Package: mosaic (via r-universe)

October 15, 2024

Type Package

Title Project MOSAIC Statistics and Mathematics Teaching Utilities

Version 1.9.1

Description Data sets and utilities from Project MOSAIC (<http://www.mosaic-web.org>) used to teach mathematics, statistics, computation and modeling. Funded by the NSF, Project MOSAIC is a community of educators working to tie together aspects of quantitative work that students in science, technology, engineering and mathematics will need in their professional lives, but which are usually taught in isolation, if at all.

Depends R (>= 4.1),

Imports dplyr, tibble, lattice (>= 0.20-21), ggformula, mosaicData, Matrix, mosaicCore (>= 0.7.0), ggplot2, rlang (>= 0.4.7), purrr, MASS, grid, tidyr, methods, utils

Suggests ggstance, ggridges, vdiffr, lubridate, magrittr, NHANES, RCurl, sp, vcd, testthat (>= 3.0.0), knitr, tools, parallel, mapproj, rgl, rmarkdown, covr, formatR, palmerpenguins, ggrepel, readr, ggdendro, gridExtra, splines, latticeExtra, glue, broom, leaflet

Enhances manipulate

VignetteBuilder knitr

License GPL (>=2)

LazyLoad yes

LazyData yes

URL https://github.com/ProjectMOSAIC/mosaic, https://www.mosaic-web.org/mosaic/

BugReports https://github.com/ProjectMOSAIC/mosaic/issues

RoxygenNote 7.3.1

Roxygen list(markdown = TRUE)

Encoding UTF-8

Config/testthat/edition 3

Repository https://projectmosaic.r-universe.dev

RemoteUrl https://github.com/projectmosaic/mosaic

RemoteRef HEAD

RemoteSha 3ff2c56706c4c8995b274a02a5cc4f23f3361466

Contents

`mosaic-package` *mosaic: the Project MOSAIC package*

Description

mosaic

Details

Data sets and utilities from Project MOSAIC (mosaic-web.org) used to teach mathematics, statistics, computation and modeling. Funded by the NSF, Project MOSAIC is a community of educators working to tie together aspects of quantitative work that students in science, technology, engineering and mathematics will need in their professional lives, but which are usually taught in isolation, if at all.

Author(s)

Randall Pruim (<`rpruim@calvin.edu`>), Daniel Kaplan (<`kaplan@macalester.edu`>), Nicholas Horton (<`nhorton@smith.edu`>)

References

http://www.mosaic-web.org

See Also

Useful links:

- https://github.com/ProjectMOSAIC/mosaic
- https://www.mosaic-web.org/mosaic/
- Report bugs at https://github.com/ProjectMOSAIC/mosaic/issues

---

`adapt_seq` *Adaptively generate sequences in an interval*

Description

`adapt_seq` is similar to `seq` except that instead of selecting points equally spaced along an interval, it selects points such that the values of a function applied at those points are (very) roughly equally spaced. This can be useful for sampling a function in such a way that it can be plotted more smoothly, for example.

Usage

```
adapt_seq(
  from,
  to,
  length.out = 200,
  f = function(x, ...) {
     1
 },
  args = list(),
  quiet = FALSE
)
```

Arguments

`from`	start of interval
`to`	end of interval
`length.out`	desired length of sequence
`f`	a function
`args`	arguments passed to `f`
`quiet`	suppress warnings about NaNs, etc.

Value

a numerical vector

Examples

```
adapt_seq(0, pi, 25, sin)
```

`aggregatingFunction1` *1-ary Aggregating functions*

Description

`aggregatinFuntion1` creates statistical summaries of one numerical vector that are formula aware.

Usage

```
aggregatingFunction1(
  fun,
  output.multiple = FALSE,
  envir = parent.frame(),
  na.rm = getOption("na.rm", FALSE),
  style = c("formula1st", "formula", "flexible")
)
```

Arguments

`fun`	a function that takes a numeric vector and computes a summary statistic, returning a numeric vector.
`output.multiple`	a boolean indicating whether `fun` returns multiple values
`envir`	an environment in which evaluation takes place.
`na.rm`	the default value for na.rm in the resulting function.
`style`	one of `"formula1st"`, `"formula2nd"` or `"flexible"`. In the first two cases, the first argument must be a formula or evaluate to an object. In the latter case, bare names will be converted into formulas.

Details

The logic of the resulting function is this: 1) If the first argument is a formula, use that formula and `data` to create the necessary call(s) to `fun`; (2) Else simply pass everything to `fun` for evaluation.

Value

a function that generalizes `fun` to handle a formula/data frame interface.

Note

Earlier versions of this function supported a "bare name + data frame" interface. This functionality has been removed since it was (a) ambiguous in some cases, (b) unnecessary, and (c) difficult to maintain.

Examples

```
if (require(mosaicData)) {
  foo <- aggregatingFunction1(base::mean)
  foo( ~ length, data = KidsFeet)
  base::mean(KidsFeet$length)
  foo(length ~ sex, data = KidsFeet)
}
```

`aggregatingFunction1or2` *1- or 2-ary aggregating functions*

Description

`aggregatingFunction1or2()` creates statistical summaries for functions like `var()` that can have either 1 or 2 numeric vector inputs.

Usage

```
aggregatingFunction1or2(
  fun,
  output.multiple = FALSE,
  na.rm = getOption("na.rm", FALSE)
)
```

Arguments

`fun`	a function that takes 1 or 2 numeric vectors and computes a summary statistic, returning a numeric vector of length 1.
`output.multiple`	a boolean indicating whether `fun` returns multiple values
`na.rm`	the default value for na.rm in the resulting function.

Details

This was designed primarily to support `var` which can be used to compute either the variance of one variable or the covariance of two variables. The logic of the resulting function is this: 1) If the first two arguments are both formulas, then those formulas are evaluated (with `data`) to compute the covariance; (2) If the first argument is a formula, and the second is `NULL`, then the formula and `data` are used to create the necessary call(s) to `fun`; (3) Else everything is simply passed to `fun` for evaluation.

Note

Earlier versions of this function supported a "bare name + data frame" interface. This functionality has been removed since it was (a) ambiguous in some cases, (b) unnecessary, and (c) difficult to maintain.

`aggregatingFunction2` *2-ary aggregating functions*

Description

`aggregatinFuntion2` creates statistical summaries of two numerical vectors that are formula aware.

Usage

```
aggregatingFunction2(fun)
```

Arguments

`fun`	a function that takes two numeric vectors and computes a summary statistic, returning a numeric vector of length 1.

Details

This was designed to support functions like `cov()` which can be used to compute numerical summaries from two numeric vectors. The logic of the resulting function is this: 1) If the first two arguments are both formulas, then those formulas are evaluated (with `data`) to compute the covariance; (2) If the first argument is a formula, and the second is `NULL`, then the left and ride sides of the formula and `data` are used to create the vectors passed to `fun`; (3) Else everything is simply passed to `fun` for evaluation.

Value

a function that generalizes `fun` to handle a formula/data frame interface.

Note

Earlier versions of this function supported a "bare name + data frame" interface. This functionality has been removed since it was (a) ambiguous in some cases, (b) unnecessary, and (c) difficult to maintain.

Examples

```
if(require(mosaicData)) {
  foo <- aggregatingFunction2(stats::cor)
  foo(length ~ width, data = KidsFeet)
  stats::cor(KidsFeet$length, KidsFeet$width)
}
```

`as.xtabs` *Convert objects to xtabs format*

Description

Convert a data frame or a matrix into an `xtabs` object.

Usage

```
as.xtabs(x, ...)

## S3 method for class 'data.frame'
as.xtabs(x, rowvar = NULL, colvar = NULL, labels = 1, ...)

## S3 method for class 'matrix'
as.xtabs(x, rowvar = NULL, colvar = NULL, ...)
```

Arguments

`x`	object (typically a data frame) to be converted to `xtabs` format
`...`	additional arguments to be passed to or from methods.
`rowvar`	name of the row variable as character string
`colvar`	name of the column variable as character string
`labels`	column of data frame that contains the labels of the row variable.

Details

The intended use is to convert a two-way contingency table stored in a data frame or a matrix into an `xtabs` object.

Value

An `xtabs` object.

Examples

```
# example from example(fisher.test)
df <- data.frame( X=c('Tea','Milk'), Tea=c(3,1), Milk=c(1,3) )
xt <- as.xtabs(df, rowvar="Guess", colvar="Truth"); xt
if (require(vcd)) { mosaic(xt) }
```

---

ashplot *Average Shifted Histograms*

Description

An ASH plot is the average over all histograms of a fixed bin width.

Usage

```
ashplot(
  x,
  data = data,
  ...,
  width = NULL,
  adjust = NULL,
  panel = panel.ashplot,
  prepanel = prepanel.default.ashplot
)

prepanel.default.ashplot(x, darg, groups = NULL, subscripts = TRUE, ...)

panel.ashplot(
  x,
  darg = list(),
  plot.points = FALSE,
  ref = FALSE,
  groups = NULL,
  jitter.amount = 0.01 * diff(current.panel.limits()$ylim),
  type = "p",
  ...,
  identifier = "ash"
)
```

Arguments

`x`	A formula or numeric vector.
`data`	A data frame.
`...`	Additional arguments passed to panel and prepanel functions or `data`, a data frame in which to find the variables used for the plot.
`width`	The histogram bin width.
`adjust`	A numeric adjustment to `width`. Primarily useful when `width` is not specified. Increasing `adjust` makes the plot smoother.
`panel`	A panel function.
`prepanel`	A prepanel function.
`darg`	a list of arguments for the function computing the ASH.

`groups`	as in other lattice plots
`subscripts`	as in other lattice prepanel functions
`plot.points`	One of `TRUE`, `FALSE`, `"jitter"`, or `"rug"`
`ref`	a logical indicating whether a reference line should be displayed
`jitter.amount`	when `plot.points="jitter"`, the value to use as the amount argument to `jitter()`.
`type`	type argument used to plot points, if requested. This is not expected to be useful, it is available mostly to protect a `type` argument, if specified, from affecting the display of the ASH.
`identifier`	A character string that is prepended to the names of i grobs that are created by this panel function.

Examples

```
ashplot( ~age | substance, groups = sex, data = HELPrct)
```

`bargraph` *Create bar graphs from raw data*

Description

`lattice::barchart()` from the `lattice` package makes bar graphs from pre-tabulated data. Raw data can be tabulated using `xtabs()`, but the syntax is unusual compared to the other lattice plotting functions. `bargraph` provides an interface that is consistent with the other `lattice` functions.

Usage

```
bargraph(
  x,
  data = parent.frame(),
  groups = NULL,
  horizontal = FALSE,
  origin = 0,
  ylab = ifelse(horizontal, "", type),
  xlab = ifelse(horizontal, type, ""),
  type = c("count", "frequency", "proportion", "percent"),
  auto.key = TRUE,
  scales = list(),
  ...
)
```

Arguments

`x`	a formula describing the plot
`data`	a data frame in which the formula `x` is evaluated
`groups`	a variable or expression used for grouping. See `lattice::barchart()`.
`horizontal`	a logical indicating whether bars should be horizontal
`origin`	beginning point for bars. For the default behavior used by `lattice::barchart()` set `origin` to `NULL`, but 0 is often a better default. If 0 is not good, perhaps you should use a different kind of plot as the results may be misleading.
`ylab`	a character vector of length one used for the y-axis label
`xlab`	a character vector of length one used for the x-axis label
`type`	one of `"frequency"`, `"count"`, `"percent"`, or `"proportion"` indicating what type of scale to use. Unique prefixes are sufficient.
`auto.key`	a logical expression indicating whether a legend should be automatically produced
`scales`	is a list determining how the x- and y-axes are drawn
`...`	additional arguments passed to `lattice::barchart()`

Details

`bargraph(formula, data=data, ...)` works by creating a new data frame from `xtabs(formula, data=data)` and then calling `lattice::barchart()` using modified version of the formula and this new data frame as inputs. This has implications on, for example, conditional plots where one desires to condition on some expression that will be evaluated in `data`. This typically does not work because the required variables do not exist in the output of `xtabs`. One solution is to first add a new variable to `data` first and then to condition using this new variable. See the examples.

