for usual matrix multiplication.

Examples

```
# simple scalar multiplication
( M <- matrix(1:6, ncol=2) )
stopifnot(kronecker(4, M)==4 * M)
# Block diagonal matrix:
stopifnot(kronecker(diag(1, 3), M) == diag(1, 3) %x% 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)
```

`labels` *Find Labels from Object*

Description

Find a suitable set of labels from an object for use in printing or plotting, for example.

Usage

```
labels(object, ...)
labels.default(object, ...)
labels.terms(object, ...)
labels.lm(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)`, for a data frame or array it is the dimnames (with `NULL` expanded to `seq(len=d[i])`), for a `terms` object it is the term labels and for an `lm` object it is the term labels for estimable terms.

Author(s)

B.D. Ripley

`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` also accepting vectors as `X`, and returning a vector or array with `dimnames` if appropriate.

Usage

```
lapply(X, FUN, ...)
sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)
```

Arguments

`X`	list or vector to be used.
`FUN`	the function to be applied. In the case of functions like `+`, `%*%`, etc., the function name must be quoted.
`...`	optional arguments to `FUN`.
`simplify`	logical; should the result be simplified to a vector if possible?
`USE.NAMES`	logical; if `TRUE` and if `X` is character, use `X` as `names` for the result unless it had names already.

See Also

`apply`, `tapply`.

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)
str(i39 <- sapply(3:9, seq))# list of vectors
sapply(i39, fivenum)
```

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("eda") # returns invisibly
.Last.value    # shows what library(.) above returned
```

`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, dim(mat)[2]),
       heights= rep(1, dim(mat)[1]),
       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 intepreted 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.

`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.

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.

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` can be used in place of the `x` and `y` arguments.

Usage

```
legend(x, y, legend, fill, col = "black", lty, lwd, pch,
       angle = NULL, density = NULL,
       bty = "o", bg = par("bg"), pt.bg = NA, cex = 1, xjust = 0, yjust = 1,
       x.intersp = 1, y.intersp = 1, adj = 0,
       text.width = NULL, merge = do.lines && has.pch, trace = FALSE,
       ncol = 1, horiz = FALSE)
```

Arguments

`x,y`	the x and y location of the legend. `x` can be a list with `x` and `y` components.
`legend`	a vector of text values or an `expression` of length ≥ 1 to appear in the legend.
`fill`	if specified, this argument will cause boxes filled with 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.
`bty`	the type of box to be drawn around the legend.
`bg`	the background color for the legend box.
`pt.bg`	the background color for the `points`.
`cex`	character expansion factor **relative** to current `par("cex")`.
`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. Defaults to the proper value computed by `strwidth(legend)`.

`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.

`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).

Details

"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.

Value

A (`invisible`) 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).

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),
       lty = c(2, -1, 1), pch = c(-1, 3, 4), merge = TRUE, bg='gray90')

##--- 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:  ------------------------------
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
str(legend(-3, .9, ex.cs1, lty=1:2, col=2:3, adj = c(0, .6)))# adj y !

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))
str(legend(4.1, 30, ex12, col = 2:3, lty=2:3, lwd=2))

## 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=.8)
legend(.8,1.2, paste("sin(",1:7,"pi * x)"), col=1:7, lty=1:7, pch = "*",cex=.8)
legend(0, -.1, paste("sin(",1:4,"pi * x)"), col=1:4, lty=1:4, ncol=2, cex=.8)
legend(0, -.4, paste("sin(",5:7,"pi * x)"), col=5:7,  pch=24, ncol=2, cex=1.5,
       pt.bg="pink")
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")
```

`length` *Length of a Vector or List*

Description

Get or set the length of vectors (including lists).

Usage

```
length(x)
length(x) <- n
```

Arguments

`x`	a vector or list.
`n`	an integer.

Details

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 `NA`s.

Value

The length of `x` as an `integer` of length 1, if `x` is (or can be coerced to) a vector or list. Otherwise, `length` returns `NA`.

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
```

`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 assignment 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.

See Also

`levels<-.factor`, `nlevels`.

levels.factor *Factor Levels Assignment*

Description

`levels<-` provides a way to alter the levels attribute of factor. `value` can be a vector of character strings with length equal to the number of levels of `x`, or a named list specifying how to rename the levels.

Usage

```
levels(x) <- value
```

See Also

`factor`, `levels`, `levels<-`, `nlevels`.

Examples

```
# assign individual levels
x <- gl(2, 4, 8)
levels(x)[1] <- "low"
levels(x)[2] <- "high"

# or as a group
y <- gl(2, 4, 8)
levels(y) <- c("low", "high")

# combine some levels
z <- gl(3, 2, 12)
levels(z) <- c("A", "B", "A")

# same, using a named list
z <- gl(3, 2, 12)
levels(z) <- list(A=c(1,3), B=2)
```

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. `.packages` returns information about package availability. `.path.package` returns information about where a package was loaded from. `.find.package` returns the directory paths of installed packages.

Usage

```
library(package, help, lib.loc = NULL, character.only = FALSE,
        logical.return = FALSE, warn.conflicts = TRUE,
        keep.source = getOption("keep.source.pkgs"),
        verbose = getOption("verbose"))
require(package, quietly = FALSE, warn.conflicts = TRUE,
        keep.source = getOption("keep.source.pkgs"))

.First.lib(libname, pkgname)
.Last.lib(libpath)

.packages(all.available = FALSE, lib.loc = NULL)
.path.package(package = .packages(), quiet = FALSE)
.find.package(package, lib.loc = NULL, use.attached, quiet = FALSE,
              verbose = getOption("verbose"))
.libPaths(new)

.Library
.Autoloaded
```

Arguments

`package, help`	name or character string giving the name of a package.
`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.
`character.only`	a logical indicating whether `package` or `help` can be assumed to be character strings.
`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`.
`keep.source`	logical. If `TRUE`, functions "keep their source" including comments, see argument `keep.source` to `options`.
`verbose`	a logical. If `TRUE`, additional diagnostics are printed.
`quietly`	a logical. If `TRUE`, a warning will not be printed if the package cannot be found.
`libname`	a character string giving the library directory where the package was found.
`pkgname`	a character string giving the name of the package.
`libpath`	a character string giving the complete path to the package.
`all.available`	logical; if `TRUE` return a character vector of all available packages in `lib.loc`.
`quiet`	logical. For `.path.package`, should this not give warnings or an error if the package(s) are not loaded? For `.find.package`, should this not give warnings or an error if the package(s) are not found?

`use.attached` a logical indicating whether attached packages should be considered in addition to the ones installed in the given libraries.

`new` a character vector with the locations of R library trees.

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 optionally gives a warning, rather than giving an error, if the package does not exist. Both functions check and update the list of currently loaded packages and do not reload code that is already loaded.

For large packages, setting `keep.source = FALSE` may save quite a bit of memory.

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.

`library(help = somename)` prints information on the package `somename`, typically by listing the most important user level objects it contains.

`.First.lib` is called when a package is loaded by `library`. 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). 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.

`.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.)

`.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`.

`.path.package` returns the paths from which the named packages were loaded, or if none were named, for all currently loaded packages. Unless `quiet = TRUE` it will warn if some of the packages named are not loaded, and given an error if none are. This function is not meant to be called by users, and its interface might change in future versions.

`.find.package` returns the paths to the locations where the given packages can be found. If `lib.loc` is `NULL`, then then attached packages are searched before the libraries. If a package is found more than once, the first match is used. Unless `quiet = TRUE` a warning will be given about the named packages which are not found, and an error if none are. If `verbose` is true, warnings about packages found more than once are given. Argument `use.attached` is deprecated. This function is not meant to be called by users, and its interface might change in future versions.

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

`.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 `unique(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 semicolon-separated list of directories at which R library trees are rooted) by calling `.libPaths` with the directories specified in `R_LIBS`. Currently, the global variable `.lib.loc` is used for storing the paths to the known library trees; however, this variable should not be accessed directly.

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.

Author(s)

R core; Guido Masarotto for the `all.available=TRUE` part of `.packages`.

See Also

`attach`, `detach`, `search`, `objects`, `autoload`, `library.dynam`, `data`, `install.packages`, `INSTALL`, `REMOVE`.

Examples

```
(.packages())               # maybe just "base"
.packages(all = TRUE)       # return all available as character vector
library()                   # list all available packages
library(lib = .Library)     # list all packages in the default library
library(help = eda)         # documentation on package 'eda'
library(eda)                # load package 'eda'
require(eda)                # the same
(.packages())               # "eda", too
detach("package:eda")

# if the package name is in a character vector, use
pkg <- "eda"
library(pkg, character.only = TRUE)
detach(pos = match(paste("package", pkg, sep=":"), search()))

.path.package()
.Autoloaded                 # maybe "ctest"

.libPaths()                 # all library trees R knows about

require(nonexistent)        # FALSE
## 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)
}
```

`library.dynam` *Loading Shared Libraries*

Description

Load the specified file of compiled code if it has not been loaded already.

Usage

```
library.dynam(chname, package = .packages(), lib.loc = NULL,
              verbose = getOption("verbose"), file.ext, ...)
```

Arguments

`chname`	a character string naming a shared library to load.
`package`	a character vector with the names of packages to search through.
`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.
`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.

Details

This is designed to be used inside a package rather than at the command line, and should really only be used inside `.First.lib()`. The system-specific extension for shared libraries ('.dll' on Windows) should not be added.

Value

The `.Dyn.libs` vector with the names of packages which have used `library.dynam` in the current R session.

It is returned as `invisible`, unless the `chname` argument is missing.

Users should never set `.Dyn.libs` directly.

See Also

`.First.lib`, `library`, `dyn.load`, `.packages`, `.libPaths`

SHLIB for how to create suitable DLLs.

Examples

```
library.dynam()# which packages have been ‘‘dynamically loaded’’
```

`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'.

`LifeCycleSavings` *Intercountry Life-Cycle Savings Data*

Description

Data on the savings ratio 1960–1970.

Usage

```
data(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., E. Kuh. E. and Welsch, R. E. (1980) *Regression Diagnostics.* New York: Wiley.

Examples

```
data(LifeCycleSavings)
pairs(LifeCycleSavings, panel = panel.smooth,
      main = "LifeCycleSavings data")
fm1 <- lm(sr ~ pop15 + pop75 + dpi + ddpi, data = LifeCycleSavings)
summary(fm1)
```

---

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, ...)
lines.default(x, y=NULL, type="l", col=par("col"), lty=par("lty"), ...)
```

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`.
`col`	color to use.
`lty`	line type to use.
`...`	Further graphical parameters (see `par`) may also be supplied as arguments, particularly, line type, `lty` and line width, `lwd`.

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` it 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.

See Also

`points`, `plot`, and the underlying "primitive" `plot.xy`.

Examples

```
data(cars)
# draw a smooth line through a scatter plot
plot(cars, main="Stopping Distance versus Speed")
lines(lowess(cars))
```

`link.html.help` *Update HTML documentation files*

Description

Functions to re-create the HTML documentation files to reflect all installed packages. `make.packages.html` creates the list of packages and `make.search.html` creates the database used by the Java search engine. `link.html.help` runs both

Usage

```
link.html.help(verbose=FALSE, lib.loc=.Library)
make.packages.html(lib.loc=.libPaths())
make.search.html(lib.loc=.Library)
```

Arguments

`verbose`	logical. If true, print out a message. For use to explain a delay when called from other functions.
`lib.loc`	character vector. List of libraries to be included.

Details

Only packages in the system library `.Library` are included in the search database by default, as cross-library links do not work on this platform.

This will give a warning if the files are not writeable.

Value

No value is returned. The file(s) 'R_HOME\doc\html\packages.html' and/or 'R_HOME\doc\html\search\index.txt' are (re-)created.

Note

You will need write permission in the 'R_HOME\doc\html' directory.

Author(s)

Guido Masarotto and Brian Ripley

`list` *Lists – Generic and Dotted Pairs*

Description

Functions to construct, coerce and check for all kinds of R lists.

Usage

```
list(...)
pairlist(...)

as.list(x, ...)
as.list.default(x, ...)
as.pairlist(x)

is.list(x)
is.pairlist(x)

alist(...)
```

Arguments

`...`	objects.
`x`	object to be coerced or tested.

Details

Most lists in R internally are *Generic Vectors*, whereas traditional *dotted pair* lists (as in LISP) are still available.

The arguments to `list` or `pairlist` are of the form `value` or `tag=value`. The functions return a list composed of its arguments with each value either tagged or untagged, depending on how the argument was specified.

`alist` is like `list`, except in the handling of tagged arguments with no value. These are handled as if they described function arguments with no default (cf. `formals`), whereas `list` simply ignores them.

`as.list` attempts to coerce its argument to list type. For functions, this returns the concatenation of the list of formals arguments and the function body. For expressions, the list of constituent calls is returned.

`is.list` returns `TRUE` iff its argument is a `list` *or* a `pairlist` of `length`> 0, whereas `is.pairlist` only returns `TRUE` in the latter case.

An empty pairlist, `pairlist()` is the same as `NULL`. This is different from `list()`.

See Also

`vector(., mode="list")`, `c`, for concatenation; `formals`.

Examples

```
data(cars)
# create a plotting structure
pts <- list(x=cars[,1], y=cars[,2])
plot(pts)

# Argument lists
f <- function()x
# Note the specification of a "..." argument:
formals(f) <- al <- alist(x=, y=2, ...=)
f
str(al)

str(pl <- as.pairlist(ps.options()))

## These are all TRUE:
is.list(pl) && is.pairlist(pl)
!is.null(list())
is.null(pairlist())
!is.list(NULL)
is.pairlist(pairlist())
is.null(as.pairlist(list()))
is.null(as.pairlist(NULL))
```

---

list.files *List the Files in a Directory/Folder*

Description

This function produces a list containing the names of files in the named directory. `dir` is an alias.

Usage

```
list.files(path = ".", pattern=NULL, all.files=FALSE, full.names=FALSE)
       dir(path = ".", pattern=NULL, all.files=FALSE, full.names=FALSE)
```

Arguments

`path`	a character vector of full path names.
`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.

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.

Author(s)

Ross Ihaka

See Also

`file.info`, `file.access` and `files` for many more file handling functions.

Examples

```
list.files(R.home())
## Only files starting with a-l or r (*including* uppercase):
dir("../..", pattern = "^[a-lr]",full.names=TRUE)
```

---

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 = NULL, ...)
```

Arguments

`formula`	a symbolic description of the model to be fit. The details of model specification are given below.
`data`	an optional data frame containing the variables in the model. By default 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. If specified, 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 `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`.
`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, defaulting to `TRUE`. *`FALSE` is not yet implemented.*

`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. An `offset` term can be included in the formula instead or as well, and if both are specified their sum is used.

`...`
: 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`.

`lm` calls the lower level functions `lm.fit`, etc, see below, for the actual numerical computations. For programming only, you may consider doing likewise.

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.

`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`.

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.

See Also

`summary.lm` for summaries and `anova.lm` for the ANOVA table; `aov` for a different interface.

The generic functions `coefficients`, `effects`, `residuals`, `fitted.values`.

`predict.lm` (via `predict`) for prediction, including confidence and prediction intervals.

`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.

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)))
```

---

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, ...)
lm.wfit(x, y, w, offset = NULL, method = "qr", tol = 1e-7, ...)
lm.fit.null (x, y,     method = "qr", tol = 1e-7, ...)
lm.wfit.null(x, y, w, method = "qr", tol = 1e-7, ...)
```

Arguments

`x`	design matrix of dimension `n * p`.
`y`	vector of observations of length `n`.
`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.
`...`	currently disregarded.

Details

The functions `lm.{w}fit.null` are called by `lm.fit` or `lm.wfit` respectively, when `x` has zero columns.

Value

a list with components

`coefficients`	`p` vector
`residuals`	`n` vector
`fitted.values`	`n` vector
`effects`	`n` vector; .....
`weights`	`n` vector — *only* for the `*wfit*` functions.
`rank`	integer, giving the rank
`df.residual`	degrees of freedom of residuals
`qr`	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))

str(lm0 <-  lm.fit.null (x=X, y=y))
str(lmw0 <- lm.wfit.null(x=X, y=y,w=w))
```

`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

```
lm.influence(lm.obj)
```

Arguments

`lm.obj`	an object as returned by `lm`.

Details

The `influence.measures()` and other functions listed in **See Also** provide a more user oriented way of computing a variety of regression diagnostics.

Value

A list containing the following components:

`hat`	a vector containing the diagonal of the "hat" matrix.
`coefficients`	the change in the estimated coefficients which results when the i-th case is dropped from the regression is contained in the i-th row of this 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.

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.

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

Belsley, D. A., Kuh, E. and Welsch, R. E. (1980) *Regression Diagnostics.* New York: Wiley.

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.
data(LifeCycleSavings)
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

```
coefficients(object, ...) ; \method
```

Arguments

`object, x`	an object of class `lm`, usually, a result of a call to `lm`.
`...`	further arguments passed to or from other methods.

Details

The generic accessor functions `coefficients`, `effects`, `fitted.values` and `residuals` can be used to extract various useful features of the value returned by `lm`.

See Also

The model fitting function `lm`, `anova.lm`.

`coefficients`, `deviance`, `df.residual`, `effects`, `fitted.values`, `glm` for **generalized** linear models, `lm.influence` 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))
```

`lm.summary` *Summarizing Linear Model Fits*

Description

`summary` method for class `"lm"`.

Usage

```
summary(object, correlation = FALSE, ...)

print(x, digits = max(3, getOption("digits") - 3),
      symbolic.cor = p > 4,
      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`.

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.
`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 residual, `residuals[i]`.
`df`	degrees of freedom, a 3-vector $(p, n-p, p*)$.

`fstatistic` 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.

See Also

The model fitting function `lm`, `summary`.

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
```

`load` *Reload Saved Datasets*

Description

This function will reload the datasets written to a file with the function `save`.

Usage

```
load(file, envir = parent.frame())
```

Arguments

`file` a connection or a character string giving the name of the file to load.

`envir` the environment where the data should be loaded.

See Also

`save`.

Examples

```
## save all data
save(list = ls(), file= "all.Rdata")

## restore the saved values to the current environment
load("all.Rdata")

## restore the saved values to the workspace
load("all.Rdata", .GlobalEnv)
```

`localeconv` *Find Details of the Numerical Representations in the Current Locale*

Description

Get details of the numerical representations in the current locale.

Usage

```
Sys.localeconv()
```

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

`locales` for ways to set locales: by default R uses the C clocal for `"LC_NUMERIC"` and `"LC_MONETARY"`.

Examples

```
Sys.localeconv()
## The results in the default 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").
old <- Sys.getlocale()
Sys.setlocale(locale = "")
Sys.localeconv()
Sys.setlocale(locale = old)

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. Must be one of `"LC_ALL"`, `"LC_COLLATE"`, `"LC_CTYPE"`, `"LC_MONETARY"`, `"LC_NUMERIC"` or `"LC_TIME"`.
`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 does set `"LC_COLLATE"` and `"LC_CTYPE"`, which allow the use of a different character set (typically ISO Latin 1) and alphabetic comparisons in that character set (including the use of `sort`) 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.

Some systems will have other locale categories, but the six described here are those specified by POSIX.

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). For portability, it is best to query categories individually. It is guaranteed that the result of `foo <- Sys.getlocale()` can used in `Sys.setlocale("LC_ALL", locale = foo)` on the same machine.

Warning

Setting `"LC_NUMERIC"` can produce output that R cannot then read by `scan` or `read.table` with their default arguments, which are not locale-specific.

See Also

`strptime` for uses of `category = "LC_TIME"`. `localeconv` for details of numerical representations.

Examples

```
Sys.getlocale()
Sys.getlocale("LC_TIME")

Sys.setlocale("LC_TIME", "de")     # Solaris: details are OS-dependent
Sys.setlocale("LC_TIME", "German") # Windows

Sys.setlocale("LC_COLLATE", "C")  # turn off locale-specific sorting
```

`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.
`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

Unless the process is terminated prematurely by the user (see below) at most `n` positions are determined.

The identification process can be terminated by clicking the second button and selecting 'Stop' from the menu, or from the 'Stop' menu on the graphics window.

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.

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.

See Also

`identify`

`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 `log(x, base)` computes logarithms with base `base` (`log10` and `log2` are only special cases).

`log1p(x)` computes $\log(1+x)$ accurately also for $|x| \ll 1$ (and less accurately when $x \approx -1$).

`exp` computes the exponential function.

Usage

```
log(x, base = exp(1))
log10(x)
log2(x)
exp(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)`.

Value

A vector of the same length as `x` containing the transformed values. `log(0)` gives `-Inf` (when available).

See Also

`Trig`, `sqrt`, `Arithmetic`.

Examples

```
log(exp(3))
all.equal(log(1:10), log(1:10, exp(1)))
log10(30) == log(30, 10)
log10(1e7)# = 7
log2(2^pi) == 2^log2(pi)
Mod(pi - log(exp(pi*1i)) / 1i) < .Machine$double.eps
Mod(1+exp(pi*1i)) < .Machine$double.eps

x <- 10^-(1+2*1:9)
cbind(x, log(1+x), log1p(x))
```

`Logic` *Logical Operators*

Description

These operators act on logical vectors.

Usage

```
! x
x & y
x && y
x | y
x || y
xor(x, y)
```

Arguments

`x, y`	logical vectors

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.

`xor` indicates elementwise exclusive OR.

See Also

`TRUE` or `logical`.

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"
```

`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.

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) and not the `codes`.

`is.logical` returns `TRUE` or `FALSE` depending on whether its argument is of logical type or not.

`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.

Examples

```
eps <- 100 * .Machine$double.eps
x <- c(0:4, rlogis(100))
all.equal(plogis(x),                 1 / (1 + exp(-x)), tol = eps)
all.equal(plogis(x, lower=FALSE),   exp(-x)/ (1 + exp(-x)), tol = eps)
all.equal(plogis(x, lower=FALSE, log=TRUE), -log(1 + exp(x)),  tol = eps)
all.equal(dlogis(x), exp(x) * (1 + exp(x))^-2, tol = eps)

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: `lm`, `nls` in package **nls**, and `gls`, `lme` and others in package **nlme**.

Usage

```
logLik(object, ...)

as.data.frame(x, row.names = NULL, optional = 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.
`x`	an object of class `logLik`.
`row.names, optional`	arguments to the `as.data.frame` method; see its documentation.

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 parameters in the model. There's 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

See Also

`logLik.lm`, `logLik.nls`, `logLik.gls`, `logLik.lme`, etc.

Examples

```
## see the method function documentation
x <- 1:5
lmx <- lm(x ~ 1)
logLik(lmx) # using print.logLik() method
str(logLik(lmx))
```

`logLik.glm` *Extract Log-Likelihood from an glm Object*

Description

Returns the log-likelihood value of the generalized linear model represented by `object` evaluated at the estimated coefficients.

Usage

```
logLik(object, ...)
```

Arguments

`object`	an object inheriting from class `"glm"`.
`...`	further arguments to be passed to or from methods.

Value

the log-likelihood of the linear model represented by `object` evaluated at the estimated coefficients.

See Also

`glm`, `logLik.lm`

Examples

```
##-- Continuing the  glm(.) example:
Eq <- function(x,y) all.equal(x,y, tol = 1e-12)
stopifnot(Eq(AIC(anorex.1), anorex.1$aic),
          Eq(AIC(g1), g1$aic),
          Eq(AIC(g2), g2$aic))
```

---

logLik.lm *Extract Log-Likelihood from an lm Object*

Description

If `REML=FALSE`, returns the log-likelihood value of the linear model represented by `object` evaluated at the estimated coefficients; else, the restricted log-likelihood evaluated at the estimated coefficients is returned.

Usage

```
logLik(object, REML = FALSE, ...)
```

Arguments

`object`	an object inheriting from class `"lm"`.
`REML`	an optional logical value. If `TRUE` the restricted log-likelihood is returned, else, if `FALSE`, the log-likelihood is returned. Defaults to `FALSE`.
`...`	further arguments to be passed to or from methods.

Value

the (restricted) log-likelihood of the linear 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.

See Also

`lm`

Examples

```
data(attitude)
(fm1 <- lm(rating ~ ., data = attitude))
logLik(fm1)
logLik(fm1, REML = TRUE)

Nnlme <- is.na(match("package:nlme", search()))
if(require(nlme)) {
  data(Orthodont)
  fm1 <- lm(distance ~ Sex * age, Orthodont)
  print(logLik(fm1))
  print(logLik(fm1, REML = TRUE))
  if(Nnlme) detach( "package:nlme")
}
```

`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 term 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

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

```
data(HairEyeColor)
## 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)`.

See Also

`dnorm` for the normal distribution.

Examples

```
dlnorm(1) == dnorm(0)
x <- rlnorm(1000)       # not yet always :
all(abs(x  -  qlnorm(plnorm(x))) < 1e4 * .Machine$double.eps * x)
```

---

longley *Longley's Regression Data*

Description

A macroeconomic data set which provides a well-known example for a highly collinear regression.

Usage

```
data(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.

Examples

```
## give the data set in the form it is used in S-PLUS:
data(longley)
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)
```

---

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 <- ma <- matrix(1:20, 4, 5)
m2[lower.tri(m2)] <- NA
m2
stopifnot(lower.tri(ma) == !upper.tri(ma, diag=TRUE))
```

`lowess` *Scatter Plot Smoothing*

Description

This function performs the computations for the *LOWESS* smoother (see the reference below). `lowess` returns a list containing components `x` and `y` which give the coordinates of the smooth. The smooth should be added to a plot of the original points with the function `lines`.

Usage

```
lowess(x, y, f=2/3, iter=3, delta=.01*diff(range(x)))
```

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`	values of `x` which lie within `delta` of each other replaced by a single value in the output from `lowess`.

References

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` (in package `modreg`), a newer formula based version of `lowess` (with different defaults!).

Examples

```
data(cars)
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` *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, see `grep`. Only names matching `pattern` are returned.

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.

See Also

`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"
```

---

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 matrix valued 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`	an unused parameter for compatibility.

Details

If weights are specified then a weighted least squares is performed with the weight given to the jth case specified by the jth 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 (stated below).
`residuals`	residuals from the fit.
`intercept`	indicates whether an intercept was fitted.
`qr`	the QR decomposition of the design matrix.

See Also

`lm` which usually is preferable; `ls.print`, `ls.diag`.

Examples

```
##-- Using the same data as the lm(.) example:
lsD9 <- lsfit(x = codes(gl(2,10)), y = weight)
ls.print(lsD9)
```

`Machine` *Machine Characteristics*

Description

`Machine()` returns information on numeric characteristics of the machine R is running on, such as the largest double or integer and the machine's precision.

`.Machine` is a variable holding this information.

Usage

```
Machine()
.Machine
```

Details

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

Value

`Machine()` returns 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.

References

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

See Also

`machine` to determine the computer type which R is running on.

Examples

```
str(Machine())
(Meps <- .Machine $ double.eps)

## All the following relations must hold :
stopifnot(
 1 +     Meps != 1,
 1 + .5* Meps == 1,
```

```
 log2(.Machine$double.xmax) == .Machine$double.max.exp,
 log2(.Machine$double.xmin) == .Machine$double.min.exp
)
```

`machine` *Determine the Machine R is Running On*

Description

This function returns a character string which specifies what kind of environment R is being run in.

Usage

```
machine()
```

See Also

`.Platform` which provides more than `machine()`; `Machine` for the computer's characteristics in arithmetics;

Examples

```
machine()# to see yours
tolower(machine()) == .Platform $ OS ## --> often TRUE
if (machine() == "Macintosh")
  cat("You are using a Macintosh computer\n")
```

`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: defauls 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 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, tol.inv = 1e-7)
```

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.
`tol.inv`	tolerance to be used for computing the inverse (if `inverted` is false), see `solve`.

Author(s)

Friedrich Leisch

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))) == apply(x*x, 1, sum))
        ##- Here, D^2 = usual Euclidean distances
Sx <- cov(x)
D2 <- mahalanobis(x, apply(x, 2, mean), Sx)
plot(density(D2, bw=.5), main="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 number, say λ resulting in power link $= \mu^\lambda$.

Value

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 all of eta is in the domain of linkinv }.

See Also

glm, family.

Examples

```
str(make.link("logit"))

l2 <- make.link(2)
l2$linkfun(0:3)# 0 1 4 9
l2$mu.eta(eta= 1:2)#= 1/(2*sqrt(eta))
```

---

make.names *Make Syntactically Valid Names*

Description

Make syntactically valid names out of character vectors.

Usage

```
make.names(names, unique = FALSE)
```

Arguments

names	character (vector) to be coerced to syntactically valid names.
unique	logical; if TRUE, the resulting elements are unique. This may be desired for, e.g., column names.

Details

A syntactically valid name consists of letters, numbers, and the dot character and starts with a letter or the dot.

All invalid characters are translated to ".".

Value

A character vector of same length as names with each changed to a syntactically valid name.

See Also

names, character, data.frame.

Examples

```
make.names(c("a and b", "a_and_b"), unique=TRUE)#-> "a.and.b"  "a.and.b1"

all(make.names(letters) == letters)# TRUE

data(state)
state.name[make.names(state.name) != state.name]# those 10 with a space
```

`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
daytime("tick.usno.navy.mil")
```

---

make.tables *Create model.tables*

Description

These are support functions for (the methods of) `model.tables` and probably not much of use otherwise.

Usage

```
make.tables.aovproj    (proj.cols, mf.cols, prjs, mf,
                        fun = "mean", prt = FALSE, ...)
make.tables.aovprojlist(proj.cols, strata.cols, model.cols, projections,
                        model, eff, fun = "mean", prt = FALSE, ...)
```

See Also

`model.tables`

---

manova *Multivariate Analysis of Variance*

Description

A class of 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.

Author(s)

B.D. Ripley

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.

`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

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`	the values to be matched.
`table`	the values to be matched against.
`nomatch`	the value to be returned in the case when no match is found.
`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 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.

Value

In both cases, a vector of the same length as `x`.

`match`: A numeric 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`.

See Also

`pmatch` and `charmatch` for (*partial*) string matching, `match.arg`, etc for function argument matching.

`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)
```

Arguments

`arg`	a character string
`choices`	a character vector of candidate values

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.

See Also

pmatch, match.fun, match.call.

Examples

```
## 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
center(x, "m")       # Error
```

---

match.call *Argument Matching*

Description

match.call returns a call in which all of the arguments are specified by their names. The most common use is to get the call of the current function, with all arguments named.

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?

Value

An object of class call.

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.
`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 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 now 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 `attach`es a list or data frame containing a character object with the same name of a system function, it will be used.

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

match.fun(outer, descend = FALSE) #-> Error:  not a function

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 coerced to a 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
```

Value

The matrix product. Use drop to get rid of dimensions which have only one level.

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
```

`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, 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 (`NA`s) 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 `type`s. 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.
`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.

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)))

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"))

data(iris)                  # is data.frame with  'Species' factor
table(iris$Species)
iS <- iris$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")
```

`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.

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.
`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.

`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.

See Also

`data.matrix`.

Examples

```
is.matrix(as.matrix(1:10))
data(warpbreaks)
!is.matrix(warpbreaks)# data.frame, NOT matrix!
str(warpbreaks)
str(as.matrix(warpbreaks))#using as.matrix.data.frame(.) method
```

`max.col` *Find Maximum Position in Matrix*

Description

Find the maximum position for each row of a matrix, breaking ties at random.

Usage

```
max.col(m)
```

Arguments

`m`	numerical matrix

Details

Ties are broken at random. The determination of "tie" assumes that the entries are probabilities.

Value

index of a maximal value for each row, an integer vector of length `nrow(m)`.

Author(s)

W. N. Venables and B. D. Ripley

References

Venables, W. N. and Ripley, B. D. (1999) *Modern Applied Statistics with S-PLUS*. New York: Springer (3nd ed).

See Also

`which.max` for vectors.

Examples

```
data(swiss)
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
```

`mean` *Arithmetic Mean*

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)
mean.default(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.

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.

See Also

`weighted.mean`, `mean.POSIXct`

Examples

```
x <- c(0:10, 50)
xm <- mean(x)
c(xm, mean(x, trim = 0.10))

all.equal(mean(x, trim = 0.5), median(x))

data(USArrests)
mean(USArrests, trim = 0.2)
```

`median` *Median Value*

Description

Compute the sample median of the vector of values given as its argument.

Usage

```
median(x, na.rm=FALSE)
```

Arguments

`x`	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.

See Also

`quantile` for general quantiles.

Examples

```
median(1:4)# = 2.5 [even number]
median(c(1:3,100,1000))# = 3 [odd, robust]
```

`Memory` *Memory Available for Data Storage*

Description

Use command line options to control the memory available for R.

Usage

```
Rgui --min-vsize=vl --max-vsize=vu --min-nsize=nl --max-nsize=nu
Rterm --min-vsize=vl --max-vsize=vu --min-nsize=nl --max-nsize=nu

mem.limits(nsize = NA, vsize = NA)
```

Arguments

`vl, vu, vsize`	Heap memory in bytes.
`nl, nu, nsize`	Number of cons cells.

Details

R has a variable-sized workspace (from version 1.2.0). There is now much less need to set memory options than previously, 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 using the option `--max-mem-size` on Windows), and since setting larger values of the minima will make R slightly more efficient on large tasks.

(On Windows the `--max-mem-size` option sets the maximum memory allocation: it has a minimum allowed value of 10M. This is intended to catch attempts to allocate excessive amounts of memory which may cause other processes to run out of resources. The default is the smaller of the amount of physical RAM in the machine and 256Mb. See also `memory.limit`.)

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 input `v` is rounded up to the nearest 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 `M`, `K`, or `k` meaning *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.

When using `read.table`, the memory requirements are in fact higher than anticipated, because the file is first read in as one long string which is then split again. Use `scan` if possible in case you run out of memory when reading in a large table.

Value

(`mem.limits`) an integer vector giving the current settings of the maxima, possibly `NA`.

Note

For backwards compatibility, options `--nsize` and `--vsize` are equivalent to `--min-nsize` and `--min-vsize`.

When using the `Rgui` console it is simplest to make a shortcut and put these command-line flags at the end of the Target field.

See Also

gc for information on the garbage collector, memory.profile for profiling the usage of cons cells.

memory.size to monitor total memory usage, memory.limit for the current limit.

Examples

```
# Start R with 10MB of heap memory and 500k cons cells, limit to
# 100Mb and 1M cells

## Unix
R --min-vsize=10M --max-vsize=100M --min-nsize=500k --max-nsize=1M
```

---

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'. There will be blanks in the list corresponding to types that are no longer in use (types 11 and 12 at the time of writing). Also FUNSXP is not included.

Value

A vector of counts, named by the types.

See Also

gc for the overall usage of cons cells.

Examples

```
memory.profile()
```

`memory.size` *Report on Memory Allocation*

Description

`memory.size` reports the current or maximum memory allocation of the `malloc` function used in this version of R.

`memory.limit` reports or increases the limit in force on the total allocation.

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

Command-line flag `--max-mem-size` sets the maximum value of obtainable memory (including a very small amount of housekeeping overhead).

Memory limits can only be increased.

Value

Size in bytes.

Examples

```
memory.size()
memory.size(TRUE)
round(memory.limit()/1048576.0, 2)
```

`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. Currently unused.
title	a character string to be used as the title of the menu

Value

The number corresponding to the selected item, or 0 if no choice was made.

Examples

```
switch(menu(c("List letters", "List LETTERS")) + 1,
       cat("Nothing done\n"), letters, LETTERS)
```

---

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, by, by.x, by.y, 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 NAs 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.

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 the `by.*` vector are of length 0, the result, `r`, is the "Cartesian product" of `x` and `y`, i.e., `dim(r) = c(nrow(x)*nrow, 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.

Value

A data frame. The rows are by default lexicographically sorted on the common columns, but are otherwise in the order in which they occurred in `x`. 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 column `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

```
authors <- data.frame(
    surname = c("Tukey", "Venables", "Tierney", "Ripley", "McNeil"),
    nationality = c("US", "Australia", "US", "UK", "Australia"),
    deceased = c("yes", rep("no", 4)))
books <- data.frame(
    name = 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)
```

`Methods` *Internal and Group Methods and Generic Functions*

Description

Many R-internal functions are *generic* and allow methods to be written for. *Group* methods in particular are available for the `"Math"`, `"Ops"`, and `"Summary"` group.

Usage

```
Math.data.frame(x, ...)
Math.factor(x, ...)

Ops.data.frame(e1, e2 = NULL)
Ops.factor(e1, e2)
Ops.ordered(e1, e2)

Summary.data.frame(x, ...)
Summary.factor(x, ...)

.Method
.Generic
.Group
.Class
```

Arguments

`x, e1, e2`	objects.
`...`	further arguments passed to methods.

Group Dispatching

There are three *groups* for which methods can be written, namely the `"Math"`, `"Ops"` and `"Summary"` groups.

A function *f* belonging to one of these groups must be `.Internal` or `.Primitive` and will automatically be using *<grp>.<class> (ob)* when *f(<ob>)* is called, *f* belongs to group *<grp>* and *<ob>* is of `class` *<class>*.

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`, `tetragamma`, `pentagamma`
   - `cumsum`, `cumprod`, `cummax`, `cummin`

2. Group "Ops":
   - "+", "-", "*", "/", "^", "%%", "%/%"
   - "&", "|", "!"
   - "==", "!=", "<", "<=", ">=", ">"
3. Group "Summary":
   - all, any
   - sum, prod
   - min, max
   - range

Simple Dispatching

The following builtin functions are *generic* as well, i.e., you can write methods for them:

[, [[

dimnames<-, dimnames, dim<-, dim

c, unlist, as.vector, is.na, is.nan

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.

Examples

```
methods("Math")
methods("Ops")
methods("Summary")

d.fr <- data.frame(x=1:9, y=rnorm(9))
data.class(1 + d.fr) == "data.frame" ##-- add to d.f. ...
```

---

methods *Class Methods*

Description

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

Usage

```
UseMethod(generic, object)
NextMethod(generic = NULL, object = NULL, ...)
methods(generic.function, class)
```

Arguments

`generic`	a character string naming a function.
`object`	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 method.
`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

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. 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, applied 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.

`methods` can be used to find out about the methods for a particular generic function or class. See the examples below for details.

Now for some obscure details that need to appear somewhere. These comments will be slightly different than those in Appendix A of the White S Book. `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 (!?). Any statements after the call to `UseMethod` will not be evaluated as `UseMethod` does not return.

`NextMethod` invokes the next method (determined by the class). It does this by creating a special call frame for that method. 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 arguments matched to `...` are handled specially. 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.

Note

The `methods` function was written by Martin Maechler.

See Also

`class`

Examples

```
methods(summary)

methods(print)

methods(class = data.frame)

methods("[")  ##- does not list the C-internal ones...
```

`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.

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) <- "<mode>"
storage.mode(x)
storage.mode(x) <- "<mode>"
```

Arguments

x	any R 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 assignment versions are currently identical. Both `mode(x) <- newmode` and `storage.mode(x) <- newmode` change the `mode` or `storage.mode` of object `x` to `newmode`.

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 assignment 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.

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)
```

---

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`	The name of a components to extract, such as `"weights"`, `"subset"`.
`type`	One of `"any"`, `"numeric"`, `"double"`. Using the either of latter two coerces the result to have storage mode `"double"`.

Details

`model.offset` and `model.response` are equivalent to `model.frame(, "offset")` and `model.frame(, "response")` respectively.

`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

```
data(esoph)
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, ...)
model.frame.default(formula, data = NULL,
                    subset = NULL, na.action = na.fail,
                    drop.unused.levels = FALSE, xlev = NULL, ...)

Methods for
        lm      glm     aovlist
```

Arguments

`formula`	a model formula
`data`	`data.frame`, list, `environment` or object coercible to `data.frame` containing the variables in `formula`.
`subset`	a specification of the rows to be used. Defaults to all rows.
`na.action`	how `NA`s 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`.
`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 `subset`, `offset` and `weights`. `NULL` arguments are treated as missing.

Details

Variables in the 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.

First all the variables needed are collected into a data frame. Then `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.

Value

A `data.frame` containing the variables used in `formula` plus those specified `....`

See Also

`model.matrix` for the "design matrix", `formula` for formulas and `expand.model.frame` for model.frame manipulation.

Examples

```
data(cars)
data.class(model.frame(dist ~ speed, data = cars))
```

`model.matrix` *Construct Design Matrices*

Description

`model.matrix` creates a design matrix.

Usage

```
model.matrix   (object, ...)
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`.
`contrasts.arg`	A list, whose entries are contrasts suitable for input to the `contrasts` function and whose names are the names of columns of `data` containing `factor`s.
`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(formula)`, using the data in `data` which must contain columns with the same names as would be created by a call to `model.frame(formula)` or, more precisely, by evaluating `attr(terms(formula), "variables")`. There may be other columns and the order is not important. If `contrasts` is specified it overrides the default factor coding for that variable.

Value

The design matrix for a regression model with the specified formula and data.

References

Chambers, J. M. and Hastie, T. J. eds (1992) *Statistical Models in S.* Chapman & Hall, London.

See Also

`model.frame`, `model.extract`, `terms`

Examples

```
data(trees)
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, ...)
model.tables.aov(x, type = "effects", se = FALSE, cterms, ...)
model.tables.aovlist(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.

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.

Author(s)

B.D. Ripley

See Also

aov

Examples

```
## From Venables and Ripley (1997) p.210.
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)
model.tables(npk.aov, "means", se=TRUE)

## as a test, not particularly sensible statistically
options(contrasts=c("contr.helmert", "contr.treatment"))
npk.aovE <- aov(yield ~  N*P*K + Error(block), npk)
model.tables(npk.aovE,  se=TRUE)
model.tables(npk.aovE, "means")
```

---

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

```
data(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

```
data(morley)
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)
```

---

mosaicplot *Mosaic Plots*

Description

Plots a mosaic on the current graphics device.

Usage

```
mosaicplot(x, main = NULL, xlab = NULL, ylab = NULL,
                   sort = NULL, off = NULL, dir = NULL,
                   color = FALSE, shade = FALSE, margin = NULL,
                   type = c("pearson", "deviance", "FT"), ...)
mosaicplot(formula, data = NULL, ..., subset)
```

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.
`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 10 at each level). There should be one offset for each dimension of the contingency table.

`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`	(`TRUE` or vector of integer colors) for color shading or (`FALSE`, the default) for empty boxes with no shading. Ignored if `shade` is not `FALSE`.
`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.
`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 to be used for plotting.

Details

This is a generic function. It currently has a default method (`mosaicplot.default`) and a formula interface (`mosaicplot.formula`).

Extended mosaic displays show the standardized residuals of a loglinear model of the counts from by the 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.

Author(s)

S-PLUS original by John Emerson ⟨emerson@stat.yale.edu⟩. Modified and enhanced for R by KH.

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.

The home page of Michael Friendly (http://hotspur.psych.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

```
data(Titanic)
mosaicplot(Titanic, main = "Survival on the Titanic", color = TRUE)
## Formula interface for tabulated data:
mosaicplot(~ Sex + Age + Survived, data = Titanic, color = TRUE)

data(HairEyeColor)
mosaicplot(HairEyeColor, shade = TRUE)
## Independence model of hair and eye color and sex.  Indicates that
## there are significantly more blue eyed blond females than expected
## in the case of independence (and too few brown eyed blond females).
mosaicplot(HairEyeColor, shade = TRUE, margin = list(c(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, but not
## ``significantly''.