Value

a trellis object describing the plot

See Also

`lattice::barchart()`

Examples

```
if (require(mosaicData)) {
data(HELPrct)
bargraph( ~ substance, data = HELPrct)
bargraph( ~ substance, data = HELPrct, horizontal = TRUE)
bargraph( ~ substance | sex, groups = homeless, auto.key = TRUE, data = HELPrct)
bargraph( ~ substance, groups = homeless, auto.key=TRUE,
            data = HELPrct |> filter(sex == "male"))
HELPrct2 <- mutate(HELPrct, older = age > 40)
bargraph( ~ substance | older, data = HELPrct2)
}
```

`binom.test` *Exact Tests for Proportions*

Description

The `binom.test()` function performs an exact test of a simple null hypothesis about the probability of success in a Bernoulli experiment from summarized data or from raw data. The mosaic `binom.test` provides wrapper functions around the function of the same name in **stats**. These wrappers provide an extended interface (including formulas).

Usage

```
binom.test(
  x,
  n = NULL,
  p = 0.5,
  alternative = c("two.sided", "less", "greater"),
  conf.level = 0.95,
 ci.method = c("Clopper-Pearson", "binom.test", "Score", "Wilson", "prop.test", "Wald",
    "Agresti-Coull", "Plus4"),
  data = NULL,
  success = NULL,
  ...
)
```

Arguments

`x`	count of successes, length 2 vector of success and failure counts, a formula, or a character, numeric, or factor vector containing raw data.
`n`	sample size (successes + failures) or a data frame (for the formula interface)
`p`	probability for null hypothesis
`alternative`	type of alternative hypothesis
`conf.level`	confidence level for confidence interval
`ci.method`	a method to use for computing the confidence interval (case insensitive and may be abbreviated). See details below.
`data`	a data frame (if missing, `n` may be a data frame)
`success`	level of variable to be considered success. All other levels are considered failure.
`...`	additional arguments (often ignored)

Details

`binom.test()` is a wrapper around `stats::binom.test()` from the stats package to simplify its use when the raw data are available, in which case an extended syntax for `binom.test()` is provided. See the examples.

Also, five confidence interval methods are provided: * `"Clopper-Pearson", "binom.test"`: This is the interval produced when using `stats::binom.test()` from the stats package. It guarantees a coverage rate at least as large as the nominal coverage rate, but may produce wider intervals than some of the methods below, which may either under- or over-cover depending on the data.

- ‘"Score", "Wilson", "prop.test": This is the usual method used by `stats::prop.test()` and is computed by inverting p-values from score tests. It is often attributed to Edwin Wilson. If specified with `"prop.test"`, the continuity correction is applied (as is the default in `prop.test()`), else the continuity correction is not applied.
    - `"Wald"` This is the interval traditionally taught in entry level statistics courses. It uses the sample proportion to estimate the standard error and uses normal theory to determine how many standard deviations to add and/or subtract from the sample proportion to determine an interval.
    - \"Agresti-Coull"‘ This is the Wald method after setting $n' = n + z^2$ and $p' = (x + z^2/2)/n$' and using $x' = n'p'$ and n' in place of x and n.
    - `"Plus4"` This is Wald after adding in two artificial success and two artificial failures. It is nearly the same as the Agresti-Coull method when the confidence level is 95%. since z^2 is approximately 4 and $z^2/2$ is approximately 2.

Value

an object of class `htest`

Note

When `x` is a 0-1 vector, 0 is treated as failure and 1 as success. Similarly, for a logical vector `TRUE` is treated as success and `FALSE` as failure.

See Also

`prop.test()`, `stats::binom.test()`

Examples

```
# Several ways to get a confidence interval for the proportion of Old Faithful
# eruptions lasting more than 3 minutes.
data(faithful)
binom.test(faithful$eruptions > 3)
binom.test(97, 272)
binom.test(c(97, 272-97))
faithful$long <- faithful$eruptions > 3
binom.test(faithful$long)
binom.test(resample(1:4, 400), p=.25)
binom.test(~ long, data = faithful)
binom.test(~ long, data = faithful, ci.method = "Wald")
binom.test(~ long, data = faithful, ci.method = "Plus4")
with(faithful, binom.test(~long))
with(faithful, binom.test(long))
```

Broyden *Multi-Dimensional Root Finding*

Description

Implementation of Broyden’s root finding function to numerically compute the root of a system of nonlinear equations

Usage

```
Broyden(system, vars, x = 0, tol = .Machine$double.eps^0.4, maxiters = 10000)
```

Arguments

`system`	A list of functions
`vars`	A character string list of variables that appear in the functions
`x`	A starting vector
`tol`	The tolerance for the function specifying how precise it will be
`maxiters`	maximum number of iterations.

---

cdist *Central portion of a distribution*

Description

This function determines the critical values for isolating a central portion of a distribution with a specified probability. This is designed to work especially well for symmetric distributions, but it can be used with any distribution.

Usage

```
cdist(
  dist = "norm",
  p,
  plot = TRUE,
  verbose = FALSE,
  invisible = FALSE,
  digits = 3L,
  xlim = NULL,
  ylim = NULL,
  resolution = 500L,
  return = c("values", "plot"),
  pattern = c("rings", "stripes"),
  ...,
```

```
  refinements = list()
)

xcgamma(
  p,
  shape,
  rate = 1,
  scale = 1/rate,
  lower.tail = TRUE,
  log.p = FALSE,
  ...
)

xct(p, df, ncp, lower.tail = TRUE, log.p = FALSE, ...)

xcchisq(p, df, ncp = 0, lower.tail = TRUE, log.p = FALSE, ...)

xcf(p, df1, df2, lower.tail = TRUE, log.p = FALSE, ...)

xcbinom(p, size, prob, lower.tail = TRUE, log.p = FALSE, ...)

xcpois(p, lambda, lower.tail = TRUE, log.p = FALSE, ...)

xcgeom(p, prob, lower.tail = TRUE, log.p = FALSE, ...)

xcnbinom(p, size, prob, mu, lower.tail = TRUE, log.p = FALSE, ...)

xcbeta(p, shape1, shape2, ncp = 0, lower.tail = TRUE, log.p = FALSE, ...)
```

Arguments

`dist`	a character string naming a distribution family (e.g., "norm"). This will work for any family for which the usual d/p/q functions exist.
`p`	the proportion to be in the central region, with equal proportions in either "tail".
`plot`	a logical indicating whether a plot should be created
`verbose`	a logical indicating whether a more verbose output value should be returned.
`invisible`	a logical
`digits`	the number of digits desired
`xlim`	x limits. By default, these are chosen to show the central 99.8\ of the distribution.
`ylim`	y limits
`resolution`	number of points used for detecting discreteness and generating plots. The default value of 5000 should work well except for discrete distributions that have many distinct values, especially if these values are not evenly spaced.
`return`	If `"plot"`, return a plot. If `"values"`, return a vector of numerical values.
`pattern`	One of `"stripes"` or `"rings"`. In the latter case, pairs of regions (from the outside to the inside) are grouped together for coloring and probability calculation.

`...`	additional arguments passed to the distribution functions. Typically these specify the parameters of the particular distribution desired. See the examples.
`refinements`	A list of refinements to the plot. See `ggformula::gf_refine()`.
`shape, scale`	shape and scale parameters. Must be positive, `scale` strictly.
`rate`	an alternative way to specify the scale.
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \le x]$, otherwise, $P[X > x]$.
`log.p`	A logical indicating whether probabilities should be returned on the log scale.
`df`	degrees of freedom (> 0, maybe non-integer). `df = Inf` is allowed.
`ncp`	non-centrality parameter δ; currently except for `rt()`, only for `abs(ncp) <= 37.62`. If omitted, use the central t distribution.
`df1, df2`	degrees of freedom. `Inf` is allowed.
`size`	number of trials (zero or more).
`prob`	probability of success on each trial.
`lambda`	vector of (non-negative) means.
`mu`	alternative parametrization via mean: see 'Details'.
`shape1, shape2`	non-negative parameters of the Beta distribution.

Value

a pair of numbers indicating the upper and lower bounds, unless `verbose` is `TRUE`, in which case a 1-row data frame is returned containing these bounds, the central probability, the tail probabilities, and the name of the distribution.

Note

This function is still experimental and changes the input or output formats are possible in future versions of the package.

Examples

```
cdist( "norm", .95)
cdist( "t", c(.90, .95, .99), df=5)
cdist( "t", c(.90, .95, .99), df=50)
# plotting doesn't work well when the parameters are not constant
cdist( "t", .95, df=c(3,5,10,20), plot = FALSE)
cdist( "norm", .95, mean=500, sd=100 )
cdist( "chisq", c(.90, .95), df=3 )
# CI
x <- rnorm(23, mean = 10, sd = 2)
cdist("t", p = 0.95, df=22)
mean(x) + cdist("t", p = 0.95, df=22) * sd(x) / sqrt(23)
confint(t.test(x))
cdist("t", p = 0.95, df=22, verbose = TRUE)
```

`chisq` *Extract Chi-squared statistic*

Description

Extract Chi-squared statistic

Usage

```
chisq(x, ...)

## S3 method for class 'htest'
chisq(x, ...)

## S3 method for class 'table'
chisq(x, correct = FALSE, ...)

## Default S3 method:
chisq(x, correct = FALSE, ...)
```

Arguments

`x`	An object of class `"htest"` a coming from a Chi-squared test, an object of class `"table"`, or the inputs to `tally()`.
`...`	additional arguments passed on to `tally` or `chisq.test`.
`correct`	a logical indicating whether a continuity correction should be applied.

See Also

`after_stat()`

Examples

```
if(require(mosaicData)) {
  Mites.table <- tally( ~ outcome + treatment, data=Mites )
  Mites.table
  chisq.test(Mites.table)
  chisq(Mites.table)
  chisq(chisq.test(Mites.table))
  ## Randomization test.  Increase replications to decrease Monte Carlo error.
  do(3) * chisq( tally( ~ outcome + shuffle(treatment),  data=Mites ) )
  Mites.rand <- do(1000) * chisq( tally( ~ outcome + shuffle(treatment),  data=Mites ) )
  tally( ~(X.squared >= chisq(Mites.table)), data=Mites.rand, format="proportion")
}
```

`CIAdata` *Return a dataset based on the CIA World Factbook*

Description

This function can be used in two different ways. Without an argument, it returns a reference table that includes information about all the CIA World Factbook tables that are available through this function. Note the `Name` column that indicates a unique name for each available dataset. If this name is passed as an argument to the function, the function will return the corresponding dataset.

Usage

```
CIAdata(name = NULL)
```

Arguments

`name`	An optional parameter specifying the name of the desired dataset. If multiple names are given, a merge will be attempted on the individual data sets.

Examples

```
## Not run:
head(CIAdata())
Population <- CIAdata("pop")
nrow(Population)
head(Population)

PopArea <-
  CIAdata(c("pop","area")) |>
  mutate(density = pop / area)
nrow(PopArea)
head(PopArea)
PopArea |>
  filter(!is.na(density)) |>
  arrange(density) |>
  tail()

## End(Not run)
```

`CIsim` *Compute confidence intervals from (multiple) simulated data sets*

Description

This function automates the calculation of coverage rates for exploring the robustness of confidence interval methods.

Usage

```
CIsim(
  n,
  samples = 100,
  rdist = rnorm,
  args = list(),
  plot = if (samples <= 200) "draw" else "none",
  estimand = 0,
  conf.level = 0.95,
  method = t.test,
  method.args = list(),
  interval = function(x) {
     do.call(method, c(list(x, conf.level = conf.level),
    method.args))$conf.int
 },
  estimate = function(x) {
     do.call(method, c(list(x, conf.level = conf.level),
    method.args))$estimate
 },
  verbose = TRUE
)
```

Arguments

`n`	size of each sample
`samples`	number of samples to simulate
`rdist`	function used to draw random samples
`args`	arguments required by `rdist`
`plot`	one of `"print"`, `"return"`, `"horizontal"`, or `"none"` describing whether a plot should be printed, returned, printed with horizontal intervals, or not generated at all.
`estimand`	true value of the parameter being estimated
`conf.level`	confidence level for intervals
`method`	function used to compute intervals. Standard functions that produce an object of class `htest` can be used here.
`method.args`	arguments required by `method`
`interval`	a function that computes a confidence interval from data. Function should return a vector of length 2.
`estimate`	a function that computes an estimate from data
`verbose`	print summary to screen?

Value

A data frame with variables `lower`, `upper`, `estimate`, `cover` ('Yes' or 'No'), and `sample` is returned invisibly. See the examples for a way to use this to display the intervals graphically.

Examples

```
# 1000 95% intervals using t.test; population is N(0,1)
CIsim(n = 10, samples = 1000)
# this time population is Exp(1); fewer samples, so we get a plot
CIsim(n = 10, samples = 100, rdist = rexp, estimand = 1)
# Binomial treats 1 like success, 0 like failure
CIsim(n = 30, samples = 100, rdist = rbinom, args = list(size = 1, prob = .7),
       estimand = .7, method = binom.test, method.args = list(ci = "Plus4"))
```

`cnorm` *Central Probability in a Normal or T Distribution*

Description

These versions of the quantile functions take a vector of *central* probabilities as its first argument.

Usage

```
cnorm(p, mean = 0, sd = 1, log.p = FALSE, side = c("both", "upper", "lower"))

ct(p, df, ncp, log.p = FALSE, side = c("upper", "lower", "both"))
```

Arguments

`p`	vector of probabilities.
`mean`	vector of means.
`sd`	vector of standard deviations.
`log.p`	logical. If TRUE, uses the log of probabilities.
`side`	One of "upper", "lower", or "both" indicating whether a vector of upper or lower quantiles or a matrix of both should be returned.
`df`	degrees of freedom (> 0, maybe non-integer). `df = Inf` is allowed.
`ncp`	non-centrality parameter δ; currently except for `rt()`, only for `abs(ncp) <= 37.62`. If omitted, use the central t distribution.

See Also

`stats::qnorm()`, `cdist()`

Examples

```
qnorm(.975)
cnorm(.95)
xcnorm(.95)
xcnorm(.95, verbose = FALSE, return = "plot") |>
  gf_refine(
  scale_fill_manual( values = c("navy", "limegreen")),
  scale_color_manual(values = c("black", "black")))
cnorm(.95, mean = 100, sd = 10)
xcnorm(.95, mean = 100, sd = 10)
```

`compareMean` *Defunct functions*

Description

The following functions were once a part of the mosaic pacakge but have been removed. In some cases, an alternative is available and is suggested if you attempt to execute the function.

Usage

```
compareMean(...)

compareProportion(...)

deltaMethod(...)

gwm(...)

r.squared(...)

mm(...)

perctable(...)

proptable(...)

xhistogram(...)
```

Arguments

`...`	arguments, ignored since the function is defunct

confint *Confidence interval methods for output of resampling*

Description

Methods for `confint` to compute confidence intervals on numerical vectors and numerical components of data frames.

Usage

```
## S3 method for class 'numeric'
confint(
  object,
  parm,
  level = 0.95,
  ...,
  method = "percentile",
  margin.of.error = "stderr" %in% method == "stderr"
)

## S3 method for class 'do.tbl_df'
confint(
  object,
  parm,
  level = 0.95,
  ...,
  method = "percentile",
  margin.of.error = "stderr" %in% method,
  df = NULL
)

## S3 method for class 'do.data.frame'
confint(
  object,
  parm,
  level = 0.95,
  ...,
  method = "percentile",
  margin.of.error = "stderr" %in% method,
  df = NULL
)

## S3 method for class 'data.frame'
confint(object, parm, level = 0.95, ...)

## S3 method for class 'summary.lm'
confint(object, parm, level = 0.95, ...)
```

Arguments

`object`	and R object
`parm`	a vector of parameters
`level`	a confidence level
`...`	additional arguments
`method`	a character vector of methods to use for creating confidence intervals. Choices are "percentile" (or "quantile") which is the default, "stderr" (or "se"), "bootstrap-t", and "reverse" (or "basic"))
`margin.of.error`	if true, report intervals as a center and margin of error.
`df`	degrees for freedom. This is required when `object` was produced using `link{do}` when using the standard error to compute the confidence interval since typically this information is not recorded in these objects. The default (`Inf`) uses a normal critical value rather than a one derived from a t-distribution.

Details

The methods of producing confidence intervals from bootstrap distributions are currently quite naive. In particular, when using the standard error, assistance may be required with the degrees of freedom, and it may not be possible to provide a correct value in all situations. None of the methods include explicit bias correction. Let q_a be the a quantile of the bootstrap distribution, let t_a, df be the a quantile of the t distribution with df degrees of freedom, let SE_b be the standard deviation of the bootstrap distribution, and let $\hat{\theta}$ be the estimate computed from the original data. Then the confidence intervals with confidence level $1 - 2a$ are

quantile (q_a, q_{1-a})

reverse $(2\hat{\theta} - q_{1-a}, 2\hat{\theta} - q_a)$

stderr $(\hat{\theta} - t_{1-a,df} SE_b, \hat{\theta} + t_{1-a,df} SE_b)$. When `df` is not provided, at attempt is made to determine an appropriate value, but this should be double checked. In particular, missing data an lead to unreliable results.

The bootstrap-t confidence interval is computed much like the reverse confidence interval but the bootstrap t distribution is used in place of a theoretical t distribution. This interval has much better properties than the reverse (or basic) method, which is here for comparison purposes only and is not recommended. The t-statistic is computed from a mean, a standard deviation, a sample size which much be named `"mean"`, `"sd"`, and `"n"` as they are when using `favstats()`.

Value

When applied to a data frame, returns a data frame giving the confidence interval for each variable in the data frame using `t.test` or `binom.test`, unless the data frame was produced using `do`, in which case it is assumed that each variable contains resampled statistics that serve as an estimated sampling distribution from which a confidence interval can be computed using either a central proportion of this distribution or using the standard error as estimated by the standard deviation of the estimated sampling distribution. For the standard error method, the user must supply the correct degrees of freedom for the t distribution since this information is typically not available in the output of `do()`.

When applied to a numerical vector, returns a vector.

References

Tim C. Hesterberg (2015): What Teachers Should Know about the Bootstrap: Resampling in the Undergraduate Statistics Curriculum, The American Statistician, https://www.tandfonline.com/doi/full/10.1080/00031305.2015.1089789.

Examples

```
if (require(mosaicData)) {
  bootstrap <- do(500) * diffmean( age ~ sex, data = resample(HELPrct) )
  confint(bootstrap)
  confint(bootstrap, method = "percentile")
  confint(bootstrap, method = "boot")
  confint(bootstrap, method = "se", df = nrow(HELPrct) - 1)
  confint(bootstrap, margin.of.error = FALSE)
  confint(bootstrap, margin.of.error = TRUE, level = 0.99,
    method = c("se", "perc") )

  # bootstrap t method requires both mean and sd
  bootstrap2 <- do(500) * favstats(resample(1:10))
  confint(bootstrap2, method = "boot")
}
lm(width ~ length * sex, data = KidsFeet) |>
  summary() |>
  confint()
```

`confint.htest` *Extract summary statistics*

Description

Extract confidence intervals, test statistics or p-values from an `htest` object.

Usage

```
## S3 method for class 'htest'
confint(object, parm, level, ...)

pval(x, ...)

## S3 method for class 'htest'
pval(x, digits = 4, verbose = FALSE, ...)

stat(x, ...)
```

```
## S3 method for class 'htest'
stat(x, ...)

## S3 method for class 'uneval'
stat(x, ...)
```

Arguments

`object`	a fitted model object or an htest object.
`parm`	a specification of which parameters are to be given confidence intervals, either a vector of numbers or a vector of names. If missing, all parameters are considered.
`level`	the confidence level required.
`...`	Additional arguments.
`x`	An object of class `htest`.
`digits`	number of digits to display in verbose output
`verbose`	a logical

Value

the extracted p-value, confidence interval, or test statistic

Examples

```
confint(t.test(rnorm(100)))
pval(t.test(rnorm(100)))
stat(t.test(rnorm(100)))
confint(var.test(rnorm(10,sd=1), rnorm(20, sd=2)))
pval(var.test(rnorm(10,sd=1), rnorm(20, sd=2)))
if (require(mosaicData)) {
data(HELPrct)
stat(t.test (age ~ shuffle(sex), data=HELPrct))
# Compare to test statistic computed with permuted values of sex.
do(10) * stat(t.test (age ~ shuffle(sex), data=HELPrct))
}
```

`cor_test.formula` *Alternative formula interface for cor.test*

Description

`stats::cor.test()` in **stats** accepts formulas of the shape `~ y + x`. The **mosaic** package allows the use of `y ~ x` as an alternative formula shape.

Usage

```
## S3 method for class 'formula'
cor_test(formula, ...)

cor.test(x, ...)

cor_test(x, ...)

## Default S3 method:
cor_test(x, y, ...)
```

Arguments

`formula`	a formula
`...`	other arguments passed to `stats::cor.test()`.
`x, y`	numeric vectors of data values. x and y must have the same length.

See Also

`stats::cor.test()` in the **stats** package.

Examples

```
# This is an example from example(stats::cor.test) done in old and new style
require(graphics)
cor.test(~ CONT + INTG, data = USJudgeRatings)
cor.test(CONT ~ INTG, data = USJudgeRatings)
```

`cross` *Factor cross products*

Description

Construct a product of factors.

Usage

```
cross(..., sep = ":", drop.unused.levels = FALSE)
```

Arguments

`...`	factors to be crossed.
`sep`	separator between levels
`drop.unused.levels`	should levels that do not appear in cross product be dropped?

Value

a factor

Examples

```
x <- letters[1:3]
y <- c(1,2,1,1,3,1,3)
cross(x, y)
cross(x, y, drop.unused.levels=TRUE)
```

`cull_for_do` *Cull objects used with do()*

Description

The `do()` function facilitates easy replication for randomization tests and bootstrapping (among other things). Part of what makes this particularly useful is the ability to cull from the objects produced those elements that are useful for subsequent analysis. `cull_for_do` does this culling. It is generic, and users can add new methods to either change behavior or to handle additional classes of objects.

Usage

```
cull_for_do(object, ...)
```

Arguments

`object`	an object to be culled
`...`	additional arguments (currently ignored)

Details

When `do(n) * expression` is evaluated, `expression` is evaluated `n` times to produce a list of `n` result objects. `cull_for_do` is then applied to each element of this list to extract from it the information that should be stored. For example, when applied to a object of class `"lm"`, the default `cull_for_do` extracts the coefficients, coefficient of determinism, an the estimate for the variance, etc.