## Formula interface for raw data: visualize crosstabulation of numbers
## of gears and carburettors in Motor Trend car data.
data(mtcars)
mosaicplot(~ gear + carb, data = mtcars, color = TRUE)
```

---

mtcars *Motor Trend 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

```
data(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 carburettors

Source

Henderson and Velleman (1981), Building multiple regression models interactively. *Biometrics*, **37**, 391–411.

Examples

```
data(mtcars)
pairs(mtcars, main = "mtcars data")
coplot(mpg ~ disp | as.factor(cyl), data = mtcars,
       panel = panel.smooth, rows = 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, cex = NA, col = NA, font = NA, vfont = NULL, ...)
```

Arguments

`text`	one or more character strings 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. For strings parallel to the axes, `adj=0` means left or bottom alignment, and `adj=1` means right or top aligment. If `adj` is not a finite value (the default), the value `par("las")` determines the adjustment. For strings plotted parallel to the axis the default is to centre the string.

`...`	Further graphical parameters (see `text` and `par`) ; currently supported are:
`cex`	character expansion factor (default = 1).
`col`	color to use.
`font`	font for text.
`vfont`	vector font for text.

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.

The arguments `side`, `line`, `at`, `at`, `adj`, the further graphical parameters and even `outer` 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 justifcation is to place the end of the string nearest the axis on the specified line.

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.

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)
```

n2mfrow *Compute Default mfrow From Number of Plots*

Description

Easy Setup for plotting multiple figures (in a rectangular layout) on one page. It allows to specify a main title and uses *smart* defaults for several par calls.

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)
```

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.

The generic function 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 generic function is.na<- sets elements to NA.

Usage

```
NA
is.na(x)
is.na.data.frame(x)

is.na(x) <- value
```

Arguments

`x`	an R object to be tested.
`value`	a suitable index vector for use with `x`.

Details

For character vectors the value `"NA"` represents missingness.

`is.na(x)` works elementwise when `x` is a `list`. The method dispatching is C-internal, rather than via `UseMethod`.

Function `is.na<-` may provide a safer way to set missingness. It behaves differently for factors, for example.

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))       #> F TRUE
is.na(paste(c(1,NA)))#> F FALSE
```

`na.action` *NA Action*

Description

`na.action` is a generic function, and `na.action.default` its default method.

Usage

```
na.action(object, ...)
na.action.default(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 `NA`s are not desired.

See Also

options("na.action"), na.omit, na.fail

Examples

```
na.action(c(1, NA))
```

---

na.fail *Handle Missing Values in Objects*

Description

These generic functions are useful for dealing with NAs 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.

Usage

```
na.fail(object, ...)
na.omit(object, ...)
na.exclude(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 NAs for cases omitted by na.exclude.

See Also

na.action; options with argument na.action for setting "NA actions"; and lm and glm for functions using these.

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")
```

`name` *Variable Names or Symbols, respectively*

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.

Note

The term "symbol" is from the lisp background of R, whereas "name" has been the standard S term for this.

See Also

`call`, `is.language`. For the internal object mode, `typeof`.

Examples

```
an <- as.name("arrg")
is.name(an) # TRUE
str(an)# symbol
```

`names` *The Names Attribute 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 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 or list.

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")).

Value

For names, NULL or a character vector of the same length as x.

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.)

Examples

```
# print the names attribute of the islands data set
names(islands)

# remove the names attribute
names(islands) <- NULL

z <- list(a=1, b="c", c=1:3)
names(z)
# change just the name of the third element.
names(z)[3] <- "c2"
z
```

---

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.
`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` and `resid` methods for modelling functions to compensate for the removal of `NA`s in the fitting process. There are used by the default, `"lm"` and `"glm"` methods, and by further methods in packages `MASS`, `rpart` and `survival`.

The default methods do nothing. The method for the `na.exclude` action to pad the object with `NA`s 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`.

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()
```

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*

Description

`nchar` takes a character vector as an argument and returns a vector whose elements contain the number of characters in the corresponding element of `x`. It only accepts character vectors as arguments if you want to operate on other objects passing them through `deparse` first will be required.

Usage

```
nchar(x)
```

Arguments

`x`	character vector.

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))
# 23  1 16 45 11 64  2 17 50 43  2 17  1
```

`nclass` *Compute the Number of Classes for a Histogram*

Description

Compute the number of classes for a histogram, for use internally in `hist`.

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.

Note

For consistency with earlier versions of R, `nclass.Sturges` rounds down. This is incompatible with S-PLUS, and probably wrong: however the other algorithms are to be preferred.

References

Venables, W. N. and Ripley, B. D. (1999) *Modern Applied Statistics with S-PLUS.* Springer, pages 118–9.

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`

`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).
`prob`	probability of success in each trial.
`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 $x = 0, 1, 2, \ldots$

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 gamma (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)`. In this parametrization the variance is `mu + mu^2/size`.

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.

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))
```

---

nhtemp *Average Yearly Temperatures in New Haven*

Description

The mean annual temperature in degrees Fahrenheit in New Haven, Connecticut, from 1912 to 1971.

Usage

```
data(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

```
data(nhtemp)
plot(nhtemp, main = "nhtemp data",
  ylab = "Mean annual temperature in New Haven, CT (deg. F)")
```

---

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
```

---

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 a vector of the length of `p` followed by any other arguments specified in `dots`.
`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 mininum 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`. `optimize` for one-dimensional minimization and `uniroot` for root finding. `deriv` to calculate analytical derivatives.

For nonlinear regression, `nls` (in package `nls`), may be of better use.

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.
demo(nlm)
```

noquote *Class for "no quote" Printing of Strings*

Description

These functions exist both as utilities and as an example of using `class` and object orientation.

Usage

```
noquote(obj)
print.noquote(x, ...)
obj[j]
```

Arguments

`obj, x`	any R object; typically a vector of `character` strings.
`...`	further options for `print`.

Value

`noquote` returns its argument as an object of class `"noquote"`. The function `"[.noquote"` ensures that the class is not lost by subsetting.

For (default) printing, `print.noquote` will be used which prints characters *without* quotes (`"..."`).

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)
```

`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.

References

Wichura, M. J. (1988) Algorithm AS 241: The Percentage Points of the Normal Distribution. *Applied Statistics*, **37**, 477–484.

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)
```

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

```
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`.

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
```

`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.

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

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 numeric type (either integer or real).

is.numeric returns TRUE if its argument is of type real or type integer and FALSE otherwise.

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.

Note

R *has no single precision data type. All real numbers are stored in double precision format.* While as.numeric is a generic function, user methods must be written for as.double, which it calls

as.numeric for factors yields the codes underlying the factor levels, not the numeric representation of the labels.

Examples

```
as.numeric(c("-.1"," 2.7 ","B")) # (-0.1, 2.7, NA)  +  warning
as.numeric(factor(5:10))
```

`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. For example, it will not detect if character storage for character strings are shared between identical elements (which it will be if `rep` was used, for example).

The calculation is of the size of the object, and excludes the space needed to store its name in the symbol table.

Value

An estimate of the memory allocation attributable to the object, in bytes.

Examples

```
object.size(letters)
object.size(ls)
## find the 10 largest objects in base
z <- sapply(ls("package:base"), function(x) object.size(get(x)))
as.matrix(rev(sort(z))[1:10])
```

`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

```
as.character(x, ...)
format(x, ...)
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 atttribute, used merely to ensure that they are printed in octal notation, specifically for Unix-like file permissions such as `755`.

See Also

These are auxiliary functions for `file.info`

`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

Value

The input value.

See Also

`model.offset`, `model.frame`, `glm`

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.

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.

See Also

`sys.on.exit` to see the current expression.

Examples

```
opar <- par(mai = c(1,1,1,1))
on.exit(par(opar))
```

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.

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. Not needed for the `"Nelder-Mead"` and `"SANN"` method. If it is `NULL` and it is needed, a finite-difference approximation will be used. It is guaranteed that `gr` will be called immediately after a call to `fn` at the same parameter values.
`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`.

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 that 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.* (1994) 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 comprehansive reference for the previous three methods.

Method `"SANN"` is 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. The next candidate point is generated from a Gaussian Markov kernel with scale proportional to the actual temperature. Temperatures are decreased according to the logarithmic cooling schedule as given in Belisle (1992, p. 890). 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` 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 controls 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`.

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

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`, `optimize`

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)
```

`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.

It uses Fortran code (from Netlib) based on algorithms given in the reference.

`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 arguments to `f`.

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.

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
```

`options` *Options Settings*

Description

`options` allows 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`. However, only the ones below are used in "base R". Further, `options('name') == options()['name']`, see the example.
`prompt`	a string, used for R's prompt; should usually end in a blank (`" "`).
`continue`	a string setting the prompt used for lines which continue over one line.
`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 valid values are in file 'Print.h' and can be changed by re-compiling R.)
`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.
`editor`	sets the default text editor, e.g., for `edit`. Set from the environment variable `VISUAL` on UNIX.
`pager`	the (stand-alone) program used for displaying ASCII files on R's console. Defaults to `"internal"`, which uses a pager similar to the GUI console. Another possibility is `"console"` to use the console itself.
`browser`	default HTML browser used by `help.start()` on UNIX.
`mailer`	default mailer used by `bug.report()`. can be `"none"`.
`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.
`expressions`	sets a limit on the number of nested expressions that will be evaluated. This is especially important on the Macintosh since stack overflow is likely if this is set too high. Valid values are 25...100000 with default 500.
`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`.
`na.action`	the name of a function for treating missing values (`NA`'s) for certain situations.
`papersize`	the default paper format used by `postscript`; set by environment variable `R_PAPERSIZE` when R is started and defaulting to `"a4"` if that is unset or invalid.
`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. See `postscript` for ways to set this up.
`show.signif.stars, show.coef.Pvalues`	logical, affecting P value printing, see `print.coefmat`.
`ts.eps`	the relative tolerance for certain time series (`ts`) computations.

`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 function `dump.frames` provides one alternative that allows post-mortem debugging.

`show.error.messages`
a logical. Should error messages be printed? Intended for use with `try` or a user-installed error handler.

`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. A top–level variable called `last.warning` is created and can be viewed 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.

`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`.

`echo` logical. Only used in non-interactive mode, when it controls whether input is echoed. Command-line option `--slave` sets this initially to `FALSE`.

`verbose` logical. Should R report extra information on progress? Set to `TRUE` by the command-line option `--verbose`.

`device` a character string giving the default device for that session. This defaults to the normal screen device (e.g. `x11`, `windows` or `gtk`) for an interactive session, and `postscript` in batch use or if a screen is not available.

`CRAN` The URL of the preferred CRAN node for use by `update.packages`. Defaults to `http://cran.r-project.org`.

`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`.

`unzip` the command used unzipping help files. Defaults to "internal" when the internal unzip DLL is used.

`de.cellwidth` integer: the cell widths (number of characters) to be used in the data editor `dataentry`. If this is unset, 0, negative or `NA`, variable cell widths are used.

`encoding` An integer vector of length 256 holding an input encoding. Defaults to `native.enc` (= `0:255`). See `connections`.

`timeout` Integer. The timeout for Internet operations, in seconds. Default 60 seconds.

`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.

`x` a character string holding one of the above option names.

Details

Invoking `options()` with no arguments returns a list with the current values of the options. 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, for S compatibility. You must use it "read only" however.

The default settings of some of these options are

prompt	"> "	continue	"+ "
width	80	digits	7
expressions	500	keep.source	TRUE
show.signif.stars	TRUE	show.coef.Pvalues	TRUE
na.action	na.omit	ts.eps	1e-5
error	NULL	warn	0
echo	TRUE	verbose	FALSE

Others are set from environment variables or are platform-dependent.

Value

A list (in any case) with the previous values of the options changed, or all options when no arguments were given.

Examples

```
options() # printing all current options
op <- options(); str(op) # nicer printing

# .Options is the same:
all(sapply(1:length(op), function(i) all(.Options[[i]] == op[[i]])))

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)
x <- NULL; x[4] <- "yes" # gives a warning

options(op)     # reset (all) initial options
options('digits')

## set contrast handling to be like S
options(contrasts=c("contr.helmert", "contr.poly"))

## on error, terminate the R session with error status 66
options(error=quote(q("no", status=66, runLast=FALSE)))
stop("test it")

## set an error action for debugging: see ?debugger.
options(error=quote(dump.frames()))
```

```
## A possible setting for non-interactive sessions
options(error=quote({dump.frames(to.file=TRUE); q()}))
```

`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

```
data(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 $* \times 8$ 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

```
data(OrchardSprays)
pairs(OrchardSprays, main = "OrchardSprays data")
```

`order` *Ordering Permutation*

Description

`order` returns a permutation which rearranges its first argument into ascending order, breaking ties by further arguments. `sort.list` is the same, using only one argument but allowing partial sorting.

Usage

```
order(..., na.last = TRUE)
sort.list(x, partial = NULL, na.last = TRUE)
```

Arguments

`...`	a sequence of vectors, all of the same length.
`x`	a vector.
`partial`	vector of indices for partial sorting.
`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.

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).

`NA` values are treated as greater than any other values so that permutations returned by `order` move `NA` values to the top end of the array. Other options are not (yet) implemented in R.

`partial` is supplied for compatibility with other implementations of S, but no other values are accepted and ordering is always complete.

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)

## 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, ]
```

`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`	A vector or array.
`Y`	A vector or array.
`FUN`	a function to use on the outer products, it may be a quoted string.
`...`	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 on arrays.

Where they exist, the [dim]names of `X` and `Y` will be preserved.

`%o%` is an alias for `outer` (where `FUN` cannot be changed from `"*"`).

Author(s)

Jonathan Rougier

See Also

`matmult` 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]
```

---

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", "bonferroni","none")
```

Arguments

`p`	vector of p values
`method`	correction method
`n`	number of comparisons

Details

The adjustment methods include the Bonferroni correction in which the p values are multiplied by the number of comparisons. Two less conservative corrections by Holm, respectively Hochberg, are also included. 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`.

Value

A vector of corrected p values.

Note

The Hochberg method is only proved to work if the p values are independent, although simulations have indicated that it works in correlated cases as well. Hence the Holm method is the default.

References

S Paul Wright: Adjusted P-values for simultaneous inference, Biometrics **48**, 1005–1013

See Also

`pairwise.*` functions in the `ctest` package, such as `pairwise.t.test`.

Examples

```
p <- runif(50)
p.adjust(p)
```

---

package.contents *Package Contents and Description*

Description

Parses and returns the 'CONTENTS' and 'DESCRIPTION' file of a package.

Usage

```
package.contents(pkg, lib.loc = NULL)
package.description(pkg, lib.loc = NULL, fields = NULL)
```

Arguments

`pkg`	a character string with the package name.
`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.
`fields`	a character vector giving the tags of fields to return (if other fields occur in the file they are ignored).

Value

`package.contents` returns `NA` if there is no 'CONTENTS' file for the given package; otherwise, a character matrix with column names `c("Entry", "Keywords", "Description")` and rows giving the corresponding entries in the CONTENTS data base for each Rd file in the package.

If a 'DESCRIPTION' for the given package is found and can successfully be read, `package.description` returns a named character vector with the values of the (given) fields as elements and the tags as names. If not, it returns a named vector of `NA`s with the field tags as names if `fields` is not null, and `NA` otherwise.

See Also

`read.dcf`

Examples

```
package.contents("mva")
package.contents("mva")[, c("Entry", "Description")]

package.description("ts")
package.description("ts")[c("Package", "Version")]
## NOTE: No subscripting using '$' or abbreviated field tags!
```

`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)
```

Arguments

`x`	A matrix of package descriptions as returned by `CRAN.packages`.
`check`	If `TRUE`, return logical vector of check results. If `FALSE`, return parsed list of 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`

`package.skeleton` *Create a skeleton for a new package*

Description

`package.skeleton` automates some of the setup for a new package. It creates directories, saves functions and data to appropriate places, and creates skeleton help files and `README` files describing further steps in packaging.

Usage

```
package.skeleton(name="anRpackage", list, environment=.GlobalEnv,
        path=".", force=FALSE)
```

Arguments

`name`	directory name for your package
`list`	vector of names of R objects to put in the package
`environment`	if `list` is omitted, the contents of this environment are packaged
`path`	path to put the package directories in
`force`	If `FALSE` will not overwrite an existing directory

Value

used for its side-effects.

References

Read the "Writing R Extensions" manual for more details

See Also

install.packages

Examples

```
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="AnExample")
```

---

packageStatus *Package Management Tools*

Description

Summarize information about installed packages and packages available at various repositories, and automatically upgrade outdated packages. These tools will replace update.packages and friends in the future and are currently work in progress.

Usage

```
packageStatus(lib.loc = NULL, repositories = getOption("repositories"))
summary(object, ...)
update(object, lib.loc=levels(object$inst$LibPath),
       repositories=levels(object$avail$Repository), ...)
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.
object	return value of packageStatus.
ask	if TRUE, the user is prompted which packages should be upgraded and which not.
...	currently not used.

Examples

```
x <- packageStatus()
print(x)
summary(x)
upgrade(x)
x <- update(x)
print(x)
```

---

page *Invoke a Pager on an R Object*

Description

Displays a representation of the object named by `x` in a pager.

Usage

```
page(x, method = c("dput", "print"), ...)
```

Arguments

`x`	the name of an R object.
`method`	The default method is to dump the object *via* `dput`. An alternative is to print to a file.
`...`	additional arguments for `file.show`. Intended for setting `pager` as `title` and `delete.file` are already used.

Author(s)

B. D. Ripley

See Also

`file.show`, `edit`, `fix`.

---

pairs *Scatterplot Matrices*

Description

A matrix of scatterplots is produced.

Usage

```
pairs(x, ...)
pairs.default(x, labels = colnames(x), 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 columns of a matrix.
`labels`	the names of the variables.
`panel`	`function(x,y,...)` which is used to plot the contents of each panel of the display.
`...`	graphical parameters can be given as arguments to `plot`.
`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.

Author(s)

Enhancements for R 1.0.0 contributed by Dr. Jens Oehlschlaegel-Akiyoshi and R-core members.

Examples

```
data(iris)
pairs(iris[1:4], main = "Anderson's Iris Data -- 3 species",
      pch = 21, bg = c("red", "green3", "blue")[codes(iris$Species)])

data(USJudgeRatings)
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)
```

`pairs.formula` *Formula Notation for Scatterplot Matrices*

Description

Produce a matrix of scatterplots using formula notation.

Usage

```
pairs(formula, data = NULL, ..., 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 the default pairs method and graphical parameters may also be passed as arguments, see `par`.
`subset`	an optional vector specifying a subset of observations to be used for plotting.

Details

This is a method of the generic function `pairs`. It operates by setting up the data from the formula specification, and then calling `pairs.default`.

See Also

`pairs.default`

Examples

```
data(swiss)
pairs(~ Fertility + Education + Catholic, data = swiss,
      subset = Education < 20, main = "Swiss data, Education < 20")
```

---

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;

`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)
heat.colors(n)
terrain.colors(n)
topo.colors(n)
cm.colors(n)
```

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`.

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`, `rgb`, `gray` and `col2rgb` for translating to RGB numbers.

Examples

```
# A Color Wheel
piechart(rep(1,12), col=rainbow(12))

##------ Some palettes ------------
ch.col <- c("rainbow(n, start=.7, end=.1)", "heat.colors(n)",
            "terrain.colors(n)", "topo.colors(n)", "cm.colors(n)")

n <- if(.Device == "postscript") 64 else 16
     # Since for screen, larger n may give color allocation problem
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=paste("color palettes;  n=",n))
for (k in 1:nt) {
  rect(i-.5,(k-1)*j+ dy, i+.4,  k*j, col=eval(parse(text=ch.col[k])))
  text(2*j,  k * j +dy/4, ch.col[k])
}
```

---

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, bg=NA, 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`.

Examples

```
data(swiss)
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 variable `.Pars`. `.Pars.readonly` contains the names of the `par` arguments which are *readonly*.

R.O. indicates ***read-only arguments***: These may only be used in queries, i.e., they do *not* set anything.

All but these ***R.O.*** and the following ***low-level arguments*** can be set as well in high-level and mid-level plot functions, such as `plot`, `points`, `lines`, `axis`, `title`, `text`, `mtext`:

- `"ask"`
- `"fig"`, `"fin"`
- `"mai"`, `"mar"`, `"mex"`
- `"mfrow"`, `"mfcol"`, `"mfg"`
- `"new"`
- `"oma"`, `"omd"`, `"omi"`
- `"pin"`, `"plt"`, `"ps"`, `"pty"`
- `"usr"`
- `"xlog"`, `"ylog"`

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 is a character string. When two or more parameters are queried, the result is a list of character strings, 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 do not annotate the plots they produce with axis and overall titles. The default is to do annotation.

`ask` logical. If `TRUE`, the user is asked for input, before a new figure is drawn.

`bg` The color to be used for the background of plots. A description of how colors are specified is given below.

`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.

`cex.axis` The magnification to be used for axis annotation relative to the current.

`cex.lab` The magnification to be used for x and y labels relative to the current.

`cex.main` The magnification to be used for main titles relative to the current.

`cex.sub` The magnification to be used for sub-titles relative to the current.

`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.

`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 in inches.

`err` (*Unimplemented*; R is silent when points outside the plot region are *not* plotted.) The degree of error reporting desired.

`fg` The color to be used for the foreground of plots. This is the default color is used for things like axes and boxes around plots. 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.

`fin` A numerical vector of the form `c(x, y)` which gives the size of the figure region in inches.

`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.

`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`.

`lab` A numerical vector of the form `c(x, y, len)` which modifies the 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 size. The default is `c(5, 5, 7)`. *Currently*, `len` *is unimplemented*.

`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.

`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(0:9, "A":"F")`) may be given, giving the length of line segments which are alternatively drawn and skipped. See section 'Line Type Specification' below.

`lwd` The line width, a *positive* number, defaulting to `1`.

`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 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.

`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.

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.

`pin` The width and height of the current plot 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 and symbols.

`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 arc should be.

`srt` The string rotation in degrees.

`tck` The length of tick marks as a fraction of the smaller of the width or height of the plotting region. If `tck=1`, grid lines are drawn. The default setting 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`.

`tmag` A number specifying the enlargement of text of the main title relative to the other annotating text of the plot.

`type` character; the default plot type desired, see `plot.default(type=...)`, defaulting to `"p"`.

`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.

`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 ensured 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 axis type. Specifying `"n"` causes an axis to be set up, but not plotted. The standard value is `"s"`: for compatibility with S values `"l"` and `"e"` are accepted but are equivalent to `"s"`.

`xlog` ***R.O.***; 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.

`yaxs` The style of axis interval calculation to be used for the y-axis. See `xaxs` above.

`yaxt` A character which specifies the axis type. Specifying `"n"` causes an axis to be set up, but not plotted.

`ylog` ***R.O.***; 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 there 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. This provides compatibility with S.

The functions `rgb`, `hsv`, `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, 3 = dotted) or directly as the lengths of on/off stretches of line. This is done with a string of up to eight characters 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.

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.

See Also

`plot.default` for some high-level plotting parameters; `colors`, `gray`, `rainbow`, `rgb`; `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
stopifnot(par("xaxp")[1:2] == 1:2 &&
          par("usr") [1:2] == 1:2)

( 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

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()
```

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

```
( ... )

{ ... }
```

See Also

`if`, `return`, etc for other objects used in the R language itself.

Examples

```
f <- get("(")
e <- expression(3 + 2 * 4)
f(e) == e           # TRUE

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. Each element of the list is of mode `expression`.

Usage

```
parse(file = "", n = NULL, text = NULL, prompt = "?", white = FALSE)
```

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`	the number of statements to parse. If `n` is negative the file is parsed in its entirety.
`text`	character. The text to parse, quoted.
`prompt`	the prompt to print when parsing from the keyboard. `NULL` means to use R's prompt, `options("prompt")[[1]]`.
`white`	if `TRUE` then any white space separates expressions otherwise only newlines or semicolons do.

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 Mac). The final line can be incomplete, that is missing the final EOL marker.

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 coerced 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 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.

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.

See Also

String manipulation with `as.character`, `substr`, `nchar`, `strsplit`; further, `cat` which concatenates and writes to a file.

Examples

```
paste(1:12) # same as as.character(1:12)
paste("A", 1:6, sep = "")
paste("Today is", date())
```

---

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",
    encoding, bg, fg, pointsize)
```

Arguments

`file`	a character string giving the name of the file.
`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, one of `"AvantGarde"`, `"Bookman"`, `"Courier"`, `"Helvetica"`, `"Helvetica-Narrow"`, `"NewCenturySchoolbook"`, `"Palatino"` or `"Times"`.
`encoding`	the name of an encoding file. Defaults to `"WinAnsi.enc"` in the 'R_HOME/afm' directory, which is used if the path does not contain a path separator. An extension `".enc"` can be omitted.
`pointsize`	the default point size to be used.
`bg`	the default background color to be used.
`fg`	the default foreground color to be used.

Details

`pdf()` opens the file `file` and the PDF commands needed to plot any graphics requested are sent to that file.

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.

There is an internal limit of 500 pages per PDF file, which should be ample.

Note

Acrobat Reader does not use the fonts specified but rather emulates them from multiple-master fonts. This can be seen in imprecise centring of characters, for example the multiply and divide signs in Helvetica.

See Also

Devices, postscript

---

persp *Perspective Plots*

Description

This function draws perspective plots of surfaces over the x–y plane. `persp` is a generic function.

Usage

```
persp(x, ...)
persp.default(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 = NULL, 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 the character strings; expressions are not accepted.
`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 of the surface facets.
`border`	the color of the line drawn around the surface facets. 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.

See Also

`contour` and `image`.

Examples

```
# (1) The Obligatory Mathematical surface.
#     Rotated sinc function.

x <- seq(-10, 10, length=50)
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
par(bg = "white")
persp(x, y, z, theta = 30, phi = 30, expand = 0.5, col = "lightblue",
      xlab = "X", ylab = "Y", zlab = "Z")
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 = "Z")

# (2) Visualizing a simple DEM model

data(volcano)
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)
persp(x, y, z, theta = 120, phi = 15, scale = FALSE, axes = FALSE)

# (3) Now something more complex
#     We border the surface, to make it more "slice like"
#     and color the top and sides of the surface differently.

zmin <- min(z) - 20
z <- rbind(zmin, cbind(zmin, z, zmin), zmin)
x <- c(min(x) - 1e-10, x, max(x) + 1e-10)
y <- c(min(y) - 1e-10, y, max(y) + 1e-10)

fill <- matrix("green3", nr = nrow(z)-1, nc = ncol(z)-1)
fill[,1] <- "gray"
fill[,ncol(fill)] <- "gray"
fill[1,] <- "gray"
fill[nrow(fill),] <- "gray"

par(bg = "lightblue")
persp(x, y, z, theta = 120, phi = 15, col = fill, scale = FALSE, axes = FALSE)
title(main = "Maunga Whau\nOne of 50 Volcanoes in the Auckland Region.",
      font.main = 4)

par(bg = "slategray")
persp(x, y, z, theta = 135, phi = 30, col = fill, scale = FALSE,
      ltheta = -120, lphi = 15, shade = 0.65, axes = FALSE)
persp(x, y, z, theta = 135, phi = 30, col = "green3", scale = FALSE,
      ltheta = -120, shade = 0.75, border = NA, box = FALSE)
```

`phones` *The World's Telephones*

Description

The number of telephones in various regions of the world (in thousands).

Usage

```
data(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

```
data(phones)
matplot(rownames(phones), phones, type = "b", log = "y",
        xlab = "Year", ylab = "Number of telephones (1000's)")
legend(1951.5, 80000, colnames(phones), col = 1:7, lty = 1:7, pch = rep(21, 7))
title(main = "phones data: log scale for response")
```

---

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.
`fg`	the foreground color for the plot.

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.

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()
##--------------------

%% LaTeX Example
\documentclass{article}
\usepackage{pictex}
\begin{document}
%...
\begin{figure}[h]
  \centerline{\input{Rplots.tex}}
  \caption{}
\end{figure}
%...
\end{document}

%%-- TeX Example --
\input pictex
$$ \input Rplots.tex $$

##--------------------
unlink("Rplots.tex")
```

---

piechart *Pie Charts*

Description

Draw a pie chart.

Usage

```
piechart(x, labels = names(x), edges = 200, radius = 0.8,
         density = NULL, angle = 45, col = NULL,
         main = NULL, ...)
```

Arguments

`x`	a vector of positive quantities. The values in `x` are displayed as the areas of pie slices.
`labels`	a vector of character strings giving names for the slices.
`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.
`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 the slices. If missing `par("bg")` is used, unless `density` is specified when `par("fg")` is used.
`main`	an overall title for the plot.
`...`	graphical parameters can be given as arguments to `piechart`.

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.

See Also

`dotchart`.

Examples

```
piechart(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")
piechart(pie.sales,
    col=c("purple", "violetred1", "green3",
    "cornsilk", "cyan", "white"))
piechart(pie.sales,
    col=gray(seq(0.4,1.0,length=6)))
piechart(pie.sales,
    density=10, angle=15+10*1:6)
```

`PkgUtils` *Utilities for Building and Checking Add-on Packages*

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

```
Rcmd build [options] pkgdirs
Rcmd 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

`Rcmd check` checks R add-on packages from their sources, performing a wide variety of diagnostic checks.

`Rcmd build` builds R source or binary packages from their sources.

Use `Rcmd foo --help` to obtain usage information on utility `foo`.

Note

These may not work correctly under Windows 95/98/ME because of problems Perl has launching programs on those limited OSes.

See Also

The chapter "Processing Rd format" in "Writing R Extensions" (see the Manuals sub-menu of the Help menu on the console).

`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

```
data(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:
data(PlantGrowth)
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))
```

`Platform` *Platform Specific Variables*

Description

`.Platform` is a list with functions and variables as components. This provides means to write OS portable R code.

Usage

```
.Platform
Platform()
```

Details

Currently, `.Platform <- Platform()` when R starts up.

Value

`.Platform` is list with at least the following components:

`OS.type`	character, giving the **O**perating **S**ystem (family) of the computer. One of the following values is returned: `"unix"`, `"mac"`, or `"windows"` (in historical order).
`file.sep`	character, giving the **file sep**arator, used on your platform, e.g., `"/"` on Unix alikes.
`dynlib.ext`	character, giving the file name **ext**ension of **dyn**amically loadable **lib**raries, e.g., `".dll"` on Windows.
`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.

See Also

`Sys.info` which gives more details about the OS, `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
}
```

---

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, xlim=range(x), ylim=range(y), type="p",
     main, xlab, ylab, ...)
```

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.
`xlim, ylim`	the ranges to be encompassed by the x and y axes.
`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.
`xlab`	a title for the x axis.
`ylab`	a title for the y axis.
`...`	graphical parameters can be given as arguments to `plot`.

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

```
data(cars)
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(stepfun)} for a better one:
plot(x <- sort(rnorm(47)), type = "s", main = "plot(x, type = \"s\")")
points(x, cex = .5, col = "dark red")
```

`plot.default` *The Default Scatterplot Function*

Description

Draw a scatter plot with "decorations" such as axes and titles in the active graphics window.

Usage

```
plot.default(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,
             col = par("fg"), bg = NA, pch = par("pch"),
             cex = par("cex"), lty = par("lty"), lab = par("lab"),
             lwd = par("lwd"), 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 (min,max) of the plot.
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.
sub	a sub title for the plot.
xlab	a label for the x axis.
ylab	a label for the y axis.
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 axes should be drawn on the plot.
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.
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.
bg	background color for open plot symbols, see points.
pch	a vector of plotting characters or symbols.
cex	a numerical value giving the amount by which plotting text and symbols should be scaled relative to the default
lty	the line type, see par.
lab	the specification for the (approximate) numbers of tick marks on the x and y axes.
lwd	the line width **not yet supported for postscript**.
asp	the y/x aspect ratio, see plot.window.
...	graphical parameters as in par may also be passed as arguments.

References

Cleveland, W. S. (1985) *The Elements of Graphing Data.* Monterey, CA: Wadsworth.

See Also

plot, plot.window, xy.coords.

Examples

```
data(cars)
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.density *Plot Method for Kernel Density Estimation*

Description

The `plot` method for density objects.

Usage

```
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.factor` *Plotting Factor Variables*

Description

This functions implements a "scatterplot" method for `factor` arguments of the *generic* `plot` function. Actually, `boxplot` or `barplot` are used when appropriate.

Usage

```
plot.factor(x, y, legend.text = levels(y), ...)
```

Arguments

`x,y`	numeric or factor. `y` may be missing.
`legend.text`	a vector of text used to construct a legend for the plot. Only used if `y` is present and a factor.
`...`	Further arguments to `plot`, see also `par`.

See Also

`plot.default`, `plot.formula`, `barplot`, `boxplot`.

Examples

```
data(PlantGrowth)
plot(PlantGrowth)                                # -> plot.data.frame
plot(weight ~ group, data = PlantGrowth)         # numeric vector ~ factor
plot(cut(weight, 2) ~ group, data = PlantGrowth) # factor ~ factor

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

```
plot(formula, data = parent.frame(), ..., subset,
             ylab = varnames[response], ask = TRUE)
points(formula, data = parent.frame(), ..., subset)
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.
`...`	Further graphical parameters may also be passed as arguments, see `par`.
`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)`.

If `y` is an object (ie. has a `class` attribute) then `plot.formula` looks for a plot method for that class first.

Value

These functions are invoked for their side effect of drawing in the active graphics device.

See Also

`plot.default`, `plot.factor`.

Examples

```
data(airquality)
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

```
plot(x, freq = equidist, density = NULL, angle = 45,
               col = NULL, border = par("fg"),
               lty = NULL, main = paste("Histogram of", x$xname),
               xlim = range(x$breaks), ylim = range(y, 0),
               xlab = x$xname, ylab, axes = TRUE, labels = FALSE,
               add = FALSE, ...)

lines(x, ...)
```

Arguments

`x`	a `histogram` object, or a list with components `intensities`, `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$counts`; if `FALSE`, *relative* frequencies ("probabilities"), i.e., `x$intensities`, 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`.
`xlim, ylim`	the range of x and y values with sensible defaults.
`main, xlab, ylab`	these arguments to `title` have useful defaults here.
`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 intensities; 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

```
data(women)
str(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.lm *Plot Diagnostics for an lm Object*

Description

Four plots (selectable by `which`) are currently provided: a plot of residuals against fitted values, a Scale-Location plot of $\sqrt{|residuals|}$ against fitted values, a Normal Q-Q plot, and a plot of Cook's distances versus row labels.

Usage

```
plot(x, which = 1:4,
        caption = c("Residuals vs Fitted", "Normal Q-Q plot",
                    "Scale-Location plot", "Cook's distance plot"),
        panel = points,
        sub.caption = deparse(x$call), main = "",
        ask = prod(par("mfcol")) < length(which) && dev.interactive(),
        ...,
        id.n = 3, labels.id = names(residuals(x)), cex.id = 0.25)
```

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 `1:4`.
`caption`	Captions to appear above the plots
`panel`	Panel function. A useful alternative to `points` is `panel.smooth`.
`sub.caption`	common title—above figures if there are multiple; used as `sub` (s.`title`) otherwise.
`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.

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).

This 'S-L' and the Q-Q 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, `lm.influence()$hat`, see also `hat`.

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.

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`.

Examples

```
## Analysis of the life-cycle savings data
## given in Belsley, Kuh and Welsch.
data(LifeCycleSavings)
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

## Fit a smmooth 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.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

```
plot.table(x, type = "h", ylim = c(0, max(x)), lwd = 2,
           xlab = NULL, ylab = deparse(substitute(x)), frame.plot = is.num, ...)
```

Arguments

`x`	a `table` (like) object.
`type`	plotting type.
`ylim`	range of y-axis.
`lwd`	line width of bars when `type = "h"` is used.
`xlab, ylab`	x- and y-axis labels.
`frame.plot`	logical indicating if a frame (`box`) should be drawn. Default are Describe `frame.plot` here
`...`	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)))")

data(state)
plot(table(state.division))

## 4-D :
data(Titanic)
plot(Titanic, main ="plot(Titanic, main= *)")
```

`plot.ts` *Plotting Time-Series Objects*

Description

Plotting methods for objects of class `"ts"` or `"mts"` (multivariate time-series).

Usage

```
plot(x, y = NULL, type = "l", frame.plot = axes,
     plot.type = c("multiple", "single"),
     xy.labels = n <= 150, xy.lines = do.lab, panel=lines, ...)

lines(x, ...)
```

Arguments

`x,y`	time series objects, usually of class `"ts"`.
`type`	the type of plot, see `plot`. When `y` is present, the default will depend on `xy.labels`, see below.
`frame.plot`	a function to give the 'frame' for each panel.
`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.
`xy.lines`	logical, indicating if `lines` should be drawn for an x-y plot. Default is true, when labels are drawn as well.
`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`.
`...`	additional graphical arguments, see `plot`, `plot.default` and `par`.

Details

With one principal argument, these functions create time series plots, for multivariate series 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(300, df = 3), 100, 3), start=c(1961, 1), frequency=12)
plot(z)                # multiple
plot(z, panel=points)  # same with points instead of lines
plot(z, plot.type="single", lty=1:3)
```

```
## A phase plot:
data(nhtemp)
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
library(ts)
plot(nhtemp, lag(nhtemp, 1), cex = .8, col="blue",
     main = "Lag plot of New Haven temperatures")


library(ts)

data(sunspots)
## xy.lines and xy.labels are FALSE for large series:
plot(lag(sunspots, 1), sunspots, pch = ".")

data(EuStockMarkets)
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)

detach("package:ts")
```

`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 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`.

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.

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.

See Also

xy.coords, plot.xy, plot.default.