Examples

```
cull_for_do(lm(length ~ width, data = KidsFeet))
do(1) * lm(length ~ width, data = KidsFeet)
```

`deg2rad` *Convert between degrees and radians*

Description

Facilitates conversion between degrees and radians.

Usage

```
deg2rad(x)

rad2deg(x)
```

Arguments

`x`	a numeric vector

Value

a numeric vector

See Also

`latlon2xyz()`, `googleMap()`, and `rgeo()`.

Examples

```
deg2rad(180)
rad2deg(2*pi)
```

`derivedVariable` *Create new variables from logicals*

Description

Utility functions for creating new variables from logicals describing the levels

Usage

```
derivedVariable(
  ...,
  .ordered = FALSE,
  .method = c("unique", "first", "last"),
  .debug = c("default", "always", "never"),
  .sort = c("given", "alpha"),
  .default = NULL,
```

```
  .asFactor = FALSE
)

derivedFactor(..., .asFactor = TRUE)
```

Arguments

`...`	named logical "rules" defining the levels.
`.ordered`	a logical indicating whether the resulting factored should be ordered Ignored if `.asFactor` is `FALSE`.
`.method`	one of `"unique"`, `"first"`, and `"last"`. If `"unique"`, exactly one rule must be `TRUE` for each position. If `"first"`, the first `TRUE` rule defines the level. If `"last"`, the last `TRUE` rule defines the level.
`.debug`	one of `"default"`, `"always"`, and `"never"`, indicating whether debugging information should be printed. If `"default"`, debugging information is printed only when multiple rules give conflicting definitions for some positions.
`.sort`	One of `"given"` (the default) or `"alpha"` or a vector of integers the same length as the number of levels indicating the order in which the levels should appear in the resulting factor. Ignored if `.asFactor` is `FALSE`.
`.default`	character vector of length 1 giving name of default level or `NULL` for no default.
`.asFactor`	A logical indicating whether the returned value should be a factor.

Details

Each logical "rule" corresponds to a level in the resulting variable. If `.default` is defined, an implicit rule is added that is `TRUE` whenever all other rules are `FALSE`. When there are multiple `TRUE` rules for a slot, the first or last such is used or an error is generated, depending on the value of `method`.

`derivedVariable` is designed to be used with `transform()` or `dplyr::mutate()` to add new variables to a data frame. `derivedFactor`() is the same but that the default value for `.asFactor` is `TRUE`. See the examples.

Examples

```
Kf <- mutate(KidsFeet, biggerfoot2 = derivedFactor(
                   dom = biggerfoot == domhand,
                   nondom = biggerfoot != domhand)
                   )
tally( ~ biggerfoot + biggerfoot2, data = Kf)
tally( ~ biggerfoot + domhand, data = Kf)

# Three equivalent ways to define a new variable
# Method 1: explicitly define all levels
modHELP <- mutate(HELPrct, drink_status = derivedFactor(
  abstinent = i1 == 0,
  moderate = (i1>0 & i1<=1 & i2<=3 & sex=='female') |
     (i1>0 & i1<=2 & i2<=4 & sex=='male'),
  highrisk = ((i1>1 | i2>3) & sex=='female') |
```

```
      ((i1>2 | i2>4) & sex=='male'),
    .ordered = TRUE)
  )
  tally( ~ drink_status, data = modHELP)

  # Method 2: Use .default for last level
  modHELP <- mutate(HELPrct, drink_status = derivedFactor(
    abstinent = i1 == 0,
    moderate = (i1<=1 & i2<=3 & sex=='female') |
       (i1<=2 & i2<=4 & sex=='male'),
    .ordered = TRUE,
    .method = "first",
    .default = "highrisk")
  )
  tally( ~ drink_status, data = modHELP)

  # Method 3: use TRUE to catch any fall through slots
  modHELP <- mutate(HELPrct, drink_status = derivedFactor(
    abstinent = i1 == 0,
    moderate = (i1<=1 & i2<=3 & sex=='female') |
       (i1<=2 & i2<=4 & sex=='male'),
    highrisk=TRUE,
    .ordered = TRUE,
    .method = "first"
    )
  )
  tally( ~ drink_status, data = modHELP)
  is.factor(modHELP$drink_status)

  modHELP <- mutate(HELPrct, drink_status = derivedVariable(
    abstinent = i1 == 0,
    moderate = (i1<=1 & i2<=3 & sex=='female') |
       (i1<=2 & i2<=4 & sex=='male'),
    highrisk=TRUE,
    .ordered = TRUE,
    .method = "first"
    )
  )
  is.factor(modHELP$drink_status)
```

`design_plot` *Interactively design plots*

Description

Proves a simple interface to let users interactively design plots in **ggformula**, **lattice**, or **ggplot2**. An option is available to show the code used to create the plot. This can be copied and pasted elsewhere to (into an RMarkdown document, for example) to recreate the plot. Only works in RStudio. Requires the **manipulate** package.

Usage

```
design_plot(
  data,
  format,
  default = format,
  system = system_choices()[1],
  show = FALSE,
  title = "",
  data_text = rlang::expr_deparse(substitute(data)),
  ...
)
```

Arguments

`data`	a data frame containing the variables that might be used in the plot. Note that for maps, the data frame must contain coordinates of the polygons comprising the map and a variable for determining which coordinates are part of the same region. See `sp2df()` for one way to create such a data frame. Typically `merge()` will be used to combine the map data with some auxiliary data to be displayed as fill color on the map, although this is not necessary if all one wants is a map.
`format`	a synonym for `default`.
`default`	default type of plot to create; one of `"scatter"`, `"jitter"`, `"boxplot"`, `"violin"`, `"sina"`, `"histogram"`, `"density"`, `"density (contours)"`, `"density (filled)"`, `"frequency polygon"`, `"xyplot"`, or `"map"`. Unique prefixes suffice.
`system`	which graphics system to use (initially) for plotting (**ggplot2** or **lattice**). A check box will allow on the fly change of plotting system.
`show`	a logical, if `TRUE`, the code will be displayed each time the plot is changed.
`title`	a title for the plot
`data_text`	A text string describing the data. It must be possible to recover the data from this string using `eval()`. Typically users will not need to modify this from the default value.
`...`	additional arguments

Details

Currently maps are only supported in **ggplot2** and not in **lattice**.

Due to an unresolved issue with RStudio, the first time this function is called, and additional plot is created to correctily initialize the mainipulate frameowrk.

Value

Nothing. Used for side effects.

Examples

```
## Not run:
  mtcars2 <-
    mtcars |>
      mutate(
        cyl2 = factor(cyl),
        carb2 = factor(carb),
        shape = c("V-shaped", "straight")[1 + vs],
        gear2 = factor(gear),
        transmission = c("automatic", "manual")[1 + am])
  design_plot(mtcars2)

## End(Not run)
```

`diffmean` *Difference in means and proportions*

Description

Wrappers around `diff(mean(...))` and `diff(prop(...))` that facilitate better naming of the result

Usage

```
diffmean(x, ..., data = parent.frame(), only.2 = TRUE)

diffprop(x, ..., data = parent.frame(), only.2 = TRUE)
```

Arguments

`x, data, ...`	as in `mean()` or `prop()`
`only.2`	a logical indicating whether differences should only be computed between two groups.

Examples

```
if (require(mosaicData)) {
diffprop( homeless ~ sex , data=HELPrct)
do(3) * diffprop( homeless ~ shuffle(sex) , data=HELPrct)
diffmean( age ~ substance, data=HELPrct, only.2=FALSE)
do(3) * diffmean(age ~ shuffle(substance), data=HELPrct, only.2=FALSE)
diffmean( age ~ sex, data=HELPrct)
do(3) * diffmean(age ~ shuffle(sex), data=HELPrct)
}
```

`do` *Do Things Repeatedly*

Description

`do()` provides a natural syntax for repetition tuned to assist with replication and resampling methods.

Usage

```
do(object, ...)

## S3 method for class 'numeric'
do(object, ...)

## Default S3 method:
do(object, ...)

Do(n = 1L, cull = NULL, mode = "default", algorithm = 1, parallel = TRUE)

## S3 method for class 'repeater'
print(x, ...)

## S4 method for signature 'repeater,ANY'
e1 * e2
```

Arguments

`object`	an object
`...`	additional arguments
`n`	number of times to repeat
`cull`	function for culling output of objects being repeated. If NULL, a default culling function is used. The default culling function is currently aware of objects of types `lme`, `lm`, `htest`, `table`, `cointoss`, and `matrix`.
`mode`	target mode for value returned
`algorithm`	a number used to select the algorithm used. Currently numbers below 1 use an older algorithm and numbers >=1 use a newer algorithm which is faster in some situations.
`parallel`	a logical indicating whether parallel computation should be attempted using the **parallel** package (if it is installed and loaded).
`x`	an object created by `do`.
`e1`	an object (in cases documented here, the result of running `do`)
`e2`	an object (in cases documented here, an expression to be repeated)

Value

do returns an object of class `repeater` which is only useful in the context of the operator `*`. See the examples.

Naming

The names used in the object returned from `do()` are inferred from the objects created in each replication. Roughly, this the strategy employed.

- If the objects have names, those names are inherited, if possible.
- If the objects do not have names, but `do()` is used with a simple function call, the name of that function is used. Example: `do(3) * mean(~height, data = Galton)` produces a data frame with a variable named `mean`.
- In cases where names are not easily inferred and a single result is produced, it is named `result`.

To get different names, one can rename the objects as they are created, or rename the result returned from `do()`. Example of the former: `do(3) * c(mean_height = mean(~height, data = resample(Galton)))`.

Note

do is a thin wrapper around `Do` to avoid collision with `dplyr::do()` from the **dplyr** package.

Author(s)

Daniel Kaplan (<`kaplan@macalaster.edu`>) and Randall Pruim (<`rpruim@calvin.edu`>)

See Also

`replicate()`, `set.rseed()`

Examples

```
do(3) * rnorm(1)
do(3) * "hello"
do(3) * 1:4
do(3) * mean(rnorm(25))
do(3) * lm(shuffle(height) ~ sex + mother, Galton)
do(3) * anova(lm(shuffle(height) ~ sex + mother, Galton))
do(3) * c(sample.mean = mean(rnorm(25)))
# change the names on the fly
do(3) * mean(~height, data = resample(Galton))
do(3) * c(mean_height = mean(~height, data = resample(Galton)))
set.rseed(1234)
do(3) * tally( ~sex|treat, data=resample(HELPrct))
set.rseed(1234)  # re-using seed gives same results again
do(3) * tally( ~sex|treat, data=resample(HELPrct))
```

`docFile` *Return the path to a documentation file in a package*

Description

Return the path to a documentation file in a package

Usage

```
docFile(file, package = "mosaic", character.only = FALSE)
```

Arguments

`file`	the name of a file
`package`	the name of a package
`character.only`	a logical. If `TRUE` package names must be specified as character, else names will be converted as a convenience as is `library()` and `library()`.

Value

a character vector specifying the path to the file on the user's system.

`dotPlot` *Dotplots*

Description

A high level function and panel function for producing a variant of a histogram called a dotplot.

Usage

```
dotPlot(x, breaks, ..., panel = panel.dotPlot)

panel.dotPlot(
  x,
  breaks,
  equal.widths = TRUE,
  groups = NULL,
  nint = if (is.factor(x)) nlevels(x) else round(1.3 * log2(length(x)) + 4),
  pch,
  col,
  lty = trellis.par.get("dot.line")$lty,
  lwd = trellis.par.get("dot.line")$lwd,
  col.line = trellis.par.get("dot.line")$col,
  alpha = trellis.par.get("dot.symbol")$alpha,
```

```
    cex = 1,
    type = "count",
    ...
)
```

Arguments

`x`	a vector of values or a formula
`breaks`, `equal.widths`, `groups`, `pch`, `col`, `lty`, `lwd`, `col.line`, `type`, `alpha`	as in `histogram()`
`...`	additional arguments
`panel`	a panel function
`nint`	the number of intervals to use
`cex`	a ratio by which to increase or decrease the dot size

Value

a trellis object

See Also

`histogram()`

Examples

```
if (require(mosaicData)) {
dotPlot( ~ age, data = HELPrct)
dotPlot( ~ age, nint=42, data = HELPrct)
dotPlot( ~ height | voice.part, data = singer, nint = 17,
          endpoints = c(59.5, 76.5), layout = c(4,2), aspect = 1,
          xlab = "Height (inches)")
}
```

`dpqrdist` *Distribution wrapper*

Description

Utility function wrapping up the d/p/q/r distribution functions

Usage

```
dpqrdist(dist, type = c("d", "p", "q", "r"), ...)
```

Arguments

`dist`	a character description of a distribution, for example `"norm"`, `"t"`, or `"chisq"`
`type`	one of `"x"`, `"p"`, `"q"`, or `"r"`
`...`	additional arguments passed on to underlying distribution function. Note that one of `d`, `p`, `q`, or `n` must be a named argument in ...

Examples

```
# 3 random draws from N(1,2)
dpqrdist("norm", "r", n = 3, mean = 1, sd = 2)
# These should all be the same
dpqrdist("norm", "d", x = 0) == dnorm(x = 0)
dpqrdist("norm", "p", q = 0, mean = 1, sd = 2) == pnorm(q = 0, mean = 1, sd = 2)
dpqrdist("norm", "q", p = 0.5, mean = 1, sd = 2) == qnorm(p = 0.5, mean = 1, sd = 2)
```

`expandFun` *Expand the left-hand side of a formula*

Description

Expands the contents of functions used in a formula.

Usage

```
expandFun(formula, ...)
```

Arguments

`formula`	A mathematical expression (see examples and `plotFun()`)
`...`	additional parameters

Value

A list with the new expanded formula and the combined formals

Examples

```
f=makeFun(x^2~x)
expandFun(f(z)~z) #Returns z^2~z
```

`factorize` *Conditionally convert vectors to factors*

Description

A generic function and several instances for creating factors from other sorts of data. The primary use case is for vectors that contain few unique values and might be better considered as factors. When applied to a data frame, this is applied to each variable in the data frame.

Usage

```
factorize(x, ...)

## Default S3 method:
factorize(x, ...)

## S3 method for class 'numeric'
factorize(x, max.levels = 5L, ...)

## S3 method for class 'character'
factorize(x, max.levels = 5L, ...)

## S3 method for class 'data.frame'
factorize(x, max.levels = 5L, ...)

factorise(x, ...)
```

Arguments

`x`	an object
`...`	additional arguments (currently ignored)
`max.levels`	an integer. Only convert if the number of unique values is no more than `max.levels`.

Examples

```
data(KidsFeet, package="mosaicData")
str(KidsFeet)
factorize(KidsFeet$birthyear)
str(factorize(KidsFeet))
# alternative spelling
str(factorise(KidsFeet))
```

`fav_stats` *Some favorite statistical summaries*

Description

Likely you mean to be using `favstats()`. Each of these computes the mean, standard deviation, quartiles, sample size and number of missing values for a numeric vector, but `favstats()` can take a formula describing how these summary statistics should be aggregated across various subsets of the data.

Usage

```
fav_stats(x, ..., na.rm = TRUE, type = 7)
```

Arguments

`x`	numeric vector
`...`	additional arguments (currently ignored)
`na.rm`	boolean indicating whether missing data should be ignored
`type`	an integer between 1 and 9 selecting one of the nine quantile algorithms detailed in the documentation for `stats::quantile()`

Value

A vector of statistical summaries

Examples

```
fav_stats(1:10)
fav_stats(faithful$eruptions)
data(penguins, package = "palmerpenguins")

# Note: this is favstats() rather than fav_stats()
favstats(bill_length_mm ~ species, data = penguins)
```

`fetchData` *Defunct functions now in the fetch package*

Description

These functions have been moved to the fetch package.

Usage

```
fetchData(...)

fetchGapminder1(...)

fetchGapminder(...)

fetchGoogle(...)
```

Arguments

`...`	arguments

`findZeros` *Find zeros of functions*

Description

Compute numerically zeros of a function or simultaneous zeros of multiple functions.

Usage

```
findZeros(
  expr,
  ...,
  xlim = c(near - within, near + within),
  near = 0,
  within = Inf,
  nearest = 10,
  npts = 1000,
  iterate = 1,
  sortBy = c("byx", "byy", "radial")
)

## S3 method for class 'formula'
solve(
  form,
  ...,
  near = 0,
  within = Inf,
  nearest = 10,
  npts = 1000,
  iterate = 1,
  sortBy = c("byx", "byy", "radial")
)
```

Arguments

`expr`	A formula. The right side names the variable with respect to which the zeros should be found. The left side is an expression, e.g. `sin(x) ~ x`. All free variables (all but the variable on the right side) named in the expression must be assigned a value via `\ldots`
`...`	Formulas corresponding to additional functions to use in simultaneous zero finding and/or specific numerical values for the free variables in the expression.
`xlim`	The range of the dependent variable to search for zeros. `Inf` is a legitimate value, but is interpreted in the numerical sense as the non-Inf largest floating point number. This can also be specified replacing `x` with the name of the variable. See the examples.
`near`	a value near which zeros are desired
`within`	only look for zeros at least this close to near. `near` and `within` provide an alternative to using `xlim` to specify the search space.
`nearest`	the number of nearest zeros to return. Fewer are returned if fewer are found.
`npts`	How many sub-intervals to divide the `xlim` into when looking for candidates for zeros. The default is usually good enough. If `Inf` is involved, the intervals are logarithmically spaced up to the largest finite floating point number. There is no guarantee that all the roots will be found.
`iterate`	maximum number of times to iterate the search. Subsequent searches take place with the range of previously found zeros. Choosing a large number here is likely to kill performance without improving results, but a value of 1 (the default) or 2 works well when searching in `c(-Inf,Inf)` for a modest number of zeros near `near`.
`sortBy`	specifies how the zeros found will be sorted. Options are 'byx', 'byy', or 'radial'.
`form`	Expression to be solved

Details

Searches numerically using `uniroot`.

Uses findZerosMult of findZeros to solve the given expression

Value

A dataframe of zero or more numerical values. Plugging these into the expression on the left side of the formula should result in values near zero.

a dataframe with solutions to the expression.

Author(s)

Daniel Kaplan (`<kaplan@macalester.edu>`)

Cecylia Bocovich

Examples

```
findZeros( sin(t) ~ t, xlim=c(-10,10) )
# Can use tlim or t.lim instead of xlim if we prefer
findZeros( sin(t) ~ t, tlim=c(-10,10) )
findZeros( sin(theta) ~ theta, near=0, nearest=20)
findZeros( A*sin(2*pi*t/P) ~ t, xlim=c(0,100), P=50, A=2)
# Interval of a normal at half its maximum height.
findZeros( dnorm(x,mean=0,sd=10) - 0.5*dnorm(0,mean=0,sd=10) ~ x )
# A pathological example
# There are no "neareset" zeros for this function.  Each iteration finds new zeros.
f <- function(x) { if (x==0) 0 else sin(1/x) }
findZeros( f(x) ~ x, near=0 )
# Better to look nearer to 0
findZeros( f(x) ~ x, near=0, within=100 )
findZeros( f(x) ~ x, near=0, within=100, iterate=0 )
findZeros( f(x) ~ x, near=0, within=100, iterate=3 )
# Zeros in multiple dimensions (not run: these take a long time)
# findZeros(x^2+y^2+z^2-5~x&y&z, nearest=3000, within = 5)
# findZeros(x*y+z^2~z&y&z, z+y~x&y&z, npts=10)
solve(3*x==3~x)

# plot out sphere (not run)
# sphere = solve(x^2+y^2+z^2==5~x&y&z, within=5, nearest=1000)
# cloud(z~x+y, data=sphere)
```

`findZerosMult` *Find the zeros of a function of two or more variables*

Description

Compute numerically zeros of a function of two or more variables. All free variables (all but the variable on the right side) named in the expression must be assigned a value via `\ldots`

Usage

```
findZerosMult(..., npts = 10, rad = 5, near = 0, sortBy = "byx")
```

Arguments

`...`	arguments for values NOTE: if the system has more than one equation and the rhs variables do not match up, there will be an error.
`npts`	number of desired zeros to return
`rad`	radius around near in which to look for zeros
`near`	center of search for zeros
`sortBy`	options for sorting zeros for plotting. Options are 'byx', 'byy' and 'radial'. The default value is 'byx'.

Details

sorts points in the domain according to the sign of the function value at respective points. Use continuity and uniroot to find zeros between points of opposite signs. Returns any number of points which may be sorted and plotted according to x, y, or radial values.

Value

A data frame of numerical values which should all result in a value of zero when input into original function

Author(s)

Cecylia Bocovich

Examples

```
findZerosMult(a*x^2-8~a&x, npts = 50)
findZerosMult(a^2+x^2-8~a&x, npts = 100, sortBy='radial')
## Not run: findZerosMult(a^2+x^2-8~a&x, npts = 1000, sortBy='radial')
```

`fitModel` *Fit a nonlinear least squares model*

Description

Allows you to specify a formula with parameters, along with starting guesses for the parameters. Refines those guesses to find the least-squares fit.

Usage

```
fitModel(formula, data = parent.frame(), start = list(), ...)

model(object, ...)

## S3 method for class 'nlsfunction'
model(object, ...)

## S3 method for class 'nlsfunction'
summary(object, ...)

## S3 method for class 'nlsfunction'
coef(object, ...)
```

Arguments

`formula`	formula specifying the model
`data`	dataframe containing the data to be used
`start`	passed as `start` to `nls()`. If and empty list, a simple starting point is used (thus avoiding the usual warning message).
`...`	additional arguments passed to `nls()`
`object`	an R object (typically a the result of fitModel)

Details

Fits a nonlinear least squares model to data. In contrast to linear models, all the parameters (including linear ones) need to be named in the formula. The function returned simply contains the formula together with pre-assigned arguments setting the parameter value. Variables used in the fitting (as opposed to parameters) are unassigned arguments to the returned function.

Value

a function

Note

This doesn't work with categorical explanatory variables. Also, this does not work with synthetic data that fit the model perfectly. See `link{nls}` for details.

See Also

`linearModel()`, `nls()`

Examples

```
if (require(mosaicData)) {
f <- fitModel(temp ~ A+B*exp(-k*time), data=CoolingWater, start=list(A=50,B=50,k=1/20))
f(time=50)
coef(f)
summary(f)
model(f)
}
```

`fitSpline` *Fit splines to data*

Description

These functions create mathematical functions from data, using splines.

Usage

```
fitSpline(
  formula,
  data = parent.frame(),
  df = NULL,
  knots = NULL,
  degree = 3,
  type = c("natural", "linear", "cubic", "polynomial"),
  ...
)
```

Arguments

`formula`	a formula. Only one quantity is allowed on the left-hand side, the output quantity
`data`	a data frame in which `formula` is evaluated.
`df`	degrees of freedom (used to determine how many knots should be used)
`knots`	a vector of knots
`degree`	parameter for splines when `type` is `"polynomial"`. 1 is locally linear, 2 is locally quadratic, etc.
`type`	type of splines to use; one of `"linear"`, `"cubic"`, `"natural"` (cubic with linear tails, the default), or `"polynomial"`.
`...`	additional arguments passed to spline basis functions (`splines::ns()` and `splines::bs()`).