Examples

```
##--- An example for the use of  'asp' :
library(mva)
data(eurodist)
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, names(eurodist), cex=0.5)
```

---

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=1, lty="solid", col=par("fg"), bg=NA, cex=1, ...)
```

Arguments

xy	A four-element list as results from xy.coords.
type	1 character code.
pch	character or integer code for kind of points/lines, see points.default.
lty	line type code, see lines.
col	color code or name, see colors, palette.
bg	background ("fill") color for open plot symbols.
cex	character expansion.
...	further graphical parameters.

See Also

plot, plot.default, points, lines.

Examples

```
points.default # to see how it calls "plot.xy(xy.coords(x, y), ...)"
```

`plotmath` *Mathematical Annotation in R*

Description

If the `text` argument to one of the text-drawing functions (`text`, `mtext`, `axis`) 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).

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 `example(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[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
`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
`infinity`	infinity 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
`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

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

`axis`, `mtext`, `text`, `title`

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),
     lab = 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")
tt <- 1.23 ; mtext(substitute(hat(theta) == that, list(that= tt)))
for(i in 2:9)
    text(i,i+1, substitute(list(xi,eta) == group("(",list(x,y),")"),
                           list(x=i, y=i+1)))

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)

######
# create tables of mathematical annotation functionality
######
make.table <- function(nr, nc) {
    savepar <- par(mar=rep(0, 4), pty="s")
    plot(c(0, nc*2 + 1), c(0, -(nr + 1)),
         type="n", xlab="", ylab="", axes=FALSE)
    savepar
}

get.r <- function(i, nr) {
```

```
    i %% nr + 1
}

get.c <- function(i, nr) {
    i %/% nr + 1
}

draw.title.cell <- function(title, i, nr) {
    r <- get.r(i, nr)
    c <- get.c(i, nr)
    text(2*c - .5, -r, title)
    rect((2*(c - 1) + .5), -(r - .5), (2*c + .5), -(r + .5))
}

draw.plotmath.cell <- function(expr, i, nr, string = NULL) {
    r <- get.r(i, nr)
    c <- get.c(i, nr)
    if (is.null(string)) {
        string <- deparse(expr)
        string <- substr(string, 12, nchar(string) - 1)
    }
    text((2*(c - 1) + 1), -r, string, col="grey")
    text((2*c), -r, expr, adj=c(.5,.5))
    rect((2*(c - 1) + .5), -(r - .5), (2*c + .5), -(r + .5), border="grey")
}

nr <- 20
nc <- 2
oldpar <- make.table(nr, nc)
i <- 0
draw.title.cell("Arithmetic Operators", i, nr); i <- i + 1
draw.plotmath.cell(expression(x + y), i, nr); i <- i + 1
draw.plotmath.cell(expression(x - y), i, nr); i <- i + 1
draw.plotmath.cell(expression(x * y), i, nr); i <- i + 1
draw.plotmath.cell(expression(x / y), i, nr); i <- i + 1
draw.plotmath.cell(expression(x %+-% y), i, nr); i <- i + 1
draw.plotmath.cell(expression(x %/% y), i, nr); i <- i + 1
draw.plotmath.cell(expression(x %*% y), i, nr); i <- i + 1
draw.plotmath.cell(expression(-x), i, nr); i <- i + 1
draw.plotmath.cell(expression(+x), i, nr); i <- i + 1
draw.title.cell("Sub/Superscripts", i, nr); i <- i + 1
draw.plotmath.cell(expression(x[i]), i, nr); i <- i + 1
draw.plotmath.cell(expression(x^2), i, nr); i <- i + 1
draw.title.cell("Juxtaposition", i, nr); i <- i + 1
draw.plotmath.cell(expression(x * y), i, nr); i <- i + 1
draw.plotmath.cell(expression(paste(x, y, z)), i, nr); i <- i + 1
draw.title.cell("Lists", i, nr); i <- i + 1
draw.plotmath.cell(expression(list(x, y, z)), i, nr); i <- i + 1
# even columns up
i <- 20
draw.title.cell("Radicals", i, nr); i <- i + 1
draw.plotmath.cell(expression(sqrt(x)), i, nr); i <- i + 1
draw.plotmath.cell(expression(sqrt(x, y)), i, nr); i <- i + 1
draw.title.cell("Relations", i, nr); i <- i + 1
draw.plotmath.cell(expression(x == y), i, nr); i <- i + 1
draw.plotmath.cell(expression(x != y), i, nr); i <- i + 1
draw.plotmath.cell(expression(x < y), i, nr); i <- i + 1
```

```
draw.plotmath.cell(expression(x <= y), i, nr); i <- i + 1
draw.plotmath.cell(expression(x > y), i, nr); i <- i + 1
draw.plotmath.cell(expression(x >= y), i, nr); i <- i + 1
draw.plotmath.cell(expression(x %~~% y), i, nr); i <- i + 1
draw.plotmath.cell(expression(x %=~% y), i, nr); i <- i + 1
draw.plotmath.cell(expression(x %==% y), i, nr); i <- i + 1
draw.plotmath.cell(expression(x %prop% y), i, nr); i <- i + 1
draw.title.cell("Typeface", i, nr); i <- i + 1
draw.plotmath.cell(expression(plain(x)), i, nr); i <- i + 1
draw.plotmath.cell(expression(italic(x)), i, nr); i <- i + 1
draw.plotmath.cell(expression(bold(x)), i, nr); i <- i + 1
draw.plotmath.cell(expression(bolditalic(x)), i, nr); i <- i + 1

# Need fewer, wider columns for ellipsis ...
nr <- 20
nc <- 2
make.table(nr, nc)
i <- 0
draw.title.cell("Ellipsis", i, nr); i <- i + 1
draw.plotmath.cell(expression(list(x[1], ..., x[n])), i, nr); i <- i + 1
draw.plotmath.cell(expression(x[1] + ... + x[n]), i, nr); i <- i + 1
draw.plotmath.cell(expression(list(x[1], cdots, x[n])), i, nr); i <- i + 1
draw.plotmath.cell(expression(x[1] + ldots + x[n]), i, nr); i <- i + 1
draw.title.cell("Set Relations", i, nr); i <- i + 1
draw.plotmath.cell(expression(x %subset% y), i, nr); i <- i + 1
draw.plotmath.cell(expression(x %subseteq% y), i, nr); i <- i + 1
draw.plotmath.cell(expression(x %supset% y), i, nr); i <- i + 1
draw.plotmath.cell(expression(x %supseteq% y), i, nr); i <- i + 1
draw.plotmath.cell(expression(x %notsubset% y), i, nr); i <- i + 1
draw.plotmath.cell(expression(x %in% y), i, nr); i <- i + 1
draw.plotmath.cell(expression(x %notin% y), i, nr); i <- i + 1
draw.title.cell("Accents", i, nr); i <- i + 1
draw.plotmath.cell(expression(hat(x)), i, nr); i <- i + 1
draw.plotmath.cell(expression(tilde(x)), i, nr); i <- i + 1
draw.plotmath.cell(expression(ring(x)), i, nr); i <- i + 1
draw.plotmath.cell(expression(bar(xy)), i, nr); i <- i + 1
draw.plotmath.cell(expression(widehat(xy)), i, nr); i <- i + 1
draw.plotmath.cell(expression(widetilde(xy)), i, nr); i <- i + 1
draw.title.cell("Arrows", i, nr); i <- i + 1
draw.plotmath.cell(expression(x %<->% y), i, nr); i <- i + 1
draw.plotmath.cell(expression(x %->% y), i, nr); i <- i + 1
draw.plotmath.cell(expression(x %<-% y), i, nr); i <- i + 1
draw.plotmath.cell(expression(x %up% y), i, nr); i <- i + 1
draw.plotmath.cell(expression(x %down% y), i, nr); i <- i + 1
draw.plotmath.cell(expression(x %<=>% y), i, nr); i <- i + 1
draw.plotmath.cell(expression(x %=>% y), i, nr); i <- i + 1
draw.plotmath.cell(expression(x %<=% y), i, nr); i <- i + 1
draw.plotmath.cell(expression(x %dblup% y), i, nr); i <- i + 1
draw.plotmath.cell(expression(x %dbldown% y), i, nr); i <- i + 1
draw.title.cell("Symbolic Names", i, nr); i <- i + 1
draw.plotmath.cell(expression(Alpha - Omega), i, nr); i <- i + 1
draw.plotmath.cell(expression(alpha - omega), i, nr); i <- i + 1
draw.plotmath.cell(expression(infinity), i, nr); i <- i + 1
draw.plotmath.cell(expression(32 * degree), i, nr); i <- i + 1
draw.plotmath.cell(expression(60 * minute), i, nr); i <- i + 1
draw.plotmath.cell(expression(30 * second), i, nr); i <- i + 1
```

```
# Need even fewer, wider columns for typeface and style ...
nr <- 20
nc <- 1
make.table(nr, nc)
i <- 0
draw.title.cell("Style", i, nr); i <- i + 1
draw.plotmath.cell(expression(displaystyle(x)), i, nr); i <- i + 1
draw.plotmath.cell(expression(textstyle(x)), i, nr); i <- i + 1
draw.plotmath.cell(expression(scriptstyle(x)), i, nr); i <- i + 1
draw.plotmath.cell(expression(scriptscriptstyle(x)), i, nr); i <- i + 1
draw.title.cell("Spacing", i, nr); i <- i + 1
draw.plotmath.cell(expression(x ~~ y), i, nr); i <- i + 1

# Need fewer, taller rows for fractions ...
# cheat a bit to save pages
par(new = TRUE)
nr <- 10
nc <- 1
make.table(nr, nc)
i <- 4
draw.plotmath.cell(expression(x + phantom(0) + y), i, nr); i <- i + 1
draw.plotmath.cell(expression(x + over(1, phantom(0))), i, nr); i <- i + 1
draw.title.cell("Fractions", i, nr); i <- i + 1
draw.plotmath.cell(expression(frac(x, y)), i, nr); i <- i + 1
draw.plotmath.cell(expression(over(x, y)), i, nr); i <- i + 1
draw.plotmath.cell(expression(atop(x, y)), i, nr); i <- i + 1

# Need fewer, taller rows and fewer, wider columns for big operators ...
nr <- 10
nc <- 1
make.table(nr, nc)
i <- 0
draw.title.cell("Big Operators", i, nr); i <- i + 1
draw.plotmath.cell(expression(sum(x[i], i=1, n)), i, nr); i <- i + 1
draw.plotmath.cell(expression(prod(plain(P)(X == x), x)), i, nr); i <- i + 1
draw.plotmath.cell(expression(integral(f(x) * dx, a, b)), i, nr); i <- i + 1
draw.plotmath.cell(expression(union(A[i], i==1, n)), i, nr); i <- i + 1
draw.plotmath.cell(expression(intersect(A[i], i==1, n)), i, nr); i <- i + 1
draw.plotmath.cell(expression(lim(f(x), x %->% 0)), i, nr); i <- i + 1
draw.plotmath.cell(expression(min(g(x), x >= 0)), i, nr); i <- i + 1
draw.plotmath.cell(expression(inf(S)), i, nr); i <- i + 1
draw.plotmath.cell(expression(sup(S)), i, nr); i <- i + 1

make.table(nr, nc)
i <- 0
draw.title.cell("Grouping", i, nr); i <- i + 1
draw.plotmath.cell(expression((x + y)*z), i, nr); i <- i + 1
draw.plotmath.cell(expression(x^y + z), i, nr); i <- i + 1
draw.plotmath.cell(expression(x^(y + z)), i, nr); i <- i + 1
# have to do this one by hand
draw.plotmath.cell(expression(x^{y + z}), i, nr, string="x^{y + z}"); i <- i + 1
draw.plotmath.cell(expression(group("(", list(a, b), "]")), i, nr); i <- i + 1
draw.plotmath.cell(expression(bgroup("(", atop(x, y), ")")), i, nr); i <- i + 1
draw.plotmath.cell(expression(group(lceil, x, rceil)), i, nr); i <- i + 1
draw.plotmath.cell(expression(group(lfloor, x, rfloor)), i, nr); i <- i + 1
draw.plotmath.cell(expression(group("|", x, "|")), i, nr); i <- i + 1
```

```
par(oldpar)
```

`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.
`table`	the values to be matched against.
`nomatch`	the value 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.

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`.

Note

Versions of R prior to 1.0.0 had a different behaviour that was seriously incompatible with S (and the current version) when `duplicates.ok = TRUE`.

Author(s)

Of this version, B. D. Ripley.

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"))
```

---

png *BMP, JPEG and PNG graphics devices*

Description

A graphics device for BMP, JPEG or PNG format bitmap files.

Usage

```
bmp(filename="Rplot.bmp", width=480, height=480, pointsize=12,
    bg="white")
jpeg(filename="Rplot.jpg", width=480, height=480, pointsize=12,
     quality=75, bg="white")
png(filename="Rplot.png", width=480, height=480, pointsize=12,
    bg="white")
```

Arguments

filename	the name of the output file.
width	the width of the device in pixels.
height	the height of the device in pixels.
pointsize	the default pointsize of plotted text, intepreted at 72 dpi, so one point is approximately one pixel.
bg	the initial bacground colour: can be overridden by setting par(”bg”).
quality	the ‘quality’ of the JPEG image, as a percentage. Smaller values will give more compression but also more degradation of the image.

Details

Plots in PNG and JPEG format can easily be converted to many other bitmap formats, and both can be displayed in most 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. The BMP format is standard on Windows, and supported elsewhere.

png supports transparent backgrounds: use bg="transparent". Not all PNG viewers render files with transparency correctly.

Value

A plot device is opened: nothing is returned to the R interpreter.

Note

These devices effectively plot on a hidden screen and then copy the image to the required format. This means that they have the same colour handling as the actual screen device, and work best if that is set to a 24-bit or 32-bit colour mode.

Author(s)

Guido Masarotto and Brian Ripley

See Also

Devices, dev.print, bitmap

Examples

```
## copy current plot to a (large) PNG file
dev.print(png, file="myplot.png", width=1024, height=768)

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()
```

---

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, ...)
points.default(x, y=NULL, type="p", pch=par("pch"), col=par("col"),
               bg=NA, cex=1, ...)
```

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`.

`pch` plotting "character", i.e. symbol to use. `pch` can either be a `character` or an integer code for 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.

`col` color code or name, see `colors`, `palette`.

`bg` background ("fill") color for open plot symbols

`cex` character expansion

`...` Further graphical parameters (see `plot.xy` and `par`) may also be supplied as arguments.

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 are permitted as arguments to this function.

See Also

`plot`, `lines`, and the underlying "primitive" `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 <- 1:(np <- 25+11); k <- floor(sqrt(np)); dd <- c(-1,1)/2
rx <- dd + range(ix <- (ipch-1) %/% k)
ry <- dd + range(iy <- 3 + (k-1)-(ipch-1) %% k)
pch <- as.list(ipch)
pch[25+ 1:11] <- as.list(c("*",".", "o","0","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)
}
```

`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 positive 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.

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 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.

Usage

```
poly(x, degree=1)
```

Arguments

`x`	a numeric vector at which to evaluate the polynomial
`degree`	the degree of the polynomial

Value

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 `"coefs"` which contains the centring and normalization constants used in constructing the orthogonal polynomials.

Note

This routine is intended for statistical purposes such as `contr.poly`: it does not attempt to orthogonalize to machine accuracy.

Author(s)

B.D. Ripley

See Also

`contr.poly`

Examples

```
poly(1:10, 3)
```

`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 = NULL, xpd = NULL, ...)
```

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. 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`	the color for filling the polygon. The default, `NA`, is to leave polygons unfilled.
`border`	the color to draw the border. The default, `NULL`, uses `par("fg")`. Use `border=0` to omit borders.
`lty`	the line type to be used, as in `par`.
`xpd`	(where) should clipping take place? Defaults to `par("xpd")`.
`...`	graphical parameters can be given as arguments to `polygon`.

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 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-intesecting polygons.

Author(s)

The code implementing polygon shading was donated by Kevin Buhr ⟨buhr@stat.wisc.edu⟩.

See Also

`segments` for even more flexibility, `lines`, `rect`, `box`, `abline`.

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))
```

---

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.

Value

A complex vector of length $n-1$, where n is `length(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)
```

`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 NULL, which is how package:base is represented.
pos.to.env(length(search()))
```

`postscript` *PostScript Graphics*

Description

`postscript` starts the graphics device driver for producing PostScript graphics.

The auxiliary function `ps.options` can be used to set and view (if called without arguments) default values for the arguments to `postscript`.

Usage

```
postscript(file = ifelse(onefile, "Rplots.ps", "Rplot%03d.ps"),
           onefile = TRUE,
           paper, family, encoding, bg, fg,
           width, height, horizontal, pointsize,
           pagecentre, print.it, command)

ps.options(paper, horizontal, width, height, family, encoding,
           pointsize, bg, fg,
           onefile = TRUE, print.it = FALSE, append = FALSE,
           reset = FALSE, override.check = FALSE)
.PostScript.Options
```

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`. For use with `onefile=FALSE` give a `printf` format such as `"Rplot%d.ps"` (the default in that case).
`...`	further options for `postscript()`.
`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.
`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 on each side.

`family` the font family to be used. EITHER a single character string OR a character vector of length four or five. See the section 'Families'.

`encoding` the name of an encoding file. Defaults to "WinAnsi.enc" in the 'R_HOME/afm' directory, which is used if the path does not contain a path separator. An extension `".enc"` can be omitted.

`pointsize` the default point size to be used.

`bg` the default background color to be used. If "white" (or a specification equivalent to white), no background is painted.

`fg` the default foreground color to be used.

`onefile` logical: if true (the default) allow multiple figures in one file. If false, generate a file name containing the page number and give EPSF header and no `DocumentMedia` comment.

`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"`.

`append` logical; currently **disregarded**; just there for compatibility reasons.

`reset, override.check`
logical arguments passed to `check.options`. See the Examples.

Details

`postscript` opens the file `file` and the PostScript commands needed to plot any graphics requested are stored in 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'.

Only the first of these will work on Windows, and the default `"printcmd"` is empty and will give an error if `print.it=TRUE` is used. Suitable commands to spool a PostScript file to a printer can be found in 'RedMon' suite available from http://www.cs.wisc.edu/~ghost/index.html. The command will be run in a minimized window. GSView 4.x provides `gsprint.exe` which may be even more convenient (it requires GhostScript version 6.0 or later).

The postscript produced by R 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"`.

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.)

As ISOLatin1 encoding is used, - is set as a minus and not as a hyphen. Supply a hyphen (character 173) if that is what you need.

`ps.options` needs to be called before calling `postscript`, and the default values it sets can be overridden by supplying arguments to `postscript`.

Families

The argument `family` specifies the 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 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 approriate metrics by using these names. To make these easier to remember, `"URWHelvetica" == "NimbusSan"` and `"URWTimes" == "NimbusRom"` are also supported.

It is also possible to specify `family="ComputerModern"`. This is 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.)

If the second form of argument `"family"` is used, it should be a character vector of four or five paths to Adobe Font Metric files for the regular, bold, italic, bold italic and (optionally) symbol fonts to be used. If these paths do not contain the file separator, they are taken to refer to files in the R directory 'R_HOME/afm'. Thus the default Helvetica family can be specified by `family = c("hv______.afm", "hvb_____.afm", "hvo_____.afm", "hvbo____.afm", "sy______.afm")`. It is the user's responsibility to check that suitable fonts are made available, and that they contain the needed characters when re-encoded. The fontnames used are taken from the `FontName` fields of the `afm` files. The software including the PostScript plot file should either embed the font outlines (usually from `.pfb` or `.pfa` files) or use DSC comments to instruct the print spooler to do so.

Encodings

Encodings describe which glyphs are used to display the character codes (in the range 0–255). By default 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.

None of this will matter if only ASCII characters (codes 32–126) are used as all the encodings 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. Three other encodings are supplied with R: `"WinAnsi.enc"` and `"MacRoman.enc"` correspond to the encodings normally used on Windows and MacOS (at least by Adobe), and `"PDFDoc.enc"` is the first 256 characters of the Unicode encoding, the standard for PDF.

If you change 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 one 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 ISOLatin1.

Author(s)

Support for Computer Modern fonts is based on a contribution by Brian D'Urso ⟨durso@hussle.harvard.edu⟩.

See Also

`Devices`, `check.options` which is called from both `ps.options` and `postscript`.

Examples

```
# open the file "foo.ps" for graphics output
postscript("foo.ps")
# produce the desired graph(s)
dev.off()              # turn off the postscript device
options(printcmd='redpr -P"\\markov\lw"')
postscript(file=tempfile("R.ps"), print.it=TRUE)
# produce the desired graph(s)
dev.off()              # send plot file to the 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")
## The resultant postscript file can be used by dvips -Ppfb -j0.



stopifnot(unlist(ps.options()) == unlist(.PostScript.Options))
ps.options(bg = "pink")
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:
str(ps.options(width=0:12, onefile=1, bg=pi,
               override.check = c(FALSE,TRUE,FALSE)))
```

`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-negative, 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.

See Also

`make.link`, `family`

Examples

```
power()
quasi(link=power(1/3))[c("linkfun", "linkinv")]
```

---

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 generate 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.

See Also

`qqplot`, `qqnorm`.

Examples

```
ppoints(4) # the same as  ppoints(1:4)
ppoints(10)
ppoints(10, a=1/2)
```

---

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

```
data(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

```
data(precip)
dotchart(precip[order(precip)], main = "precip data")
title(sub = "Average annual precipitation (in.)")
```

---

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.

The function `predict.lm` makes predictions based on the results produced by `lm`.

Usage

```
predict (object, ...)
```

Arguments

object	a model object for which prediction is desired.
...	additional arguments affecting the predictions produced.

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.

See Also

predict.lm.

Examples

```
## All the "predict" methods available in your current search() path:
for(fn in methods("predict"))
   cat(fn,":\n\t",deparse(args(get(fn))),"\n")
```

---

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

```
predict(object, newdata = NULL, type = c("link", "response", "terms"),
            se.fit = FALSE, dispersion = NULL, terms = NULL, ...)
```

Arguments

object	a fitted object of class inheriting from "glm".
newdata	optionally, a new data frame from which to make the predictions. 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
...	further arguments passed to or from other methods.

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.

Author(s)

B.D. Ripley

See Also

`glm`

Examples

```
## example from Venables and Ripley (1997, pp. 231-3.)
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.lm` *Predict method for Linear Model Fits*

Description

Predicted values based on linear model object

Usage

```
predict(object, newdata, se.fit = FALSE, scale = NULL, df = Inf,
        interval = c("none", "confidence", "prediction"),
        level = 0.95, type = c("response", "terms"),
        terms = NULL, ...)
```

Arguments

`object`	Object of class inheriting from `"lm"`
`newdata`	Data frame in which to predict
`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)
`...`	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`.

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 predictions
`residual.scale`	residual standard deviations
`df`	degrees of freedom for residual

Note

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.

See Also

The model fitting function `lm`, `predict`.

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")
```

---

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`.

---

presidents *Approval Rating 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

```
data(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

```
data(presidents)
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

```
data(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

```
data(pressure)
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")
```

`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

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`.

. . . . . .

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.

The advantage of `.Primitive` over `.Internal` functions is the potential efficiency of argument passing.

Usage

```
.Primitive(name)
```

Arguments

`name`	name of the R function.

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, ...)

print.factor(x, quote = FALSE, ...)
print.ordered(x, quote = FALSE, ...)
```

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.

Details

The default method, `print.default` has its own help page. Use `methods("print")` to get all the methods for the `print` generic.

See `noquote` as an example of a class whose main purpose is a specific `print` method.

See Also

The default method `print.default`, and help for the methods above; further `options`, `noquote`.

Examples

```
ts(1:20)#-- print  is the ‘‘Default function’’ --> print.ts(.) is called
rr <- for(i in 1:3) print(1:i)
rr
```

`print.coefmat` *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

```
print.coefmat(x, digits=max(3, getOption("digits") - 2),
              signif.stars = getOption("show.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(",
              na.print = "", ...)
```

Arguments

`x`	a numeric matrix like object, to be printed.
`digits`	number of 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`.
`dig.tst`	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 `link{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 *iff* `P.values` (above).
`na.print`	a character string to code `NA` values in printed output.
`...`	further arguments for `print`.

Details

Despite its name, this is *not* (yet) a method for the generic `print` function, because there is no class `"coefmat"`.

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)")
print.coefmat(cmat[,1:3])
print.coefmat(cmat)
options(show.coef.Pvalues = FALSE)
print.coefmat(cmat, digits=2)
print.coefmat(cmat, digits=2, P.values = TRUE)
options(show.coef.Pvalues = TRUE)# revert
```

---

print.data.frame *Printing Data Frames*

Description

These functions create or manipulate 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

```
print(x, ..., digits = NULL, quote = FALSE, right = TRUE)
```

Arguments

`x`	object of class `data.frame`.
`...`	optional arguments to `print` or `plot` methods.
`digits`	the number of significant digits to be printed.
`quote`	logical, indicating whether or not strings (`character`s) should be printed with surrounding quotes.
`right`	logical, indicating whether or not strings should be right-aligned. The default is left-alignment.

Value

For the `print` method (`print.data.frame`), see `print.matrix`.

See Also

`data.frame`.

`print.default` *Default Printing*

Description

`print.default` is the *default* method of the generic `print` function which prints its argument.

`print.atomic` is almost the same and exists purely for compatibility reasons.

Usage

```
print.default(x, digits = NULL, quote = TRUE, na.print = NULL,
              print.gap = NULL, right = FALSE, ...)
print.atomic(x, quote = TRUE, ...)
```

Arguments

`x`	the object to be printed.
`digits`	a non-null value for `digits` specifies the number of significant digits to be printed in values. If `digits` is `NULL`, the value of `digits` set by `options` is used.
`quote`	logical, indicating whether or not strings (`character`s) should be printed with surrounding quotes.
`na.print`	a character string which is used to indicate `NA` values in printed output, or `NULL` meaning `"NA"`.
`print.gap`	an integer, giving the spacing between adjacent columns in printed matrices and arrays, or `NULL` meaning 1.
`right`	logical, indicating whether or not strings should be right-aligned. The default is left-alignment.
`...`	(further arguments, currently disregarded)

See Also

The generic `print`, `options`. The `"noquote"` class and print method.

Examples

```
pi
print(pi, digits = 16)
LETTERS[1:16]
print(LETTERS, quote = FALSE)
```

`print.matrix` *Print Matrices*

Description

Pseudo-method for the `print` generic. Especially useful with the `right` argument which does not (yet) exist for `print.default`.

Usage

```
print.matrix(x, rowlab=character(0), collab=character(0),
             quote=TRUE, right=FALSE, na.print = NULL,
             print.gap = 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 `NA`s are printed. If this is non-null, its value is used to represent `NA`.
`print.gap`	not yet used.
`...`	arguments for other methods.

Details

`print.matrix` and `print.default` both print matrices, and each has at least an optional argument that the other lacks. Also, both directly dispatch into `.Internal` code directly instead of relying on each other. This mainly stems from historic compatibility and similar reasons should be changed in the future.

`prmatrix` is currently just an alias for `print.matrix`.

Value

Invisibly returns its argument, `x`.

See Also

`print.default`, and other `print` methods.

Examples

```
print.matrix(m6 <- diag(6), row = rep("",6), coll=rep("",6))

chm <- matrix(scan(system.file("help", "AnIndex", package = "eda"),
                   what = ""), , 2, byrow = TRUE)
chm  #-> print.default(.)  = 'same' as  print.matrix(chm)
print.matrix(chm, collab = paste("Column",1:3), right=TRUE, quote=FALSE)
```

print.ts *Printing Time-Series Objects*

Description

Print method for time series objects.

Usage

```
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)
```

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; it is common for them to be recorded to of the order of 1/100 second, and elapsed time is rounded to the nearest 1/100.

It is most useful for "timing" the evaluation of R expressions, which can be done conveniently with system.time.

Note

CPU times will be returned as `NA` on Windows 9x/ME systems, but are genuine times on NT4 and 2000 systems. Times of child processes are not available and will always be given as `NA`.

See Also

`system.time` for timing a valid R expression, `gc.time` for how much of the time was spent in garbage collection.

Examples

```
ptm <- proc.time()
for (i in 1:50) mad(runif(500))
proc.time() - ptm
```

---

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 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.

See Also

`sum`, `cumprod`, `cumsum`.

Examples

```
print(prod(1:7)) == print(gamma(8))
```

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` in package **nls**, `profile.glm` in package **MASS**, ...

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, ...)
proj.aov    (object, onedf = FALSE, unweighted.scale = FALSE, ...)
proj.aovlist(object, onedf = FALSE, unweighted.scale = FALSE, ...)
proj.default(object, onedf = TRUE, ...)
proj.lm     (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)

B.D. Ripley

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)
```

`prompt` *Produce Prototype of an R Documentation File*

Description

Facilitate the constructing of files documenting R functions.

Usage

```
prompt(object, ...)

prompt.default(object,
    filename = paste(name, ".Rd", sep = ""),
    force.function = FALSE, ...)

prompt.data.frame(object,
    filename = paste(name, ".Rd", sep = ""), ...)
```

Arguments

`object`	an R object, typically a function
`filename`	name of the output file
`force.function`	treat `object` as function in any case
`...`	further arguments passed to or from other methods.

Details

An ASCII file `filename` is produced containing the proper function and argument names of `object`. You have to edit it before adding the documentation to the source tree, i.e., (currently) to '$R_HOME/src/library/base/man/'.

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` library. We are trying to settle on a preferred format for the documentation.

Author(s)

Douglas Bates for `prompt.data.frame`

See Also

`help` and the chapter on "Writing R documentation" in "Writing R Extensions" (see the 'doc/manual' subdirectory of the R source tree).

Examples

```
prompt(plot.default)
prompt(interactive, force.function = TRUE)
unlink("plot.default.Rd")
unlink("interactive.Rd")

data(women) # data.frame
prompt(women)
unlink("women.Rd")

data(sunspots) # non-data.frame data
prompt(sunspots)
unlink("sunspots.Rd")
```

---

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)
```

`qqnorm` *Quantile-Quantile Plots*

Description

`qqnorm` is a generic functions 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, ...)
qqnorm(y, ylim, main = "Normal Q-Q Plot",
       xlab = "Theoretical Quantiles",
       ylab = "Sample Quantiles", plot.it = TRUE, ...)
qqline(y, ...)
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.
`plot.it`	logical. Should the result be plotted?
`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 corresponding y coordinates

See Also

`ppoints`.

Examples

```
y <- rt(200, df = 5)
qqnorm(y); qqline(y, col = 2)
qqplot(y, rt(300, df = 5))
data(precip)
qqnorm(precip, ylab = "Precipitation [in/yr] for 70 US cities")
```

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 (for complex matrices) the LAPACK routine ZGEQP3.

Usage

```
qr(x, tol=1e-07)
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)

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.
qr	a QR decomposition of the type computed by qr.
y, b	a vector or matrix of right-hand sides of equations.
a	A matrix or QR decomposition.
k	effective rank.

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 $\boldsymbol{Q}$ matrix.

All the above functions keep dimnames (and names) of x and y if there are.

qr.solve solves systems of equations via the QR decomposition.

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/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).
`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 complex case.
`pivot`	information on the pivoting strategy used during the decomposition.

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`.

The complex case uses column pivoting and does not attempt to detect rank-deficient matrices.

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

`qr.Q`, `qr.R`, `qr.X` for reconstruction of the matrices. `solve.qr`, `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 = 1)
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 $\boldsymbol{Q}$ or $\boldsymbol{X}$ matrices is to be made, or whether the $\boldsymbol{R}$ 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 $\boldsymbol{X}$. The default when `complete` is `FALSE` is the original $\boldsymbol{X}$ from which the qr object was constructed. The default when `complete` is `TRUE` is a square matrix with the original $\boldsymbol{X}$ 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 $\boldsymbol{Q}$ will be multiplied by the corresponding diagonal value.

Value

`qr.X` returns $\boldsymbol{X}$, the original matrix from which the qr object was constructed. 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 $\mathbf{Q}$, the order-nrow(X) orthogonal (unitary) transformation represented by `qr`. If `complete` is `TRUE`, $\mathbf{Q}$ has `nrow(X)` columns. If `complete` is `FALSE`, $\mathbf{Q}$ has `ncol(X)` columns. When `Dvec` is specified, each column of $\mathbf{Q}$ is multiplied by the corresponding value in `Dvec`.

`qr.R` returns $\mathbf{R}$, the upper triangular matrix such that `X == Q %*% R`. The number of rows of $\mathbf{R}$ is `nrow(X)` or `ncol(X)`, depending on whether `complete` is `TRUE` or `FALSE`.

See Also

`qr`, `qr.qy`.

Examples

```
data(LifeCycleSavings)
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 :
all((1 - X /( Q %*% R))< 100*.Machine$double.eps)#TRUE

dim(Qc <- qr.Q(qrstr, complete=TRUE)) # Square: dim(Qc) == rep(nrow(x),2)
all((crossprod(Qc) - diag(nrow(x))) < 10*.Machine $double.eps)

QD <- qr.Q(qrstr, D=1:p)      # QD == Q %*% diag(1:p)
all(QD - Q %*% diag(1:p)  < 8* .Machine$double.eps)

dim(Rc <- qr.R(qrstr, complete=TRUE)) # == dim(x)
dim(Xc <- qr.X(qrstr, complete=TRUE)) # square: nrow(x) ^ 2
all(Xc[,1:p] == X)
```

`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

```
data(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 a 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

```
data(quakes)
pairs(quakes, main = "Fiji Earthquakes, N = 1000", cex.main=1.2, pch=".")
```

---

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, probs = seq(0, 1, 0.25), na.rm = FALSE,
         names = TRUE, ...)
```

Arguments

`x`	numeric vectors whose sample quantiles are wanted.
`probs`	numeric vector 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`.
`...`	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.

`quantile(x,p)` as a function of `p` linearly interpolates the points ((i-1)/(n-1), ox[i]), where `ox <- order(x)` (the "order statistics") and `n <- length(x)`.

This gives `quantile(x, p) == (1-f)*ox[i] + f*ox[i+1]`, where `r <- 1 + (n-1)*p`, `i <- floor(r)`, `f <- r - i` *and* `ox[n+1] := ox[n]`.

`NA` and `NaN` values in `probs` are propagated to the result.

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)

n <- length(x) ## the following is exact, because 1/(1001-1) is exact:
stopifnot(sort(x) == quantile(x, probs = ((1:n)-1)/(n-1), names=FALSE))

n <- 777
ox <- sort(x <- round(rnorm(n),1))# round() produces ties
ox <- c(ox, ox[n]) #- such that ox[n+1] := ox[n]
p <- c(0,1,runif(100))
i <- floor(r <- 1 + (n-1)*p)
f <- r - i
all(abs(quantile(x,p) - ((1-f)*ox[i] + f*ox[i+1])) < 20*.Machine$double.eps)
```

---

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.

See Also

`.First` for setting things on startup.

Examples

```
## Unix-flavour example
.Last <- function() {
  cat("Now sending PostScript graphics to the printer:\n")
  system("lpr Rplots.ps")
  cat("bye bye...\n")
}
quit("yes")
```

R.home *Return the R Home Directory*

Description

Return the R home directory.

Usage

```
R.home()
```

Value

A character string giving the current home directory.

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 an `.Alias` to it for S compatibility), whereas `R.version.string` is simple `character` string, useful for plotting, etc.

Usage

```
R.Version()
R.version
R.version.string
```

Value

`R.Version` returns a list with components

`platform`	the platform for which R was built. Under Unix, a triplet of the form CPU-VENDOR-OS, as determined by the configure script. E.g, `"i586-unknown-linux"`.
`arch`	the architecture (CPU) R was built on/for.
`os`	the underlying operating system
`system`	CPU and OS.
`status`	the status of the version (e.g., `"Alpha"`)
`status.rev`	the status revision level
`major`	the major version number
`minor`	the minor version number
`year`	the year the version was released
`month`	the month the version was released
`day`	the day the version was released
`language`	always `"R"`.

See Also

`machine`.

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.

`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)
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.
`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 `"Marsaglia-Multicarry"`.

`"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}.

`"user-supplied":` Use a user-supplied generator. See `Random.user` for details.

`normal.kind` can be `"Kinderman-Ramage"` (the default) or `"Ahrens-Dieter"` or `"Box-Muller"` or `"user-supplied"`.

`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 `"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 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.

`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`.

`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`.

Author(s)

of RNGkind: Martin Maechler. Current implementation, B. D. Ripley

References

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.

See Also

`runif`, `rnorm`, ...

Examples

```
runif(1); .Random.seed; runif(1); .Random.seed
## If there is no seed,  a ‘‘random’’ new one is created:
rm(.Random.seed); runif(1); .Random.seed

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])
```

`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

```
##  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 SHLIB urand.c
R
> dyn.load("urand.so")
> RNGkind("user")
> runif(10)
> .Random.seed
> RNGkind(, "user")
```

```
> rnorm(10)
> RNGkind()
[1] "user-supplied" "user-supplied"
```

`randu` *Random Numbers from Congruential Generator*

Description

400 triples of successive random numbers were taken from the VAX FORTRAN function RANDU running under VMS 1.5.

Usage

```
data(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

```
## We could re-generate the dataset by the following R code
seed <- as.double(1)
RANDU <- function() {
    seed <<- ((2^16 + 3) * seed)
    seed/(2^31)
}
for(i in 1:400) {
    U <- c(RANDU(), RANDU(), RANDU(), RANDU(), RANDU())
    print(round(U[1:3], 6))
}
```

`range` *Range of Values*

Description

`range` returns a vector containing the minimum and maximum of all the given arguments.

Usage

```
range(..., na.rm = FALSE)
range.default(..., 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

This is a generic function; currently, it has only a default method (`range.default`).

It is also a member of the `Summary` group of functions, see `Methods`.

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`.

See Also

`min`, `max`, `Methods`.

Examples

```
print(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 numeric vector. Ties result in ranks being averaged.

Usage

```
rank(x, na.last = TRUE)
```

Arguments

x	a numeric vector.
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.

See Also

order and sort.

Examples

```
(r1 <- rank(x1 <- c(3,1,4,59,26)))
(r2 <- rank(x2 <- c(3,1,4,5,9,2,6,5,3,5))) # ties

## rank() is "idempotent": rank(rank(x)) == rank(x) :
stopifnot(rank(r1) == r1, rank(r2) == r2)
```

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

```
Rcmd Rdconv [options] file
Rcmd Rdindex [options] files
Rcmd Rd2dvi.sh [options] files
Rcmd Rd2txt [options] file
Rcmd 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) format 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.

`Rdindex` creates an index table from Rd files.

`Sd2Rd` converts S version 3 documentation format to Rd format.

Use `Rcmd foo --help` to obtain usage information on utility `foo`.

See Also

The chapter "Processing Rd format" in "Writing R Extensions" (see the Manuals sub-menu of the Help menu on the console).

`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 object, data and demo indices and 'TITLE' 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.

`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, digits = getOption("digits"))
as.table(x, ...)
```

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"`.
`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.

`as.table.ftable` converts a contingency table in flat matrix form to one in standard array form. This is a method for the generic function `as.table`.

References

Agresti, A. (1990) *Categorical data analysis.* New York: Wiley.

See Also

`ftable` for more information on flat contingencty 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)
```

`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, sep = "\t", as.is = FALSE,
         skip = 0, row.names, col.names, n = -1, ...)
```

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).
`sep`	character; the separator used internally; should be a character that does not occur in the file.
`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.
`...`	further arguments to be passed to `read.table`.

Details

Fields that are of zero-width or are wholly beyond the end of the line in `file` are replaced by `NA`.

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
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)
```

`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)
}
finger("root")  ## only works if your site provides a finger daemon
```

`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 = FALSE, 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 = "#")

read.csv(file, header = TRUE, sep = ",", quote="\"", dec=".",
         fill = TRUE, ...)

read.csv2(file, header = TRUE, sep = ";", quote="\"", dec=",",
         fill = TRUE, ...)

read.delim(file, header = TRUE, sep = "\t", quote="\"", dec=".",
         fill = TRUE, ...)

read.delim2(file, header = TRUE, sep = "\t", quote="\"", dec=",",
         fill = TRUE, ...)
```

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. `file` can also be a complete URL.
`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 or newlines.
`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 this conversion. Its value is either a vector of logicals (values are recycled if necessary), or a vector of numeric indices which specify which columns should not be converted to factors.

Note: to suppress all conversions including those of numeric columns, set `colClasses = "character"`.

`na.strings` a vector of strings which are to be interpreted as `NA` values. Blank fields are also considered to be missing values.

`colClasses` character. A vector of classes to be assumed for the columns. Recycled as necessary. If this is not one of the atomic vector classes (logical, integer, numeric, complex and character), there needs to be an `as` method for conversion from `"character"` to the specified class, or `NA` when `type.convert` is used.

`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.

`fill` logical. If `TRUE` then in case the rows have unequal length, blank fields are implicitly added.

`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.

`...` Further arguments to `read.table`.

Details

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.

`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 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.

Comment characters are allowed unless `comment.char = ""`, and 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.

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 five 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.

See Also

The *R Data Import/Export* manual.

`scan`, `type.convert`, `read.fwf` for reading *f*ixed *w*idth *f*ormatted input; `read.table.url` for "reading" data from the internet; `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)

readChar(con, nchars)
writeChar(object, con, nchars = nchar(object), eos = "")
```

Arguments

`con`	A connection object or a character string.
`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"`.
`n`	integer. The (maximal) number of records to be read. You can use an over-estimate 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 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.
`nchars`	integer, giving the lengths of (unterminated) character strings to be read or written.
`eos`	character. The terminator to be written after each string, followed by an ASCII `nul`; use `NULL` for no terminator at all.

Details

If the `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.

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 `NA`s, and the extended precision sizes are unlikely to be portable across platforms.

`readBin` and `writeBin` read and write C-style zero-terminated character strings. `readChar` and `writeChar` allow more flexibility, and can also be used on text-mode connections.

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 `readChar`, a character vector of length the number of items read (which might be less than `length(nchars)`).

For `writeBin` and `writeChar`, none.

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`.

Note that as R character strings cannot contain ASCII `nul`, strings read by `readChar` which contain such characters will appear to be shorted than requested, but the additional bytes are read from the file.

If the character length requested for `readChar` is longer than the string, as from version 1.4.0 what is available is returned.

See Also

The *R Data Import/Export* manual.

`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)

## test fixed-length strings
zz <- file("testbin", "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("testbin", "rb")
readChar(zz, nc)
readChar(zz, nchar(x)+3) # need to read the terminator explicitly
close(zz)
unlink("testbin")

## 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")
```

`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.

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)
```

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.

Details

If the `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 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.

Value

A character vector of length the number of lines read.

See Also

`connections`, `writeLines`, `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.

See Also

`do.call` and `call`.

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
```

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.

The format of recorded plots was changed in R 1.4.0: plots saved in earlier versions can still be replayed.

Value

`recordPlot` returns an object of class `"recordedplot"`, a list with components:

`displaylist`	The saved display list, as a pairlist.
`gpar`	The graphics state, as an integer vector.

`replayPlot` has no return value.

`rect` *Draw a Rectangle*

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 = NULL, border = NULL, lty = NULL, lwd = par("lwd"), xpd = NULL)
```

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. Non-positive values of `density` also inhibit the drawing of shading lines.
`angle`	angle (in degrees) of the shading lines.
`col`	color(s) to fill or shade the rectangle(s) with. The default `NULL`, or also `NA` do not fill, i.e., draw transparent rectangles.
`border`	color for rectangle border(s).
`lty`	line type for borders; defaults to `"solid"`.
`lwd`	width for borders.
`xpd`	logical (“e**xp**an**d**”); defaults to `par("xpd")`. See `par(xpd= )`.

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.

Examples

```
## set up the plot region:
op <- par(bg = "thistle")
plot(c(100, 250), c(300, 450), type = "n", xlab="",
     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\"))")
par(op)
```

---

reg.finalizer *Finalization of objects*

Description

Registers an R function to be called upon garbage collection of object.

Usage

```
reg.finalizer(e, f)
```

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.

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.

Examples

```
f <- function(e) print("cleaning....")
g <- function(x){e<-environment(); reg.finalizer(e,f)}
g()
gc() # trigger cleanup
```

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`.

Author(s)

B. D. Ripley

See Also

`factor`, `contr.treatment`

Examples

```
data(warpbreaks)
warpbreaks$tension <- relevel(warpbreaks$tension, ref="M")
summary(lm(breaks ~ wool + tension, data=warpbreaks))
```

REMOVE *Remove Add-on Packages*

Description

Use `Rcmd REMOVE pkgs` to remove the packages in `pkgs` from the default library tree (which is rooted at '$R_HOME/library').

To remove from the library tree `lib` instead of the default one, use `Rcmd REMOVE -l lib pkgs`.

Usage

```
Rcmd REMOVE [-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.

See Also

INSTALL

---

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 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, supplied individually and/or as a character vector
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.

See Also

ls, objects

Examples

```
tmp <- 1:4
## work with tmp  and cleanup
rm(tmp)


## 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())
```

`remove.packages` *Remove Installed Packages*

Description

Removes installed packages and updates index information as necessary.

Usage

```
remove.packages(pkgs, lib)
```

Arguments

`pkgs`	a character vector with the names of the packages to be removed.
`lib`	a character string giving the library directory to move the packages from.

See Also

`install.packages` for installing packages.

`rep` *Replicate Elements of Vectors and Lists*

Description

`rep` replicates the values in `x` according to the values given in `times` and `length.out`.

Usage

```
rep(x, times, length.out)
```

Arguments

`x`	a vector (of any mode including a list) or a pairlist.
`times`	integer. A vector giving the number of times to repeat each element. Either of length 1 or `length(x)`.
`length.out`	integer. (Optional.) The desired length of the output vector.

Details

If `times` consists of a single integer, the result consists of the values in `x` repeated this many times. If `times` is a vector of the same length as `x`, 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 `length.out` and `times` are specified, `times` determines the replication, and `length.out` can be used to truncate the output vector (or extend it by `NA`s).

Non-integer values of `times` will be truncated towards zero. If `times` is a computed quantity it is prudent to add a small fuzz.

Note

If the original vector has names, these are also replicated and so will almost always contain duplicates.

See Also

`seq`, `sequence`.

Examples

```
rep(1:4, 2)
rep(1:4, c(2,2,2,2))     # same as above, length 8.
rep(1:4, c(2,1,2,1))
rep(1:4, c(2,2,2,2), 4)  # first 4 only.
rep(1:4, c(2,2,2,2), 10) # 8 intgers plus two NAs

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)
```

`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.

`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)

B. D. Ripley

See Also

`model.tables`

Examples

```
## From Venables and Ripley (1997) p.210.
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 dataframe 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, fix.row.names = TRUE,
    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/individual
`idvar`	The variable in long format that identifies multiple records from the same group/individual. This variable 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`	`"wide" to reshape to wide format, "long" to reshape to long format`
`fix.row.names`	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 by specifying just the `direction` argument. 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

```
data(Indometh,package="nls")
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
data(state)
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))
```

`reshapeLong-deprecated` *Reshape data frame to long format*

Description

(EXPERIMENTAL). Convert data frame with repeated measurements in wide format with repeated observations in multiple variables across rows to long format, 1 row per observation.

Usage

```
reshapeLong(x, jvars, ilev=row.names(x), jlev=names(x)[jvars],
            iname="reshape.i", jname="reshape.j", vname="reshape.v")
```

Arguments

`x`	The data frame to convert
`jvars`	The variables to reshape by
`ilev`	Levels of 1st indexing factor
`jlev`	Levels of 2nd indexing factor
`iname`	Name of 1st indexing factor
`jname`	Name of 2nd indexing factor
`vname`	Name of variable holding the combined values of the "across" variables

Details

This causes the values in `jvars` to be combined into a single variable, all other variables being replicated the relevant number of times. Two factors are added to the data frame indicating rows and columns of the original data format.

Value

The reshaped data frame.

Bugs

There ought to be a way to have multiple sets of `jvars` variables.

Note

The same substitution tricks are used for `jvars` as for the `select` argument to `subset`. I.e. the argument is treated as an expression and variables are replaced with their number in the data frame, allowing ranges of variables to be specified.

See Also

`reshapeWide`, `stack`

Examples

```
(dd<-as.data.frame(matrix(1:24,4)))
reshapeLong(dd,V3:V5)
```

`reshapeWide-deprecated`

Reshape data frame to wide format

Description

(EXPERIMENTAL). Convert data frame with repeated measurements in long format, 1 row per observation to wide format with repeated observations in multiple variables across rows.

Usage

```
reshapeWide(x, i=reshape.i, j=reshape.j,
            val=reshape.v, jnames=levels(j))
```

Arguments

`x`	The data frame to convert
`i`	Factor or numeric. Indicates observations in same row of the wide format
`j`	Factor or numeric. Indicates observations in same column of wide format
`val`	Value to reshape for.
`jnames`	Names of new variables in reshaped frame

Details

`val`, `i`, and `j` are interpreted relative to `x`. This essentially places the values of `val` in a table defined by `i` and `j`, which are assumed to span the data set (exactly one observation in each cell of their cross-tabulation). Any other variables in the data frame will be assumed to have the same value for all values of `j` (given `i`) and are collapsed into a single value.

Value

The reshaped data frame.

WARNING

There is no check that the names given to the new columns are valid identifiers or unique within the data frame. Variables in `x` are not checked to have constant values given `i`.

Bugs

There ought to be a way to have multiple `val` variables.

Note

The defaults are set to use the names that result from a call to `reshapeLong`.

See Also

reshapeLong, unstack

Examples

```
dd<-as.data.frame(matrix(1:24,4))
(dd1<-reshapeLong(dd,V3:V5))
reshapeWide(dd1)
reshapeWide(dd1,jnames=c("A","B","C"))
```

`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 'residuals' and not 'resid'.)

Methods can make use of naresid methods to compensate for the omission of missing values. The default method does.

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`.

See Also

`coefficients`, `fitted.values`, `glm`, `lm`.

`restart` *Restart an Expression*

Description

`restart` performs a type of non-local return.

Usage

```
restart(on = TRUE)
```

Arguments

`on`	if true a jump point is set; if false the jump point is removed.

Details

When `restart` is called with `on = TRUE` the evaluator marks that function as a return point. Any errors or signals (such as control-C on Unix) cause control to return to the start of the function containing the call to `restart`. The most recently established function is always entered first.

Note

The direct use of `restart` is likely to result in an infinite loop. Use `try` unless you are **sure** you know what you are doing.

See Also

`options` for setting error handlers and suppressing the printing of error messages.

`rev` *Reverse a Vector's Elements*

Description

`rev` provides a reversed version of its argument. It can be used in combination with `sort` to obtain vectors sorted into descending order.

Usage

```
rev(x)
```

Arguments

x	a vector.

See Also

seq, sort.

Examples

```
x <- c(1:5,5:3)
# sort into descending order
rev(sort(x))
stopifnot(rev(1:7) == 7:1)#- don't need 'rev' here
```

---

rgb *RGB Color Specification*

Description

This function creates "colors" corresponding to the given intensities (between 0 and 1) of the red, green and blue primaries. The names argument may be used to provide names for the colors.

The values returned by rgb can be used with a col= specification in graphics functions or in par.

Usage

```
rgb(red, green, blue, names=NULL)
```

Arguments

red, blue, green	vector in $[0, 1]$.
names	character. The names for the resulting vector.

See Also

col2rgb the "inverse" for translating R colors to RGB vectors.
rainbow, hsv, gray.

Examples

```
rgb(0,1,0)
u01 <- seq(0,1, length=11); all(rgb(u01,u01,u01) == gray(u01))
reds <- rgb((0:15)/15, g=0,b=0, names=paste("red",0:15,sep="."))
reds
```

`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

```
data(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.

`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)
print(x, digits = getOption("digits"), ...)
inverse.rle(x, ...)
```

Arguments

`x`	a simple vector [as argument of rle()], a class "rle" object [as argument of `print()` or `inverse.rle()`].
`digits, ...`	potentially further arguments to the corresponding method.

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`.

`round` rounds the values in its first argument to the specified number of decimal places (default 0). Note that for rounding off a 5, the IEEE standard is used, *"go to the even digit"*. Therefore `round(0.5)` is `0` and `round(-1.5)` is `-2`.

`signif` rounds the values in its first argument to the specified number of significant digits.

`trunc` takes a single numeric argument `x` and returns a numeric vector containing the integers by truncating the values in `x` toward `0`.

`zapsmall` determines a `digits` argument `dr` for calling `round(x, digits = dr)` such that values "close to zero" values are "zapped", i.e., treated as `0`.

Usage

```
ceiling(x)
floor(x)
round(x, digits = 0)
signif(x, digits = 6)
trunc(x)
zapsmall(x, digits= getOption("digits"))
```

Arguments

`x`	a numeric vector.
`digits`	integer indicating the precision to be used.

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))
stopifnot(
 trunc(x1) == as.integer(x1),
 non.int == (ceiling(x1) != trunc(x1) | trunc(x1) != floor(x1)),
 (signif(x1, 1) != round(x1,1)) == (non.int & abs(x1) > 1)
)

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

```
round(x, units=c("secs", "mins", "hours", "days"))
trunc.POSIXt(x, units=c("secs", "mins", "hours", "days"))
```

Arguments

x	an object inheriting from "POSIXt".
units	one of the units listed. Can be abbreviated.

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.

Value

An object of class "POSIXlt".

Note

trunc is not generic, so trunc.POSIXt has to be called explicitly.

See Also

DateTimeClasses

Examples

```
round(.leap.seconds + 1000, "hour")
trunc.POSIXt(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`.

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/colnames` *Row and Columnn Names*

Description

Retrieve or set the row or column names of an object (the first or second component of its `dimnames`).

Usage

```
rownames(x, do.NULL = TRUE, prefix = "row")
rownames(x) <- namevector

colnames(x, do.NULL = TRUE, prefix = "col")
colnames(x) <- namevector
```

Arguments

`x`	
`do.NULL`	logical. Should this create names if they are `NULL`?
`prefix`	for created names.

Details

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 `dimnames(x)[[i]]` (`i` = 1 or 2) is `NULL`.

See Also

`dimnames`, `case.names`, `variable.names`.

Examples

```
m0 <- matrix(NA, 4, 0)
m2 <- cbind(1,1:4)
rownames(m0)

colnames(m2, do.NULL = FALSE)
colnames(m2) <- c("x","Y")
rownames(m2) <- rownames(m2, do.NULL = FALSE, prefix = "Obs.")
m2
```

rowsum *Give Row Sums of a Matrix, Based on a Grouping Variable*

Description

Compute sums across rows of a matrix for each level of a grouping variable.

Usage

```
rowsum(x, group, reorder = TRUE)
```

Arguments

x	a matrix or vector of numeric data. Missing values are allowed.
group	a vector giving the grouping, with one element per row of x. Missing values are not allowed.
reorder	if TRUE, then the result will be in order of sort(unique(group)), if FALSE, it will be in the order that rows were encountered (and may run faster for large matrices). The default is to reorder the data, so as to agree with tapply (see example below).

Value

a matrix containing the sums. There will be one row per unique value of group.

Author(s)

Terry Therneau

See Also

tapply

Examples

```
x <- matrix(runif(100), ncol=5)
group <- sample(1:8, 20, TRUE)
xsum <- rowsum(x, group)

## same result another way, slower, and temp may be much larger than x
temp <- model.matrix(~ a - 1, data.frame(a=as.factor(group)))
xsum2<- t(temp) %*% x

## same as last one, but really slow
xsum3 <- tapply(x, list(group[row(x)], col(x)), sum)
```

`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)
```

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.

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. The Perl script `Rcmd Rprof` can be used to process the output file to produce a summary of the usage; use `Rcmd Rprof --help` for usage information.

Exactly what the time interval measures is subtle: it is time that the R process is running and executing an R command. It is not however just CPU time, for if `readline()` is waiting for input, that counts (on Windows, but not on Unix).

See Also

The chapter on "Tidying and profiling R code" in "Writing R Extensions" (see the 'doc/manual' subdirectory of the R source tree).

Examples

```
Rprof()
## some code to be profiled
Rprof(NULL)
## some code NOT to be profiled
Rprof(append=TRUE)
## some code to be profiled
Rprof(NULL)
...
## Post-process the output by
##   Rcmd Rprof Rprof.out
## at the command prompt.
```

`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,
    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, default is black.
`quiet`	logical indicating if there should be a warning about clipped values.

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 coloured the same as the ticks. You can always replot the box in black if you don't like this feature.

Author(s)

B. D. Ripley

See Also

`jitter` which you may want for ties in `x`.

Examples

```
data(faithful)
attach(faithful)
plot(density(eruptions, bw=0.15))
rug(eruptions)
rug(jitter(eruptions, amount = .01), side = 3, col = "light blue")
detach("faithful")
```

`Rwin configuration` *R for Windows Configuration*

Description

The file 'Rconsole' configures the R GUI console in this Windows port. The file 'Rdevga' configures the graphics devices `windows`, `win.graph`, `win.metafile` and `win.print`.

Details

There are system copies of these files in 'R_HOME\etc'. Users can have personal copies of the files: these are looked for in the location given by the environment variable `R_USER`. The system files are read only if a corresponding personal file is not found.

If the environment variable `R_USER` is not set, the R system sets it to `HOME` if that is set (stripping any trailing slash), otherwise to `HOMEDIR:HOMEPATH` if `HOMEDIR` is set otherwise to the working directory.

Value

Each of the files contains details in its comments of how to set the values.

At the time of writing 'Rdevga' configured the mapping of font numbers to fonts, and 'Rconsole' configured the appearance (single or multiple document interface, toolbar, statusbar on MDI), size, font and colours of the GUI console, and whether resizing the console sets `options("width")`.

The file 'Rconsole' also configures the internal pager. This shares the font and colours of the console, but can be sized separately.

Note

The `GUI preferences` item on the `Edit` menu brings up an form which can be used to edit the console settings, and to save them to a file.

Author(s)

Guido Masarotto

See Also

`windows`

`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`	A positive 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, sampling takes place from `1:x`.

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 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.

Examples

```
x <- 1:12
# a random permutation
sample(x)
# bootstrap sampling
sample(x,replace=TRUE)

# 100 Bernoulli trials
sample(c(0,1), 100, replace = TRUE)
```

`save` *Save R Objects*

Description

`save` writes a 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`.

`save.image()` is just a short-cut for "save my current environment", i.e., `save(list = ls(all=TRUE), file = ".RData")`. It is what also happens with `q("yes")`.

Usage

```
save(..., list = character(0), file = "", ascii = FALSE,
     version = NULL, envir = parent.frame(), compress = FALSE)
save.image(file = ".RData", version = NULL, ascii = FALSE,
           compress = FALSE, safe = TRUE)

sys.load.image(name, quiet)
sys.save.image(name)
```

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. This is useful for transporting data between machines of different types. 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 is 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, or if `gzfile` is not supported on this platform.
`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.
`name`	name of image file to save or load.
`quiet`	logical specifying whether a message should be printed.

Details

Almost all current R platforms (including Windows and Macintosh) use the XDR representation of binary objects in binary save-d files, and these are portable across all XDR-compliant implementations.

Default values for `save.image` options can be modified with the `save.image.defaults` option. This mechanism is experimental and subject to change.

`sys.save.image` is a system function that is called by `q()` and its GUI analogs; `sys.load.image` is called by the startup code. These functions should not be called directly and are subject to change.

See Also

`dput`, `dump`, `load`.

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.image dafaults using option:
options(save.image.defaults=list(ascii=TRUE, safe=FALSE))
save.image()
unlink(".RData")
```

---

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)
```

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.

Details

This works in Rgui and Rterm but not in embedded/DCOM versions.

Note

If you want to save the history (almost) every session, you can put a call to `savehistory()` in `.Last`.

savePlot *Save Windows Plot to a File*

Description

Saves the current plot on a `windows` device to a file.

Usage

```
savePlot(filename="Rplot",
         type=c("wmf", "png", "jpeg", "jpg", "bmp", "ps", "pdf"),
         device=dev.cur())
```

Arguments

`filename`	The filename under which to save the plot, without the extension.
`type`	The type of plot, Windows metafile, PNG, JPEG, BMP (Windows bitmap format), PostScript or PDF.
`device`	A device number of a `windows` device, by default the current device.

Details

This is equivalent to selecting the 'Save as' menu item on the 'File' menu of a `windows` device.

Using `filename` as `"clipboard"` or `""` with `type = "wmf"` will copy to the clipboard.

Value

None, but a plot file will be created.

Author(s)

Guido Masarotto, B. D. Ripley

See Also

`windows`, `dev.print`

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.

See Also

sweep which allows centering (and scaling) with arbitrary statistics.

Examples

```
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 (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 = "#")
```

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). 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. `file` can also be a complete URL.
`what`	the type of `what` gives the type of data to be read. If `what` is a list, it is assumed that the lines of the data file are records each containing `length(what)` items ("fields").
`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 (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 a newline unless it is quoted.
`quote`	the set of quoting characters as a single character string.
`dec`	decimal point character.
`skip`	the number of lines of the input file to skip before beginning to read data values.
`nlines`	the maximum number of lines of data to be read.
`na.strings`	character vector. Elements of this vector are to be interpeted as missing (`NA`) values.
`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.

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.

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 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.

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 file line if the file ends in a linefeed and not it 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) are treated as blank lines.

Note

The default for `multi.line` differs from S. To read one record per line, use `flush = TRUE` and `multi.line = FALSE`.

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.

See Also

`read.table` for more user-friendly reading of data matrices; `readLines` to read a file a line at a time. `write`.

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
str(scan("ex.data", what = list("","",""))) # flush is F -> read "7"
str(scan("ex.data", what = list("","",""), flush = TRUE))
unlink("ex.data") # tidy up
```

`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.

`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.
`screen`	A number giving the screen to be split.
`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`).
`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 WARNINGS below). Split-screen mode is exited by the command `close.screen(all = TRUE)`

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.

Warning

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.

The functions should not be used with multiple devices.

See Also

`par`, `layout`, `Devices`, `dev.*`

Examples

```
if (interactive()) {
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
}
```

---

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 dataframe, 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.aov` *Internal Functions Used by model.tables*

Description

Internal function for use by `model.tables`.

Usage

```
se.aov(object, n, type = "means")
se.aovlist(object, dn.proj, dn.strata, factors, mf, efficiency,
           n, type = "diff.means", ...)
```

Author(s)

B. D. Ripley

See Also

`model.tables`

`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, ...)
se.contrast.aov(object, contrast.obj,
  coef = contr.helmert(ncol(contrast))[, 1], data = NULL, ...)
se.contrast.aovlist(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 as returned by `contrasts`.
`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 directly with the coefficients.

In multistratum models, the contrasts can appear in more than one stratum; the contrast and standard error are computed in the lowest stratum and adjusted for efficiencies and comparisons between strata.

Value

A vector giving the standard errors for each contrast.

Author(s)

B.D. Ripley

See Also

`contrasts`, `model.tables`

Examples

```
## From Venables and Ripley (1997) p.210.
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)
options(contrasts=c("contr.treatment", "contr.poly"))
npk.aov1 <- aov(yield ~ block + N + K, npk)
se.contrast(npk.aov1, list(N=="0", N=="1"), data=npk)
```

search *Give Search Path for R Objects*

Description

Gives a list of attached *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.

See Also

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, where = NA, origin = "start", rw = "", ...)
truncate(con, ...)

isSeekable(con)
```

Arguments

con	a connection.
where	integer. 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".
...	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 file connections do. The value returned by `seek(where=NA)` appears to be unreliable on Windows systems, at least for text files.

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.

`truncate` truncates a file opened for writing at its current position. It works only for `file` connections, and is not implemented on all platforms.

Value

`seek` returns the current position (before any move), as a byte offset, if relevant, or `0` if not.

`truncate` returns `NULL`: it stops with an error if it fails (or is not implemented).

`isSeekable` returns a logical value, whether the connection is support `seek`.

See Also

connections

---

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`.
`...`	further graphical parameters (from `par`).

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` and `lty` can be used to specify a color and line texture for the line segments (`col` may be a vector).

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')
```

`select.list` *Select Items from a List*

Description

Select item(s) from a character vector.

Usage

```
select.list(list, preselect = NULL, multiple = FALSE)
```

Arguments

`list`	character. A list of items.
`preselect`	a character vector of length one, or `NULL`. If non-null and if the string appears in the list, the item is selected initially.
`multiple`	logical: can more than one item be selected?

Details

This brings up a modal dialog box with a (scrollable) list of items, which can be selected by the mouse. If `multiple` is true, further items can be selected or deselected by holding the control key down whilst selecting, and shift-clicking can be used to select ranges.

Normal termination is via the `OK` button or by hitting Enter. The selection can be aborted via the `Cancel` button or pressing Escape.

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.

Examples

```
select.list(sort(.packages(all.available = TRUE)))
```

`seq` *Sequence Generation*

Description

Generate regular sequences.

Usage

```
from:to
seq(from, to)
seq(from, to, by=)
seq(from, to, length=)
seq(along)
```

Arguments

`from`	starting value of sequence.
`to`	(maximal) end value of the sequence.
`by`	increment of the sequence.
`length`	desired length of the sequence.
`along`	take the length from the length of this argument.

Details

The operator `:` and the first `seq(.)` form generate the sequence `from, from+1`, ..., `to`. `seq` is a generic function.

The second form generates `from, from+by`, ..., `to`.

The third generates a sequence of `length` equally spaced values from `from` to `to`.

The last generates the sequence `1, 2`, ..., `length(along)`.

If `from` and `to` are factors of the same length, then `from : to` returns the "cross" of the two.

Value

The result is of `mode "integer"` if `from` is (numerically equal to an) integer and `by` is not specified.

See Also

`rep`, `sequence`, `row`, `col`.

Examples

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

seq(0,1, length=11)
str(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)
stopifnot(
 3 == seq(3,3,  by=pi),
 3 == seq(3,3.1,by=pi),
 seq(1,6,by=3) == c(1,4),
 seq(10,4.05,by=-3) == c(10,7)
)

for (x in list(NULL, letters[1:6], list(1,pi)))
  cat("x=",deparse(x),";  seq(along = x):",seq(along = x),"\n")

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

---

seq.POSIXt *Generate Regular Sequences of Dates*

Description

The method for `seq` for data-time classes.

Usage

```
seq(from, to, by, length.out=NULL, along.with=NULL, ...)
```

Arguments

`from`	starting date. Required
`to`	end date. Optional. If supplied must be after `from`.
`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"`, `"week"`, `"month"` or `"year"`. This can optionally be preceded by an integer and a space.

Value

A vector of class "POSIXct".

See Also

DateTimeClasses

Examples

```
## first days of years in the (commonly used) 20th century
seq(ISOdate(1910,1,1), ISOdate(1999,1,1), "years")
## by month
seq(ISOdate(2000,1,1), by="month", length=12)
## quarters
seq(ISOdate(1990,1,1), ISOdate(2000,1,1), by="3 months")
```

---

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`.

Author(s)

B. D. Ripley

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
```

`shell` *Invoke a System Command, using a Shell*

Description

`shell` runs the command specified by `cmd`, usually under a shell.

Usage

```
shell(cmd, shell, flag="/c", intern=FALSE, wait=TRUE,
      translate=FALSE, mustWork=FALSE, ...)
```

Arguments

`cmd`	the system command to be invoked, as a string.
`shell`	a string giving the name of the shell to be used, or `NULL` (no shell). If missing, a suitable shell is chosen: see 'details'
`flag`	the switch to run a command under the shell. If the shell is `bash` or `tcsh` the default is changed to `"-c"`.
`intern`	a logical, indicates whether to make the output of the command an R object.
`wait`	should the R interpreter wait for the command to finish? The default is to wait, and the interpreter will always wait if `intern = TRUE`.
`translate`	If `TRUE`, `"/"` in `cmd` is translated to `"\"`.
`mustWork`	a logical; if `TRUE` failure to run the command will give an R error.
`...`	additional arguments to `system`.

Details

If no `shell` is specified, the environment variables `R_SHELL`, `SHELL` and `COMSPEC` are tried in turn: `COMSPEC` should always succeed. Using `shell=NULL` invokes the command `cmd` directly, in which case an extension of `.exe` is assumed. It is possible to use batch files directly if their extension is given: Windows (rather than R) then chooses a shell.

See `system` for fuller details: `shell` is a more user-friendly wrapper for `system`.

Value

If `intern=TRUE`, a character vector giving the output of the command, one line per character string, or an error message if the command could not be run.

If `intern=FALSE`, the return value is a error code, given the invisible attribute (so needs to be printed explicitly). If the command could not be run for any reason, the value is `-1` and an R warning is generated. Otherwise if `wait=FALSE` the value is the error code returned by the command, and if `wait=TRUE` it is the zero (the conventional success value),

If `intern=FALSE` and `wait=TRUE` (the defaults) the text output from a command that is a console application will appear in the R console (`Rgui`) or the window running R (`Rterm`).

Author(s)

B. D. Ripley

See Also

`system`

`shell.exec` *Open a File using Windows File Associations*

Description

Opens the specified file using the application specified in the Windows file associations.

Usage

```
shell.exec(file)
```

Arguments

`file`	file to be opened.

Value

No value.

Author(s)

B. D. Ripley

See Also

`system`, `shell`

`SHLIB` *Build a DLL for Dynamic Loading*

Description

The given source files are first compiled. All specified object files are then linked into a shared library which can be loaded into R using `dyn.load` or `library.dynam`.

Usage

```
Rcmd SHLIB [-o dllname] files
```

Arguments

files	a list of names of (typically) source files to be compiled and included in the library. You can also include the name of object files which are automagically made from their sources.
dllname	the full name of the shared library to be built, including the extension '.dll'. If not given, the name of the DLL is taken from the first source file.

See Also

dyn.load, library.dynam

---

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"
close(getConnection(3))

showConnections()
```

---

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

See Also

abs

Examples

```
sign(pi) # == 1
sign(-2:3)# -1 -1 0 1 1 1
```

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`	numbers of observations in the sample. Must be positive integers less than 50.
`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.

Value

`dsignrank` gives the density, `psignrank` gives the distribution function, `qsignrank` gives the quantile function, and `rsignrank` generates random deviates.

Author(s)

Kurt Hornik ⟨hornik@ci.tuwien.ac.at⟩

See Also

`dwilcox` etc, for the *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,")"))
}
```

---

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"))
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.

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. These too can diverted by `type = "message"` (see below).

`sink()` or `sink(file=NULL)` ends the last diversion (of the specified type). As from R version 1.3.0 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 if 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

For `sink`.

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.

Examples

```
sink("sink-examp.txt")
i <- 1:10
outer(i, i, "*")
sink()
unlink("sink-examp.txt")

## 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"))
```

`sleep` *Students' 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

```
data(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

```
data(sleep)
## ANOVA
anova(lm(extra ~ group, data = sleep))
```

`slotOp` *Extract or Replace Slots*

Description

Operators to extract or replace tbe contents of a slot in a object with a formal class structure.

Usage

```
object@name
object@name <- value
```

Arguments

`object`	An object from a formally defined class.
`name`	The character-string name of the slot.

Details

These operators support the formal classes of package `methods`. See `slot` for further details.

See Also

`Extract`, `slot`

`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, tol = 1e-7, ...)
```

Arguments

`a`	a numeric matrix containing the coefficients of the linear system.
`b`	a numeric vector or matrix giving the right-hand side(s) of the linear system. If omitted, `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`.
`...`	further arguments passed to or from other methods

Details

As from R version 1.3.0, `a` or `b` can be complex, in which case LAPACK routine `ZESV` is used. This uses double complex arithmetic which might not beavailable on all platforms.

See Also

`backsolve`, `qr.solve`.

Examples

```
hilbert <- function(n) { i <- 1:n; 1 / outer(i - 1, i, "+") }
h8 <- hilbert(8); h8
solve(h8) # gives error: 'singular'
sh8 <- solve(h8, tol = 1e-10)
round(sh8 %*% h8, 3)

A <- hilbert(4)
A[] <- as.complex(A)
## might not be supported on all platforms
try(solve(A))
```

---

sort *Sorting a Vector*

Description

Sort a numeric or complex vector (partially) into ascending order.

Usage

```
sort(x, partial = NULL, na.last = NA)
is.unsorted(x, na.rm = FALSE)
```

Arguments

`x`	a numeric or complex vector.
`partial`	a vector of indices for partial sorting.
`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.
`na.rm`	logical. Should missing values be removed?

Details

If `partial` is non `NULL`, it is taken to contain indexes 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.

`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 `NA`s).

See Also

order, rank.

Examples

```
data(swiss)
x <- swiss$Education[1:25]
x; sort(x); sort(x, partial = c(10, 15))
median # shows you another example for 'partial'

stopifnot(!is.unsorted(sort(x)),
          !is.unsorted(LETTERS),
           is.unsorted(c(NA,1:3,2), na.rm = TRUE))
```

---

source *Redirect Input*

Description

`source` causes R to accept its input from the named file (the name must be quoted). Input is read from that file until the end of the file is reached. `parse` is used to scan the expressions in, they are then 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)
```

Arguments

`file`	a connection or a character string giving the name 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`, the R working directory is changed to the directory containing `file` for evaluating.

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 Mac). The final line can be incomplete, that is missing the final EOL marker.

See Also

`demo` which uses `source`; `eval`, `parse` and `scan`; `options("keep.source")`.

`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)
digamma(x)
trigamma(x)
tetragamma(x)
pentagamma(x)
choose(n, k)
lchoose(n, k)
```

Arguments

`a, b, x`	numeric vectors.
`n, k`	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 functions `gamma` and `lgamma` return the gamma function $\Gamma(x)$ and the natural logarithm of the absolute value of the gamma function.

The functions `digamma`, `trigamma`, `tetragamma` and `pentagamma` return the first, second, third and fourth derivatives of the logarithm of the gamma function.

$$\texttt{digamma(x)} = \psi(x) = \frac{d}{dx}\ln\Gamma(x) = \frac{\Gamma'(x)}{\Gamma(x)}$$

The functions `choose` and `lchoose` return binomial coefficients and their logarithms.

References

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.

Examples

```
choose(5, 2)
for (n in 0:10) print(choose(n, k = 0:n))

curve(gamma(x),-3,4, n=1001, ylim=c(-10,100),
      col="red", lwd=2, main="gamma(x)")
abline(h=0,v=0, lty=3, col="midnightblue")

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
  if (is.deriv) dy <- diff(y) / dx
  nm <- paste(ch, "gamma", sep = "")
  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(2, 2))
```

---

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")
spline(x, y = NULL, n = 3*length(x), method = "fmm",
       xmin = min(x), xmax = max(x))
```

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.

Details

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 which will perform cubic spline interpolation of the given data points. This is often more useful than `spline`.

References

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` in package `modreg` 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)

f <- splinefun(x, y)
ls(envir = environment(f))
splinecoef <- eval(expression(z), envir = environment(f))
curve(f(x), 1, 10, col = "green", lwd = 1.5)
points(splinecoef, col = "purple", cex = 2)
par(op)
```

`split` *Divide into Groups*

Description

`split` divides the data in the vector `x` into the groups defined by `f`.

Usage

```
split(x, f)
split.default(x, f)
split.data.frame(x, f)
```

Arguments

`x`	vector containing the values to be divided into groups.
`f`	a "factor" such that `factor(f)` defines the grouping, or a list of such factors in which case their interaction is used for the grouping.

Details

`f` is recycled as necessary and if the length of `x` is not a multiple of the length of `f` a warning is printed.

Value

The value returned is a list of vectors containing the values for the groups. The components of the list are named by the factor levels given be `f`. If `f` is longer than `x` some of these will be of zero length.

See Also

`cut`

Examples

```
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)

## 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)
```

`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, ...)
stack.default(x, ...)
stack.data.frame(x, select, ...)
unstack(x, ...)
unstack.default(x, form, ...)
unstack.data.frame(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

```
data(PlantGrowth)
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
```

---

stackloss *Brownlee's Stack Loss Plant Data*

Description

Operational data of a plant for the oxidation of ammonia to nitric acid.

Usage

```
data(stackloss)
```

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

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

```
data(stackloss)
summary(lm.stack <- lm(stack.loss ~ stack.x))
```

---

standardGeneric *Formal Method System Placeholders*

Description

These routines are primitives used with the `methods` package. They should not be used without it and do not need to be called directly in any case.

`standardGeneric`: dispatch the method defined for generic function named `f`, using the actual arguments in the frame from which `standardGeneric` is called.

`objWithClass`: return the result of setting the class of `object` to `value`. Defined as a separate primitive function because R types cannot generally be changed in place.

`dataClass`: returns a single string for the class of `object` even in the case that the object has an old-style class attribute with multiple strings.

`topicName`: the string used internally to find documenation of the given type and topic. Called by the `?` operator and the special prompt functions in the methods package.

Usage

```
standardGeneric(f)
objWithClass(object, value)
dataClass(object)
topicName(type, topic)
```

Author(s)

John Chambers

---

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)),
      ...)
```

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 chosing 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.

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

Examples

```
data(mtcars)
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)

data(USJudgeRatings)
## scale linearly (not affinely) to [0, 1]
USJudge <- apply(USJudgeRatings, 2, function(x) x/max(x))
Jnam <- case.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!")
```

`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`.

`Startup` *Initialization at Start of an R Session*

Description

In R, the startup mechanism is as follows.

Unless `--no-environ` was given, 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, '$R_HOME/etc/Renviron.site' is used. 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'. This code is loaded into package `base`.

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`).

Finally, if a function `.First` exists, it is executed as `.First()`.

The functions `.First` and `.Last` can be defined in the appropriate startup profiles or reside in '.RData'.

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-init-file`, `--no-restore` and `--no-environ`.

Usage

```
.First <- function() { ..... }

.Rprofile <startup file>
```

Details

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` is of the form `${foo-bar}`, the value is that of the environmental variable `foo` if that exists and is set to a non-empty value, otherwise `bar`. This construction can be nested, so `bar` can be of the same form (as in `${foo-${bar-blah}}`).

Leading and trailing white space in `value` are stripped. `value` is processed in a similar to a Unix shell. In particular quotes are stripped, and backslashes are removed except inside quotes.

Note

Prior to R version 1.4.0, the environment files searched were '.Renviron' in the current directory, the file pointed to by `R_ENVIRON` if set, and '.Renviron' in the user's home directory.

Prior to R version 1.2.1, '.Rprofile' was sourced after '.RData' was loaded, although the documented order was as here.

The format for site and user environment files was changed in version 1.2.0. Older files are quite likely to work but may generate warnings on startup if they contained unnecessary `export` statements.

Values in environment files were not processed prior to version 1.4.0.

See Also

`.Last` for final actions before termination.

Examples

```
## Some examples with a Unix flavour
# ~/.Renviron
R_LIBS=~/R/library
PAGER=/usr/local/bin/less

# .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")

## if .Renviron contains
FOOBAR="coo\bar"doh\ex"abc\"def'"

## then we get
> cat(Sys.getenv("FOOBAR"), "\n")
coo\bardoh\exabc"def'
```

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 weighted residual sum of squares.
`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.

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)
```

state *States of the U.S.A.*

Description

Data sets related to the 50 states of the United States of America.

Usage

```
data(state)
```

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*.

`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.

Examples

```
data(islands)
stem(islands)
stem(log10(islands))
```

`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.
`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 (or the initial model is the upper model) 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`.

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 function `extractAIC.glm` 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 compute the correct AIC.

This is a minimal implementation. Use `stepAIC` for a wider range of object classes.

Author(s)

B. D. Ripley

See Also

`stepAIC`, `add1`, `drop1`

Examples

```
example(lm)
step(lm.D9)

data(swiss)
summary(lm1 <- lm(Fertility ~ ., data = swiss))
slm1 <- step(lm1)
summary(slm1)
slm1$anova
```

---

stop *Stop Function Execution*

Description

`stop` stops execution of the current expression, prints the message given as its argument, then executes an error action.

`geterrmessage` gives the last error message.

Usage

```
stop(message = NULL, call. = TRUE)
geterrmessage()
```

Arguments

`message`	a character vector (of length 1) or `NULL`.
`call.`	logical, indicating if the call should become part of the error message.

Details

The error action is controlled by the current error handler set by `options(error=)`. The default behaviour (the `NULL` error-handler) in interactive use is to return to the top level prompt, and in non-interactive use to (effectively) call `q("no", status=1, runLast=FALSE)`.

Value

`geterrmessage` gives the last error message, as character string ending in "
n".

See Also

`warning`, `restart` to catch errors and retry, and `options` for setting error handlers. `stopifnot` for validity testing.

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,long,calling,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* element of `...` which was not true.

Usage

```
stopifnot(...)
```

Arguments

`...`	any number of (`logical`) R expressions which should evaluate to `TRUE`.

Details

`stopifnot(A, B)` is conceptually equivalent to `{ if(!all(A)) stop(...) ; if(!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

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(error = NULL)# revert to default error handler
```

`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.

`ls.str` and `lsf.str` are useful "versions" of `ls`, calling `str` on each object. They are not foolproof and should rather not be used for programming, but are provided for their usefulness.

Usage

```
str(object, ...)
str.data.frame(object, ...)
str.default(object, max.level = 0, vec.len = 4, digits.d = 3,
    nchar.max = 128, give.attr = TRUE, give.length = TRUE,
    wid = getOption("width"), nest.lev = 0,
    indent.str = paste(rep(" ", max(0, nest.lev + 1)), collapse = ".."),
    ...)

 ls.str(pos = 1, pattern, ..., mode = "any",
        max.level = 1, give.attr = FALSE)
lsf.str(pos = 1, pattern, ...)
```

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 0: 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. Default 4.
`digits.d`	number of digits for numerical components (as for `print`).
`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:...]`).
`wid`	the page width to be used. The default is the currently active `options("width")`.
`nest.lev`	current nesting level in the recursive calls to `str`.
`indent.str`	the indentation string to use.

`...`	potential further arguments (required for Method/Generic reasons).
`pos`	integer indicating `search` path position.
`pattern`	regular expression passed to `ls`. Only names matching `pattern` are considered.
`mode`	character specifying the `mode` of objects to consider. Passed to `exists` and `get`.

Value

`str` does not return anything, for efficiency reasons. The obvious side effect is output to the terminal.

`ls.str` and `lsf.str` invisibly return a character vector of the matching names, similarly to `ls`.

Author(s)

Martin Maechler ⟨maechler@stat.math.ethz.ch⟩ since 1990.

See Also

`summary`, `args`.

Examples

```
## The following examples show some of 'str' capabilities
str(1:12)
str(ls)
str(args)#- more useful than  args(args) !
data(freeny); str(freeny)
str(str)
str(.Machine, digits = 20)
str( lsfit(1:9,1:9))
str( lsfit(1:9,1:9),  max =1)
op <- options(); str(op)#- save first; otherwise internal options() is used.
need.dev <- !exists(".Device") || is.null(.Device)
if(need.dev) postscript()
str(par()); if(need.dev) graphics.off()

nchar(longch <- paste(rep(letters,100), collapse=""))
str(longch)
str(longch, nchar.max = 52)

lsf.str()#- how do the functions look like which I am using?
ls.str(mode = "list")#- what are the structured objects I have defined?
## which base functions have "file" in their name ?
lsf.str(pos = length(search()), pattern = "file")
```

`stripchart` *1-D Scatter Plots*

Description

`stripchart` produces one dimensional scatter plots (or dot plots) of the given data. These plots are are 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,
           xlim=NULL, ylim=NULL, main="", ylab="", xlab="",
           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 vector, or as list of vectors, each corresponding to a component plot. Alternatively a symbolic specification of the form `x ~ g` can be given, indicating the 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 jittering 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.
`xlim, ylim, main, ylab, xlab, pch, col, cex`	Graphical parameters.

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 <- round(rnorm(50), 1)
stripchart(x)
```

`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

```
format.POSIXct(x, format = "", tz = "", usetz = FALSE, ...)
format.POSIXlt(x, format = "", usetz = FALSE, ...)

as.character(x, ...)

strftime(x, format="%Y-%m-%d %X", usetz = FALSE, ...)
strptime(x, format)

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 vector. 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.
`...`	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 time, 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.

Details

`strftime` is an alias for `format.POSIXlt`, and `format.POSIXct` first converts to class `"POSIXct"` by calling `as.POSIXct`. 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 POSIX standard for `strftime` and are likely to be widely available. Any character in the format string other than the `%` escapes is interpreted literally (and `%%` gives `%`).

`%a` Abbreviated weekday name.

`%A` Full weekday name.

`%b` Abbreviated month name.

`%B` Full month name.

`%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.

`%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 are prefixed by 20 and 70–99 by 19.

`%Y` Year with century.

`%Z` (output only.) Time zone as a character string (empty if not available). Note: do not use this on Linux unless the `TZ` environment variable is set.

Where leading zeros are shown they will be used on output but are optional on input.

`ISOdatetime` and `ISOdate` are convenience wrappers for `strptime`, that differ only in their defaults.

Value

The `format` methods and `strftime` return character vectors representing the time.

`strptime` turns character representations into an object of class `"POSIXlt"`.

`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 `"2001-02-03"` 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 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.

References

International Organization for Standardization (1988, 1997, ...) *ISO 8601. Data elements and interchange formats – Information interchange – Representation of dates and times.* The 1997 version is available on-line at ftp://ftp.qsl.net/pub/g1smd/8601v03.pdf

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.

Examples

```
## locale-specific version of date()
format(Sys.time(), "%a %b %d %X %Y")
## we would include the timezone as in
## format(Sys.time(), "%a %b %d %X %Y %Z")
## but this crashes some Linux systems

## 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)
z <- strptime(x, "%m/%d/%y %H:%M:%S")
z
```

---

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)
```

Arguments

`x`	character vector, to be split.
`split`	character vector containing a regular expression 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`	if `TRUE`, extended regular expression matching is used, and if `FALSE` basic regular expressions are used.

Value

A list of length `length(x)` the `i`-th element of which contains the vector of splits of `x[i]`.

See Also

`paste` for the reverse, `grep` and `sub` for string search and manipulation; further `nchar`, `substr`.

Examples

```
noquote(strsplit("A text I want to display with spaces", NULL)[[1]])

x <- c("asfef", "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"
```

`structure` *Attribute Specification*

Description

`structure` returns the given object with its 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.

Examples

```
structure(1:6, dim = 2:3)
```

`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.
`units`	character indicating in which units `s` is measured; must be one of `"user"`, `"inches"`, `"figure"` .
`cex`	character expansion to which is applies. Per default, the current `par("cex")` is used.

Value

integer vector with the same length as `s`, giving the width for each `s[i]`.

See Also

`text`, `nchar`

Examples

```
str.ex <- c("W","w","I",".","WwI.")
op <- par(pty='s'); plot(1:100,1:100); par('usr')
sw <- strwidth(str.ex); sw
sum(sw[1:4] == sw[5])#- since the last string contains the others
sw / strwidth(str.ex, cex = .5)
# between 1.5 and 4.2 (!),  font dependent

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")
table(shL)# all have same heights ...
mean(shL) / par("cin")[2] # should be 1 (exactly?)

swL <- strwidth(all.lett)
swL <- 3 * swL / min(swL)
```

```
all(swL == round(swL))#- TRUE !
swL <- as.integer(swL)
n.classes <- length(tL <- table(swL)); tL
iL <- order(swL)
structure(swL[iL], names = all.lett[iL])
lett.classes <- structure(vector("list", n.classes), names= names(tL))
for(i in 1:n.classes)
  lett.classes[[i]] <- all.lett[swL == as.numeric(names(tL)[i])]
lett.classes

sumex <- expression(sum(x[i], i=1,n), e^{i * pi} == -1)
strwidth(sumex)
strheight(sumex)

rm(sumex); par(op)#- reset to previous setting
```

`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
`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, it possible sentence ends at line breaks are not considerd 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(), "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> "))
```

---

subset *Subsetting Vectors and Data Frames*

Description

Return subsets of vectors or data frames which meet conditions.

Usage

```
subset(x, ...)
subset.default(x, subset, ...)
subset.data.frame(x, subset, select, ...)
```

Arguments

`x`	object to be subsetted
`...`	how to subset, depends on object
`subset`	logical expression
`select`	expression, indicating variables to select from a data frame

Details

For ordinary vectors, the result is simply `x[subset & !is.na(subset)]`.

For dataframes, the `subset` argument works similarly on the rows. Note that `subset` will be evaluated in the dataframe.

The `select` argument exists only for dataframes. It works by first replacing variable names in the selection expression with the corresponding column numbers in the dataframe and then using the resulting integer vector to index the columns. This allows the use of the standard indexing conventions so that for examples ranges of variables can be specified easily.

Value

Selected rows and columns of the object `x`.

Author(s)

Peter Dalgaard

See Also

[, transform

Examples

```
data(airquality)
subset(airquality, Temp > 80, select = c(Ozone, Temp))
subset(airquality, Day == 1, select = -Temp)
subset(airquality, select = Ozone:Wind)

attach(airquality)
subset(Ozone, Temp > 80)
```

---

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=<<see below>>)
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 `delay()`, 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.

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.

See Also

`missing` for argument "missingness".

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.

When extracting, if `start` is larger than the string length then `""` is returned.

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 as `x`.

For `substring`, a character vector of length the longest of the arguments.

Note

The S4 version of `substring<-` ignores `last`; this version does not.

See Also

`strsplit`, `paste`, `nchar`.

Examples

```
substr("abcdef",2,4)
print(ss <- substring("abcdef",1:6,1:6))
stopifnot(ss == strsplit ("abcdef",NULL)[[1]])
## strsplit is more efficient ...

substr(rep("abcdef",4),1:4,4:5)
x <- c("asfef", "qwerty", "yuiop[", "b", "stuff.blah.yech")
stopifnot(substr(x, 2, 5) == substring(x, 2, 5))
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. 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.

Usage

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

Arguments

`...`	numeric vectors.
`na.rm`	logical. Should missing values be removed?

`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, ...)

summary.default   (object, ..., digits = max(3, getOption("digits")-3))
summary.data.frame(object, maxsum = 7,
                   digits = max(3, getOption("digits")-3), ...)
summary.factor    (object, maxsum = 100, ...)
summary.matrix    (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 factors, 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.

See Also

`anova`, `summary.glm`, `summary.lm`.

Examples

```
data(attenu)
summary(attenu, digits = 4) #-> summary.data.frame(...), default precision
summary(attenu $ station, maxsum = 20) #-> summary.factor(...)
```

---

summary.manova *Summary Method for Multivariate Analysis of Variance*

Description

A `summary` method for class `"manova"`.

Usage

```
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).

Value

A list with components

`SS`	A names list of sums of squares and product matrices.
`Eigenvalues`	A matrix of eigenvalues,
`stats`	A matrix of the statistics, approximate F value and degrees of freedom.

Author(s)

B.D. Ripley

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`

Examples

```
## Example on producing plastic filem 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
```

`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,
              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 computes.
`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.
`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.

See Also

`density`

Examples

```
data(iris)
## '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")
```

---

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

```
data(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 (package ts) has a longer (and a bit different) series.

Examples

```
data(sunspots)
plot(sunspots, main = "sunspots data", xlab = "Year",
     ylab = "Monthly sunspot numbers")
```

---

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))
La.svd(x, nu = min(n, p), nv = min(n, p), method = c("dgesdd", "dgesvd"))
```

Arguments

x	a matrix whose SVD decomposition is to be computed.
nu	the number of left singular vectors to be computed. This must be one of 0, nrow(x) and ncol(x), except for method = "dgesdd".
nv	the number of right singular vectors to be computed. This must be one of 0 and ncol(x).
method	The LAPACK routine to use in the real case.