Value

a function of the explanatory variable

See Also

`splines::bs()` and `splines::ns()` for the bases used to generate the splines.

Examples

```
f <- fitSpline( weight ~ height, data=women, df=5 )
xyplot( weight ~ height, data=women )
plotFun(f(height) ~ height, add=TRUE)

g <- fitSpline( length ~ width, data = KidsFeet, type='natural', df=5 )
h <- fitSpline( length ~ width, data = KidsFeet, type='linear', df=5 )
xyplot( length ~ width, data = KidsFeet, col='gray70', pch=16)
plotFun(g, add=TRUE, col='navy')
plotFun(h, add=TRUE, col='red')
```

`fortify.hclust` *mosaic tools for clustering*

Description

mosaic tools for clustering

Usage

```
## S3 method for class 'hclust'
fortify(
  model,
  data,
  which = c("segments", "heatmap", "leaves", "labels", "data"),
  k = 1,
  ...
)

## S3 method for class 'hclust'
mplot(
  object,
  data,
  colorize = TRUE,
  k = 1,
  labels = FALSE,
  heatmap = 0,
  enumerate = "white",
  ...
)
```

Arguments

`model`	a model
`data`	a data-like object
`which`	which kind of fortification to compute
`k`	number of clusters
`...`	additional arguments passed on to `link{dendro_data}`
`object`	an object of class `"hclust"`
`colorize`	whether to show clusters in different colors
`labels`	a logical indicating whether labels should be used to identify leaves of the tree.
`heatmap`	the ratio of size of heatmap to size of dendrogram. Use `0` or `FALSE` to omit the heatmap.
`enumerate`	a color used for numbers within heatmap. Use `"transparent"` to hide.

Examples

```
KidsFeet |> select(-name, -birthmonth) |> rescale() -> KidsFeet2
  M <- dist(KidsFeet2)
  Cl <- hclust(M)
  fortify(Cl, k=5) |> head(3)
  fortify(Cl, which="heatmap", data=KidsFeet2) |> head(3)
  fortify(Cl, which="data", data=KidsFeet2) |> head(3)
  fortify(Cl, which="labels") |> head(3)
  mplot(Cl, data=KidsFeet2, k=4, heatmap=2)
  mplot(Cl, data=KidsFeet2, k=4, heatmap=0.5, enumerate="transparent")
  mplot(Cl, data=KidsFeet2, k=4, heatmap=2, type="triangle")
  mplot(Cl, data=KidsFeet2, k=4, heatmap=0, type="triangle")
```

`fortify.summary.lm` *Extract data from R objects*

Description

Extract data from R objects

Usage

```
## S3 method for class 'summary.lm'
fortify(model, data = NULL, level = 0.95, ...)

## S3 method for class 'summary.glm'
fortify(model, data = NULL, level = 0.95, ...)

## S3 method for class 'TukeyHSD'
fortify(model, data, order = c("asis", "pval", "difference"), ...)
```

Arguments

`model`	an R object
`data`	original data set, if needed
`level`	confidence level
`...`	additional arguments
`order`	one of `"pval"`, `"diff"`, or `"asis"` determining the order of the `pair` factor, which determines the order in which the differences are displayed on the plot.

`freqpoly` *Turn histograms into frequency polygons*

Description

Turn histograms into frequency polygons

Usage

```
freqpoly(x, plot = TRUE, ...)

hist2freqpolygon(hist)

## S3 method for class 'freqpolygon'
plot(
  x,
  freq = equidist,
  col = graphics::par("fg"),
  lty = NULL,
  lwd = 1,
  main = paste("Frequency polygon of", paste(x$xname, collapse = "\n")),
  sub = NULL,
  xlab = x$xname,
  ylab,
  xlim = range(x$x),
  ylim = NULL,
  axes = TRUE,
  labels = FALSE,
  add = FALSE,
  ann = TRUE,
  ...
)
```

Arguments

`x`	a vector of values for which a frequency polygon is desired.
`plot`	a logical indicating if a plot should be generated.
`...`	additional arguments passed on to `hist()`.
`hist`	a histogram object produced by `link{hist}()`.
`freq`	A logical indicating whether the vertical scale should be frequency (count).
`col`	A color for the frequency polygon.
`lty`	An integer indicating the line type.
`lwd`	An integer indicating the line width.
`main`	A title for the plot.

`sub`	A sub-title for the plot.
`xlab`	Label for the horizontal axis.
`ylab`	Label for the vertical axis.
`xlim`	A numeric vector of length 2.
`ylim`	A numeric vector of length 2.
`axes`	A logical indicating whether axes should be drawn.
`labels`	A logical indicating whether labels should be printed or a character vector of labels to add.
`add`	A logical indicating whether the plot should be added to the current plot
`ann`	A logical indicating whether annotations (titles and axis titles) should be plotted.

Value

An object of class `"freqpoly"` (invisibly). Additionally, if `plot` is `TRUE`, a plot is generated.

Examples

```
freqpoly(faithful$eruptions)
bks <- c(0, 1, 1.5, 2, 3, 3.5, 4, 4.5, 5, 7)
hist(faithful$eruptions, breaks = bks)
freqpoly(faithful$eruptions, col = rgb(0,0,1,.5), lwd = 5, breaks = bks, add = TRUE)
```

`freqpolygon` *Frequency Polygons*

Description

Frequency polygons are an alternative to histograms that make it simpler to overlay multiple distributions.

Usage

```
freqpolygon(
  x,
  ...,
  panel = "panel.freqpolygon",
  prepanel = "prepanel.default.freqpolygon"
)

prepanel.default.freqpolygon(
  x,
  darg = list(),
  plot.points = FALSE,
  ref = FALSE,
```

```
    groups = NULL,
    subscripts = TRUE,
    jitter.amount = 0.01 * diff(current.panel.limits()$ylim),
    center = NULL,
    nint = NULL,
    breaks = NULL,
    width = darg$width,
    type = "density",
    ...
  )

  panel.freqpolygon(
    x,
    darg = list(),
    plot.points = FALSE,
    ref = FALSE,
    groups = NULL,
    weights = NULL,
    jitter.amount = 0.01 * diff(current.panel.limits()$ylim),
    type = "density",
    breaks = NULL,
    nint = NULL,
    center = NULL,
    width = darg$width,
    gcol = trellis.par.get("reference.line")$col,
    glwd = trellis.par.get("reference.line")$lwd,
    h,
    v,
    ...,
    identifier = "freqpoly"
  )
```

Arguments

`x`	a formula or a numeric vector
`...`	additional arguments passed on to `histogram()` and `panel`.
`panel`	a panel function
`prepanel`	a prepanel function
`darg`	a list of arguments for the function computing the frequency polygon. This exists primarily for compatibility with `densityplot` and is unlikely to be needed by the end user.
`plot.points`	one of `TRUE`, `FALSE`, `"jitter"`, or `"rug"` indicating how points are to be displayed
`ref`	a logical indicating whether a horizontal reference line should be added (roughly equivalent to `h=0`)
`groups`, `weights`, `jitter.amount`, `identifier`	as in `densityplot()` or `histogram()`

`subscripts`	as in other lattice prepanel functions
`center`	center of one of the bins
`nint`	an approximate number of bins for the frequency polygon
`breaks`	a vector of breaks for the frequency polygon bins
`width`	width of the bins
`type`	one of `'density'`, `'percent'`, or `'count'`
`gcol`	color of guidelines
`glwd`	width of guidelines
`h, v`	a vector of values for additional horizontal and vertical lines

Value

a trellis object

Note

This function make use of `histogram` to determine overall layout. Often this works reasonably well but sometimes it does not. In particular, when `groups` is used to overlay multiple frequency polygons, there is often too little head room. In the latter cases, it may be necessary to use `ylim` to determine an appropriate viewing rectangle for the plot.

Examples

```
freqpolygon(~age | substance, data=HELPrct, v=35)
freqpolygon(~age, data=HELPrct, labels=TRUE, type='count')
freqpolygon(~age | substance, data=HELPrct, groups=sex)
freqpolygon(~age | substance, data=HELPrct, groups=sex, ylim=c(0,0.11))
## comparison of histogram and frequency polygon
histogram(~eruptions, faithful, type='density', width=.5)
ladd( panel.freqpolygon(faithful$eruptions, width=.5 ))
```

`FunctionsFromData` *Create function from data*

Description

These functions create mathematical functions from data, by smoothing, splining, or linear combination (fitting). Each of them takes a formula and a data frame as an argument

Usage

```
spliner(formula, data = NULL, method = "fmm", monotonic = FALSE)

connector(formula, data = NULL, method = "linear")

smoother(formula, data, span = 0.5, degree = 2, ...)

linearModel(formula, data, ...)
```

Arguments

`formula`	a formula. Only one quantity is allowed on the left-hand side, the output quantity
`data`	a data frame
`method`	a method for splining. See `spline()`.
`monotonic`	a `TRUE/FALSE` flag specifying whether the spline should respect monotonicity in the data
`span`	parameter to smoother. How smooth it should be.
`degree`	parameter to smoother. 1 is locally linear, 2 is locally quadratic.
`...`	additional arguments to `stats::loess()` or `stats::lm()`

Details

These functions use data to create a mathematical, single-valued function of the inputs. All return a function whose arguments are the variables used on the right-hand side of the formula. If the formula involves a transformation, e.g. `sqrt(age)` or `log(income)`, only the variable itself, e.g. `age` or `income`, is an argument to the function.

`linearModel` takes a linear combination of the vectors specified on the right-hand side. It differs from `project` in that `linearModel` returns a function whereas `project` returns the coefficients. NOTE: An intercept term is not included unless that is explicitly part of the formula with `+1`. This conflicts with the standard usage of formulas as found in `lm`. Another option for creating such functions is to combine `lm()` and `makeFun()`.

`spliner` and `connector` currently work for only one input variable.

See Also

`project()` method for formulas

Examples

```
if (require(mosaicData)) {
data(CPS85)
f <- smoother(wage ~ age, span=.9, data=CPS85)
f(40)
g <- linearModel(log(wage) ~ age + educ + 1, data=CPS85)
g(age=40, educ=12)
# an alternative way to define g (Note: + 1 is the default for lm().)
g2 <- makeFun(lm(log(wage) ~ age + educ, data=CPS85))
g2(age=40, educ=12)
x<-1:5; y=c(1, 2, 4, 8, 8.2)
f1 <- spliner(y ~ x)
f1(x=8:10)
f2 <- connector(x~y)
}
```

`getVarFormula` *Extract data from a data frame using a formula interface*

Description

Uses the full model syntax.

Usage

```
getVarFormula(formula, data = parent.frame(), intercept = FALSE)
```

Arguments

`formula`	a formula. The right-hand side selects variables; the left-hand side, if present, is used to set row names. A `.` on the right-hand side indicates to use all variables not in the LHS.
`data`	a data frame
`intercept`	a logical indicating whether to include the intercept in the model default: FALSE (no intercept)

Examples

```
getVarFormula( ~ wt + mpg, data = mtcars)
```

`googleMap` *Display a point on earth on a Google Map*

Description

Creates a URL for Google Maps for a particular latitude and longitude position. This function has been deprecated due to changes in Google's access policies. Give `leaflet_map()` a try as an alternative.