Details

The singular value decomposition plays an important role in many statistical techniques.

svd provides an interface to the LINPACK routine DSVDC. La.svd provides an interface to the LAPACK routines DGESVD and DGESDD. The latter is usually substantially faster if singular vectors are required: see http://www.cs.berkeley.edu/~demmel/DOE2000/Report0100.html. Most benefit is seen with an optimized BLAS system.

La.svd is preferred to svd for new projects, but it is not an exact replacement as it returns the transpose of the right singular vector matrix, and the signs of the singular vectors may differ from those given by svd. (They may also differ between methods and between platforms.)

Both functions handle complex matrices via LAPACK routine ZGESVD.

Computing the singular vectors is the slow part for large matrices.

Using method="dgesdd" requires IEEE 754 arithmetic. Should this not be supported on your platform, method="dgesvd" is used, with a warning.

Value

The SVD decomposition of the matrix as computed by 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 components in the returned value correspond directly to the values returned by DSVDC.

`d`	a vector containing the singular values of `x`.
`u`	a matrix whose columns contain the left singular vectors of `x`.
`v`	a matrix whose columns contain the right singular vectors of `x`.

For `La.svd` the return value replaces `v` by `vt`, the (conjugated if complex) transpose of `v`.

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

`eigen`, `qr`.

`capabilities` to test for IEEE 754 arithmetic.

Examples

```
hilbert <- function(n) { i <- 1:n; 1 / outer(i - 1, i, "+") }
str(X <- hilbert(9)[,1:6])
str(s <- svd(X))
Eps <- 100 * .Machine$double.eps

D <- diag(s$d)
stopifnot(abs(X - s$u %*% D %*% t(s$v)) < Eps)#  X = U D V'
stopifnot(abs(D - t(s$u) %*% X %*% s$v) < Eps)#  D = U' X V

X <- cbind(1, 1:7)
str(s <- svd(X)); D <- diag(s$d)
stopifnot(abs(X - s$u %*% D %*% t(s$v)) < Eps)#  X = U D V'
stopifnot(abs(D - t(s$u) %*% X %*% s$v) < Eps)#  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 be quoted.
`...`	optional arguments to `FUN`.

Value

An array with the same shape as `x`, but with the summary statistics swept out.

See Also

`apply` on which `sweep` is based; `scale` for centering and scaling.

Examples

```
data(attitude)
med.att <- apply(attitude, 2, median)
sweep(data.matrix(attitude), 2, med.att)# subtract the column medians
```

`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

```
data(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 are available at http://opr.princeton.edu/archive/eufert/switz.html.

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."

Examples

```
data(swiss)
pairs(swiss, panel = panel.smooth, main = "swiss data",
      col = 3 + (swiss$Catholic > 50))
summary(lm(Fertility ~ . , data = swiss))
```

`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`.

Examples

```
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(ch, a=1,    b=2:3),            "\n")
for(ch in ccc) cat(ch,":",switch(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))
```

---

symbols *Draw symbols 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, circles, squares, rectangles, stars,
        thermometers, boxplots, inches=TRUE, add=FALSE,
        fg=1, bg=NA, xlab = NULL, ylab = NULL, main = NULL,
        xlim = NULL, ylim = NULL, ...)
```

Arguments

`x`	a vector giving the x coordinates of the symbols.
`y`	a vector giving the y coordinates of the symbols.
`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 rectangle symbols.
`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 from their base to this proportion of their height. If there are four columns, the third and fourth columns are taken as proportions. The thermometers are filled between these two proportions of their heights.
`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`	If `inches` is `FALSE`, the units are taken to be those of the x axis. If `inches` is `TRUE`, the symbols are scaled so that the largest symbol is one inch in height. If a number is given the symbols are scaled to make largest symbol this height in inches.
`add`	if `add` is `TRUE`, the symbols are added to an existing plot, otherwise a new plot is created.
`fg`	colors the symbols are to be drawn in (the default is the value of the `col` graphics parameter).
`bg`	if specified, the symbols are filled with this color. The default is to leave the symbols unfilled.
`xlab`	the x label of the plot if `add` is not true; this applies to the following arguments as well. Defaults to the `deparse`d expression used for `x`.
`ylab`	the y label of the plot.
`main`	a main title for the plot.
`xlim`	numeric of length 2 giving the x limits for the plot.
`ylim`	numeric of length 2 giving the y limits for the plot.
`...`	graphics parameters can also be passed to this function.

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 rays which are `NA` is reset to zero.

References

W. S. Cleveland (1985) *The Elements of Graphing Data.* Monterey, California: Wadsworth.

See Also

`stars` for drawing *stars* with a bit more flexibility; `sunflowerplot`.

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)

data(trees)
## Note that  example(trees)  shows more sensible plots!
N <- nrow(trees)
attach(trees)
## Girth is diameter in inches
symbols(Height, Volume, circles=Girth/24, inches=FALSE,
        main="Trees' Girth")# xlab and ylab automatically
## Colors 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"); detach()
```

---

symnum *Symbolic Number Coding*

Description

Symbolically encode a given numeric or logical vector or array.

Usage

```
symnum(x, cutpoints=c(0.3, 0.6, 0.8, 0.9, 0.95),
       symbols=c(" ", ".", ",", "+", "*", "B"),
       legend = length(symbols) >= 3,
       na="?", eps=1e-5,
       corr = missing(cutpoints), show.max = if(corr) "1", show.min = NULL,
       lower.triangular = corr & 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 $(c_j, c_{j+1}]$. For `logical` argument `x`, 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.
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 minmal cutpoint is coded especially.
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 (TRUE by default when there more than 2 classes), it 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].

Author(s)

Martin Maechler ⟨maechler@stat.math.ethz.ch⟩

See Also

as.character

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)# use for logical

##-- Symbolic correlation matrices:
data(attitude)
symnum(cor(attitude), diag = FALSE)

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))
```

`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.

Note

`getenv` is an alias for backwards compatibility.

See Also

`Sys.putenv`, `getwd` for the working directory.

Examples

```
Sys.getenv(c("R_HOME", "R_PAPERSIZE", "R_PRINTCMD", "HOST"))
```

`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.

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 information is obtained from Windows system calls. It is likely to be most complete on Windows NT and 2000 systems.

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.)

Author(s)

B. D. Ripley

See Also

`.Platform`

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(n = 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 generations to go back if negative. (See the Details section.)
`n`	the number of frame generations to go back.

Details

`.GlobalEnv` is given number 0 in the list of frames. Each subsequent function evaluation increases the frame stack by 1 and the environment for evaluation of that function is returned by `sys.frame` with the appropriate index.

The parent 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.frame` returns the environment associated with a given frame number.

`sys.call` and `sys.frame` both accept integer values for the argument `which`. Non-negative values of `which` are normal frame numbers whereas negative values are counted back from the frame number of the current evaluation.

`sys.nframe` returns the number of the current frame in that list. `sys.function` gives the definition of the function curently being evaluated in the frame `n` generations back.

`sys.frames` gives a list of all the active frames and `sys.parents` gives the indices of the parent frames of each of the 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`.

`sys.on.exit()` retrieves 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).

See Also

`eval` for the usage of `sys.frame` and `parent.frame`.

Examples

```
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 nos", 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
```

`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`.

See Also

`Sys.getenv`, `setwd` for the working directory.

Examples

```
print(Sys.putenv("R_TEST"="testit", ABC=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 lauched 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.

Value

Invisible `NULL`.

Note

This function is not implemented on all systems.

Author(s)

B. D. Ripley

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 = NULL, 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.

See Also

`source`, and `library` which uses `sys.source`.

`Sys.time` *Get Current Time and Timezone*

Description

`Sys.time` returns the system's idea of the current time and `Sys.timezone` returns the current time zone.

Usage

```
Sys.time()
Sys.timezone()
```

Value

`Sys.time` returns an object of class `"POSIXct"` (see DateTimeClasses).

`Sys.timezone` returns an OS-specific character string, possibly an empty string.

See Also

`date` for the system time in a fixed-format character string.

Examples

```
Sys.time()
## locale-specific version of date()
format(Sys.time(), "%a %b %d %X %Y")

Sys.timezone()
```

`system` *Invoke a System Command*

Description

`system` invokes the system command specified by `command`.

Usage

```
system(command, intern = FALSE, wait = TRUE, input = "",
       show.output.on.console = FALSE,
       minimized = FALSE, invisible = FALSE)
```

Arguments

`command`	the system command to be invoked, as a string.
`intern`	a logical, indicates whether to make the output of the command an R object.
`wait`	should the R interpreter wait for the command to finish? The default is to wait, and the interpreter will always wait if `intern = TRUE`.
`input`	if a character vector is supplied, this is copied one string per line to a temporary file, and the standard input of `command` is redirected to the file.
`show.output.on.console`	a logical, indicates whether to capture the output of the command and show it on the R console (not used by `Rterm`, which captures the output unless `wait` is false).
`minimized`	a logical, indicates whether the command window should be initially displayed as a minimized window.
`invisible`	a logical, indicates whether the command window should be visible on the screen.

Details

The command is run directly as a Windows command by the Windows API call `CreateProcess`: extensions of `.exe`, `.com`, `.cmd` and `.bat` are tried in turn if none is supplied. (To use DOS internal commands use `command.com /c cmd` under Windows 9X/ME.) The search path for `command` may be system-dependent: it will include the R `bin` directory, the working directory and the Windows system directories before `PATH`.

Precisely what is seen by the user depends on whether `Rgui` or `Rterm` is being used. For `Rgui` a new console will always be used, so a commands window will appear for the duration of console applications unless `invisible` is true. For `Rterm` a separate commands window will appear for console applications only if `wait=FALSE`.

`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 a R error is generated.

If `intern=FALSE`, the return value is a error code, given the invisible attribute (so needs to be printed explicitly). If the command could not be run for any reason, the value is `-1` and an R warning is generated. Otherwise if `wait=FALSE` the value is the error code returned by the command, and if `wait=TRUE` it is the zero (the conventional success value),

If `intern=FALSE` and `show.output.on.console=TRUE` the text output from a command that is a console application will appear in the R console (`Rgui`) or the window running R (`Rterm`).

WARNING

The command cannot be interrupted by the R process.

Do not run console applications that require user input from `Rgui` setting `intern=TRUE` and/or `show.output.on.console=TRUE`. They will not work, may hang and then will probably hang `Rgui` too.

Author(s)

Guido Masarotto and Brian Ripley

See Also

`shell` for a less raw interface.

Examples

```
# launch an editor, wait for it to quit
system("notepad myfile.txt")
# launch a Windows 9x process monitor (from Win9x KernelToys)
system("wintop", wait=FALSE)
# launch your favourite (!) shell:
system("command.com")
```

`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, pkg, lib)
```

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.
`pkg`	previous name for argument `package`. Deprecated.
`lib`	previous name for argument `lib.loc`. Deprecated.

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 = "lqs") # The root of package 'lqs'
system.file("INDEX")
system.file("help", "AnIndex", package = "stepfun")
```

`system.time` *CPU Time Used*

Description

Return CPU (and other) times that `expr` used.

Usage

```
system.time(expr)
unix.time(expr)
```

Arguments

`expr`	Valid R expression to be "timed"

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.

The values returned by the `proc.time` are (on Unix) those returned by the C library function `times`(3v).

`unix.time` is an alias of `system.time`, for compatibility reasons.

Value

A numeric vector of length 5 containing the user cpu, system cpu, elapsed, subproc1, subproc2 times. The subproc times are the the user and system cpu time used by child processes (and so are usually zero). On Windows the subproc times are not available and so are always `NA`. The first two components are not available on Windows 9x, and so are reported as `NA`; they do return real values on Windows NT4 and 2000.

The resolution of the times will be system-specific; it is common for them to be recorded to of the order of 1/100 second, and elapsed time is rounded to the nearest 1/100.

See Also

`proc.time`, `time` which is for time series.

Examples

```
system.time(for(i in 1:50) mad(runif(500)))
exT <- function(n = 100) {
  # 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):
exT()                 #- '1.4' on -O-optimized Ultra1
system.time(exT())    #~ +/- same
```

---

t *Matrix Transpose*

Description

Given a matrix or `data.frame` `x`, `t` returns the transpose (matrix or data.frame) of `x`.

Usage

```
t(x)
```

Arguments

`x`	a matrix or data frame.

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(! a[, j] == ta[j, ]) stop("wrong transpose")
```

---

table *Cross Tabulation*

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, deparse.level = 1)
as.table(x, ...)
is.table(x)
as.data.frame.table(x, row.names = NULL, optional = FALSE, ...)
```

Arguments

`...`	objects which can be interpreted as factors (including character strings), or a list (or data frame) whose components can be so interpreted
`exclude`	values to use in the exclude argument of `factor` when interpreting non-factor objects
`dnn`	the names to be given to the dimensions in the result ('the dimname names').
`deparse.level`	controls how the default `dnn` is constructed. See details.
`x`	an arbitrary R object.
`row.names`	a character vector giving the row names for the data frame.
`optional`	a logical controlling whether row names are set. Currently not used.

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.

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` in package **ctest** currently only handles 2-d tables).

`as.table` and `is.table` coerce to and test for contingency table, respectively.

`as.data.frame.table` is a method for the generic function `as.data.frame` to convert the array-based representation of a contingency table to a data frame containing the classifying factors and the corresponding counts (the latter as component `Freq`). This is the inverse of `xtabs`.

Examples

```
## Simple frequency distribution
table(rpois(100,5))
data(warpbreaks)
attach(warpbreaks)
## Check the design:
table(wool, tension)
data(state)
table(state.division, state.region)

data(airquality)
attach(airquality)
# simple two-way contingency table
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() :
data(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))
```

---

tabulate *Tabulation for Vectors*

Description

`tabulate` takes the integer valued vector `bin` and counts the number of times each integer occurs in it. `tabulate` is used as the basis of the `table` function.

Usage

```
tabulate(bin, nbins = max(1, bin))
```

Arguments

`bin`	a vector of integers, or a factor.
`nbins`	the number of bins to be used.

Details

If `bin` is a factor, its internal integer representation is tabulated. If the elements of `bin` are not integers, they are rounded to the nearest integer. Elements outside the range `1,...,nbin` are (silently) ignored in the tabulation.

See Also

`factor`, `table`.

Examples

```
tabulate(c(2,3,5))
tabulate(c(2,3,3,5), nb = 10)
tabulate(c(-2,0,2,3,3,5), nb = 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.

See Also

the convenience function `aggregate` (using `tapply`); `apply`, `lapply` with its version `sapply`.

Examples

```
groups <- as.factor(rbinom(32, n = 5, p = .4))
tapply(groups, groups, length) #- is almost the same as
table(groups)

data(warpbreaks)
## 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)

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*

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.

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)


# 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)
```

`taskCallbackManager` *Create an R-level task callback manager*

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 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-leve 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*

Usage

```
getTaskCallbackNames()
```

Arguments

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

This is an experimental feature and the interface may be changed in the future.

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")
```

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, log = FALSE)
pt(q, df, ncp=0, lower.tail = TRUE, log.p = FALSE)
qt(p, df,        lower.tail = TRUE, log.p = FALSE)
rt(n, df)
```

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).
`ncp`	non-centrality parameter δ; currently `ncp <= 37.62`.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, 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 a 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 `pchisq`.

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.

References

Lenth, R. V. (1989). *Algorithm AS 243* — Cumulative distribution function of the non-central t distribution, *Appl. Statist.* **38**, 185–189.

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)")
```

`tempfile` *Create Names for Temporary Files*

Description

Returns a vector of character strings which can be used as names for temporary files.

Usage

```
tempfile(pattern = "file")
```

Arguments

`pattern`	a non-empty character vector giving the initial part of the 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 filenames will be in the directory given by the first found of the environment variables `TMP`, `TEMP` and `R_USER` (see `Rconsole`). If the path to the directory contains a space in any of the components, the path returned will use the shortnames version of the path.

Value

A character vector giving the names of possible (temporary) files.

Note that no files are generated by `tempfile`.

See Also

`unlink` for deleting files.

`termplot` *Plot regression terms*

Description

Plots regression terms against their predictors, optionally with standard errors and partial residuals added.

Usage

```
termplot(model, data=model.frame(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"),
         ask = interactive() && nb.fig < n.tms && .Device != "postscript",
         ...)
```

Arguments

`model`	fitted model object
`data`	data frame in which the 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=.)`.

`...` 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.

It is often necessary to specify the `data` argument, because it is not possible to reconstruct eg `x` from a model frame containing `sin(x)`. The `data` argument must have exactly the same rows as the model frame of the model object so, for example, missing data must have been removed in the same way.

See Also

For (generalized) linear models, `plot.lm` and `predict.glm`.

Examples

```
rs <- require(splines)
x <- 1:100
z <- factor(rep(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, rug= TRUE,
         main="termplot(..., partial = TRUE, rug = TRUE)")
termplot(model, partial=TRUE, se = TRUE, main = TRUE)
if(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).

Value

An object of class `c("terms", "formula")` which contains the *terms* representation of a symbolic model. See `terms.object` for its structure.

See Also

`terms.object`, `terms.formula`, `lm`, `glm`, `formula`.

`terms.formula` *A function to 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

```
terms.formula(x, specials=NULL, abb=NULL, data=NULL, neg.out=TRUE,
              keep.order=FALSE, ...)
```

Arguments

`x`	A formula.
`specials`	What functions in the formula should be marked as special in the terms object.
`abb`	Unused 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`	`TRUE` if terms with a minus, `-` should be removed. If FALSE these are kept in and indicate a negative order (for fractionate?).
`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.
`...`	further arguments passed to or from other methods.

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 is returned.

See Also

`terms.object`, `terms.default`

`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.

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.

Value

An object with the following attributes:

`factors`	A matrix of variables by terms showing which variables appear in which terms.
`term.labels`	A character vector containing the labels for each of the terms in the model.
`variables`	A 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`	

`offset` If the model contains `offset` terms there is an `offset` attribute indicating which terms are offsets

`specials` If the `specials` argument was given to `terms.formula` there is a `specials` attribute, a list of vectors indicating the terms that contain these special functions.

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.

An example of the `specials` argument can be seen in the `aov` function.

See Also

`terms`, `terms.default`, `formula`.

`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, ...)
text.default (x, y = NULL, labels = seq(along = x), adj = NULL,
       pos = NULL, offset = 0.5, vfont = NULL,
       cex = 1, col = NULL, font = NULL, xpd = 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.

`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.
`col, font`	the color and font to be used; these default to the values of the global graphical parameters in `par()`.
`xpd`	(where) should clipping take place? Defaults to `par("xpd")`.
`...`	further graphical parameters (from `par`).

Details

`labels` 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.

`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).

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 l', but not ■ ...")
mtext("ńISO-accents■: ś 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)")
```

`textConnection` *Text Connections*

Description

Input and output text connections.

Usage

```
textConnection(object, open = "r")
```

Arguments

`object`	character. A description of the connection. For an input is an R character vector object, and for an output connection the name for the R character vector to receive the output.
`open`	character. Either `"r"` (or equivalently `""`) for an input connection or `"w"` or `"a"`for an output 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. 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.

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. If none is found (even if an object exists of the right name but the wrong type) a new character vector wil be created, with a warning.

You cannot `seek` on a text connection, and `seek` will always return zero as the position.

Value

A connection object of class `"textConnection"` which inherits from class `"connection"`.

See Also

`connections`, `showConnections`, `pushBack`

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

# 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")
```

`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, 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.

See Also

`ts`, `start`, `tsp`, `window`.

`date` for clock time, `system.time` for CPU usage.

Examples

```
data(presidents)
cycle(presidents)
# a simple series plot: c() makes the x and y arguments into vectors
plot(c(time(presidents)), c(presidents), type="l")
```

`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

```
data(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

```
data(Titanic)
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 ...
```

`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.axis")` and color `par("col.axis")`.
`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 from the plot.
`outer`	a logical value. If `TRUE`, the titles are placed in the outer margins of the plot.
`...`	further graphical parameters (from `par`).

Details

The labels passed to title can be simple strings or expressions, or they can be a list containing the string to be plotted, and a selection of the optional modifying graphical parameters `cex=`, `col=`, `font=`.

See Also

`mtext`, `text`; `plotmath` for details on mathematical annotation.

Examples

```
data(cars)
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))

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),
     lab = expression(-pi, -pi/2, 0, pi/2, pi))
abline(h = 0, v = pi/2 * c(-1,1), lty = 2, lwd = .1, col = "gray70")
```

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

```
data(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

```
data(ToothGrowth)
coplot(len ~ dose | supp, data = ToothGrowth, panel = panel.smooth,
       xlab = "ToothGrowth data: given is supplement type")
```

---

toString — *toString Converts its Argument to a Character String*

Description

This is a helper function for `format`. It converts its argument to a string. If the argument is a vector then its elements are concatenated with a `,` as a separtor. Most methods should honor the width argument. The minimum value for `width` is six.

Usage

```
toString(x, ...)
toString.default(x, width, ...)
```

Arguments

`x`	The object to be converted.
`width`	The returned value is at most the first `width` characters.
`...`	Optional arguments for methods.

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 *Trace All Calls to a Function.*

Description

These are both in very primitive form. If `trace` is called the function named is marked and each time it is entered the call is printed on the console. To stop tracing a function use `untrace`.

Usage

```
trace(fun)
untrace(fun)
```

Arguments

`fun`	any interpreted R function (not quoted).

See Also

debug.

Examples

```
trace(names)
data.frame(x=1:2, y=3:4)
untrace(names)
```

traceback *Print Call Stack of Last Error*

Description

`traceback()` prints the call stack of the last error, i.e., the sequence of calls that lead to the error. This is useful when an error occurs with an unidentifiable error message. This stack is stored as a list in `.Traceback`, which `traceback` prints in a user-friendly format.

Usage

```
traceback()
```

Value

`traceback()` returns nothing, but prints the deparsed call stack deepest call first. The calls may print on more that one line, and the first line is labelled by the frame number.

Examples

```
foo <- function(x) { print(1); bar(2) }
bar <- function(x) { x + a.variable.which.does.not.exist }

foo(2) # gives a strange error
traceback()
## 2: bar(2)
## 1: foo(2)
bar
## Ah, this is the culprit ...
```

`transform` *Transform an Object, e.g. a Data Frame*

Description

`transform` is a generic function, which—at least currently—only does anything useful with dataframes. `transform.default` converts its first argument to a dataframe if possible and calls `transform.data.frame`.

Usage

```
transform(x, ...)
transform.default(x, ...)
transform.data.frame(x, ...)
```

Arguments

`x`	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 dataframe `x`. The tags are matched against `names(x)`, and for those that match, the value replace the corresponding variable in `x`, and the others are appended to `x`.

Value

The modified value of `x`.

Note

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

```
data(airquality)
transform(airquality, Ozone = -Ozone)
transform(airquality, new = -Ozone, Temp = (Temp-32)/1.8)

attach(airquality)
transform(Ozone, logOzone = log(Ozone)) # marginally interesting ...
detach(airquality)
```

---

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

```
data(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

```
data(trees)
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
```

`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 vector

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 $\pi/2$).

Examples

```
cos(0) == 1
sin(3*pi/2) == cos(pi)
x <- rnorm(99)
all.equal( sin(-x), - sin(x))
all.equal( cos(-x), cos(x))
x <- abs(x); y <- abs(rnorm(x))
all(abs(atan2(y, x) - atan(y/x)) <= .Machine$double.eps)# TRUE
table(abs(atan2(y, x) - atan(y/x)) / .Machine$double.eps) # depends!

x <- 1:99/100
all(Mod(1 - (cos(x) + 1i*sin(x)) / exp(1i*x)) < 1.1 * .Machine$double.eps)
 2* abs(1 - x / acos(cos(x))) / .Machine$double.eps #-- depends ?
all(abs(1 - x / asin(sin(x))) <= .Machine$double.eps) # TRUE
all(abs(1 - x / atan(tan(x))) <= .Machine$double.eps) # TRUE
```

`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, first = TRUE)
```

Arguments

`expr`	an R expression to try
`first`	not for user use!

Details

`try` is a user-friendly wrapper to `restart`. The argument `first` is used to record if `restart` has already been used, and so ensure that `restart` is called only once.

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 if fails. The normal error handling will print the same message unless `options("show.error.messages")` is false.

See Also

`options` for setting error handlers and suppressing the printing of error messages; `geterrmessage` for retrieving the last error message.

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)

## 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")
}
options(show.error.messages = FALSE)
## alternative 1
res <- lapply(1:100, function(i) try(doit(x)))
```

```
## alternative 2
res <- vector("list", 100)
for(i in 1:100) res[[i]] <- try(doit(x))
options(show.error.messages = TRUE)
unlist(res[sapply(res, function(x) !inherits(x, "try-error"))])
```

`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 vector or matrix of the observed time-series values.
`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.

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.

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.

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 will use the tsp attribute of the object if it has one to set the start and end times and frequency.

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. Standard package **ts** for many additional time-series functions.

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)

# Ensure working arithmetic for 'ts' objects :
stopifnot(z == z)
stopifnot(z-z == 0)


## A phase plot:
data(nhtemp)
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
library(ts)
plot(nhtemp, lag(nhtemp, 1), cex = .8, col="blue",
     main = "Lag plot of New Haven temperatures")
```

---

ts-methods *Methods for Time Series Objects*

Description

Methods for objects of class "ts", typically the result of ts.

Usage

```
diff(x, lag=1, differences=1, ...)
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.

Author(s)

B. D. Ripley

See Also

`diff`; `na.omit`, `na.fail`, `na.contiguous`.

`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"` class of `x`.

See Also

`ts`, `time`, `start`.

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, 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

```
system.time(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)))
```

---

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.
`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.

See Also

`mode`, `storage.mode`.

Examples

```
typeof(2)
mode(2)
```

`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

```
data(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://hotspur.psych.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

```
data(UCBAdmissions)
## 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)
```

`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.
`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.

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
 runif(10, 2,2) == 2

var(runif(10000))#- ~ = 1/12 = .08333
```

---

unique *Extract Unique Elements*

Description

`unique` returns a vector or data frame like `x` but with duplicate elements removed. If an element is equal to one with a smaller index, it is removed.

Usage

```
unique(x, incomparables = FALSE)
```

Arguments

`x`	an atomic vector or a data frame.
`incomparables`	a vector of values that cannot be compared. Currently, `FALSE` is the only possible value, meaning that all values can be compared.

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
my.unique <- function(x) x[!duplicated(x)]
for(i in 1:4)
 { x <- rpois(100, pi); stopifnot(unique(x) == my.unique(x)) }

data(iris)
unique(iris)
stopifnot(dim(unique(iris)) == c(149, 5))
```

`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 arguments to `f`.

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.

If the algorithm does not converge in `maxiter` steps, a warning is printed and the current approximation is returned.

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`.

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,,-1000,0, tol=1e-20)
str(r, digits= 15)##> around -745.1332191

exp(r$r)        # = 0, but not for r$r * 0.999...
minexp <- r$r * (1 - .Machine$double.eps)
exp(minexp)     # typically denormalized
```

---

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.

`cm` translates inches in to cm (centimeters).

Usage

```
xinch(x = 1, warn.log = TRUE)
yinch(y = 1, warn.log = TRUE)
xyinch(xy = 1, warn.log = TRUE)
cm(x)
```

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"

cm(1)# = 2.54

## plot labels offset 0.12 inches to the right
## of plotted symbols in a plot
data(mtcars)
attach(mtcars)
plot(mpg, disp, pch=19, main= "Motor Trend Cars")
text(mpg + xinch(0.12), disp, rownames(mtcars),adj=0, cex = .7, col='blue')
detach(mtcars)
```

`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 recusively?

Details

If `recusive = FALSE` directories are not deleted, not even empty ones.

`file.remove` can only remove files, but gives more detailed error information.

Value

`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.

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`	A list or vector.
`recursive`	logical. Should unlisting be applied to list components of `x`?
`use.names`	logical. Should names be preserved?

Details

If `recursive = FALSE`, the function will not recurse beyond the first level items in `x`.

`x` can be a 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.

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

A vector of an appropriate mode to hold the list components.

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.frames. *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)))
str(table(breaks)) # The names are quite larger than the data ...
barplot(unname(table(breaks)), axes= FALSE)
```

`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, ...)
update.default(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.

Author(s)

B.D. Ripley

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

```
update.formula(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.formula(y ~ x,    ~ . + x2) #> y ~ x + x2
update.formula(y ~ x, log(.) ~ . ) #> log(y) ~ x
```

update.packages *Download Packages from CRAN*

Description

These functions can be used to automatically compare the version numbers of installed packages with the newest available version on CRAN and update outdated packages on the fly.

Usage

```
update.packages(lib.loc = NULL, CRAN = getOption("CRAN"),
                contriburl = contrib.url(CRAN),
                method, instlib = NULL,
                ask=TRUE, available=NULL, destdir=NULL)

installed.packages(lib.loc = NULL)
CRAN.packages(CRAN = getOption("CRAN"), method = "auto",
              contriburl = contrib.url(CRAN))
old.packages(lib.loc = NULL, CRAN = getOption("CRAN"),
             contriburl = contrib.url(CRAN),
             method, available = NULL)

download.packages(pkgs, destdir, available = NULL,
                  CRAN = getOption("CRAN"),
                  contriburl = contrib.url(CRAN), method = "auto")
install.packages(pkgs, lib, CRAN = getOption("CRAN"),
                 contriburl = contrib.url(CRAN),
                 method, available = NULL, destdir = NULL)
```

Arguments

`lib.loc`	A character vector describing the location of R library trees to search through (and update packages therein).
`CRAN`	The base URL of the CRAN mirror to use, i.e., the URL of a CRAN root such as `"http://cran.r-project.org"` (the default) or its Statlib mirror, `"http://lib.stat.cmu.edu/R/CRAN"`.
`contriburl`	URL of the contrib section of CRAN. Use this argument only if your CRAN mirror is incomplete, e.g., because you burned only the contrib section on a CD. Overrides argument `CRAN`.
`method`	Download method, see `download.file`.
`pkgs`	A character vector of the short names of packages whose current versions should be downloaded from `CRAN`.
`destdir`	Directory where downloaded packages are stored.
`available`	List of packages available at CRAN as returned by `CRAN.packages`.
`lib,instlib`	A character string giving the library directory where to install the packages.
`ask`	If `TRUE`, ask before packages are actually downloaded and installed.

Details

`installed.packages` scans the 'DESCRIPTION' files of each package found along `lib.loc` and returns a list of package names, library paths and version numbers. `CRAN.packages` returns a similar list, but corresponding to packages currently available in the contrib section of CRAN, the comprehensive R archive network. The current list of packages is downloaded over the internet (or copied from a local CRAN mirror). Both functions use `read.dcf` for parsing the description files. `old.packages` compares the two lists and reports installed packages that have newer versions on CRAN.

`download.packages` takes a list of package names and a destination directory, downloads the newest versions of the package sources and saves the in `destdir`. If the list of available

packages is not given as argument, it is also directly obtained from CRAN. If CRAN is local, i.e., the URL starts with `"file:"`, then the packages are not downloaded but used directly.

The main function of the bundle is `update.packages`. First a list of all packages found in `lib.loc` is created and compared with the packages available on CRAN. Outdated packages are reported and for each outdated package the user can specify if it should be automatically updated. If so, the pre-compiled packages are downloaded from CRAN and installed in the respective library path (or `instlib` if specified).

`install.packages` can be used to install new packages, it takes a vector of package names and a destination library, downloads the packages from CRAN and installs them. If the library is omitted it defaults to the first directory in `.libPaths()`, with a warning. Argument `pkgs` can also be a character vector of file names of zip files if `CRAN=NULL`. The zip files are then unpacked directly.

For `install.packages` and `update.packages`, `destdir` is the directory to which packages will be downloaded. If it is `NULL` (the default) a temporary directory is used, and the user will be given the option of deleting the temporary files once the packages are installed.

Note

`wget.exe` is available from http://www.stats.ox.ac.uk/pub/Rtools/wget.zip. `lynx.exe` is available from http://www.fdisk.com/doslynx/lynxport.htm.

See Also

`library`, `.packages`, `read.dcf`, `download.file`

`url.show` *Display a text URL*

Description

Extension of `file.show` to display text files on a remote server.

Usage

```
url.show(url, title = url, file = tempfile(),
         delete.file = TRUE, method, ...)
```

Arguments

`url`	The URL to read from.
`title`	Thitel 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

```
url.show("http://lib.stat.cmu.edu/datasets/csb/ch3a.txt")
```

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

```
data(USArrests)
```

Format

A data frame with 50 observations on 5 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

```
data(USArrests)
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

```
data(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

```
data(USJudgeRatings)
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, and 1960.

Usage

```
data(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

```
data(USPersonalExpenditure)
USPersonalExpenditure
require(eda)
medpolish(log10(USPersonalExpenditure))
```

---

uspop *The Population of the United States*

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

```
data(uspop)
```

Format

A time series of 19 values.

Source

McNeil, D. R. (1977) *Interactive Data Analysis.* New York: Wiley.

Examples

```
data(uspop)
plot(uspop, log = "y", main = "uspop data", xlab = "Year",
     ylab = "U.S. Population (millions)")
```

VADeaths *Death Rate Data*

Description

Death rates per 100 in Virginia in 1940.

Usage

```
data(VADeaths)
```

Format

A matrix with 5 rows and 5 columns.

Details

The death rates 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

```
data(VADeaths)
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)
```

---

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). The attributes of `x` are removed.

`is.vector` returns `TRUE` if `x` is a vector (of mode logical, integer, real, complex, character or list if not specified) 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 "any".
length	A non-negative integer specifying the desired length.
x	An object.

Details

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.

Note that factors are *not* vectors; is.vector returns FALSE and as.vector converts to character mode.

Value

For vector, a vector of the given length and mode. Logical vector elements are initialized to FALSE, numeric vector elements to 0 and character vector elements to "".

See Also

c, is.numeric, is.list, etc.

Examples

```
df <- data.frame(x=1:3, y=5:7)
## Error:
  as.vector(data.frame(x=1:3, y=5:7), mode="numeric")


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")
```

---

volcano *Topographic Information for the 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

```
data(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

```
data(volcano)
filled.contour(volcano, color = terrain.colors, asp = 1)
title(main = "volcano data: filled contour map")
```

`warning` *Warning Messages*

Description

Generates a warning message that corresponds to its argument(s) and the expression or function from which it was called.

Usage

```
warning(...)
```

Arguments

`...`	character vectors (which are pasted together with no separator) or `NULL`.

Details

The result *depends* on the value of `options("warn")`.

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.

If `warn` is zero (the default), a top-level variable `last.warning` is created. It contains the warnings which can be printed via a call to `warnings`.

See Also

`stop` for fatal errors, `warnings`, and `options` with argument `warn=`.

Examples

warnings *Print Warning Messages*

Description

`warnings` prints the top-level variable `last.warning` in a pleasing form.

Usage

```
warnings(...)
```

Arguments

`...`	arguments to be passed to `cat`.

See Also

`warning`.

Examples

```
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
```

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

```
data(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

```
data(warpbreaks)
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)
```

`weekdays` *Extract Parts of a POSIXt 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 = FALSE)
months(x, abbreviate = FALSE)
quarters(x, ...)
julian(x, origin = as.POSIXct("1970-01-01", tz="GMT"), ...)
```

Arguments

`x`	an object inheriting from class `"POSIXt"`.
`abbreviate`	logical. Should the names be abbreviated?
`origin`	an length-one object inheriting from class `"POSIXt"`.
`...`	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`

Examples

```
weekdays(.leap.seconds)
months(.leap.seconds)
quarters(.leap.seconds)
```

`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 $x > 0$. The cumulative is $F(x) = 1 - \exp(-(x/\sigma)^a)$, the mean is $E(X) = \sigma\Gamma(1 + 1/a)$, and the $Var(X) = \sigma^2(\Gamma(1 + 2/a) - \Gamma(1 + 1/a))$.

Value

`dweibull` gives the density, `pweibull` gives the distribution function, `qweibull` gives the quantile function, and `rweibull` generates random deviates.

Note

The cumulative hazard $H(t) = -\log(1 - F(t))$ is `-pweibull(t, a, b, lower = FALSE, log = TRUE)` which is just $H(t) = (t/b)^a$.

See Also

`dexp` for the Exponential which is a special case of a 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 the usual 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 `lm.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)
```

`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 who's 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.

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?
str(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.

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.

Examples

```
x <- c(1:4,0:5,11)
which.min(x)
which.max(x)

data(presidents)
presidents[1:30]
range(presidents, na.rm = TRUE)
which.min(presidents)# 28
which.max(presidents)# 2
```

`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.
`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.

Value

`dwilcox` gives the density, `pwilcox` gives the distribution function, `qwilcox` gives the quantile function, and `rwilcox` generates random deviates.

Author(s)

Kurt Hornik ⟨hornik@ci.tuwien.ac.at⟩

See Also

`dsignrank` etc, for the *one-sample* Wilcoxon rank statistic.

Examples

```
x <- -1:(4*6 + 1)
fx <- dwilcox(x, 4, 6)
all(fx == dwilcox(x, 6, 4))
Fx <- pwilcox(x, 4, 6)
all(abs(Fx - cumsum(fx)) < 10 * .Machine$double.eps)

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")
```

---

winDialog *Dialog Boxes under Windows*

Description

Put up a Windows dialog box to communicate with the user. There are various types, either for the user to select from a set of buttons or to edit a string.

Usage

```
winDialog(type = c("ok", "okcancel", "yesno", "yesnocancel"), message)
winDialogString(message, default)
```

Arguments

`type`	character. The type of dialog box. It will have the buttons implied by its name.
`message`	character. The information field of the dialog box.
`default`	character. The default string.

Value

For `winDialog` a character string giving the name of the button pressed (in capitals) or `NULL` (invisibly) if the user had no choice.

For `winDialogString` a string giving the contents of the text box when `Ok` was pressed, or `NULL` if `Cancel` was pressed.

Note

The standard keyboard accelerators work with these dialog boxes: where appropriate `Return` accepts the default action, `Esc` cancels and the underlined initial letter (`Y` or `N`) can be used.

See Also

`winMenuAdd`
`file.choose` to select a file
package `windlgs` in the package source distribution for ways to program dialogs in C in the `GraphApp` toolkit.

Examples

```
winDialog("yesno", "Is it OK to delete file blah")
```

`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, ...)
window.default(x, start = NULL, end = NULL,
               frequency = NULL, deltat = NULL, extend = FALSE, ...)
window.ts(x, start = NULL, end = NULL,
          frequency = NULL, deltat = NULL, extend = FALSE, ...)
```

Arguments

`x`	a time-series or other object.
`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.

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.

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 NA if needed.

See Also

time, ts.

Examples

```
data(presidents)
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)
```

---

windows *Windows graphics devices*

Description

A graphics device is opened. For win.graph, windows, x11 and X11 this is a graphics window on the current Windows display: the multiple names are for compatibility with other systems. win.metafile prints to a file and win.print to the Windows print system.

Usage

```
windows(width = 7, height = 7, pointsize = 12,
        record = getOption("graphics.record"),
        rescale = c("R", "fit", "fixed"), xpinch, ypinch,
        canvas = "white", gamma = getOption("gamma"))
win.graph(width = 7, height = 7, pointsize = 12)
x11(width = 7, height = 7, pointsize = 12)
X11(width = 7, height = 7, pointsize = 12)
win.metafile(filename = "", width = 7, height = 7, pointsize = 12)
win.print(width = 7, height = 7, pointsize = 12)
```

Arguments

display	indicates the purpose of the device.
filename	the name of the output file: it will be an enhanced Windows metafile, usually given extension .emf or .wmf.
width	the (nominal) width of the plotting window in inches.

`height` the (nominal) height of the plotting window in inches.

`pointsize` the default pointsize of plotted text.

`record` logical: sets the initial state of the flag for recording plots.

`rescale` controls the action for resizing plots.

`xpinch, ypinch`
double. Pixels per inch, horizontally and vertically.

`canvas` color. The color of the canvas which is visible when the background color is transparent.

`gamma` the gamma correction factor. This value is used to ensure that the colors displayed are linearly related to RGB values. By default this is taken from `options("gamma"`, or is 1 (no correction) if that is unset. It sets `par("gamma")` for the device.

Details

All these devices are implemented as `windows` devices, the `display` parameter selects which is actually used.

The size of a window is computed from information provided about the display: it depends on the system being configured accurately. By default a screen device asks Windows for the number of pixels per inch. This can be overridden (it is often wrong) by specifying `xpinch` and `ypinch` or the corresponding options `"xpinch"` and `"ypinch"`.

A graphics window is not allowed to be specified at more that 85% of the screen width or height: the width and height are rescaled proportionally. The window can be resized to a larger size.

If the `filename` is omitted for a `win.metafile` device, the output is copied to the clipboard when the device is closed. A `win.metafile` device can only be used for a single page.

If a screen device is re-sized, the default behaviour is to redraw the plot(s) as if the new size had been specified originally. Using `"fit"` will rescale the existing plot(s) to fit the new device region, preserving the aspect ratio. Using `"fixed"` will leave the plot size unchanged, adding scrollbars if part of the plot is obscured.

A graphics window will never be created at more that 85% of the screen width or height, but can be resized to a larger size. For the first two `rescale` options the width and height are rescaled proportionally if necessary, and if `rescale = "fit"` the plot(s) are rescaled accordingly. If `rescale = "fixed"` the initially displayed portion is selected within these constraints, separately for width and height.

Using `strwidth` or `strheight` after a window has been rescaled (when using `"fit"`) gives dimensions in the original units, but only approximately as they are derived from the metrics of the rescaled fonts (which are in integer sizes)

The displayed region may be bigger than the 'paper' size, and areas outside the 'paper' are coloured in the Windows application background colour. Graphics parameters such as `"din"` refer to the scaled plot if rescaling is in effect.

The different colours need to be distinguished carefully. The area outside the device region is coloured in the Windows application background colour. The device region is coloured in the canvas colour. This is over-painted by the background colour of a plot when a new page is called for, but that background colour can be transparent (and is by default). The difference between setting the canvas colour and the background colour is that when the device is copied the background colour is copied but the canvas colour is not.

Recorded plot histories are of class "SavedPlots". They have a print method, and a subset method. As the individual plots are of class "recordedplot" they can be replayed by printing them: see recordPlot.

Value

A plot device is opened: nothing is returned to the R interpreter.

Author(s)

Guido Masarotto and Brian Ripley

See Also

savePlot, bringToTop, Devices, postscript

---

winextras *Auxiliary Functions for the Windows Port*

Description

Auxiliary functions for the Windows port

Usage

```
flush.console()
win.version()
zip.unpack(zipname, dest)
```

Arguments

zipname	character string giving name of zip file.
dest	character string giving directory within which to unpack.

Details

flush.console flushes the console output buffer in Rgui and does nothing under other front-ends.

win.version is an auxiliary function for bug.report which returns a character string describing the version of Windows in use.

zip.unpack unpacks the zip file zipname in directory dest: it is an internal version of unzip zipfile -d dest (but will use an external unzip if one is set in options("unzip")).