Usage

```
googleMap(
  latitude,
  longitude,
  position = NULL,
  zoom = 12,
  maptype = c("roadmap", "satellite", "terrain", "hybrid"),
  mark = FALSE,
  radius = 0,
  browse = TRUE,
  ...
)
```

Arguments

`latitude`, `longitude`	vectors of latitude and longitude values
`position`	a data frame containing latitude and longitude positions
`zoom`	zoom level for initial map (1-20)
`maptype`	one of `'roadmap'`, `'satellite'`, `'terrain'`, and `'hybrid'`
`mark`	a logical indicating whether the location should be marked with a pin
`radius`	a vector of radii of circles centered at position that are displayed on the map
`browse`	a logical indicating whether the URL should be browsed (else only returned as a string)
`...`	additional arguments passed to `browseURL`

Value

a string containing a URL. Optionally, as a side-effect, the URL is visited in a browser

See Also

`leaflet_map()`, `deg2rad()`, `latlon2xyz()` and `rgeo()`.

Examples

```
## Not run:
googleMap(40.7566, -73.9863, radius=1)   # Times Square
googleMap(position=rgeo(2), radius=1)    # 2 random locations

## End(Not run)
```

`inferArgs` *Infer arguments*

Description

The primary purpose is for inferring argument settings from names derived from variables occurring in a formula. For example, the default use is to infer limits for variables without having to call them `xlim` and `ylim` when the variables in the formula have other names. Other uses could easily be devised by specifying different `variants`.

Usage

```
inferArgs(
  vars,
  dots,
  defaults = alist(xlim = , ylim = , zlim = ),
  variants = c(".lim", "lim")
)
```

Arguments

`vars`	a vector of variable names to look for
`dots`	a named list of argument values
`defaults`	named list or alist of default values for limits
`variants`	a vector of optional postfixes for limit-specifying variable names

Value

a named list or alist of limits. The names are determined by the names in `defaults`.

If multiple `variants` are matched, the first is used.

Examples

```
inferArgs(c('x','u','t'), list(t=c(1,3), x.lim=c(1,10), u=c(1,3), u.lim=c(2,4)))
inferArgs(c('x','u'), list(u=c(1,3)), defaults=list(xlim=c(0,1), ylim=NULL))
```

`is.wholenumber` *Check for whole number values*

Description

Unlike `is.integer()`, which checks the type of argument is `integer`, this function checks whether the value of the argument is an integer (within a specified tolerance).

Usage

```
is.wholenumber(x, tol = .Machine$double.eps^0.5)
```

Arguments

`x`	a vector
`tol`	a numeric tolerance

Details

This function is borrowed from the examples for `is.integer()`

Value

a logical vector indicating whether `x` has a whole number value

Examples

```
is.wholenumber(1)
all(is.wholenumber(rbinom(100,10,.5)))
is.wholenumber((1:10)/2)
```

`ladd` *Add to Lattice Plots*

Description

Simplified lattice plotting by adding additional elements to existing plots.

Usage

```
ladd(x, data = NULL, ..., plot = trellis.last.object())
```

Arguments

`x`	callable graphical element to be added to a panel or panels in a lattice plot
`data`	a list containing objects that can be referred to in `x`. Panel functions also have access to the data already used in the panel by the underlying lattice plot. See `latticeExtra::layer()` for details.
`...`	additional arguments passed to `latticeExtra::layer()`.
`plot`	a lattice plot to add to. Defaults to previous lattice plot.

Details

`ladd` is a wrapper around `latticeExtra::layer()` that simplifies certain common plotting additions. The same caveats that apply to that function apply here as well. In particular, `ladd` uses non-standard evaluation. For this reason care must be taken if trying to use `ladd` within other functions and the use of `data` may be required to pass information into the environment in which `x` will be evaluated.

Value

a trellis object

Author(s)

Randall Pruim (<`rpruim@calvin.edu`>)

See Also

`latticeExtra::layer()`

Examples

```
p <- xyplot(rnorm(100) ~rnorm(100))
print(p)
ladd(panel.abline(a=0,b=1))
ladd(panel.abline(h=0,col='blue'))
ladd(grid.text('Hello'))
ladd(grid.text(x=.95,y=.05,'text here',just=c('right','bottom')))
```

```
q <- xyplot(rnorm(100) ~rnorm(100)|factor(rbinom(100,4,.5)))
q <- update(q, layout=c(3,2))
ladd(panel.abline(a=0,b=1), plot=q)
ladd(panel.abline(h=0,col='blue'))
ladd( grid.text("(2,1)",gp=gpar(cex=3,alpha=.5)), columns=2, rows=1)
ladd( grid.text("p5",gp=gpar(cex=3,alpha=.5)), packets=5)
q
ladd( grid.text(paste(current.column(), current.row(),sep=','), gp=gpar(cex=3,alpha=.5)) )
histogram( ~eruptions, data=faithful )
# over would probably be better here, but the demonstrates what under=TRUE does.
ladd(panel.densityplot(faithful$eruptions, lwd=4), under=TRUE)
```

`leaflet_map` *Simple Leaflet Maps*

Description

Primarily designed to work with `rgeo()` to display randomly sampled points on the globe.

Usage

```
leaflet_map(
  latitude = NULL,
  longitude = NULL,
  position = NULL,
  zoom = 12,
  mark = FALSE,
  radius = 0,
  units = c("km", "miles", "meters", "feet"),
  ...
)
```

Arguments

`latitude, longitude`	vectors of latitude and longitude values. If `latitude` is a data frame, then it is treated as `position`. This facilitates "piping" from `rgeo()`. See examples.
`position`	a data frame containing latitude and longitude positions
`zoom`	zoom level for initial map (1-20)
`mark`	a logical indicating whether the location should be marked with a pin
`radius`	a vector of radii of circles (in miles) centered at position that are displayed on the map
`units`	units for radii of circles (km, miles, meters, or feet).
`...`	additional arguments passed to `leaflet::addCircles()`

Value

a leaflet map

See Also

`deg2rad()`, `latlon2xyz()` and `rgeo()`.

Examples

```
# the leaflet package is required
if (require(leaflet)) {
  # Times Square
  leaflet_map(40.7566, -73.9863, radius = 1, units = "miles")
  # 3 random locations; 5 km circles
  leaflet_map(position = rgeo(3), radius = 5, mark = TRUE, color = "red")
  # using pipes
  rgeo(4, latlim = c(25,50), lonlim = c(-65, -125)) |>
    leaflet_map(radius = 5, mark = TRUE, color = "purple")
}
```

`linear.algebra` *Functions for teaching linear algebra.*

Description

These functions provide a formula based interface to the construction of matrices from data and for fitting. You can use them both for numerical vectors and for functions of variables in data frames. These functions are intended to support teaching basic linear algebra with a particular connection to statistics.

Usage

```
mat(formula, data = parent.frame(), A = formula)

singvals(formula, data = parent.frame(), A = formula)
```

Arguments

`formula`	a formula. In `mat` and `singvals`, only the right-hand side is used.
`data`	a data frame from which to pull out numerical values for the variables in the formula
`A`	an alias for `formula` for backward compatibility. `mat` returns a model matrix To demonstrate singularity, use `singvals`.

Value

`mat` returns a matrix

`singvals` gives singular values for each column in the model matrix

See Also

`project()`

`linearModel()`, which returns a function.

Examples

```
a <- c(1,0,0); b <- c(1,2,3); c <- c(4,5,6); x <- rnorm(3)
# Formula interface
mat(~a+b)
mat(~a+b+1)
if (require(mosaicData)) {
mat(~length+sex, data=KidsFeet)
singvals(~length*sex*width, data=KidsFeet)
}
```

`MAD` *All pairs mean and sum of absolute differences*

Description

The functions compute the sum or mean of all pairwise absolute differences. This differs from `stats::mad()`, which computes the median absolute difference of each value from the median of all the values. See the `ISIwithR` package (and the textbook it accompanies) for examples using these functions in the context of simulation-based inference.

Usage

```
MAD(x, ..., data = NULL, groups = NULL, na.rm = getOption("na.rm", FALSE))

SAD(x, ..., data = NULL, groups = NULL, na.rm = getOption("na.rm", FALSE))
```

Arguments

`x`	a numeric vector or a formula.
`...`	additional arguments passed through to `MAD_` or `SAD_`. If `x` is a formula, `...` should include an argument named `data` if the intent is to interpret the formula in a data frame.
`data`	a data frame in which to evaluate formulas (or bare names). Note that the default is `data = parent.frame()`. This makes it convenient to use this function interactively by treating the working environment as if it were a data frame. But this may not be appropriate for programming uses. When programming, it is best to use an explicit `data` argument – ideally supplying a data frame that contains the variables mentioned.

`groups`	a grouping variable, typically a name of a variable in `data`
`na.rm`	a logical indicating whether NAs should be removed before calculating.

Value

the mean or sum of the absolute differences between each pair of values in `c(x,...)`.

See Also

`mad()`, `MAD_()`

Examples

```
SAD(1:3)
MAD(1:3)
MAD(~eruptions, data = faithful)
```

`MAD_` *All pairs mean and sum of absolute differences*

Description

All pairs mean and sum of absolute differences

Usage

```
MAD_(x, ..., na.rm = getOption("na.omit", FALSE))

SAD_(x, ..., na.rm = getOption("na.omit", FALSE))
```

Arguments

`x`	a numeric vector or a formula.
`...`	additional arguments appended to `x`
`na.rm`	a logical indicating whether NAs should be removed before calculating.

Value

the mean or sum of the absolute differences between each pair of values in `c(x,...)`.

See Also

`mad()`

`maggregate` *Aggregate for mosaic*

Description

Compute function on subsets of a variable in a data frame.

Usage

```
maggregate(
  formula,
  data = parent.frame(),
  FUN,
  groups = NULL,
  subset,
  drop = FALSE,
  ...,
  .format = c("default", "table", "flat"),
  .overall = mosaic.par.get("aggregate.overall"),
  .multiple = FALSE,
  .name = deparse(substitute(FUN)),
  .envir = parent.frame()
)
```

Arguments

`formula`	a formula. Left side provides variable to be summarized. Right side and condition describe subsets. If the left side is empty, right side and condition are shifted over as a convenience.
`data`	a data frame. Note that the default is `data = parent.frame()`. This makes it convenient to use this function interactively by treating the working environment as if it were a data frame. But this may not be appropriate for programming uses. When programming, it is best to use an explicit `data` argument – ideally supplying a data frame that contains the variables mentioned in `formula`.
`FUN`	a function to apply to each subset
`groups`	grouping variable that will be folded into the formula (if there is room for it). This offers some additional flexibility in how formulas can be specified.
`subset`	a logical indicating a subset of `data` to be processed.
`drop`	a logical indicating whether unused levels should be dropped.
`...`	additional arguments passed to `FUN`
`.format`	format used for aggregation. `"default"` and `"flat"` are equivalent.
`.overall`	currently unused
`.multiple`	a logical indicating whether FUN returns multiple values Ignored if `.multiple` is not `NULL`.
`.name`	a name used for the resulting object
`.envir`	an environment in which to evaluate expressions

Value

a vector

Examples

```
if (require(mosaicData)) {
maggregate( cesd ~ sex, HELPrct, FUN = mean )
# using groups instead
maggregate( ~ cesd, groups = sex, HELPrct, FUN = sd )
# the next four all do the same thing
maggregate( cesd ~ sex + homeless, HELPrct, FUN = mean )
maggregate( cesd ~ sex | homeless, HELPrct, FUN = sd )
maggregate( ~ cesd | sex , groups= homeless, HELPrct, FUN = sd )
maggregate( cesd ~ sex, groups = homeless, HELPrct, FUN = sd )
# this is unusual, but also works.
maggregate( cesd ~ NULL , groups = sex, HELPrct, FUN = sd )
}
```

`makeColorscheme` *Create a color generating function from a vector of colors*

Description

Create a color generating function from a vector of colors

Usage

```
makeColorscheme(col)
```

Arguments

`col`	a vector of colors

Value

a function that generates a vector of colors interpolated among the colors in `col`

Examples

```
cs <- makeColorscheme( c('red','white','blue') )
cs(10)
cs(10, alpha=.5)
```

---

makeMap *Make a map with* `ggplot2`

Description

`makeMap` takes in two sources of data that refer to geographical regions and merges them together. Depending on the arguments passed, it returns this merged data or a ggplot object constructed with the data.