`winMenus` *User Menus under Windows*

Description

Enables users to add, delete and program menus under Windows.

Usage

```
winMenuAdd(menuname)
winMenuAddItem(menuname, itemname, action)
winMenuDel(menuname)
winMenuDelItem(menuname, itemname)
```

Arguments

`menuname`	a character string naming a menu.
`itemname`	a character string naming a menu item on an existing menu.
`action`	a character string describing the action when that menu is selected, or `"enable"` or `"disable"`.

Details

User menus are added to the right of existing menus, and items are added at the bottom of the menu.

By default the action character string is treated as R input, being echoed on the command line and parsed and executed as usual.

Specifying an existing item in `winMenuAddItem` enables the action to be changed.

Submenus can be specified by separating the elements in `menuname` by slashes: as a consequence menu names may not contain slashes.

If the `action` is specified as `"none"` no action is taken: this can be useful to reserve items for future expansion.

Value

`NULL`, invisibly. If an error occurs, an informative error message will be given.

See Also

`winDialog`

Examples

```
winMenuAdd("Testit")
winMenuAddItem("Testit", "one", "aaaa")
winMenuAddItem("Testit", "two", "bbbb")
winMenuAdd("Testit/extras")
winMenuAddItem("Testit", "-", "")
winMenuAddItem("Testit", "two", "disable")
winMenuAddItem("Testit", "three", "cccc")
winMenuAddItem("Testit/extras", "one more", "ddd")
winMenuAddItem("Testit/extras", "and another", "eee")
```

`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.

See Also

evalq.

Examples

```
#examples from glm:

library(MASS)
data(anorexia)
with(anorexia, {
    anorex.1 <- glm(Postwt ~ Prewt + Treat + offset(Prewt),
                    family = gaussian)
    summary(anorex.1)
})


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:
data(ToothGrowth)
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"))
```

`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

```
data(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

```
data(women)
plot(women, xlab = "Height (in)", ylab = "Weight (lb)",
     main = "women data: American women aged 30-39")
```

---

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)
```

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.
`ncolumns`	the number of columns to write the data in.
`append`	if `TRUE` the data `x` is appended to file `file`.

See Also

`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 6 9 will form the first row
write(t(x))

# the file data now contains the data in x,
# two rows, five cols but the first row is 1 2 3 4 5
write(x)
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 already) to `file`. The entries in each line (row) are separated by the value of `sep`.

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"))
```

Arguments

`x`	the object to be written, typically a 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 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 the variable (columns) to quote. In both cases, row and columns names are quoted if they are written, but not if `quote is FALSE`.
`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.
`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.
`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.

Details

Normally there is no column name for a column of row names. If `col.names=NA` a blank column name is added. This can be used to write CSV files for input to spreadsheets.

`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. Function `write.matrix` in package `MASS` may be much more efficient if `x` is a matrix or can be represented in a numeric matrix.

See Also

The 'R Data Import/Export' manual.

`read.table`, `write`.

`write.matrix`.

Examples

```
## To write a CSV file for input to Excel one might use
write.table(x, file = "foo.csv", sep = ",", col.names = NA)
## and to read this file back into R one needs
read.table("file.csv", header = TRUE, sep = ",", row.names=1)
```

---

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 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 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, LF on Macintosh). 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`, `readLines`, `cat`

`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 = "Rplots.fig", onefile = FALSE, ...)
```

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`. For use with `onefile=FALSE` give a `printf` format such as `"Rplot%d.fig"` (the default in that case).
`onefile`	logical: if true (the default) allow multiple figures in one file. If false, assume only one page per file and generate a file number containing the page number.
`...`	further options for `xfig()`, such as:
`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"`.
`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.

Note

On 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.

---

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	a data frame, list or environment containing the variables to be cross-tabulated.
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.
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 in package **ctest** 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

```
data(esoph)
## '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))

data(UCBAdmissions)
## 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 independece ...
summary(xtabs(Freq ~ ., DF))

data(warpbreaks)
## Create a nice display for the warp break data.
warpbreaks$replicate <- rep(1:9, len = 54)
ftable(xtabs(breaks ~ wool + tension + replicate, data = warpbreaks))
```

`xy.coords` *Extracting Plotting Structures*

Description

`xy.coords` is used by many function to obtain x and y coordinates for plotting. The use of this common mechanism across all R functions produces a measure of consistency.

`plot.default` and `lowess` are examples of functions which use this mechanism.

Usage

```
xy.coords(x, y, 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 be provided. In this case, an attempt is made to interpret the argument in a way suitable for plotting. If the argument is a formula `yvar ~ xvar`, `xvar` and `yvar` are used as x and y variables; if the argument is a list containing components `x` and `y`, these are used are assumed to define plotting coordinates; if the argument contains a time series, the x values are taken to be time and the y values to be the time series; if the argument

	is a matrix with two columns, the first is assumed to contain the x values and the second the y values; in any other case, the argument is coerced to a vector and the values plotted against their indices.
`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.

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`.

Examples

```
xy.coords(fft(c(1:10)), NULL)
data(cars) ; attach(cars)
xy.coords(dist ~ speed, NULL)$xlab # = "speed"

str(xy.coords(1:3, 1:2, recycle=TRUE))
str(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, z, 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. Alternatively, a single argument `x` can be be provided. In this case, an attempt is made to interpret the argument 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 with three columns, the first is assumed to contain the x values, etc. Alternatively, two arguments `x` and `y` can be be provided. 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

```
str(xyz.coords(data.frame(10*1:9, -4),y=NULL,z=NULL))

str(xyz.coords(1:6, fft(1:6),z=NULL,xlab="X", ylab="Y"))

y <- 2 * (x2 <- 10 + (x1 <- 1:10))
str(xyz.coords(y ~ x1 + x2,y=NULL,z=NULL))

str(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 ...
```

---

zcbind *Bind Two or More Time Series*

Description

Bind Two or More Time Series which have common frequency.

Usage

```
.cbind.ts(sers, nmsers, dframe = FALSE, union = TRUE)
```

Arguments

`sers`	a list of two or more univariate or multivariate time series, or objects which can coerced to time series.
`nmsers`	a character vector of the same length as `sers` with the names for the time series.
`dframe`	logical; if `TRUE` return the result as a data frame.
`union`	logical; if `TRUE`, act as `ts.union` or `ts.intersect`.

Details

This is an internal function which is not to be called by the user.

`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.

Details

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 used by `tempfile`.

Value

The name of the original or extracted file.

Warning

This function is intended for internal use only: it may be altered at any time, and may differ between platforms. Do NOT use in user code.

Note

The implementation differs by platform: it might do nothing.

Author(s)

B. D. Ripley

Chapter 2

The ctest package

`ansari.test` *Ansari-Bradley Test*

Description

Performs the Ansari-Bradley two-sample test for a difference in scale parameters.

Usage

```
ansari.test(x, y, alternative = c("two.sided", "less", "greater"),
            exact = NULL, conf.int = FALSE, conf.level = 0.95, ...)
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 data frame containing the variables in the model 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.

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

Myles Hollander & Douglas A. Wolfe (1973), *Nonparametric statistical inference.* New York: John Wiley & Sons. Pages 83–92.

David F. Bauer (1972), Constructing confidence sets using rank statistics. *Journal of the American Statistical Association* **67**, 687–690.

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.

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)
```

---

bartlett.test *Bartlett Test for 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, g, ...)
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 data frame containing the variables in the model 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 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 for 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

```
data(InsectSprays)
plot(count ~ spray, data = InsectSprays)
bartlett.test(InsectSprays$count, InsectSprays$spray)
bartlett.test(count ~ spray, data = InsectSprays)
```

`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.

Conover, W. J. (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.
```

`chisq.test` *Pearson's Chi-squared Test for Count Data*

Description

`chisq.test` performs chi-squared tests on contingency tables.

Usage

```
chisq.test(x, y = NULL, correct = TRUE,
           p = rep(1/length(x), length(x)),
           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.
`p`	a vector of probabilities of the same length of `x`.
`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 simulation.

Details

If `x` is a matrix with one row or column, or if `x` is a vector and `y` is not given, `x` is treated as a one-dimensional contingency table. 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, and hence its entries should be nonnegative 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 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`. Otherwise, if `simulate.p.value` is `TRUE`, the p-value is computed by Monte Carlo simulation with `B` replicates. This is done by random sampling from the set of all contingency tables with given marginals, and works only if the marginals are positive.

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.

Examples

```
data(InsectSprays)              # 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
## Case B. Raw data
x <- trunc(5 * runif(100))
chisq.test(table(x))            # NOT 'chisq.test(x)'!
```

`cor.test` *Test for Association 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, y,
         alternative = c("two.sided", "less", "greater"),
         method = c("pearson", "kendall", "spearman"),
         exact = NULL, conf.level = 0.95, ...)
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. Only used for Kendall's τ. 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 data frame containing the variables in the model 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 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.
data(USJudgeRatings)
pairs(USJudgeRatings)
cor.test(~ CONT + INTG, data = USJudgeRatings)
```

`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,
            or = 1, alternative = "two.sided", conf.level = 0.95)
```

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.
`hybrid`	a logical indicating whether the exact probabilities (default) or a hybrid approximation thereof should be computed. In the hybrid case, asymptotic chi-squared probabilities are only used provided that the "Cochran" conditions are satisfied.
`or`	the hypothesized odds ratio. Only used in the 2 by 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 by 2 case.
`conf.level`	confidence level for the returned confidence interval. Only used in the 2 by 2 case.

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.

In the one-sided 2 by 2 cases, p-values are obtained directly using the 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 & Joe (1993). The FORTRAN code can be obtained from http://www.netlib.org/toms/643.

In the 2 by 2 case, 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).

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 by 2 case.
`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 by 2 case.
`null.value`	the odds ratio under the null, `or`. Only present in the 2 by 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

Alan Agresti (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.

Cyrus R. Mehta & Nitin R. Patel (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.

Douglas B. Clarkson, Yuan-an Fan & Harry Joe (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.

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 women'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), Convictions of like-sex twins in criminals
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.level = 0.95)$conf.int
fisher.test(Convictions, conf.level = 0.99)$conf.int
```

`fligner.test` *Fligner-Killeen Test for 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, g, ...)
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 data frame containing the variables in the model 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 $a(i) = \text{qnorm}((1 + i/(n + 1))/2)$. 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 for homogeneity of variances"`.
`data.name`	a character string giving the names of the data.

References

W. J. Conover & Mark E. Johnson & Myrie 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

```
data(InsectSprays)
plot(count ~ spray, data = InsectSprays)
fligner.test(InsectSprays$count, InsectSprays$spray)
fligner.test(count ~ spray, data = InsectSprays)
## Compare this to bartlett.test()
```

`friedman.test` *Friedman Rank Sum Test*

Description

Performs a Friedman rank sum test with unreplicated blocked data.

Usage

```
friedman.test(y, groups, blocks, ...)
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 data frame containing the variables in the model 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

data(warpbreaks)
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)
```

`kruskal.test` *Kruskal-Wallis Rank Sum Test*

Description

Performs a Kruskal-Wallis rank sum test.

Usage

```
kruskal.test(x, g, ...)
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 data frame containing the variables in the model 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.
data(airquality)
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 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.
`exact`	`NULL` or a logical indicating whether an exact p-value should be computed. See Details for the meaning of `NULL`. Only used in the two-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 of the null that the distribution function underlying `x` is `y` with parameters specified by `...` is carried out.

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.

Exact p-values are only available for the two-sided two-sample test with no ties. In that case, if `exact = NULL` (the default) an exact p-value is computed if the product of the sample sizes is less than 10000. Otherwise, asymptotic distributions are used whose approximations may be inaccurate in small samples.

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

Conover, W. J. (1971), *Practical nonparametric statistics*. New York: John Wiley & Sons. Pages 295–301 (one-sample "Kolmogorov" test), 309–314 (two-sample "Smirnov" test).

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 scale 2?
ks.test(x+2, "pgamma", 3, 2) # two-sided
ks.test(x+2, "pgamma", 3, 2, alternative = "gr")
```

`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.

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
data(UCBAdmissions)
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.)
```

`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)
## => very strong association between the two successive ratings
```

---

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, y, alternative = c("two.sided", "less", "greater"), ...)
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 data frame containing the variables in the model 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

Conover, W. J. (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)
```

`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 data frame containing the variables in the model 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")`.

`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 preformed. 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

```
data(sleep)
## 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))
```

`pairwise.prop.test` *Pairwise comparisons of 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

```
data(airquality)
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 `t.test`

Value

Object of class `"pairwise.htest"`

See Also

`wilcox.test`, `p.adjust`

Examples

```
data(airquality)
attach(airquality)
Month <- factor(Month, labels = month.abb[5:9])
pairwise.wilcox.test(Ozone, Month)
pairwise.wilcox.test(Ozone, Month, p.adj = "bonf")
detach()
```

`power.prop.test` *Power calculations two sample test for of 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"))
```

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

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.

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"))
```

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

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.

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")
```

`print.pairwise.htest` *Print method for pairwise tests*

Description

Display results of pairwise comparison procedures

Usage

```
print(x, ...)
```

Arguments

x	Object of class "pairwise.htest"
...	further arguments to be passed to or from methods.

Value

None

See Also

pairwise.t.test, et al.

---

print.power.htest *Print method for power calculation object*

Description

Print object of class "power.htest" in nice layout.

Usage

```
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

`prop.test` *Test for 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.

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))
```

`quade.test` *Quade Test*

Description

Performs a Quade test with unreplicated blocked data.

Usage

```
quade.test(y, groups, blocks, ...)
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 data frame containing the variables in the model 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

`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.

W. 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)
```

`shapiro.test` *Shapiro-Wilk Normality Test*

Description

Performs the Shapiro-Wilk test for 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))
```

---

t.test *Student's t-Test*

Description

Performs one and two sample t-tests on vectors of data.

Usage

```
t.test(x, y = NULL, alternative = c("two.sided", "less", "greater"),
       mu = 0, paired = FALSE, var.equal = FALSE,
       conf.level = 0.95, ...)
t.test(formula, data, subset, na.action, ...)
```

Arguments

`x`	a numeric vector of data values.
`y`	an optional numeric vector 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 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 data frame containing the variables in the model 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.

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.

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
data(sleep)
plot(extra ~ group, data = sleep)
## Traditional interface
attach(sleep)
t.test(extra[group == 1], extra[group == 2])
detach()
## Formula interface
t.test(extra ~ group, data = sleep)
```

`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, y, ratio = 1, alternative = c("two.sided", "less", "greater"),
         conf.level = 0.95, ...)
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 data frame containing the variables in the model 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.
```

`wilcox.test` *Wilcoxon Rank Sum and Signed Rank Tests*

Description

Performs one and two sample Wilcoxon tests on vectors of data.

Usage

```
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, ...)
wilcox.test(formula, data, subset, na.action, ...)
```

Arguments

`x`	numeric vector of data values.
`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 location parameter.
`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 data frame containing the variables in the model 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) is carried out. In this case, the null hypothesis is that the location of the distributions of `x` and `y` differ by `mu`.

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.

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`.)

References

Myles Hollander & Douglas A. Wolfe (1973), *Nonparametric statistical inference.* New York: John Wiley & Sons. Pages 27–33 (one-sample), 68–75 (two-sample).

David F. Bauer (1972), Constructing confidence sets using rank statistics. *Journal of the American Statistical Association* **67**, 687–690.

See Also

`kruskal.test` for testing homogeneity in location parameters in the case of two or more samples; `t.test` for a parametric alternative under normality assumptions.

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.
data(airquality)
boxplot(Ozone ~ Month, data = airquality)
wilcox.test(Ozone ~ Month, data = airquality,
            subset = Month %in% c(5, 8))
```

Chapter 3

The eda package

`line` *Robust Line Fitting*

Description

Fit a line robustly as recommended in *Exploratory Data Analysis*.

Usage

```
line(x, y)

coef(object, ...)
residuals(object, type, ...)
fitted(object, ...)
print(x, digits = max(3, getOption("digits") - 3), ...)
```

Arguments

`x,y`	the arguments can be any way of specifying x-y pairs.
`object`	a `tukeyline` object, typically the result of `line(*)`.
`digits`	number of significant digits to use, see `print`.
`type, ...`	potentially further arguments, required by generic.

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

```
library(eda)
data(cars)
plot(cars)
(z <- line(cars))
abline(coef(z))
## Tukey-Anscombe Plot :
plot(residuals(z) ~ fitted(z), main = deparse(z$call))
```

`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)
```

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?

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`.

`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)
```

---

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)

  print(x, ...)
summary(object, ...)
```

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".
object	(and x in print(.)): object of class "tukeysmooth", typically the result of smooth(.).
...	potentially further arguments, required by generic.

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 in package 'modreg'.

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)))

data(presidents)
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)
```

Chapter 4

The `lqs` package

`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(x, cor = FALSE, quantile.used = floor((n + p + 1)/2),
        nsamp = "best", seed)
cov.mcd(x, cor = FALSE, quantile.used = floor((n + p + 1)/2),
        nsamp = "best", seed)
```

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.

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 (1997) 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")
```

---

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, ...)
lqs.formula(formula, data, ...,
            method = c("lts", "lqs", "lms", "S", "model.frame"),
            subset, na.action = na.fail, model = TRUE,
            x = FALSE, y = FALSE, contrasts = NULL)
lqs.default(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. An alternative is `na.omit`, which leads to omission of cases with missing values on any required variable. (NOTE: If given, this argument must be named exactly.)
`model`	logical. If `TRUE` the model frame is returned.
`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.

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" 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?

Value

An object of class "lqs".

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")
```

`predict.lqs` *Predict from an lqs Fit*

Description

Predict from an resistant regression fitted by `lqs`.

Usage

```
predict(object, newdata, ...)
```

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.
`...`	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 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 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)
```

Chapter 5

The `methods` package

`.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 libraries, 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, coerceFlag=TRUE)

as(object, Class, coerceFlag=TRUE) <- value

setAs(from, to, def, replace, where = 1)
```

Arguments

`object`	Any object.
`Class`	The name of the class to which `object` should be coerced.
`coerceFlag`	A logical flag. If `FALSE`, only inheritance relations will be used; i.e., `is(object, Class)` must be `TRUE`, and if not, `NULL` is returned. If the flag is `TRUE`, however, coerce methods will be called if inheritance is not true (and an error will result if coercion fails).
`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`	A 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 database on which to store the resulting method for `coerce`; by default, the global environment.

Summary of Functions

`as`: Returns the version of this object coerced to be the given `Class`.

If the corresponding `is` relation is true, it will be used. In particular, if the relation has a coerce method, the method will be invoked on `object`.

If the `is` relation is FALSE, and `coerceFlag` is `TRUE`, the coerce function will be called (which will throw an error if there is no valid way to coerce the two objects). Otherwise, `NULL` is returned.

Coerce methods are pre-defined for basic classes (including all the types of vectors, functions and a few others). The object `asFunctions` contains the list of such pre-defined relations: `names(asFunctions)` gives the names of all the classes.

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 a 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 `coerce` exists almost entirely as a repository for such methods, to be selected as desribed 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.

Author(s)

John Chambers

References

The web page http://www.omegahat.org/RSMethods/index.html is the primary documentation.

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.

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)new("ca", a=from@b, id = from@id))

as(CB, "numeric")
```

`BasicFunctions` *Group Generic Functions*

Description

These are group generic functions. Methods defined for them will be used for any of the specific functions belonging to the particular group, provided no specific methods override.

These functions should never be called directly (a suitable error message will result if they are).

Usage

```
Arith(e1, e2)
Compare(e1, e2)
Ops(e1, e2)

Math(x)
Math2(x, digits)

Summary(x, ..., na.rm = FALSE)

Complex(z)
```

Arguments

`e1, e2`	Arguments to the various binary operators.
`x`	The argument to the `Math` or `Summary` groups of functions.

Details

The functions belonging to the various groups are as follows:

```
Arith "+", "-", "*", "^", "%%", "%/%", "/"

Compare "==", ">", "<", "!=", "<=", ">="

Ops "Arith", "Compare"

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"
```

All the functions in these groups (other than the group generics themselves) are basic functions in R. They are not by default generic functions, and many of them are defined as primitives, meaning that they do not have formal arguments. However, you can still define methods for them. The effect of doing so is to create a 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.

Author(s)

John Chambers

References

The web page http://www.omegahat.org/RSMethods/index.html is the primary documentation.

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.

---

class *Class of an Object*

Description

Returns the name of the object's class as a character string.

In contrast to the version of this function in the base package, this version of `class` never returns `NULL`. For objects that do not have a formal class definition, and do not have the `"class"` attribute set, the value returned is effectively the same as `data.class`.

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.

Usage

```
class(object)

class(object) <- value
```

Arguments

`object`	Any R object (including basic objects for which no class is currently defined). When assigning the class, however, it must be possible to coerce the object to the specified class: the semantics of assigning a class to `object` are equivalent to `object <- as(object, value)`.

Author(s)

John Chambers

References

The web page http://www.omegahat.org/RSMethods/index.html is the primary documentation.

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

as

`Classes` *Class Definitions*

Description

Class definitions are objects that contain the formal definition of a class of R objects.

Details

When a class is defined, an object is stored that contains the information about that class, including:

slots Each slot is a component object. Like elements of a list these may be extracted (by name) and set. However, they differ from list components in important ways.

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.

Classes don't need to have any slots, and many useful classes do not. These objects usually extend other, simple objects, such as numeric or character vectors. Finally, classes can have no data at all—these are known as *virtual* classes and are in fact very important programming tools. They are used to group together ordinary classes that want to share some programming behavior, without necessarily restricting how the behavior is implemented.

extends The names of 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.

In other programming languages, this relationship is sometimes expressed by saying that `Simple` is a superclass of `Fancy`, or that `Fancy` is a subclass of `Simple`.

The actual class definition object contains the names of all the classes this class extends. But those classes can themselves extend other classes also, so the complete extension can only be known by obtaining all those class definitions.

Class extension is usually defined when the class itself is defined, by including the names of superclasses as unnamed elements in the representation argument to setClass.

An object from a given class will then have all the slots defined for its own class *and* all the slots defined for its superclasses as well.

Note that `extends` relations can be defined in other ways as well, by using the setIs function.

prototype Each class definition contains a prototype object from the class. This must have all the slots, if any, defined by the class definition.

The prototype most commonly just consists of the prototypes of all its slots. But that need not be the case: 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. 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.

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.

Author(s)

John Chambers

References

The web page http://www.omegahat.org/RSMethods/index.html is the primary documentation.

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

ClassUtils *Utilities for Managing Class Definitions*

Description

These are various functions to support the definition and use of formal classes. Most of them are rarely suitable to be called directly.

Usage

```
testVirtual(properties, extends, prototype)

makePrototypeFromClassDef(properties, prototype, extends)

newEmptyObject()

completeClassDefinition(Class, ClassDef)

getFromClassDef(ClassDef, what)

setInClassDef(ClassDef, what, value, synchronize=TRUE)
```

```
synchronizeClassDef(ClassDef)

getProperties(ClassDef)

setProperties(ClassDef, value)

getSlots(x, complete = TRUE)

getExtends(ClassDef)

getAccess(ClassDef)

getAllSuperClasses(ClassDef)

superClassDepth(ClassDef, soFar)

setExtends(ClassDef, value)

getPrototype(ClassDef)

setPrototype(ClassDef, value)

getVirtual(ClassDef)

isVirtualClass(Class)

setVirtual(ClassDef, value)

getSubclasses(ClassDef)

setSubclasses(ClassDef, value)

getClassName(ClassDef)

setClassName(ClassDef, value)

assignClassDef(Class, def, where=.GlobalEnv)

newClassEnvironment(name, properties, extends, prototype, subclasses,
                    virtual, validity, access)

newBasic(Class, ..., .Force=FALSE)

makeExtends(extends)

reconcilePropertiesAndPrototype(name, properties, prototype, extends)

tryNew(Class)

empty.dump()
```

```
showClass(Class, complete=TRUE, printTo=stdout(),
          propertiesAreCalled="Properties")

showExtends(ext, printTo = stdout())

print.classRepEnvironment(x, ...)

print.environment(x, ...)

getFromClassMetaData(name)

assignToClassMetaData(name, value)

removeFromClassMetaData(name)

extendsCoerce(fromClass, Class, formFunction)

extendsReplace(fromClass, Class)

findExtends(class1, class2)

completeExtends(ClassDef, soFar)

classMetaName(name)

methodsMetaName(prefix, name)

as.data.frame(x, row.names=NULL, optional=FALSE)

requireMethods(functions, signature, message)

checkSlotAssignment(obj, name, value)

defaultPrototype()

isClassDef(object)
```

Summary of Functions

`testVirtual`: Test for a Virtual Class. Figures out, as well as possible, whether the class with these properties, extension, and prototype is a virtual class. Can be forced to be virtual by extending "VIRTUAL".

Otherwise, a class is virtual only if it has no slots, extends no non-virtual classes, and has a NULL Prototype

`makePrototypeFromClassDef`: Makes the prototype implied by the class definition.

The following three rules are applied in this order.

If the class has slots, then the prototype for each slot is used by default, but a corresponding element in the explicitly supplied prototype, if there is one, is used instead (but it must be coercible to the class of the slot).

If there are no slots but a non-null prototype was specified, this is returned.

If there is a single non-virtual superclass (a class in the extends list), then its prototype is used.

If all three of the above fail, the prototype is `NULL`.

`newEmptyObject`: Utility function to create an empty object into which slots can be set.

Currently just creates an empty list with class "NULL"

Later version should create a special object reference that marks an object currently with no slots and no data.

`completeClassDefinition`: Completes the definition of Class, relative to the current session.

The completed definition is stored in the session's class metadata, to be retrieved the next time that getClass is called on this class, and is returned as the value of the call.

If `ClassDef` is omitted, the initial definition is obtained from the first package having a meta-object for this class.

`getFromClassDef`: Extracts one of the intrinsically defined class definition properties (".Poperties", etc.) Strictly a utility function

`setInClassDef`: Set Property in Class Definition set one of the intrinsically defined class definition properties (".Poperties", etc.) Strictly a utility function

`synchronizeClassDef`: Does whatever is needed to synchronize information in the class definition.

Basically computes derived information used to make object manipulations more efficient but that need to be revised if information changes.

(Nothing at the moment)

`getProperties`: Extracts the class's Properties information from the class representation (only, not from the name of the class).

`setProperties`: Sets the class's Properties information given the class representation (only, not from the name of the class)

`getSlots`: Returns a named character vector. The names are the names of the slots, the values are the classes of the corresponding slots. If `complete` is `TRUE`, all slots from all superclasses will be included. The argument `x` can either be the name of a class or an object having that class.

`getExtends`: Extracts the class's Extends information from the class representation (only, not from the name of the class)

Contrast with the `findExtends` and `is` functions, both of which use indirect information as well.

`getAllSuperClasses, superClassDepth`: Get the names of all the classes that this class definition extends.

`getAllSuperClasses` is a utility function used to complete a class definition. It returns all the superclasses reachable from this class, in breadth-first order (which is the order used for matching methods); that is, the first direct superclass followed by all its superclasses, then the next, etc. (The order is relevant only in the case that some of the superclasses have multiple inheritance.)

`superClassDepth`, which is called from `getAllSuperClasses`, returns the same information, but as a list with components `label` and `depth`, the latter for the number of generations back each class is in the inheritance tree. The argument `soFar` is used to avoid loops in the network of class relationships.

`setExtends`: set the class's Extends information given the class representation (only, not from the name of the class)

`getPrototype`: extract the class's Prototype information from the class representation (only, not from the name of the class)

`getAccess`: extract the class's Access information from the class representation (only, not from the name of the class)

`setPrototype`: set the class's Prototype information given the class representation (only, not from the name of the class)

`getVirtual`: extract the class's Virtual information from the class representation (only, not from the name of the class)

`isVirtualClass`: Is the named class a virtual class?

A class is virtual if explicitly declared to be, and also if the class is not formally defined.

`setVirtual`: set the class's Virtual information given the class representation (only, not from the name of the class)

`getSubclasses`: extract the class's Subclasses information from the class representation (only, not from the name of the class)

`setSubclasses`: set the class's Subclasses information given the class representation (only, not from the name of the class)

`getClassName`: The internal property in the class definition for the class name.

`setClassName`: set the name of the class inside the class definition

`assignClassDef`: assign the definition of the class to the specially named object

`newBasic`: the implementation of the function `new` for basic classes that don't have a formal definition.

Any of these could have a formal definition, except for Class="NULL" (disallowed because NULL can't have attributes). For all cases except "NULL", the class of the result will be set to Class.

See `new` for the interpretation of the arguments.

`makeExtends`: convert the argument to a list defining the extension mechanism.

`reconcilePropertiesAndPrototype`: makes a list or a structure look like a prototype for the given class.

Specifically, returns a structure with attributes corresponding to the slot names in properties and values taken from prototype if they exist there, from `new(classi)` for the class, `classi` of the slot if that succeeds, and `NULL` otherwise.

The prototype may imply slots not in the properties list, since properties does not include inherited slots (these are left unresolved until the class is used in a session).

`tryNew`: Tries to generate a new element from this class, but if the attempt fails (as, e.g., when the class is undefined or virtual) just returns NULL.

This is inefficient and also not a good idea when actually generating objects, but is useful in the initial definition of classes.

`showClass`: Print the information about a class definition.

If complete==TRUE, include the indirect information about extensions.

`showExtends`: Print the elements of the list of extensions.

Also used to print extensions recorded in the opposite direction, via a subclass list

`print.environment, showNonVector`: These routines provide a default printing mechanism for objects from non-vector classes, implemented as environments. The `print.environment` function exists to intercept printing based on S3-style methods; it calls `showNonVector` to do the work.

`extendsCoerce, extendsReplace:` Returns the function to perform coercion or replacement, respectivelyl, based on the `is` relation between two classes.

A method may be explicitly stored in the metadata or inferred. If the latter, a function will always be constructed if explicit test and/or coerce methods (in a call to setIs) were encountered, or if `formFunction=TRUE`. If `formFunction=FALSE`, the purpose of the call is to determine if the relation is explicit or intrinsic (according to whether a function object is returned).

`findExtends:` Find the information that says whether class1 extends class2, directly or indirectly.

This can be either a logical value or an object containing various functions to test and/or coerce the relationship.

`completeExtends:` complete the extends information in the class definition, by following transitive chains.

Elements in the immediate extends list may be added and current elements may be replaced, either by replacing a conditional relation with an unconditional one, or by adding indirect relations.

The resulting extends list is presented in depth-first order; that is, the first immediate superclass followed by all the indirect relations through it, then the next immediate superclass, etc.

Depth first order is required for consistent elaboration of inherited methods during dispatch, because the method dispatcher stores the inherited method under the immediate class name. Under rather obscure situations of multiple inheritance, the result could be ambiguous (depending on the order in which signatures are seen by the dispatcher for a particular generic function), unless searching is done depth first.

Used recursively, with `soFar` defining what has been included in previous calls at this level.

`classMetaName:` a name for the object storing this class's definition

`methodsMetaName:` a name mangling device to simulate the meta-data in S4

`requireMethods:` Require a subclass to implement methods for the generic functions, for this signature.

For each generic, `setMethod` will be called to define a method that throws an error, with the supplied message.

The `requireMethods` function allows virtual classes to require actual classes that extend them to implement methods for certain functions, in effect creating an API for the virtual class.

Otherwise, default methods for the corresponding function would be called, resulting in less helpful error messages or (worse still) silently incorrect results.

`checkSlotAssignment:` Check that the value provided is allowed for this slot, by consulting the definition of the class. Called from the C code that assigns slots.

For privileged slots (those that can only be set by accesor functions defined along with the class itself), the class designer may choose to improve efficiency by validating the value to be assigned in the accessor function and then calling `slot<-` with the argument `check=FALSE`, to prevent the call to `checkSlotAssignment`.

`defaultPrototype:` **The prototype for a class which will have slots, is not a virtual class, and do**

SessionClassMetaData: Contains the name of the special table in which class information is cached during the session.

`.InitBasicClasses`, `.InitMethodsListClass`, `.setCoerceGeneric`: These functions perform part of the initialization of classes and methods, and are called (only!) from `.First.lib`.

`isClassDef`: Is `object` a representation of a class?

`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)

standardGeneric(f)

dumpMethod(f, signature, file, where, def)

existsFunction(f, generic = TRUE, where)

findFunction(f, generic=TRUE)

dumpMethods(f, file, signature, methods, where)

signature(...)

removeMethods(f, where)

setReplaceMethod(f, ...)

getGenerics(where)

allGenerics(where)

callGeneric(...)
```

Arguments

`f`	The character string naming the function.
`where`	Where on the search list of attached packages to look for functions or methods. By default, use the whole search list to find the relevant object(s).

`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).

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, either on database `where` or anywhere in the current search list. 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`: Remove the generic function of this name.

`standardGeneric`: Dispatches a method from the current function call for the generic function `fname`.

`getMethods`: The list of methods for the specified generic.

`dumpMethod`: Dump the method for this generic function and signature.

`existsFunction`: Is there a function of this name. If `generic==FALSE`, generic functions are not counted.

`findFunction`: return all the elements of the search list on which a function definition for `name` exists.

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.

`selectMethod`: Returns the method (a function) that R would use to evaluate a call to this generic, with arguments corresponding to the specified signature.

f = the name of the generic function sig = the signature of classes to match to the arguments of f.

`dumpMethods`: Dump all the methods for this generic.

`signature`: Returns a named list of classes to be matched to arguments of a generic function.

`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`.

`getMethods`: If the function is not a generic function, returns NULL. The `f` argument can be either the character string name of the generic or the object itself

The `where` argument optionally says where to look for the function, if `f` is given as the name.

`dumpMethod`: The resulting source file will recreate the method.

`findFunction`: If `generic` is FALSE, ignore generic functions.

`selectMethod`: The vector of strings for the classes can be named or not. If named, the names must match formal argument names of f. If not named, the signature is assumed to apply to the arguments of f in order. mustFind = If TRUE, an error results if there is no method (or no unique method) corresponding to this signature. Otherwise may return NULL or a MethodsList object.

`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, it will be removed. The assignment or removal can be controlled by optional argument `where`, which defaults to the first element of the search list having a function called `f`.

Author(s)

John Chambers

References

The web page http://www.omegahat.org/RSMethods/index.html is the primary documentation.

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

setGeneric, setClass, showMethods

---

getClass *Get Class Definition*

Description

Get the definition of a class.

Usage

```
getClass(Class, .Force = FALSE)
getClassDef(Class, where=-1)
```

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

where to search for the definition; by default, anywhere on the current search list.

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 that database, unadorned. It's usually getClass you want.

The statement that getClass fails if the class is undefined is not quite true at the moment. A few special vector types and informally defined classes are allowed, and getClass returns NULL for these. But don't count on it; eventually these holes in the class system should be plugged.

In any case, 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.

Author(s)

John Chambers

References

The web page `http://www.omegahat.org/RSMethods/index.html` is the primary documentation.

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, `setClass`, `isClass`.

Examples

```
getClass("numeric") ## a built in class
```

---

getMethod *Get the Definition of a Method*

Description

A call to `getMethod` returns the method corresponding a particular generic function and signature (set classes for formal arguments). The functions `MethodsListSelect` and `selectMethod` select methods using an environment in which arguments to the function have been assigned (they differ in the treatment of inheritance). Neither is intended to be called directly, except perhaps in debugging. The function `getMethods` returns *all* the methods for the generic function.

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.

Usage

```
getMethod(f, signature=character(), where, optional=FALSE)

findMethod(f, signature, where)

getMethods(f, where=-1)

MethodsListSelect(f, env, mlist, fEnv, finalDefault, evalArgs, useInherited)
selectMethod(f, env, optional=False, useInherited, mlist=getMethods(f))
```

Arguments

`f`	The character-string name of the generic function. In `getMethods` only, this argument may be a function definition, in which case the special methods list object, if any, stored in the environment of the function is returned. (This usage is largely for internal purposes; you aren't likely to have such a function definition for direct use.)
`signature`	The signature of classes to match to the arguments of `f`. The vector of strings for the classes can be named or not. If named, the names must match formal argument names of `f`. If not named, the signature is assumed to apply to the arguments of `f` in order.
`where`	Where to look for the method. By default, looks in all the currently attached databases (i.e., the complete 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`	In `selectMethod`, the `MethodsList` object can be explicitly supplied. (Unlikely to be used, except in the recursive call that finds matches to more than one argument.)
`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`	Internal-use arguments for the function's environment, the method to use as the overall default, whether to evaluate arguments, and which arguments should use inheritance.

Details

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 `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.

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.)

The `selectMethod` function is similar to `MethodsListSelect` (and calls it) but differs in that it is intended to be used with control over which arguments are allowed to use inheritance. The restriction is needed to make sensible method selection for coerce methods (the `as` function).

Value

The call to `selectMethod` or `getMethod` returns a function object, the selected method, if a unique selection exists. 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.

Author(s)

John Chambers

References

The web page `http://www.omegahat.org/RSMethods/index.html` is the primary documentation.

The functions in this package implement a facility for classes and methods as described in *Programming with Data*, (John M. Chambers, Springer, 1998). See this book for further details and examples.

Examples

`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)

setIs(class1, class2, test=NULL, coerce=NULL, replace=NULL, where=-1)
```

Arguments

`object`	Any R object.
`class1, class2`	The names of the classes between which `is` relations are to be defined.
`maybe`	What value to return if the relationship is conditional.
`test, coerce, replace`	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`.
`where`	Where to store the metadata defining the relationship. Default is the global environment.

Details

`setIs`:

The relationship can be conditional, if a function is supplied as the `test` argument. If a function is supplied as the `coerce` argument, this function will be applied to any `class1` object in order to turn it into a `class2` object.

Extension may imply that a `class1` object contains a `class2` object. The default sense of containing is that all the slots of the simpler class are found in the more elaborate one. If the `replace` argument is supplied as an S replacement function, this function will be used to implement `as(obj, class2) <- value`.

Author(s)

John Chambers

References

The web page http://www.omegahat.org/RSMethods/index.html is the primary documentation.

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.

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)))
## Define a multi-curve to extend a single curve ONLY
## if the y data is one variable.
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")), curve = as.numeric(slot(obj, "curve")))})
```

`languageEl` *Elements of Language Objects*

Description

Internal routines to support some operations on language objects.

Usage

```
languageEl(object, which)

isGrammarSymbol(symbol)
```

Summary of Functions

`languageEl`**:** extract an element of a language object, consistently for different kinds of objects.

The 1st., etc. elements of a function are the corresponding formal arguments, with the default expression if any as value.

The first element of a call is the name or the function object being called.

The 2nd, 3rd, etc. elements are the 1st, 2nd, etc. arguments expressions. Note that the form of the extracted name is different for R and S-Plus. When the name (the first element) of a call is replaced, the languageEl replacement function coerces a character string to the internal form for each system.

The 1st, 2nd, 3rd elements of an `if` expression are the test, first, and second branch.

The 1st element of a `for` object is the name (symbol) being used in the loop, the second is the expression for the range of the loop, the third is the body of the loop.

The first element of a `while` object is the loop test, and the second the body of the loop.

isGrammarSymbol: Checks whether the symbol is part of the grammar. Don't use this function directly.

`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; **Class Inheritance and Method Selection** provides more details on how class definitions determine which methods are used.

The section **Changes with the Methods Package** outlines possible effects on other computations when running with package `methods`.

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. In R, the "collection" is an object of class `"MethodsList"`, which contains a named list of methods (the `methods` slot), and the name of one of the formal arguments to the function (the `argument` slot). The names of the methods are the names of classes, and the corresponding element defines the method or methods to be used if the corresponding argument has that class. For example, 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 selection of methods depending on the remaining formal arguments, in this example, `y`. The same selection process takes place, recursively, using the new methods list. Eventually, the selection returns either a function or `NULL`, meaning that no method matched the actual arguments.

Each method selected 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"`

The actual selection is done recursively, but you can see the methods arranged by signature by calling the function `showMethods`, and objects with the methods arranged this way (in two different forms) are returned by the functions `listFromMlist` and `linearizeMlist`.

In an R session, each generic function has a single methods list object defining all the currently available methods. The session methods list object is created the first time the function is called by merging all the relevant method definitions currently visible. Whenever something happens that might change the definitions (such as attaching or detaching a package with methods for this function, or explicitly defining or removing methods), the merged methods list object is removed. The next call to the function will recompute the merged definitions.

When methods list are merged, they can come from two sources:

1. Methods list objects for the same function on one or more currently attached databases. These are merged so that methods in a database earlier in the search list override methods for the same function later in the search list. A method overrides only another method for the same signature. See the comments on class `"ANY"` in the section on **Inheritance**.
2. Methods list objects corresponding to the generic function itself and to the group generic functions, if any, for this function. Any generic function can be defined to belong to a group generic. The methods for the group generic are available as methods for this function. The group generic can itself be defined as belong to a group; as a result there is a list of group generic functions. A method defined for a function and a particular signature overrides a method for the same signature for that function's group generic

Merging is done first over databases for a particular function, and then over the generic and its group generics.

The result is a single methods list object that contains all the methods *directly* defined for this function. As calls to the function occur, this information may be supplemented by *inherited* methods, which we consider next.

Class Inheritance and Method Selection

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 extends. (These are sometimes called the *superclasses* of the new class.) These classes themselves may extend other classes. Putting all this information together produces the full list of superclasses for this class. (You can see this list for any class `"A"` from the expression `extends("A")`.) In addition, any class implicitly extends class `"ANY"`.

A method will be selected by inheritance if we can find a method in the methods list for a signature corresponding to any combination of superclasses for each of the relevant arguments. The search for such a method is performed by the function `MethodsListSelect`, working as follows.

For the first formal argument of the function, a list of classes is made up from the class itself, all its superclasses, and class `"ANY"`. For each of these, the selection computation looks for an element of the methods with the corresponding name. Each time it finds one, it then calls the selection process recursively if necessary to select a method directly or by inheritance for the remaining arguments.

Each one of these recursive calls can fail or it can return a function (the method). As long as the calls fail, the selection process moves on to the next superclass and tries again. The last step corresponds to class `"ANY"`, the default method defined at this level.

The effect of this definition of the selection process is to order all possible inherited methods, first by the superclasses for the first argument, then within this by the superclasses for the second argument, and so on. Superclasses are ordered by how direct they are: first, the direct superclasses, then the superclasses of these classes.

Changes with the Methods Package

The methods package is designed to leave other computations in R unchanged. There are, however, a few areas where the default functions and behavior are overridden when running with the methods package attached. This section outlines those known to have some possible effect.

`class`: The methods package enforces the notion that every object has a class; in particular, `class(x)` is never `NULL`, as it would be for basic vectors, for example, when not using methods.

In addition, when assigning a class, the value is required to be a single string. (However, objects can have multiple class names if these were generated by old-style class computations. The methods package does not hide the "extra" class names.)

Computations using the notion of `NULL` class attributes or of class attributes with multiple class names are not really compatible with the ideas in the methods package. Formal classes and class inheritance are designed to give more flexible and reliable implementations of similar ideas.

If you do have to mix the two approaches, any operations that use class attributes in the old sense should be written in terms of `attr(x, "class")`, not `class(x)`. In particular, test for no class having been assigned with `is.null(attr(x, "class"))`.

Printing To provide appropriate printing automatically for objects with formal class definitions, the methods package overrides `print.default`, to look for methods for the generic function `show`, and to use a default method for objects with formal class definitions.

The revised version of `print.default` is intended to produce identical printing to the original version for any object that does *not* have a formally defined class, including honoring old-style print methods. So far, no exceptions are known.

`plot` A version of the `plot` function is included in the current methods package, differing from the one in the base package in that it has a `y` argument (necessary if methods for plot are to be defined for the `y` data; see the examples for setMethod). This version will move into base as soon as it is tested.

Author(s)

John Chambers

References

The web page `http://www.omegahat.org/RSMethods/index.html` is the primary documentation.

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

`setGeneric`, `setClass`

`MethodsList` *MethodsList Objects*

Description

These functions create and manipulate `MethodsList` objects, the objects used in R to store methods for dispatch. You should not call any of these functions from code that you want to port to S-Plus. Instead, use the functions described in the references.

Usage

```
MethodsList(.ArgName, ...)

makeMethodsList(object, level=1)

SignatureMethod(names, signature, definition)

insertMethod(mlist, signature, args, def, whichMethods, fromClass, envir)

matchArg(object, thisClass, mlist, ev)

matchArgClass(Class, classes, methods)
```

```
showMlist(mlist, includeDefs = TRUE, inherited = TRUE,
   classes, useArgNames, printTo = stdout() )

print.MethodsList(x, ...)

listFromMlist(mlist, prefix = list())

linearizeMlist(mlist, inherited = TRUE)

finalDefaultMethod(mlist, fname = "")

mergeMethods(m1, m2)
```

Details

Note that `MethodsList` objects represent methods only in the R implementation. You can use them to find or manipulate information about methods, but avoid doing so if you want your code to port to S-Plus.

Details

`MethodsList:` Create a MethodsList object out of the arguments.

Conceptually, this object is a named collection of methods to be dispatched when the (first) argument in a function call matches the class corresponding to one of the names. A final, unnamed element (i.e., with name `""`) corresponds to the default method.

The elements can be either a function, or another MethodsList. In the second case, this list implies dispatching on the second argument to the function using that list, given a selection of this element on the first argument. Thus, method dispatching on an arbitrary number of arguments is defined.

MethodsList objects are used primarily to dispatch OOP-style methods and, in R, to emulate S4-style methods.

`SignatureMethod:` construct a MethodsList object containing (only) this method, corresponding to the signature; i.e., such that signature[[1]] is the match for the first argument, signature[[2]] for the second argument, and so on. The string "missing" means a match for a missing argument, and "ANY" means use this as the default setting at this level.

The first argument is the argument names to be used for dispatch corresponding to the signatures.

`insertMethod:` insert the definition `def` into the MethodsList object, `mlist`, corresponding to the signature. By default, insert it in the slot `which="methods"`, but `cacheMethod` inserts it into the `which="allMethods"` slot (used for dispatch but not saved).

`matchArg, matchArgClass`: Utility functions to match the object or the class to the elements of a methods list. Used in finding inherited methods, and not meant to be called directly.

`showMlist`: Prints the contents of the MethodsList. If `includeDefs` the signatures and the corresonding definitions will be printed; otherwise, only the signatures.

The function calls itself recursively. `prev` is the previously selected classes.

`listFromMlistForPrint`: Undo the recursive nature of the methods list, making a list of function defintions, with the names of the list being the corresponding signatures (designed for printing; for looping over the methods, use `listFromMlist` instead).

The function calls itself recursively: `prev` is the previously selected classes.

`finalDefaultMethod` The true default method for the methods list object `mlist` (the method that matches class `"ANY"` for as many arguments as are used in methods matching for this generic function).

`mergeMethods` Merges the methods in the second MethodsList object into the first, and returns the merged result. Called from `getAllMethods`.

Author(s)

John Chambers

References

The web page http://www.omegahat.org/RSMethods/index.html is the primary documentation.

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.

Examples

`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.

Creating Objects

```
 new('MethodsList', methods = ...., # Object of class list argument = ....,
# Object of class name allMethods = ...., # Object of class list fromClass
= ...., # Object of class character )
```

Slots

`methods`: Object of class "list"

`argument`: Object of class "name"

`allMethods`: Object of class "list"

`fromClass`: Object of class "character"

Extends

Class "OptionalMethods" directly.

MethodSupport — *Additional (Support) Functions for Methods*

Description

These are support routines for computations on formal methods.

Usage

```
getMethodsForDispatch(f)

cacheMethod(f, sig, def, args = names(sig))

hasMethod(f, signature=character())

resetGeneric(f)
```

Summary of Functions

`hasMethod`: returns `TRUE` if `f` is the name of a generic function with an (explicit) method for this signature.

`resetGeneric`: reset the currently defined methods for this generic by removing the corresponding definition from the methods metadata (to be remerged when the function is next called). Returns `TRUE` or `FALSE` according to whether information for the function was found in the metadata.

You *must* call this function when you change relevant inheritance information during a session, to guarantee that the new information is used if this generic has already been called.

`cacheMethod`: Store the definition for this function and signature in the method metadata for the function. Used to store extensions of coerce methods found through inheritance.

No persistent effect, since the method metadata is session-scope only.

`getMethodsForDispatch`: Get the current methods list object representing the methods for function f, merged from the various libraries and with any additional caching information stored in the `allMethods` slot.

If methods have not yet been merged, calling `getMethodsForDispatch` will cause the merge to take place.

methodUtilities — *Utility Functions for Methods and S-Plus Compatibility*

Description

These are utilities, currently in the methods package, that either provide some functionality needed by the package (e.g., element matching by name), or add compatibility with S-Plus, or both.

Usage

```
functionBody(fun=sys.function(sys.parent()))

allNames(x)

getFunction(name, generic=TRUE, mustFind=TRUE, where)

el(object, where)

elNamed(x, name, mustFind=FALSE)

formalArgs(def)

Quote()

message(...)

showDefault(object, printTo = stdout(), oldMethods = TRUE)
```

Summary of Functions

`allNames`: the character vector of names (unlike names(), never returns NULL)

`getFunction`: find the object as a function.

`elNamed`: get the element of the vector corresponding to name. Unlike the `[`, `[[`, and `$` operators, this function requires `name` to match the element name exactly (no partial matching).

`formalArgs`: Returns the names of the formal arguments of this function.

`existsFunction`: Is there a function of this name? If `generic==FALSE`, generic functions are not counted.

`findFunction`: return all the indices of the search list on which a function definition for `name` exists.

If `generic` is `FALSE`, ignore generic functions.

`message`: Output all the arguments, pasted together with no intervening spaces.

`showDefault`: Utility, used to enable `show` methods to be called by the automatic printing (via `print.default`).

`new` *Generate an Object from a Class*

Description

Given the 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, ..., .Force=FALSE)
```

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.
`.Force`	Normally, an attempt to apply `new` to a virtual or an undefined class generates an error. If `.Force` is `TRUE`, an "empty" object is returned instead. Users are very unlikely to need this argument.

Details

The function begins by copying the prototype object from the class definition. Then information is inserted according to the ... arguments, if any, first from the superclasses (the unnamed arguments) then from the named slots. Thus, explicit slots override inherited information for the same slot, regardless of the order in which the arguments appear.

Note that the basic vector classes, `"numeric"`, etc. are implicitly defined, so one can use `new` for these classes.

Author(s)

John Chambers

References

The web page http://www.omegahat.org/RSMethods/index.html is the primary documentation.

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)
```

`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 QA facilities at present

Usage

```
promptClass(clName, filename, type = "class", where)
```

Arguments

`clName`	character string naming the class to be documented
`filename`	Usually, the filename on which the documentation shell should be written. By default it is the topic name for the class documentation, followed by ".Rd". See the example below. The argument can also be any writable connection.
`type`	The documentation type to be declared in the output file.
`where`	where to look for the definition of the class and of methods that use it. By default, the function searches for the class definition on the currently attached databases.

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 on the same database 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, the documentation shell is written to a file, which 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.

Value

The name of the file to which the shell is written (the value is `invisible`). A message is also printed notifying the user about the file.

Author(s)

VJ Carey, ⟨stvjc@channing.harvard.edu⟩ and John Chambers

References

The web page http://www.omegahat.org/RSMethods/index.html is the primary documentation.

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

```
> promptClass("track")
A shell of class documentation has been written to the
file "track-class.Rd".
```

---

promptMethods *Generate a Shell for Documentation of Formal Methods*

Description

Generates a shell of documentation for the methods of a generic function, with

Usage

```
promptMethods(f, filename, addTo = FALSE, inherited=FALSE, type = "methods")
```

Arguments

`f`	The name of the generic function whose methods are to be documented.
`filename`	Optional file on which to write the documentation shell. If `FALSE`, the text is returned, presumably to be used in the documentation of the generic function itself (see prompt). By default, this is the same as the coded topic name for these methods (currently `"NAME-methods.Rd"`, where `NAME` is the function name, argument `f`).
`addTo`	Optionally, text to add to the shell. This would typically be part of the contents of the documentation file for the generic function, if you want the generic and the methods documented together.
`inherited`	Should inherited methods be included? Default `FALSE`.
`type`	The documentation type to be declared in the output file.

Value

If `filename` is `FALSE`, the text generated; otherwise, the name of the file written.

Author(s)

John Chambers

References

The web page http://www.omegahat.org/RSMethods/index.html is the primary documentation.

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 and promptClass

---

recover *Browsing on 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`.

WARNING: Do not use the special `Q` command to go directly from the browser to the prompt level of the evaluator. This 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 the usage section. 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.

References

Programming with Data, (John M. Chambers, Springer, 1998). See the compatibility note above, however.

See Also

See `dump.frames` and `debugger` for the more traditional approach to error recovery.

`representation` *Describe A Class Representation*

Description

Constructs a description of a class (without verifying the validity of the description). A convenience function in calls to `setClass`.

Usage

```
representation(...)
```

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.

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.

Value

The value is just the list of arguments, after these have been checked for validity.

Author(s)

John Chambers

References

The web page `http://www.omegahat.org/RSMethods/index.html` is the primary documentation.

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

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")
```

`RMethodUtils` *RMethodUtils*

Description

These are utility functions to support the definition and use of formal methods. Most of them will not normally be called directly by the user.

`makeGeneric`: Makes a generic function object corresponding to the given function name. and optional definition.

`makeStandardGeneric`: a utility function that makes a valid function calling standardGeneric for name f Works (more or less) even if the actual definition, fdef, is not a proper function, that is, it's a primitive or internal

`conformMethod`: If the formal arguments, `mnames`, are not identical to the formal arguments to the function, `fnames`, `conformMethod` determines whether the signature and the two sets of arguments conform, and returns the signature, possibly extended.

The method assignment conforms if either method and function have identical formal argument lists. It can also conform if the method omits some of the formal arguments of the function but: (1) the non-omitted arguments are a subset of the function arguments, appearing in the same order; (2) there are no arguments to the method that are not arguments to the function; and (3) the omitted formal arguments do not appear as explicit classes in the signature.

`defaultDumpName`: the default name to be used for dumping a method.

`getAllMethods`: a generic function (with methods) representing the merge of all the versions of `f` on the specified packages (anything on the current search path by default).

If the generic `f` has a group generic, methods for this group generic (and further generations of group generics, if any) are also merged.

The merging rule is as follows: each generic is merged across packages, and the group generics are then merged, finally adding the directly defined methods of `f`.

The effect of the merging rule is that any method directly defined for `f` on any included package overrides a method for the same signature defined for the group generic; similarly for the group generic and its group, if any, etc.

For `f` or for a specific group generic, methods override in the order of the packages being searched. A method for a particular signature on a particular package overrides any methods for the same signature on packages later on in the list of packages being searched.

The slot "allMethods" of the merged methods list is set to a copy of the methods slot; this is the slot where inherited methods are stored.

`doPrimitiveMethod`: do a primitive call to builtin function `name` the definition and call provided, and carried out in the environment `ev`.

A call to `doPrimitiveMethod` is used when the actual method is a .Primitive. (because primitives don't behave correctly as ordinary functions, not having either formal arguments nor a function body).

`getGeneric`: return the definition of the function named f as a generic.

If there is no definition in the current search list, throws an error or returns NULL according to the value of mustFind.

Primitive functions are dealt with specially, since there is never a formal generic definition for them. The value returned is the formal definition used for assigning methods to this primitive. Not all primitives can have methods; if this one can't, then `getGeneric` returns `NULL` or throws an error.

`getGroup`: return the groups to which this generic belongs.

If `recursive=TRUE`, also all the group(s) of these groups.

`getGroupMembers, setGroupMembers`: Get or set the known members of the group generic function `f`.

`matchSignature` Matches the signature object (a partially or completely named subset of the arguments of `fun`), and return a vector of all the classes in the order specified by `names`. The classes not specified by 'signature' will be `"ANY"` in the value.

The formal arguments of `fun` must agree with `names` (usually the formal arguments of the generic function) as well, and `matchSignature` checks this.

`getMethodsMetaData, assignMethodsMetaData, mlistMetaName`: utilities to manage methods list objects in a particular database. Not to be called directly.

`getFromMethodMetaData, assignToMethodMetaData, removeFromMethodMetaData` Functions to manage the session metadata for methods. Don't call these directly.

`MethodAddCoerce` Possibly modify one or more methods to explicitly coerce this argument to `methodClass`, the class for which the method is explicitly defined. Only modifies the method if an explicit coerce is required to coerce from `thisClass` to `methodClass`.

`is.primitive` Is this object a primitive function (either a builtin or special)?

`getFromMethodMetaData, assignToMethodMetaData, removeFromMethodMetaData` Functions to manage the session metadata for methods. Don't call these directly.

`is.primitive` Is this object a primitive function (either a builtin or special)?

`removeMethodsObject`: remove the metadata object containing methods for `f`.

`findUnique`: Find the first position on the search list containing object `what`; if more than one is found, a warning message is generated, using `message` to identify what was being searched for. If `doFind` is supplied, it's the version of `find` used to do the search (e.g., `findFunction`.

`cacheMetaData, cacheGenericsMetaData, setPrimitiveMethods`: Utilities for ensuring that the session-scope information about class and method definitions is up to date. Should normally be called automatically whenever needed (for example, when a method or class definition changes, or when a database is attached or detached.

The `environment` must be one of the environments on the current search list; note in particular that even on detaching (`attach=FALSE`), the environment will normally still be on the search list.

The `setPrimitiveMethods` function resets the caching information for a particular primitive function. Don't call it directly.

`printNoClass,print.default` : `printNoClass` is equivalent to the version of `print.default` in the base package. The methods package overrides the latter function to provide meaningful printing for formally defined classes, and `printNoClass` is used to get the original default printing.

`.saveImage`: Flag, used in dynamically initializing the methods package from `.First.lib`

Usage

```
makeGeneric(f, fdef, keepMethods=TRUE, useAsDefault=NA, group=character(), valueClass=charac

makeStandardGeneric(f, fdef)

generic.skeleton(name, fdef, fdefault)

defaultDumpName(generic, signature)

getAllMethods(f, libs=search())

setAllMethodsSlot(mlist)

doPrimitiveMethod(name, def, call=sys.call(-1), ev=sys.frame(-2))

conformMethod(signature, mnames, fnames)

getGeneric(f, mustFind=FALSE)

getGroup(fdef, recursive=FALSE)

getGroupMembers(f, fdef = getGeneric(f))
setGroupMembers(f, members, fdef = getGeneric(f))

matchSignature(names, signature, fun)

## manage method metadata
getFromMethodMetaData(name)

assignToMethodMetaData(name, value)

removeFromMethodMetaData(name)

removeMethodsObject(f, where)

findUnique(what, doFind, message)

MethodAddCoerce(method, argName, thisClass, methodClass)

is.primitive(fdef)

copyEnvironment(object, exceptions)

cacheMetaData(envir, attach = TRUE)
```

```
cacheGenericsMetaData(generics, attach = TRUE, envir)

setPrimitiveMethods(f, fdef, code, generic, mlist)

printNoClass(x, digits, quote, na.print, print.gap, right, ...)

print.default(x, ...)
```

`Session` *Session Data and Debugging Tools*

Description

The function `sessionData` provides a work-around in R for the absence of a session database (what you get in S with the argument `where=0` to functions such as `get` and `assign`).

The functions `traceOn`, `traceOff` and `browseAll` are debugging tools with roughly similar functionality to `trace`, `untrace` and `browser` as described in the reference. Calling `traceOn` creates a new version of a function with arbitrary code inserted at the beginning of the call and/or on exit. Calling `traceOff` undoes the change. The function `browseAll` is a useful interactive call to insert for tracing. It behaves basically like `debugger`.

Note: use `traceOn` only for testing functions in a library, *not* for functions defined in the global environment. Because the session data is in position 2 on the search list, the temporary versions stored there will not override functions in the global environment.

A more satisfactory version of these facilities is likely to be provided in the future. Use the current versions but don't write code that depends on them.

Usage

```
sessionData()

traceOn(what, tracer=browseAll, exit=NULL)

traceOff(what)

browseAll()
```

Arguments

`what`	The object to be traced or untraced.
`tracer, exit`	The code to use at the entrance or exit trace. Either a function (which will be called with no arguments) or an expression (typically constructed from `link{quote}` or `substitute`) to be evaluated.

Details

`sessionData`: return the index of the session data in the search list, attaching it if it is not attached

`traceOn`: initialize tracing on calls to function `what`. The function or expression `tracer` is called on entry, and the function or expression `exit` on exit.

`traceOff`: turn off tracing of this function

`browseAll`: browse the current stack of function calls.

Uses the function `debugger` to set up browser calls on the frames. On exit from that function, computation continues after the call to browseAll. Computations done in the frames will have no effect.

Author(s)

John Chambers

References

See *Programming with Data* (John M. Chambers, Springer, 1998) for the equivalent functions.

`setClass` *Create a Class Definition*

Description

Create a formally defined class with specified slots and/or relationships to other classes. Also functions to remove a class definition, to test whether a class has been defined, to test whether an object is a class definition, and to reset the internal definition of a class.

Usage

```
setClass(Class, representation, prototype,
         contains=character(), validity, access, where=1, version=FALSE)

removeClass(Class, where=-1)

isClass(Class, formal=TRUE)
isClassDef(object)

getClasses(where)

resetClass(Class)
```

Arguments

`Class`	character string name for the class
`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` What environment to use to store or remove the definition (as metadata). By default, uses the global environment for `setClass` and searches for a definition to remove, for `removeClass`.

`validity, access, version`
Control arguments included for compatibility with the S-Plus API, but not currently used.

`x` an arbitrary object.

`formal` Should a formal definition be required?

`object` any R object.

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. Calling this *always* resets the version of the class cached for the session. If `where=0`, that's all it does. Otherwise, it removes the version from the specified environment or database (from the global environment by default).

`isClass:` Is this a the name of a formally defined class? (Argument `formal` is for compatibility and is ignored.)

`isClassDef:` Is this object a class definition (it will be, for example, if it is the value of a call to `getClass`, the complete definition of a class with its extensions, or to `getClassDef`, the local definition of the class).

`getClasses:` The names of all the classes formally defined on 'where'. If called with no argument, all the classes currently known in the session (which does not include classes that may be defined on one of the attached libraries, but have not yet been used in the session).

`unclass:` Returns the object containing the values of all the slots in this object's class definition (specifically, ithe returned object has attributes corresponding to each slot), in the case that the object's class is formally defined with slots.

For classes that extend a single other class (e.g., a basic class such as `"numeric"`) the result is an object of that class.

`resetClass:` Reset the internal definition of a class. The effect is that the next time the definition of this class is needed, it will be recomputed from the information on the attached databases.

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).

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 basic classes, perhaps with the intension of modifying R's standard behavior for that class. Perfectly legal and sometimes quite helpful, but you may need to be more careful in this case: your new class will inherit much of the behavior of the basic (informally defined) class, and the results can be surprising. Just proceed with caution and plenty of testing.

As an example, the class `"matrix"` is included in the pre-defined classes, to behave essentially as matrices do without formal class definitions. Suppose we don't like all of this; in particular, we want the default matrix to have 0 rows and columns (not 1 by 1 as it is now).

```
setClass("myMatrix", "matrix", prototype = matrix(0,0,0))
```

The arguments above illustrate two short-cuts relevant to such examples. We abbreviated the `representation` argument to the single superclass, because the new class doesn't add anything to the representation of class `"matrix"`. Also, we provided an object from the superclass as the prototype, not a list of slots.

Author(s)

John Chambers

References

The web page http://www.omegahat.org/RSMethods/index.html is the primary documentation.

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`, `setSClass`

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")))})
```

`setGeneric` *Define a New Generic Function*

Description

Create a new generic function of the given name, for which formal methods can then be defined.

Usage

```
setGeneric(name, def, group=NULL, valueClass=NULL, where=1, doAssign)

setGroupGeneric(name, def, group=NULL, valueClass=NULL, knownMembers, where=1)
```

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`. See also argument `useAsDefault`.
`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 string defining the class to asserted to be given to the value returned by this generic function. By default, the generic function can return any object. (At the moment, the assertion supplied in this argument is not enforced or checked.)
`where`	Where to store the resulting initial methods definition, and possibly the generic function; by default, stored into the global environment.

`doAssign` Should a new generic version of the function be assigned? The default action depends on whether a generic version currently exists. If not, one will be assigned to `where`, except that primitive functions in the base package are never turned into explicit generic functions (they are dispatched from the internal code in the evaluator).

You can supply `doAssign` as `TRUE` to force assigning a generic function even if one already exists, or as `FALSE` to prevent assigning the generic in any case (for example, because you know one exists on another package). But the rule of not assigning generics for primitives cannot be overriden by setting `doAssign` to `TRUE`.

`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 prepartion for defining some methods for that function.

The simplest and most common situation is that `name` is already a function, either an ordinary non-generic function or else a generic function in another environment. In the second case, essentially all you will be doing is to make a copy of the existing generic, without its methods, in the current environment. *Only* the `name` argument will be used. The new generic *must* agree with the current one in essentially everything except for the methods defined.

The other simple and common situation is that you already have a non-generic function (probably in the same place you plan to save the generic), and now want to turn this function into a generic. In this case, too, you will most often supply only `name`. The existing function becomes the default method, and the special `group` and `valueClass` properties remain unspecified.

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`.

Author(s)

John Chambers

References

The web page `http://www.omegahat.org/RSMethods/index.html` is the primary documentation.

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 for a discussion of other functions to specify and manipulate the methods of generic functions.

`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=1, valueClass)

removeMethod(f, signature, where)
```

Arguments

`f`	The character-string name of the generic 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; by default, the current global environment. 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.)

Details

R methods for a particular generic function are stored in an object of class MethodsList, which in turn is stored with the definition of the generic function. The effect of calling `setMethod` is to store `definition` in a `MethodsList` object in a definition of the generic function on database `where`. If no such function exists (on that database) one will be created, by copying the generic function from where it is found in the current search list. Finally, 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.

Methods are stored in a hierarchical structure, by formal arguments to `f`: see MethodsList for details. 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.

While `f` can correspond to methods defined on several packages or databases, 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. In particular, all the versions of `f` and all the methods must correspond to the same formal arguments (including, in the present definition, the same default expressions for the arguments). For compatibility with S-Plus, the current implementation enforces this partly with a warning and a reconstruction of a method that fails to match, but don't count on this for the future: Make the formal arguments of `definition` match those of the generic..

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.

Author(s)

John Chambers

References

The web page http://www.omegahat.org/RSMethods/index.html is the primary documentation.

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, `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.
setMethod("plot", c("trackCurve", "missing"),
function(x, y, ...) {
  plot(as(x, "track"))
  if(length(slot(x, "smooth") > 0))
    lines(slot(x, "x"), slot(x, "smooth"))
  }
)
## the plot of tc1 alone has an added curve; other uses of tc1
## are treated as if it were a "track" object.
plot(tc1)
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>")
```

`setSClass` *Low-level Class definition*

Description

This is the lower-level version of `setClass`. Unless you really know why you are calling it, use `setClass` instead.

Usage

```
setSClass(name, properties=list(), extends=character(),
   prototype=NULL, generatorFunction, where=1,
   subclasses=character(), virtual=NA, validity, access)
```

Arguments

`name`	character string name for the class
`properties`	the `representation` argument to `setClass`
`extends`	what classes does this class extend
`prototype`	an object (usually a list) providing the default data for the slots specified in the representation.
`generatorFunction`	an optional function to use in generating new objects from the class
`where`	where to store the resulting definition (as metadata). By default, uses the global environment.
`subclasses`	Classes that extend *this* class.
`virtual`	Is this known to be a virtual class?
`validity, access`	Optional arguments for validity method and access restrictions. At the time of writing, these are included for compatibility and are not used. In the future, however, the validity method is likely to be relevant. See *Programming with Data* for its interpretation.

Author(s)

John Chambers

References

The web page `http://www.omegahat.org/RSMethods/index.html` is the primary documentation.

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

`setClass`

`show` *Show an Object*

Description

Display the object, by printing, plotting or whatever suits its class.

The function `show` exists to be specialized by methods; the default method calls `showDefault`.

With library `methods` attached, methods for `show` will usually be invoked for automatic printing (see the details).

The function `showDefault` allows redirection of output and optional use of old-style print methods, but normally will not be called directly.

Usage

```
show(object)

showDefault(object, printTo = stdout(), oldMethods = TRUE)
```

Arguments

`object`	Any R object
`printTo`	Either a file or connection, or else `FALSE`. In the latter case, the lines of text that would have been printed are returned as the value of the call (in a character vector with one element per line of output).
`oldMethods`	Should old-style print methods be used for this object? `TRUE` by default if called directly, but `FALSE` when called from the methods package for automatic printing (to avoid potential recursion; see the details below).

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`.

For `showDefault`, if `printTo` is `FALSE`, the value is a character vector containing the lines that would otherwise have been printed.

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

  [,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

## and also for tc1, an object of a class that extends "track"
tc1

  [,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
```

`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=-1, classes=NULL, includeDefs=FALSE,
    inherited=TRUE, printTo = stdout())
```

Arguments

`f`	One or more function names. If omitted, all the functions on database `where` will be examined.
`where`	If `where` is supplied, the methods definition from that database will be used; otherwise, the current definition is used (which will include inherited methods that have arisen so far in the session).
`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.
`printTo`	The connection on which the printed information will be written. 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 output style is different from S-Plus in that it does not show the database from which the definition comes, but can optionally include the method definitions.

Value

If `printTo` is `FALSE`, the character vector that would have been printed is returned; otherwise the value is the connection or filename.

Author(s)

John Chambers

References

The web page http://www.omegahat.org/RSMethods/index.html is the primary documentation.

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

`setMethod`, and `GenericFunctions` for other tools involving methods; `show` for

Examples

```
## assuming the methods for plot
## are set up as in the documentation for setMethod,
## print (without definitions) the methods that involve
## class "track"
showMethods("plot", classes = "track")

Function "plot":
x = ANY, y = track
x = track, y = missing
x = track, y = ANY
```

`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) <- 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`	
`x`	Either the name of a class or an object from that class. Print `getClass(class)` to see the full description of the slots.

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.

In the case of the `slot` function, the slot 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 diretly 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.

Author(s)

John Chambers

References

Chambers, J. M. (1998) *Programming with Data*, Springer.

The web page http://www.omegahat.org/RSMethods/index.html is the primary documentation.

See Also

`@`, `Classes`, `Methods`, `getClass`

Examples

```
slot(myTrack, "x")
slot(myTrack, "y") <- log(slot(myTrack, "x"))

slotNames("track")
```

`substituteDirect` *SubstituteDirect*

Description

Substitute for the variables named in the second argument the corresponding objects, substituting into `object`. The argument `frame` is a named list; if omitted, the environment of the caller is used.

This function differs from the ordinary `substitute` in that it treats its first argument in the standard S way, by evaluating it. In contrast, `substitute` does not evaluate its first argument.

The goal is to replace this with an `eval=` argument to `substitute`.

Usage

```
substituteDirect(object, frame, cleanFunction=T)
```

Examples

`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`).

The functions `getValidity` and `setValidity` get and set the validity method of a class. This method is a function of one object that returns `TRUE` or a description of the non-validity.

Usage

```
validObject(object, test)

getValidity(ClassDef)
setValidity(ClassDef, method)
```

Arguments

`object`	Any object, but not much will happen unless the object's class has a formal definition.
`test`	If `test` is `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.
`ClassDef`	The name 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. Note that validity methods do not have to check validity of any slots or 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": first the validity of the object's slots, if any, is tested. Then 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.

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.

Author(s)

John Chambers

References

The web page `http://www.omegahat.org/RSMethods/index.html` is the primary documentation.

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

`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
t1@x <- 1:20
## This should generate an error
try(validObject(t1))
```

Chapter 6

The `modreg` package

`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.

Author(s)

B. D. Ripley

Examples

```
data(cars)
attach(cars)
plot(speed, dist)
lines(ksmooth(speed, dist, "normal", bandwidth=2), col=2)
lines(ksmooth(speed, dist, "normal", bandwidth=5), col=3)
lines(ksmooth(speed, dist, "normal", bandwidth=10), col=4)
detach()
```

---

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 response and one or more numeric predictors (best specified via an interaction, but can also be specified additively).
`data`	an optional data frame within which to look first for the response, predictors and weights.
`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 to stop.
`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.

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

```
data(cars)
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'.

Author(s)

B.D. Ripley

See Also

`loess`

`modreg-internal` *Internal modreg functions*

Description

Internal modreg functions.

Usage

```
predLoess(y, x, newx, s, weights, robust, span, degree, normalize,
    parametric, drop.square, surface, cell, family, kd, divisor,
    se = FALSE)
simpleLoess(y, x, weights, span = 0.75, degree = 2, parametric = FALSE,
    drop.square = FALSE, normalize = TRUE, statistics = "approximate",
    surface = "interpolate", cell = 0.2, iterations = 1,
    trace.hat = "exact")
pointwise(results, coverage)
```

Details

These are not to be called by the user.

`plot.ppr` *Plot Ridge Functions for Projection Pursuit Regression Fit*

Description

Plot ridge functions for projection pursuit regression fit.

Usage

```
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

```
data(rock)
attach(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)")
```

---

ppr *Projection Pursuit Regression*

Description

Fit a projection pursuit regression model.

Usage

```
ppr(formula, data = sys.parent(), weights,
    subset, na.action, contrasts = NULL,
    ww = rep(1,q), nterms, max.terms = nterms, optlevel = 2,
    sm.method = c("supsmu", "spline", "gcvspline"),
    bass = 0, span = 0, df = 5, gcvpen = 1)

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 response variables and the explanatory variables.
x	matrix of explanatory variables. Rows represent observations, and columns represent variables. Missing values are not accepted.
y	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`	data 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 `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`	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.

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 `nterm` 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.

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. (1999) *Modern Applied Statistics with S-PLUS*. Springer.

See Also

`plot.ppr`, `supsmu`, `smooth.spline`

Examples

```
# Note: your numerical values may differ
data(rock)
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)")

detach()
```

`predict.loess` *Predict Loess Curve or Surface*

Description

Predictions from a `loess` fit, optionally with standard errors.

Usage

```
predict(object, newdata = NULL, se = FALSE, ...)
```

Arguments

`object`	an object fitted by `loess`.
`newdata`	an optional data frame specifying points at which to do the predictions. 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.

Author(s)

B.D. Ripley, based on the `cloess` package of Cleveland, Grosse and Shyu.

See Also

`loess`

Examples

```
data(cars)
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.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

```
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`.

Author(s)

B.D. Ripley

See Also

`smooth.spline`

Examples

```
data(cars)
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)
```

rock *Measurements on Petroleum Rock Samples*

Description

Measurements on 48 rock samples from a petroleum reservoir.

Usage

```
data(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.

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, span = 2/3, degree = 1,
    family = c("symmetric", "gaussian"),
    xlab = deparse(substitute(x)), ylab = deparse(substitute(y)),
    ylim = range(y, prediction$y), evaluation = 50, ...)
loess.smooth(x, y, span = 2/3, degree = 1,
    family = c("symmetric", "gaussian"), evaluation=50, ...)
```

Arguments

`x`	x coordinates for scatter plot.
`y`	y coordinates for scatter plot.
`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.

Value

None.

Author(s)

B.D. Ripley

See Also

`loess`

Examples

```
data(cars)
attach(cars)
scatter.smooth(speed, dist)
detach()
```

`smooth.spline` *Fit a Smoothing Spline*

Description

Fits a cubic smoothing spline to the supplied data.

Usage

```
smooth.spline(x, y, w = rep(1, length(x)), df = 5, spar = NULL,
              cv = FALSE, all.knots = FALSE, 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
`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' (`FALSE`) cross-validation.
`all.knots`	if `TRUE`, all points in `x` are uses as knots. If `FALSE`, a suitably fine grid of knots is used.
`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. **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 ten distinct values.

The computational λ used (as a function of $s = spar$) is $\lambda = r * 256^{3s-1}$ where $r = tr(X'W^2X)/tr(\Sigma)$, 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^2 is the diagonal matrix of scaled weights, `W = diag(w)/n` (i.e., the identity for default weights), 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 $L = (y-f)'W^2(y-f) + \lambda c'\Sigma c$, and hence c is the solution of the (ridge regression) $(X'W^2X + \lambda\Sigma)c = X'W^2y$.

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 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 be come 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.
`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.
`lev`	leverages, the diagonal values of the smoother matrix.
`cv.crit`	(generalized) cross-validation score.
`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`.
`call`	the matched call.

Author(s)

B.D. Ripley and Martin Maechler (spar/lambda, etc).

See Also

`predict.smooth.spline`

Examples

```
data(cars)
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()

##-- 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)
```

`supsmu` *Friedman's SuperSmoother*

Description

Smooth the (x, y) values by Friedman's "super smoother".

Usage

```
supsmu(x, y, wt = rep(1, length(y)), span = "cv", periodic = FALSE,
       bass = 0)
```

Arguments

`x`	x values for smoothing
`y`	y values for smoothing
`wt`	case weights
`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.

Author(s)

B. D. Ripley

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

```
data(cars)
attach(cars)
plot(speed, dist)
lines(supsmu(speed, dist))
lines(supsmu(speed, dist, bass=7), lty=2)
detach()
```

Chapter 7

The `mva` package

`ability.cov` *Ability and Intelligence Tests*

Description

Six tests were given to 112 individuals. The covariance matrix is given in this object.

Usage

```
data(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

```
data(ability.cov)
(ability.FA <- factanal(factors = 1, covmat=ability.cov))
update(ability.FA, factors=2)
update(ability.FA, factors=2, rotation="promax")
```

---

as.hclust *Convert Objects to Class hclust*

Description

Converts objects from other hierarchical clustering functions to class `"hclust"`.

Usage

```
as.hclust(x, ...)
as.hclust.twins(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'.

Value

An object of class `"hclust"`.

See Also

`hclust`, `diana`, `agnes`

Examples

```
x <- matrix(rnorm(30), ncol=3)
hc <- hclust(dist(x), method="complete")

library(cluster)
ag <- agnes(x, method="complete")

x11()
par(mfrow=c(1,2))
plot(hc)
mtext("hclust", side=1)
plot(as.hclust(ag))
mtext("agnes", side=1)
```

`biplot` *Biplot of Multivariate Data*

Description

Plot a biplot on the current graphics device.

Usage

```
biplot(x, ...)

biplot.default(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, ...)
```

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.
`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 get the two sets on 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, ...`	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`.

Side Effects

a plot is produced on the current graphics device.

Author(s)

B.D. Ripley

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`.

Usage

```
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

```
data(USArrests)
biplot(princomp(USArrests))
```

`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

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

```
data(LifeCycleSavings)
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)
```

`cmdscale` *Classical (Metric) Multidimensional Scaling*

Description

Classical multidimensional scaling of a data matrix.

Usage

```
cmdscale(d, k = 2, eig = 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 $\{1, 2, .., n-1\}$.
`eig`	indicates whether eigenvalues 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.

Value

If `eig = FALSE`, 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 eigenvalues computed during the scaling process.

Note

The S version of this function provides for computing an additional "fiddle" factor suggested by Torgerson. R will provide this option from version 1.5.0 on.

References

Cox, F.C. and Cox, M.A.A. (1994) *Multidimensional Scaling*. Chapman and Hall.

Mardia, K. V., J. T. Kent and J. M. Bibby (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

```
data(eurodist)
loc <- cmdscale(eurodist)
x <- loc[,1]
y <- -loc[,2]
plot(x, y, type="n", xlab="", ylab="", main="cmdscale(eurodist)")
text(x, y, names(eurodist), cex=0.8)
```

`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 of groups or the cut height.

Usage

```
cutree(tree, k=NULL, h=NULL)
```

Arguments

`tree`	a tree as produced by `hclust`
`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).

See Also

`hclust`

Examples

```
require(mva)
data(USArrests)

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"])
```

`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. Currently the code is in alpha stage and the API may change at any point in time.

Usage

```
plot(x, type=c("rectangle", "triangle"),
                center=FALSE, xlab="", ylab="", ...)
cut(x, h, ...)
```

Arguments

`x`	Object of class `"dendrogram"`.
`type`	Type of plot.
`center`	If `TRUE`, nodes are plotted centered with respect to the leaves in the branch. Default is to plot them in the middle of all direct child nodes.
`h`	Height at which the tree is cutted.
`..., xlab, ylab`	graphical parameters, or arguments for other methods.

Details

Warning: This documentation is preliminary.

The dendrogram is directly represented as a nested list, where each list 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:

`members` number of leaves in the branch

`height` Height at which the node is plotted

`midpoint` Horizontal distance of the node from the left border of the branch

`text` Text label of the node

`edgetext` Text label for the edge leading to the node

Terminal nodes (leaves of the tree) can have arbitrary value but must not be a list.

`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`.

Objects of class `"hclust"` can be converted to class `"dendrogram"` using method `as.dendrogram`.

`plotNode()` and `plotNodeLimit()` are helper functions.

Examples

```
library(mva)
data(USArrests)
hc <- hclust(dist(USArrests), "ave")
str(dend1 <- as.dendrogram(hc))
plot(dend1)
dend2 <- cut(dend1, h=70)
plot(dend2$upper)
plot(dend2$lower[[3]])
```

---

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)

print.dist(x, diag = NULL, upper = NULL, ...)
as.matrix.dist(x)
as.dist(m, diag = FALSE, upper = FALSE)
```

Arguments

`x`	A matrix or (data frame). Distances between the rows of `x` will be computed.
`method`	The distance measure to be used. This must be one of `"euclidean"`, `"maximum"`, `"manhattan"`, `"canberra"` or `"binary"`. Any unambiguous substring can be given.
`diag`	A logical value indicating whether the diagonal of the distance matrix should be printed by `print.dist`.
`upper`	A logical value indicating whether the upper triangle of the distance matrix should be printed by `print.dist`.
`m`	A matrix of distances to be converted to a `"dist"` object (only the lower triangle is used, the rest is ignored).
`...`	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.

Missing values are allowed, and are excluded from all computations involving the rows within which they occur. If some columns are excluded in calculating a Euclidean, Manhattan or Canberra 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 functions `as.matrix.dist()` and `as.dist()` can be used for conversion between objects of class `"dist"` and conventional distance matrices and vice versa.

Value

An object of class `"dist"`.

The lower triangle of the distance matrix stored by columns in a single vector. The vector has the attributes `"Size"`, `"Diag"`, `"Upper"`, `"Labels"` and `"class"` equal to `"dist"`.

References

Mardia, K. V., Kent, J. T. and Bibby, J. M. (1979) *Multivariate Analysis.* London: Academic Press.

See Also

`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))
names(d) <- LETTERS[1:5]
print(d, digits = 3)

## 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)
```

`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`	Either 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`	A data frame.
`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.) 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.
`n.obs`	The number of observations if available, or `NA`.
`call`	The matched call.
`na.action`	If relevant.

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.

Author(s)

B. D. Ripley

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)
```

`Harman23.cor` *Harman Example 2.3*

Description

A correlation matrix of eight physical measurements on 305 girls between ages seven and seventeen.

Usage

```
data(Harman23.cor)
```

Source

Harman, H. H. (1976) *Modern Factor Analysis*, Third Edition Revised, University of Chicago Press, Table 2.3.

Examples

```
data(Harman23.cor)
(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

```
data(Harman74.cor)
```

Source

Harman, H. H. (1976) *Modern Factor Analysis*, Third Edition Revised, University of Chicago Press, Table 7.4.

Examples

```
data(Harman74.cor)
(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)
```

`hclust` *Hierarchical Clustering*

Description

Performs a hierarchical cluster analysis on a set of dissimilarities.

Usage

```
hclust(d, method = "complete", members=NULL)

plot(x, labels = NULL, hang = 0.1,
            axes = TRUE, frame.plot = FALSE, ann = TRUE,
            main = "Cluster Dendrogram",
            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`.
`x`	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.

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.

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).

Author(s)

The `hclust` function is based on Fortran code contributed to STATLIB by F. Murtagh.

References

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. (1981). *Classification*. London: Chapman and Hall.

Murtagh, F. (1985). "Multidimensional Clustering Algorithms", in *COMPSTAT Lectures 4*. Wuerzburg: Physica-Verlag (for algorithmic details of algorithms used).

See Also

kmeans.

Examples

```
library(mva)
data(USArrests)
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, apply(USArrests[memb==k,,drop=FALSE], 2, mean))
}
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)
```

---

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

```
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 amde 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. If 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, or by clicking outside the graphics window.

Value

Either a list of data point index vectors or a list of return values of `FUN`.

See Also

`hclust`, `rect.hclust`

Examples

```
library(mva)
data(USArrests)
hca <- hclust(dist(USArrests))
plot(hca)
x <- identify.hclust(hca)
x

data(iris)
hci <- hclust(dist(iris[,1:4]))
plot(hci)
identify.hclust(hci, function(k) print(table(iris[k,5])))

x11()
dev.set(2)
plot(hci)
identify.hclust(hci, function(k) barplot(table(iris[k,5])), DEV.FUN=3)
```

`kmeans` *K-Means Clustering*

Description

Perform k-means clustering on a data matrix.

Usage

```
kmeans(x, centers, iter.max = 10)
```

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 cluster centers. If the first, a random set of rows in `x` are chosen as the initial centers.
`iter.max`	The maximum number of iterations allowed.

Details

The data given by `x` is clustered by the k-means algorithm. When this terminates, 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.

Value

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.

References

Hartigan, J.A. and Wong, M.A. (1979). A K-means clustering algorithm. *Applied Statistics* **28**, 100–108.

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))
cl <- kmeans(x, 2, 20)
plot(x, col = cl$cluster)
points(cl$centers, col = 1:2, pch = 8)
```

`loadings` *Print Loadings in Factor Analysis*

Description

Extract or print loadings in factor analysis (or principal components analysis).

Usage

```
loadings(x)
print.loadings(x, digits = 3, cutoff = 0.1, sort = FALSE, ...)
print.factanal(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`.

Author(s)

B. D. Ripley

See Also

`factanal`, `princomp`

`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, retx = TRUE, center = TRUE, scale. = FALSE, tol = NULL)
```

Arguments

`x`	a 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. Alternately, 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.

Details

The calculation is done by a singular value decomposition of the (centered and scaled) data matrix, not by using `eigen` on the covariance matrix. This is generally the preferred method for numerical accuracy. The `print` method for the these objects prints the results in a nice format and the `plot` method produces a scree plot.

Value

`prcomp` returns an 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 data multiplied by the `rotation` matrix) is returned.

References

Mardia, K. V., J. T. Kent, and J. M. Bibby (1979) *Multivariate Analysis*, London: Academic Press.

Venables, W. N. and B. D. Ripley (1997, 9) *Modern Applied Statistics with S-PLUS*, Springer-Verlag.

See Also

princomp, cor, cov, svd, eigen.

Examples

```
## the variances of the variables in the
## USArrests data vary by orders of magnitude, so scaling is appropriate
data(USArrests)
prcomp(USArrests)  # inappropriate
prcomp(USArrests, scale = TRUE)
plot(prcomp(USArrests))
summary(prcomp(USArrests, scale = TRUE))
```

princomp *Principal Components Analysis*

Description

princomp performs a principal components analysis on the given data matrix and returns the results as an object of class princomp.

Usage

```
princomp(x, data = NULL, subset, na.action, ...)
princomp(x, cor = FALSE, scores = TRUE, covmat = NULL,
         subset = rep(TRUE, nrow(as.matrix(x))), ...)
```

Arguments

x	a formula or matrix (or data frame) which provides the data for the principal components analysis.
data	an optional data frame containing the variables in the formula x. By default the variables are taken from environment(x).
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.
cor	a logical value indicating whether the calculation should use the correlation matrix or the covariance matrix.
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, cov.mve or cov.mcd. 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.

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 the 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.

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.
`call`	the matched call.
`na.action`	If relevant.

References

Mardia, K. V., J. T. Kent and J. M. Bibby (1979). *Multivariate Analysis*, London: Academic Press.

Venables, W. N. and B. D. Ripley (1997, 9). *Modern Applied Statistics with S-PLUS*, 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
data(USArrests)
(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(~ ., data = USArrests, na.action=na.exclude, cor = TRUE)
pc.cr$scores
```

`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

```
library(mva)
data(USArrests)
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
```

`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 (1997, 9). *Modern Applied Statistics with S-PLUS*, Springer-Verlag.

See Also

`princomp`.

Examples

```
## The variances of the variables in the
## USArrests data vary by orders of magnitude, so scaling is appropriate
data(USArrests)
(pc.cr <- princomp(USArrests, cor = TRUE))  # inappropriate
screeplot(pc.cr)

data(Harman74.cor)
fit <- princomp(covmat=Harman74.cor)
screeplot(fit)
screeplot(fit, npcs=24, type="lines")
```

summary.princomp — *Summary method for Principal Components Analysis*

Description

The summary method for class "princomp".

Usage

```
summary(object, loadings = FALSE, cutoff = 0.1,
        digits = 3, ...)
```

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.
digits	the number of significant digits to be used in listing loadings.
...	arguments to be passed to or from other methods.

Value

object, invisibly.

See Also

princomp

Examples

```
data(USArrests)
summary(pc.cr <- princomp(USArrests, cor=TRUE))
summary(princomp(USArrests, cor=TRUE),
        loadings = TRUE, cutoff = 0.2, digits = 2)
```

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`.
`rotmat`	The 'rotation matrix.

Author(s)

B. D. Ripley

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

```
data(swiss)
## varimax with normalize = T is the default
fa <- factanal( ~., 2, data = swiss)
varimax(fa$loadings, normalize = FALSE)
promax(fa$loadings)
```

Chapter 8

The `nls` package

`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)
```

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.

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

```
data(BOD)
# 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 )
```

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.

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

```
data(ChickWeight)
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 )
```

---

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

```
data( women )
lapply( women, mean )               # has a names attribute
clearNames( lapply( women, mean ) ) # removes the names
```

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*

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

```
data(CO2)
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)
```

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.

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

```
data(DNase)
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)
```

formula.nls *Extract Model Formula from nls Object*

Description

Returns the model used to fit `object`.

Usage

```
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

```
data(Orange)
fm1 <- nls(circumference ~ A/(1+exp((B-age)/C)), Orange,
  start = list(A=160, B=700, C = 350))
formula(fm1)
```

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

```
data(Puromycin)
PurTrt <- Puromycin[ Puromycin$state == "treated", ]
getInitial( rate ~ SSmicmen( conc, Vm, K ), PurTrt )
```

`Indometh` *Pharmacokinetics of Indomethicin*

Description

The `Indometh` data frame has 66 rows and 3 columns of data on the pharmacokinetics of indomethicin.

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

```
data(Indometh)
fm1 <- nls(conc ~ SSbiexp(time, A1, lrc1, A2, lrc2),
           data = Indometh, subset = Subject == 1)
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.

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

```
data(Loblolly)
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)))
```

`nls` *Nonlinear Least Squares*

Description

Determine the nonlinear least squares estimates of the nonlinear model parameters and return a class `nls` object.

Usage

```
nls(formula, data = parent.frame(), start, control = nls.control(),
    algorithm = "default", trace = FALSE, subset,
    weights, na.action)
```

Arguments

`formula`	a nonlinear model formula including variables and parameters
`data`	an optional data frame in which to evaluate the variables in `formula`
`start`	a named list or named numeric vector of starting estimates
`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. The other alternative is "plinear", the Golub-Pereyra algorithm for partially linear least-squares models.
`trace`	logical value indicating if a trace of the iteration progress should be printed. Default is `FALSE`. If `TRUE` the residual 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.
`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. *not yet implemented*
`na.action`	a function which indicates what should happen when the data contain `NA`s.

Details

An `nls` object is a type of fitted model object. It has methods for the generic functions `coef`, `formula`, `resid`, `print`, `summary`, `AIC`, and `fitted`.

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.

Author(s)

Douglas M. Bates and Saikat DebRoy

References

Bates, D.M. and Watts, D.G. (1988) *Nonlinear Regression Analysis and Its Applications*, Wiley

See Also

nlsModel

Examples

```
data( DNase )
DNase1 <- DNase[ DNase$Run == 1, ]
## using a selfStart model
fm1DNase1 <- nls( density ~ SSlogis( log(conc), Asym, xmid, scal ), DNase1 )
summary( fm1DNase1 )
## using conditional linearity
fm2DNase1 <- nls( density ~ 1/(1 + exp(( xmid - log(conc) )/scal ) ),
                  data = DNase1,
                  start = list( xmid = 0, scal = 1 ),
                  alg = "plinear", trace = TRUE )
summary( fm2DNase1 )
## without conditional linearity
fm3DNase1 <- nls( density ~ Asym/(1 + exp(( xmid - log(conc) )/scal ) ),
                  data = DNase1,
                  start = list( Asym = 3, xmid = 0, scal = 1 ),
                  trace = TRUE )
summary( fm3DNase1 )
## weighted nonlinear regression
data(Puromycin)
Treated <- Puromycin[Puromycin$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, 18:48

    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)
```

`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`

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)
```

`nlsModel` *Create an nlsModel Object*

Description

This is the constructor for `nlsModel` objects, which are function closures for several functions in a list. The closure includes a nonlinear model formula, data values for the formula, as well as parameters and their values.

Usage

```
nlsModel(form, data, start)
```

Arguments

`form`	a nonlinear model formula
`data`	a data frame or a list in which to evaluate the variables from the model formula
`start`	a named list or named numeric vector of starting estimates for the parameters in the model

Details

An `nlsModel` object is primarily used within the `nls` function. It encapsulates the model, the data, and the parameters in an environment and provides several methods to access characteristics of the model. It forms an important component of the object returned by the `nls` function.

Value

The value is a list of functions that share a common environment.

`resid`	returns the residual vector evaluated at the current parameter values
`fitted`	returns the fitted responses and their gradient at the current parameter values
`formula`	returns the model formula
`deviance`	returns the residual sum-of-squares at the current parameter values
`gradient`	returns the gradient of the model function at the current parameter values
`conv`	returns the relative-offset convergence criterion evaluated at the current parmeter values
`incr`	returns the parameter increment calculated according to the Gauss-Newton formula
`setPars`	a function with one argument, `pars`. It sets the parameter values for the `nlsModel` object and returns a logical value denoting a singular gradient array.
`getPars`	returns the current value of the model parameters as a numeric vector
`getAllPars`	returns the current value of the model parameters as a numeric vector
`getEnv`	returns the environment shared by these functions

`trace`	the function that is called at each iteration if tracing is enabled
`Rmat`	the upper triangular factor of the gradient array at the current parameter values
`predict`	takes as argument `newdata`,a `data.frame` and returns the predicted response for `newdata`.

Author(s)

Douglas M. Bates and Saikat DebRoy

References

Bates, D.M. and Watts, D.G. (1988), *Nonlinear Regression Analysis and Its Applications*, Wiley

See Also

`nls`

Examples

```
data( DNase )
DNase1 <- DNase[ DNase$Run == 1, ]
mod <-
 nlsModel(density ~ SSlogis( log(conc), Asym, xmid, scal ),
          DNase1, list( Asym = 3, xmid = 0, scal = 1 ))
mod$getPars()     # returns the parameters as a list
mod$deviance()    # returns the residual sum-of-squares
mod$resid()       # returns the residual vector and the gradient
mod$incr()        # returns the suggested increment
mod$setPars( unlist(mod$getPars()) + mod$incr() )  # set new parameter values
mod$getPars()     # check the parameters have changed
mod$deviance()    # see if the parameter increment was successful
mod$trace()       # check the tracing
mod$Rmat()        # R matrix from the QR decomposition of the gradient
```

`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

```
data( Loblolly )
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

```
data( DNase )
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

```
data( DNase )
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

```
data( DNase )
DNase.2 <- DNase[ DNase$Run == "2", ]
DN.srt <- sortedXyData( expression(log(conc)), expression(density), DNase.2 )
NLSstRtAsymptote( DN.srt )
```

`numericDeriv` *Evaluate derivatives numerically*

Description

`numericDeriv` numerically evaluates the gradient of an expression.

Usage

```
numericDeriv(expr, theta, rho=parent.frame())
```

Arguments

`expr`	The expression to be differentiated. The value of this expression should be a numeric vector.
`theta`	A character vector of names of variables used in `expr`
`rho`	An environment containing all the variables needed to evaluate `expr`

Details

This is a front end to the C function `numeric_deriv`, which is described in *Writing R Extensions*.

Value

The value of `eval(expr, env = 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)
```

`Orange` *Growth of orange trees*

Description

The `Orange` data frame has 35 rows and 3 columns of records of the growth of orange trees.

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

```
data(Orange)
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)))
```

`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

```
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

```
data( BOD )
# 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)
```

`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

```
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 with values of the input variables for the model in `object`. 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 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 predictions
`residual.scale`	residual standard deviations
`df`	degrees of freedom for residual

See Also

The model fitting function `nls`, `predict`.

Examples

```
data( BOD )
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)
```

`profile.nls` *Method for Profiling nls Objects*

Description

Investigates behavior of the log-likelihood function near the solution represented by `fitted`.

Usage

```
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. By default, all 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)

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`, `profiler.nls`, `plot.profile.nls`

Examples

```
data( BOD )
# 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
```

`profiler` *Constructor for Profiler Objects for Nonlinear Models*

Description

Create a profiler object for the model object `fitted`.

Usage

```
profiler(fitted, ...)
```

Arguments

`fitted`	the original fitted model object.
`...`	Additional parameters. See documentation on individual methods.

Value

An object of class `"profiler"` which is a list with function elements

`getFittedPars()`
: the parameters in `fitted`

`setDefault(varying, params)`
: this is used for changing the default settings for profiling. In absence of both parameters, the default is set to the original fitted parameters with all parameters varying. The arguments are

  `varying`: a logical, integer or character vector giving parameters to be varied. `params`: the default value at which profiling is to take place.

`getProfile(varying, params)`
: this can be used in conjunction with `setDefault` without any arguments. Alternatively, the parameters to be varied and the values for fixed parameters can be specified using the arguments. The arguments are

  `varying`: a logical vector giving parameters to be varied. This can be omitted if params is a named list or numeric vector.

  `params`: values for parameters to be held fixed.

  It returns a list with elements

  `parameters`: the parameter values for the profiled optimum.

  `fstat`: a profile statistics. See individual methods for details.

  `varying`: a logical vector indicating parameters which were varied.

Author(s)

Douglas M. Bates and Saikat DebRoy

See Also

`profiler.nls`, `profile`

Examples

```
# see documentation on individual methods
```

`profiler.nls` *Constructor for Profiler Objects from nls Objects*

Description

Create a profiler object for the model object `fitted` of class `nls`.

Usage

```
profiler(fitted, ...)
```

Arguments

`fitted`	the original fitted model object of class `nls`.
`...`	Additional parameters. None are used.

Value

An object of class `profiler.nls` which is a list with function elements

`getFittedModel()`
: the `nlsModel` object corresponding to `fitted`

`getFittedPars()`
: See documentation for `profiler`

`setDefault(varying, params)`
: See documentation for `profiler`

`getProfile(varying, params)`
: In the returned list, `fstat` is the ratio of change in sum-of-squares and the residual standard error.

  For other details, see documentation for `profiler`

WARNING

When using setDefault and getProfile together, the internal state of the fitted model may get changed. So after completing the profiling for a parameter, the internal states should be restored by a call to setDefault without any arguments. For example see below or the source for `profile.nls`.

Author(s)

Douglas M. Bates and Saikat DebRoy

References

Bates, D.M. and Watts, D.G. (1988), *Nonlinear Regression Analysis and Its Applications*, Wiley

See Also

`nls`, `nlsModel`, `profiler`, `profile.nls`

Examples

```
data( BOD )
## obtain the fitted object
fm1 <- nls(demand ~ SSasympOrig( Time, A, lrc ), data = BOD)
## get the profile for the fitted model
prof1 <- profiler( fm1 )
## profile with A fixed at 16.0
prof1$getProfile(c(FALSE, TRUE), 16.0)
## vary lrc
prof1$setDefault(varying = c(FALSE, TRUE))
## fix A at 14.0 and starting estimate of lrc at -0.2
prof1$setDefault(params = c(14.0, -0.2))
## and get the profile
prof1$getProfile()
## finally, set defaults back to original estimates
prof1$setDefault()
```

`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.

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

```
data(Puromycin)
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)
```

`selfStart` *Construct Self-starting Nonlinear Models*

Description

This function is generic; methods functions can be written to handle specific classes of objects. Available methods include `selfStart.default` and `selfStart.formula`. See the documentation on the appropriate method function.

Usage

```
selfStart(model, initial, parameters, template)
```

Arguments

`model`	a function object defining a nonlinear model.
`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, template`	arguments used by some methods..

Value

a function object of the `selfStart` class.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

`selfStart.default`, `selfStart.formula`

Examples

```
## see documentation for the methods
```

`selfStart.default` *Construct Self-starting Nonlinear Models*

Description

A method for the generic function 'selfStart' for formula objects.

Usage

```
selfStart(model, initial, parameters, template)
```

Arguments

`model`	a function object defining a nonlinear model.
`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, template`	these arguments are included for consistency with the generic function, but are not used in the `default` method. See the documentation on `selfStart.formula`.

Value

a function object of class `selfStart`, corresponding to a self-starting nonlinear model function. 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

`selfStart.formula`

Examples

```
# '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

SSfol <- selfStart(first.order.log.model, first.order.log.initial)
```

`selfStart.formula` *Construct Self-starting Nonlinear Models*

Description

A method for the generic function 'selfStart' for formula objects.

Usage

```
selfStart(model, initial, parameters, template)
```

Arguments

`model`	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.

Value

a function object of class `selfStart`, 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

`selfStart.default`, `deriv`

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"))
```

`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
```

`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

```
data( DNase )
DNase.2 <- DNase[ DNase$Run == "2", ]
sortedXyData( expression(log(conc)), expression(density), DNase.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

```
data( Loblolly )
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

```
data( CO2 )
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

```
data( Loblolly )
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

```
data( Indometh )
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)
```

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

```
data( Theoph )
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

```
data( ChickWeight )
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

```
data(DNase )
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

```
data( ChickWeight )
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

```
data( Puromycin )
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

```
data(ChickWeight)
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)
```

`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.

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

```
data(Theoph)
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)
```

Chapter 9

The splines package

`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

```
data( women )
ispl <- interpSpline( weight ~ height,  women )
pred <- predict(ispl)
class(pred)
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

```
data( women )
ispl <- interpSpline( women$height, women$weight )
bspl <- backSpline( ispl )
plot( bspl )                    # plots over the range of the knots
points( women$weight, women$height )
```

`bs` *Generate a 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.
`df`	degrees of freedom; one can specify `df` rather than `knots`; `bs()` then chooses `df-degree-1` knots at suitable quantiles of `x`.
`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()` written by Douglas Bates. 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.

Beware of making predictions with new `x` values when `df` is used as an argument. Either use `safe.predict.gam()`, or else specify `knots` and `Boundary.knots`.

See Also

`ns`, `poly`, `smooth.spline`, `predict.bs`.

Examples

```
data(women)
bs(women$height, df = 5)
summary(fm1 <- lm(weight ~ bs(height, df = 5), data = women))
```

`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 `NA`s. 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

```
data( women )
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.
`df`	degrees of freedom. One can supply `df` rather than knots; `ns()` then chooses `df - 1 - intercept` knots at suitably chosen quantiles of `x`.
`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.

Beware of making predictions with new `x` values when `df` is used as an argument. Either use `safe.predict.gam()`, or else specify `knots` and `Boundary.knots`.

See Also

bs, poly, predict.ns

Examples

```
data(women)
ns(women$height, df = 5)
summary(fm1 <- lm(weight ~ ns(height, df = 5), data = women))
```

`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

```
data( women )
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

```
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. Presently 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

```
data(women)
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

```
predict(object, x, nseg=50, deriv=0, ...)
plot.spline(x, ...)
```

Arguments

`object`	An object that inherits from the `bSpline` or the `polySpline` class. For `plot.spline` this argument is called `x`.
`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. For `plot.spline` the `x` argument is as described under `object` above.
`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.
`...`	`predict`: further arguments passed to or from other methods. `plot`: additional graphical parameters (see `link{par}`).

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

```
data( women )
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)
spline.des(knots, x, ord = 4, derivs)
```

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. 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`.

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)
```

`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

```
data( women )
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

```
data( women )
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

```
data( women )
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 10

The stepfun package

`ecdf` *Empirical Cumulative Distribution Function*

Description

Compute or plot an empirical cumulative distribution function.

Usage

```
ecdf(x)

plot(..., verticals = FALSE, col.01line = "gray70")
print(x, digits= getOption("digits") - 2, ...)
summary(object, ...)
```

Arguments

`x`	numeric vector of "observations" in `ecdf`; for the methods `x` is as `object` below.
`...`	arguments to be passed to `plot.stepfun`, the first of which should be an R object of class `"ecdf"`.
`verticals`	see `plot.stepfun`.
`col.01line`	numeric or character specifying the color of the horizontal lines at y=0 and 1, see `colors`.
`object`	(or `x`:) object of class `"ecdf"`, typically.
`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 jump $1/n$ at each observation (possibly with multiple jumps at one place if there are ties).

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⟩.

See Also

`stepfun`, the more general class of step functions, `approxfun` and `splinefun`.

Examples

```
##-- Simple didactical  ecdf  example:
Fn <- ecdf(rnorm(12))
Fn; summary(Fn)
12*Fn(knots(Fn)) == 1:12 ## == 1:12  if and only if there are no ties !

y <- round(rnorm(12),1); y[3] <- y[1]
Fn12 <- ecdf(y)
Fn12
print(knots(Fn12), dig=2)
12*Fn12(knots(Fn12)) ## ~= 1:12  if there where no ties

summary(Fn12)
summary.stepfun(Fn12)
print(ls.Fn12 <- ls(env= environment(Fn12)))
##[1] "f"  "method"  "n"  "ties"   "x"  "y"  "yleft"  "yright"

12 * Fn12((-20:20)/10)

###----------------- 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) dis-regarded
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='b', cex=.1)#- plot.default
plot(Fn12, col.h='red', add= TRUE)  #- plot method
abline(v=knots(Fn12),lty=2,col='gray70')
plot(Fn12, verticals=TRUE, col.p='blue', col.h='red',col.v='bisque')
par(op)

##-- this works too (automatic call to  ecdf(.)):
```

```
plot.ecdf(rnorm(24))
```

`plot.stepfun` *Plot Step Functions*

Description

Method of the generic `plot` for `stepfun` objects and utility for plotting piecewise constant functions.

Usage

```
plot(x, xval, xlim, 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"), ...)
```

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`	numeric(2); range of `x` values to use.
`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, f = 1)

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); plot(sfun1, xval=tt, add=TRUE, 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)
  plot(list(sfun0,sfun.2,sfun1)[[i]], add=TRUE, 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)

##-- this works too (automatic call to  ecdf(.)):
plot.stepfun(rt(50, df=3), col.vert = "gray20")
```

`stepfun` *Step Functions*

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 $fn(x_i) = y_i$ for $i = 1, \ldots, n$.

The value of the constant c_i above depends on the "continuity" parameter `f`. For the default, `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.

Usage

```
stepfun(x, y, f = 0, ties = "ordered")

is.stepfun(x)
knots(Fn, ...)

print(x, digits= getOption("digits") - 2, ...)
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`.
`Fn, object`	an R object inheriting from `"stepfun"`.
`digits`	number of significant digits to use, see `print`.
`...`	potentially further arguments (require 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(.)`).

normal-bracket97bracket-normal The knots are also available by `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)
sfun0
summary(sfun0)
summary(sfun.2)

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))
```

Chapter 11

The `tools` package

`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 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.

See Also

`.C`, `.Fortran`; `Foreign`.

Examples

```
checkFF(package = "ts", verbose = TRUE)
```

checkTnF *Check R Code for T/F*

Description

Checks the specified R code for occurrences of `T` or `F`, and reports 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 of the package is 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 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`.

Warning

This function is still experimental. Both name and interface might change in future versions.

codoc *Check Code/Documentation Consistency*

Description

Find inconsistencies between actual and documented usage of R function objects in a package, by comparing names and optionally also corresponding positions and default values of the arguments of the functions.

Usage

```
codoc(package, dir, lib.loc = NULL,
      use.values = FALSE, use.positions = TRUE,
      ignore.generic.functions = FALSE,
      keep.tempfiles = FALSE,
      verbose = getOption("verbose"))
```

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.positions`	a logical indicating whether to use the positions of function arguments when comparing.
`use.values`	a logical indicating whether to use function default values when comparing code and docs.
`ignore.generic.functions`	if `TRUE`, functions the body of which contains `"UseMethod"` are ignored.
`keep.tempfiles`	if `TRUE`, keep temporary code and docs files used for comparison. This used to be useful for debugging, but is now deprecated and will be removed in future versions.
`verbose`	a logical. If `TRUE`, additional diagnostics are printed.

Details

The purpose of this function 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 manipulated in order to give function stubs corresponding to the indicated usage, which are then sourced in another new environment. For interpreted functions in both the code and docs environment, the formals are compared according to the values of the arguments `use.positions` and `use.values`.

Currently, synopsis sections are used, but multiple usage examples (such as in `abline`) are not combined when building the stubs.

Value

A list the names of which are the names of the functions where an inconsistency was found. The elements of the list are lists of length 2 with elements `code` and `docs`, giving the corresponding arguments obtained from the function's code and documented usage.

See Also

`undoc`

QA *QA Checks for R Code and/or Documentation*

Description

Functions for performing various quality checks.

Usage

```
checkAssignFuns(package, dir, lib.loc = NULL)
checkDocArgs(package, dir, lib.loc = NULL)
checkDocStyle(package, dir, lib.loc = NULL)
checkMethods(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

checkAssignFuns checks whether assignment functions in the package R code have their final argument named value.

checkDocArgs 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.

checkDocStyle investigates how (S3) methods are shown in the usages of the Rd files in a package. It reports if methods are shown along with their generic, which typically causes problems for the documentation of the primary argument of the generic. It also finds the methods shown by their full name (rather than that of the generic using the Rd \method markup).

checkMethods checks whether all 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. The generics are sought first in the given package and then in the base package. The rules when ... is involved are subtle: see the source code.

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.

Warning

These functions are still experimental. Both names and interfaces might change in future versions.

tools-internal *Internal tools functions*

Description

Internal tools functions.

Usage

```
listFilesWithExts(dir, exts, path = TRUE)
sQuote(s)
```

Details

These are not to be called by the user.

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 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

A character vector containing the names of the undocumented objects.

Examples

```
undoc("eda")                  # Undocumented objects in 'eda'
```

Chapter 12

The ts package

`acf` *Autocovariance and Autocorrelation 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 = NULL, plot = TRUE, na.action = na.fail, ...)
ccf(x, y, lag.max = NULL, type = c("correlation", "covariance"),
    plot = TRUE, na.action = na.fail, ...)
```

Arguments

`x, y`	a univariate or multivariate (not `ccf`) time series object or a numeric vector or matrix.
`lag.max`	maximum number of lags at which to calculate the acf. Default is $10\log_{10}(N)$ where N is the number of observations.
`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 acf is plotted.
`na.action`	function to be called to handle missing values.
`demean`	logical. Should the covariances be about the sample means?
`...`	further arguments to be passed to `plot.acf`.

Details

For `type` = `"correlation"` and `"covariance"`, the estimates are based on the sample covariance.

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.

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 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`

Examples

```
## Examples from Venables & Ripley
data(lh)
acf(lh)
acf(lh, type = "covariance")
pacf(lh)

data(UKLungDeaths)
acf(ldeaths)
acf(ldeaths, ci.type = "ma")
acf(ts.union(mdeaths, fdeaths))
ccf(mdeaths, fdeaths) # just the cross-correlations.
```

`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, aic = TRUE, order.max = NULL,
        na.action = na.fail, demean = TRUE, series, var.method = 1, ...)
ar.yw(x, aic = TRUE, order.max = NULL,
      na.action = na.fail, demean = TRUE, series, ...)
ar.mle(x, aic = TRUE, order.max = NULL, na.action = na.fail, demean = TRUE,
       series, ...)

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.
`asy.var.coef`	(univariate case.) 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.

Examples

```
data(lh)
ar(lh)
ar(lh, method="burg")
ar(lh, method="ols")
ar(lh, FALSE, 4) # fit ar(4)

data(LakeHuron)
ar(LakeHuron)
ar(LakeHuron, method="burg")
ar(LakeHuron, method="ols")

data(sunspot)
sunspot.ar <- ar(sunspot.year)
sunspot.ar
ar(x = sunspot.year, method = "burg")
ar(x = sunspot.year, method = "ols")
## next is slow and may have convergence problems,
## as it cares about invertibility
ar(x = sunspot.year, method = "mle")

predict(sunspot.ar, n.ahead=25)

data(BJsales)
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.
`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

```
data(lh)
ar(lh, method="burg")
ar.ols(lh)
ar.ols(lh, FALSE, 4) # fit ar(4)

data(BJsales)
ar.ols(ts.union(BJsales, BJsales.lead))

data(EuStockMarkets)
x <- diff(log(EuStockMarkets))
ar.ols(x, order.max=6, demean=FALSE, intercept=TRUE)
```

`arima0` *ARIMA Modelling of Time Series – Preliminary Version*

Description

Fit an ARIMA model to a univariate time series by exact maximum likelihood, 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, na.action = na.fail,
       delta = 0.01, transform.pars = 2)

predict(object, n.ahead = 1, newxreg, se.fit = TRUE, ...)

arima0.diag(fit, gof.lag = 10)
```

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)`).
`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).
`na.action`	Function to be applied to remove missing values.
`delta`	A value to indicate at which point 'fast recursions' should be used. See the Details section.
`transform.pars`	If greater than 0, the ARMA parameters are transformed to ensure that they remain in the region of invertibility. If equal to 2, the optimization is rerun on the original scale to find the Hessian.
`object, fit`	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?
`gof.lag`	Number of lags to be used in goodness-of-fit test.
`...`	arguments passed to or from other methods.

Details

Different definitions of ARIMA 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.

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 using the Fortran code of Gardener *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.

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.

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`. If `transform.pars = 1`, only the portion corresponding to the untransformed parameters is returned.
`loglik`	the maximized log-likelihood (of the differenced data).
`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.
`resid`	the residuals.
`call`	the matched call.
`series`	the name of the series `x`.
`convergence`	the value returned by `optim`.

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.

Note

This is a preliminary version, and will be replaced in due course.

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.

Author(s)

B.D. Ripley

References

Brockwell, P. J. and Davis, R. A. (1996) *Introduction to Time Series and Forecasting.* Springer, New York. Sections 3.3 and 8.3.

Gardener, 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, section 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.

See Also

ar

Examples

```
data(lh)
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)

data(USAccDeaths)
fit <- arima0(USAccDeaths, order=c(0,1,1), seasonal=list(order=c(0,1,1)))
fit
predict(fit, n.ahead=6)

data(LakeHuron)
arima0(LakeHuron, order=c(2,0,0), xreg=1:98)
```

---

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

```
data(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

```
data(beavers)
```

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

```
data(beavers)

(yl <- range(beaver1$temp, beaver2$temp))

beaver.plot <- function(bdat, ...) {
  nam <- deparse(substitute(bdat))
  attach(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

```
data(BJsales)
```

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.

`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 the time series `x` is computed.

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")
```

`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))
data(lh)
cpgram(lh)
lh.ar <- ar(lh, order.max = 9)
cpgram(lh.ar$resid, main = "AR(3) fit to lh")

data(UKLungDeaths)
cpgram(ldeaths)
```

`diffinv` *Discrete Integrals: Inverse of Differencing*

Description

Computes the inverse function of the lagged differences function `diff`.

Usage

```
diffinv(x, lag = 1, differences = 1,
        xi = rep(0.0, lag*differences*NCOL(x)), ...)
```

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.
`...`	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 representing the discrete integral of `x`.

Author(s)

A. Trapletti

See Also

`diff`

Examples

```
s <- 1:10
d <- diff(s)
diffinv(d, xi = 1)
```

`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)
```

`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

```
data(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.

`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.

Author(s)

B.D. Ripley

See Also

`convolve`

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)
data(presidents)
filter(presidents, rep(1,3))

## A simple simulation function for ARMA processes
arma.sim <- function(n, ar = NULL, ma = NULL, sigma = 1.0)
{
    x <- ts(rnorm(n+100, 0, sigma), start = -99)
    if(length(ma)) x <- filter(x, c(1, ma), sides=1)
    if(length(ar)) x <- filter(x, ar, method = "recursive")
    as.ts(x[-(1:100)])
}
arma.sim(63, c(0.8897,-0.4858), c(-0.2279, 0.2488), sigma=sqrt(0.1796))
```

---

kernapply *Apply Smoothing Kernel*

Description

`kernapply` computes the convolution between an input sequence and a specific kernel.

Usage

```
kernapply(x, k, circular = FALSE, ...)
kernapply(k1, k2)
```

Arguments

`k, k1, k2`	smoothing `"tskernel"` objects.
`x`	an input vector, matrix, or time series to be smoothed.
`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.

Usage

```
kernel(coef, m, r, name)

df.kernel(k)
bandwidth.kernel(k)
is.tskernel(k)

print(x, digits = max(3,getOption("digits")-3), ...)
plot(x, ...)
```

Arguments

`coef`	the upper half of the smoothing kernel coefficients (inclusive of coefficient zero) *or* the name of a kernel (currently `"daniell"`, `"dirichlet"`, `"fejer"` or `"modified.daniell"`.
`m`	the kernel dimension. The number of kernel coefficients is `2*m+1`.
`name`	the name of the kernel.
`r`	the kernel order for a Fejer kernel.

`digits`	the number of digits to format real numbers.
`k, x`	a `"tskernel"` object.
`...`	arguments passed to or from other methods.

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).

`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 (1991), p. 201, with a continuity correction.

Value

`kernel` returns a list with class `"tskernel"`, and components the coefficients `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

```
data(EuStockMarkets)    # Demonstrate a simple trading strategy for the
x <- EuStockMarkets[,1]  # financial time series German stock index DAX.
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

data(sunspot)     # Reproduce example 10.4.3 from Brockwell and Davies (1991)
spectrum(sunspot.year, kernel=kernel("daniell", c(11,7,3)), log="no")
```

`lag` *Lag a Time Series*

Description

Computed a lagged version of a time series, shifting the time base back by a given number of observations.

Usage

```
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*.

Author(s)

B.D. Ripley

See Also

`diff`, `deltat`

Examples

```
data(UKLungDeaths)
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,
         font.main=par("font.main"), cex.main=par("cex.main"),
         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`	positive integer vector allowing to specify 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`.
`font.main, cex.main`	attributes for the title, see `par()`.
`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; if true, the user is asked 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`.

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

```
data(nhtemp)
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")

data(sunspots) # no lines for larges series :
lag.plot(sqrt(sunspots), set = c(1:4, 9:12), pch = ".", col = "gold")

## Multivariate (but non-stationary! ...)
data(freeny)
lag.plot(freeny.x, lag = 3)
```

LakeHuron *Level of Lake Huron 1875–1972*

Description

Annual measurements of the level, in feet, of Lake Huron 1875–1972.

Usage

```
data(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

```
data(lh)
```

Source

P.J. Diggle (1990) *Time Series: A Biostatistical Introduction.* Oxford, table A.1, series 3

`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

```
data(lynx)
```

Source

Brockwell, P. J. and Davis, R. A. (1991) *Time Series and Forecasting Methods.* Second edition. Springer. Series G (page 557).

References

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.

`na.contiguous` *NA Handling Routines for Time Series*

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(frame)
```

Arguments

`frame`	a univariate or multivariate time series.

Value

A time series without missing values. The class of `frame` will be preserved.

Author(s)

B. D. Ripley

See Also

`na.omit` and `na.omit.ts`; `na.fail`

Examples

```
data(presidents)
na.contiguous(presidents)
```

`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

```
data(nottem)
```

Source

Anderson, O.D. (1976) *Time Series Analysis and Forecasting: The Box-Jenkins approach.* Butterworths. Series R.

`plot.acf` *Plotting Autocovariance and Autocorrelation Functions*

Description

Plotting method for objects of class `"acf"`.

Usage

```
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. `xpd = NA` used to be the default for R version <= 1.4.0.
`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.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

```
plot(x, add = FALSE, ci = 0.95, log = c("yes", "dB", "no"),
     xlab = "frequency", ylab = NULL, type = "l", ci.col = "blue",
     main = NULL, sub = NULL,
     plot.type = c("marginal", "coherency", "phase"),
     ci.lty = 3, ...)
```

Arguments

`x`	an object of class `"spec"`.
`add`	logical. If `TRUE`, add to already existing plot.
`ci`	Coverage probability for confidence interval. Plotting of the confidence bar 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"`.
`xlab`	the x label of the plot.
`ylab`	the y label of the plot.
`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.
`main`	overall title for the plot.

sub	a sub title for the plot.
plot.type	For multivariate time series, the type of plot required. Only the first character is needed.
ci.lty	line type for confidence intervals for coherency and phase.
...	Further graphical parameters.

See Also

spectrum

PP.test *Phillips-Perron Unit Root Test*

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)
```

---

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.

Author(s)

B.D. Ripley

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. (1997) *Modern Applied Statistics with S-PLUS.* Second edition. Springer. (Especially page 448.)

See Also

ar, spectrum.

Examples

```
data(lh)
spec.ar(lh)

data(UKLungDeaths)
spec.ar(ldeaths)
spec.ar(ldeaths, method="burg")
```

---

spec.pgram *Estimate Spectral Density of a Time Series from 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 = FALSE, 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`	proportion 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 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. (1997) *Modern Applied Statistics with S-PLUS.* Second edition. Springer. (Especially pp. 437–442.)

See Also

`spectrum`, `spec.taper`, `plot.spec`, `fft`

Examples

```
## Examples from Venables & Ripley
data(UKLungDeaths)
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")

data(EuStockMarkets)
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")

data(BJsales)
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*

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 total proportion to be tapered, either a scalar or a vector of the length of the number of series.

Details

The cosine-bell taper is applied to the first and last `p[i]/2` observations of time series `x[, i]`.

Value

A new time series object.

Note

From package 'MASS'.

Author(s)

Kurt Hornik, B.D. Ripley

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"), plot = TRUE, na.action = na.fail,
         ...)
```

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"`.
`plot`	logical. If `TRUE` then the spectral density is plotted.
`na.action`	`NA` action function.
`...`	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. (1997) *Modern Applied Statistics with S-PLUS.* Second edition. Springer. (Especially pages 437–442.)

See Also

`spec.ar`, `spec.pgram`; `plot.spec`.

Examples

```
## Examples from Venables & Ripley
## spec.pgram
par(mfrow=c(2,2))
data(lh)
spectrum(lh)
spectrum(lh, spans=3)
spectrum(lh, spans=c(3,3))
```

```
spectrum(lh, spans=c(3,5))

data(UKLungDeaths)
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")
```

`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 'modreg' (which is not actually used in `stl`).

Examples

```
data(nottem)
plot(stl(nottem, "per"))
plot(stl(nottem, s.win = 4, t.win = 50, t.jump = 1))

data(co2)
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 :
data(UKLungDeaths)
        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.

Usage

```
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, ...)
print(x, ...)
summary(object, digits = getOption("digits"), ...)
```

Arguments

`x, object`	`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.
`digits`	significant figures used in printing.
`...`	further arguments passed to or from other methods.

See Also

`plot.ts` and `stl`, particularly for examples.

`sunspot` — *Yearly Sunspot Data, 1700–1988. Monthly Sunspot Data, 1749–1997.*

Description

Monthly and yearly number of sunspots.

Usage

```
data(sunspot)
```

Format

The univariate time series `sunspot.year` and `sunspot.month` contain 289 and 2988 observations, respectively. The objects are of class `"ts"`.

Source

Monthly data: Sunspot Index Data Center, 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

Yearly data: H. Tong (1996) *Non-Linear Time Series.* Clarendon Press, Oxford, p. 471.

See Also

`sunspot.month` is a longer version of `sunspots` in base R, that runs until 1988.

Examples

```
## Compare the monthly series from 'base' and 'ts':
data(sunspots, package = base)
data(sunspot,  package = ts)
plot (sunspot.month, main = "sunspot.month [ts]", 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))
```

`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)
```

`treering` *Yearly Treering Data, -6000–1979.*

Description

Contains normalized tree-ring widths in dimensionless units. Each tree ring corresponds to one year. Tree: Methuselah Walk, Pilo; Location: California, Gt Basin B C pine 2805M, 3726-11810; Author: Donald A. Graybill, 1980.

Usage

```
data(treering)
```

Format

A univariate time series with 7981 observations. The object is of class `"ts"`.

Source

Time Series Data Library: http://www-personal.buseco.monash.edu.au/~hyndman/TSDL/

`ts-internal` *Internal ts functions*

Description

Internal ts functions.

Usage

```
arma0f(p)
```

Details

These are not to be called by the user.

`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

Value

None.

Note

Although this can be used for a single time series, `plot` is easier to use and is preferred.

Author(s)

B.D. Ripley

See Also

`plot.ts`

Examples

```
data(UKLungDeaths)
ts.plot(ldeaths, mdeaths, fdeaths,
        gpars=list(xlab="year", ylab="deaths", lty=c(1:3)))

data(nottem)
nott <- window(nottem, end=c(1936,12))
fit <- arima0(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)))
```

`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.

Author(s)

B. D. Ripley

See Also

`cbind`.

Examples

```
data(UKLungDeaths)
ts.union(mdeaths, fdeaths)
cbind(mdeaths, fdeaths) # same as the previous line
ts.intersect(window(mdeaths, 1976), window(fdeaths, 1974, 1978))
data(BJsales)
sales1 <- ts.union(BJsales, lead = BJsales.lead)
ts.intersect(sales1, lead3 = lag(BJsales.lead, -3))
```

`UKDriverDeaths` *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

```
data(UKDriverDeaths)
```

Source

Harvey, A.C. (1989) *Forecasting, Structural Time Series Models and the Kalman Filter.* Cambridge University Press, pp. 519–523.

`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

```
data(UKLungDeaths)
```

Source

P. J. Diggle (1990) *Time Series: A Biostatistical Introduction.* Oxford, table A.3

Examples

```
data(UKLungDeaths)
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

```
data(USAccDeaths)
```

Source

P. J. Brockwell and R. A. Davis (1991) *Time Series: Theory and Methods.* Springer, New York.

Chapter 13

The `tcltk` package

`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 the pseudo-list `tclvar`.

Usage

```
.Tcl(...)
.Tcl.args(...)
.Tcl.callback(...)
.Tk.ID(win)
.Tk.newwin(ID)
.Tk.subwin(parent)
.TkWin
.TkRoot

tkdestroy(win)
is.tkwin(x)

tclvar$name           # Deprecated
tclvar$name <- value  # Deprecated

tclVar(init="")
as.character.tclVar(x)
tclvalue(x)
tclvalue(x) <- value

addTclPath(path = ".")
tclRequire(package, warn = TRUE)
```

Arguments

`win`	a window structure
`x`	an object
`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 a text string.

`.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 `-`

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 the function will exist at that 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` is used to access Tcl variables. The Tcl variable name is used as if it were a list element name in `tclvar`, but in reality `tclvar` is an object of class `tclvar` and `$` and `$<-` have special methods for that class. NOTE: This interface is deprecated, use the `tclVar` object class instead.

`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.

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"

Examples

```
## 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()"HI!"
.Tcl.callback(f)
.Tcl.args(text="Push!", command=f) # NB: Different address

xyzzy <- tclVar(7913)
tclvalue(xyzzy)
tclvalue(xyzzy) <- "foo"
as.character(xyzzy)
tkcmd("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
```

`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

```
tkcmd(...)
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(...)

## 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(...)
tkmessageBox(...)

## File handling functions
tkfile.tail(...)
tkfile.dir(...)
tkopen(...)
tkclose(...)
tkputs(...)
```

Arguments

`x`	A window object
`...`	Handled via `.Tcl.args`

Details

`tkcmd` provides a generic interface to calling any Tk or Tcl command by simply running `.Tcl.args` on the arguments and passing the result to `.Tcl`. Most of the other commands simply call `tkcmd` 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. Except for a few exceptions, the pattern is that Tcl subcommands like `pack configure` are converted to function names like `tkpack.configure`.

See Also

TclInterface, TkWidgets, TkWidgetcmds

Examples

```
## 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)
```

`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 at the beginning of each file
`title`	title to use for the window
`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`

Examples

`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, ...)
tkdump(widget, ...)
tkentrycget(widget, ...)
tkentryconfigure(widget, ...)
tkfind(widget, ...)
tkflash(widget, ...)
tkfraction(widget, ...)
tkget(widget, ...)
tkgettags(widget, ...)
tkicursor(widget, ...)
tkidentify(widget, ...)
tkimage.cget(widget, ...)
tkimage.configure(widget, ...)
tkimage.create(widget, ...)
tkimage.names(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

```
## 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=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)
```

`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

```
## 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)
```

Index