Usage

```
makeMap(
  data = NULL,
  map = NULL,
  key = c(key.data, key.map),
  key.data,
  key.map,
  tr.data = identity,
  tr.map = identity,
  plot = c("borders", "frame", "none")
)
```

Arguments

`data`	A dataframe with regions as cases
`map`	An object that can be fortified to a dataframe (ex: a dataframe itself, or a SpatialPolygonsDataFrame)
`key`	The combination of `key.data` and `key.map`
`key.data`	The column name in the `data` that holds the unique names of each region
`key.map`	The column name in the `map` that holds the unique names of each region
`tr.data`	A function of the transformation to be performed to the `key.data` column
`tr.map`	A function of the transformation to be performed to the `key.map` column
`plot`	The plot desired for the output. `plot` = "none" returns the merged data that is the result of merging the `data` and `map` together; `plot`="frame" returns an empty (unplottable) ggplot object; `plot` = "border" (the default) returns a ggplot object with one geom_polygon layer that shows the borders of the regions.

`mean_` *Aggregating functions*

Description

The `mosaic` package makes several summary statistic functions (like `mean` and `sd`) formula aware.

Usage

```
mean_(x, ..., data = NULL, groups = NULL, na.rm = getOption("na.rm", FALSE))

mean(x, ...)

median(x, ..., data = NULL, groups = NULL, na.rm = getOption("na.rm", FALSE))

range(x, ..., data = NULL, groups = NULL, na.rm = getOption("na.rm", FALSE))

sd(x, ..., data = NULL, groups = NULL, na.rm = getOption("na.rm", FALSE))

max(x, ..., data = NULL, groups = NULL, na.rm = getOption("na.rm", FALSE))

min(x, ..., data = NULL, groups = NULL, na.rm = getOption("na.rm", FALSE))

sum(x, ..., data = NULL, groups = NULL, na.rm = getOption("na.rm", FALSE))

IQR(x, ..., data = NULL, groups = NULL, na.rm = getOption("na.rm", FALSE))

fivenum(x, ..., data = NULL, groups = NULL, na.rm = getOption("na.rm", FALSE))

iqr(x, ..., data = NULL, groups = NULL, na.rm = getOption("na.rm", FALSE))

prod(x, ..., data = NULL, groups = NULL, na.rm = getOption("na.rm", FALSE))

sum(x, ..., data = NULL, groups = NULL, na.rm = getOption("na.rm", FALSE))

favstats(x, ..., data = NULL, groups = NULL, na.rm = TRUE)

quantile(x, ..., data = NULL, groups = NULL, na.rm = getOption("na.rm", FALSE))

var(x, y = NULL, na.rm = getOption("na.rm", FALSE), ..., data = NULL)

cor(x, y = NULL, ..., data = NULL)

cov(x, y = NULL, ..., data = NULL)
```

Arguments

`x`	a numeric vector or a formula

`...`	additional arguments
`data`	a data frame in which to evaluate formulas (or bare names). Note that the default is `data = parent.frame()`. This makes it convenient to use this function interactively by treating the working environment as if it were a data frame. But this may not be appropriate for programming uses. When programming, it is best to use an explicit `data` argument – ideally supplying a data frame that contains the variables mentioned.
`groups`	a grouping variable, typically a name of a variable in `data`
`na.rm`	a logical indicating whether `NA`s should be removed before computing
`y`	a numeric vector or a formula

Details

Many of these functions mask core R functions to provide an additional formula interface. Old behavior should be unchanged. But if the first argument is a formula, that formula, together with `data` are used to generate the numeric vector(s) to be summarized. Formulas of the shape `x ~ a` or `~ x | a` can be used to produce summaries of `x` for each subset defined by `a`. Two-way aggregation can be achieved using formulas of the form `x ~ a + b` or `x ~ a | b`. See the examples.

Note

Earlier versions of these functions supported a "bare name + data frame" interface. This functionality has been removed since it was (a) ambiguous in some cases, (b) unnecessary, and (c) difficult to maintain.

Examples

```
mean(HELPrct$age)
mean( ~ age, data = HELPrct)
mean( ~ drugrisk, na.rm = TRUE, data = HELPrct)
mean(age ~ shuffle(sex), data = HELPrct)
mean(age ~ shuffle(sex), data = HELPrct, .format = "table")
# wrap in data.frame() to auto-convert awkward variable names
data.frame(mean(age ~ shuffle(sex), data = HELPrct, .format = "table"))
mean(age ~ sex + substance, data = HELPrct)
mean( ~ age | sex + substance, data = HELPrct)
mean( ~ sqrt(age), data = HELPrct)
sum( ~ age, data = HELPrct)
sd(HELPrct$age)
sd( ~ age, data = HELPrct)
sd(age ~ sex + substance, data = HELPrct)
var(HELPrct$age)
var( ~ age, data = HELPrct)
var(age ~ sex + substance, data = HELPrct)
IQR(width ~ sex, data = KidsFeet)
iqr(width ~ sex, data = KidsFeet)
favstats(width ~ sex, data = KidsFeet)

cor(length ~ width, data = KidsFeet)
cov(length ~ width, data = KidsFeet)
```

```
tally(is.na(mcs) ~ is.na(pcs), data = HELPmiss)
cov(mcs ~ pcs, data = HELPmiss)            # NA because of missing data
cov(mcs ~ pcs, data = HELPmiss, use = "complete")  # ignore missing data
# alternative approach using filter explicitly
cov(mcs ~ pcs, data = HELPmiss |> filter(!is.na(mcs) & !is.na(pcs)))
```

`mid` *midpoints along a sequence*

Description

Compute a vector of midpoints between values in a numeric vector

Usage

```
mid(x)
```

Arguments

`x`	a numeric vector

Value

a vector of length 1 less than `x`

Examples

```
mid(1:5)
mid((1:5)^2)
```

`mosaic.options` *Setting options for mosaic package functions*

Description

A mechanism for setting options in the mosaic package.

Usage

```
mosaic.options(...)

mosaic.getOption(name)

mosaic.par.set(name, value, ..., theme, warn = TRUE, strict = FALSE)

mosaic.par.get(name = NULL)
```

```
restoreLatticeOptions()

mosaicLatticeOptions()
```

Arguments

`...`	additional arguments that are turned into a list if a list cannot be inferred from `theme`, `name`, and `value`.
`name`	the name of the option being set
`value`	the value to which to set the option
`theme`	a list appropriate for a mosaic theme
`warn`	a logical. UNUSED at present.
`strict`	a logical or numeric.

Details

`restoreLatticeOptions` returns any lattice options that were changed when the mosaic package was loaded back to their pre-mosaic state.

`mosaicLatticeOptions` sets a number of defaults for lattice graphics.

`mPlot` *Interactive plotting*

Description

These functions provide a menu selection system (via **manipulate**) so that different aspects of a plot can be selected interactively. The **ggplot2** or **lattice** command for generating the plot currently being displayed can be copied to the console, whence it can be copied to a document for later direct, non-interactive use.

Usage

```
mPlot(
  data,
  format,
  default = format,
  system = system_choices()[1],
  show = FALSE,
  title = "",
  data_text = rlang::expr_deparse(substitute(data)),
  ...
)

mMap(
  data,
```

```
  default = "map",
  system = "ggplot2",
  show = FALSE,
  title = title,
  data_text = rlang::expr_deparse(substitute(data)),
  ...
)

mScatter(
  data,
  default = c("scatter", "jitter", "boxplot", "violin", "line", "sina",
    "density (contours)", "density (filled)"),
  system = "ggformula",
  show = FALSE,
  title = "",
  data_text = rlang::expr_deparse(substitute(data))
)

mUniplot(
  data,
  default = c("histogram", "density", "frequency polygon", "ASH plot"),
  system = system_choices()[1],
  show = FALSE,
  title = "",
  data_text = rlang::expr_deparse(substitute(data))
)
```

Arguments

`data`	a data frame containing the variables that might be used in the plot. Note that for maps, the data frame must contain coordinates of the polygons comprising the map and a variable for determining which coordinates are part of the same region. See `sp2df()` for one way to create such a data frame. Typically `merge()` will be used to combine the map data with some auxiliary data to be displayed as fill color on the map, although this is not necessary if all one wants is a map.
`format`	a synonym for `default`.
`default`	default type of plot to create; one of `"scatter"`, `"jitter"`, `"boxplot"`, `"violin"`, `"sina"`, `"histogram"`, `"density"`, `"density (contours)"`, `"density (filled)"`, `"frequency polygon"`, `"xyplot"`, or `"map"`. Unique prefixes suffice.
`system`	which graphics system to use (initially) for plotting (**ggplot2** or **lattice**). A check box will allow on the fly change of plotting system.
`show`	a logical, if `TRUE`, the code will be displayed each time the plot is changed.
`title`	a title for the plot
`data_text`	A text string describing the data. It must be possible to recover the data from this string using `eval()`. Typically users will not need to modify this from the default value.
`...`	additional arguments

Details

Only `mPlot` is required by end users. The other plotting functions are dispatched based on the value of `default`. Furthermore, `mplot()` will dispatch `mPlot` when provided a data frame.

Currently maps are only supported in **ggplot2** and not in **lattice**.

Due to an unresolved issue with RStudio, the first time this function is called, and additional plot is created to correctily initialize the mainipulate frameowrk.

Value

Nothing. Just for side effects.

Note

Due to an unresolved issue with RStudio, the first time this function is called, and additional plot is created to correctily initialize the mainipulate frameowrk.

Examples

```
## Not run:
mPlot(HELPrct, format = "scatter")
mPlot(HELPrct, format = "density")

## End(Not run)
```

`mplot` *Generic plotting*

Description

Generic function plotting for R objects. Currently plots exist for `data.frame`s, `lm`s, (including `glm`s).

Usage

```
mplot(object, ...)

## Default S3 method:
mplot(object, ...)

## S3 method for class 'lm'
mplot(
  object,
  which = c(1:3, 7),
  system = c("ggplot2", "lattice", "base"),
  ask = FALSE,
  multiplot = "package:gridExtra" %in% search(),
  par.settings = theme.mosaic(),
```

```
    level = 0.95,
    title = paste("model: ", deparse(object$call), "\n"),
    rows = TRUE,
    id.n = 3L,
    id.size = 5,
    id.color = "red",
    id.nudge = 1,
    add.smooth = TRUE,
    smooth.color = "red",
    smooth.alpha = 0.6,
    smooth.size = 0.7,
    span = 3/4,
    ...
  )

  ## S3 method for class 'data.frame'
  mplot(
    object,
    format,
    default = format,
    system = c("ggformula", "ggplot2", "lattice"),
    show = FALSE,
    data_text = rlang::expr_deparse(substitute(object)),
    title = "",
    ...
  )

  ## S3 method for class 'summary.lm'
  mplot(
    object,
    system = c("ggplot2", "lattice"),
    level = 0.95,
    par.settings = trellis.par.get(),
    rows = TRUE,
    ...
  )

  ## S3 method for class 'TukeyHSD'
  mplot(
    object,
    system = c("ggplot2", "lattice"),
    ylab = "",
    xlab = "difference in means",
   title = paste0(attr(object, "conf.level") * 100, "% family-wise confidence level"),
    par.settings = trellis.par.get(),
    order = c("asis", "pval", "difference"),
    ...
  )
```

Arguments

`object`	an R object from which a plot will be constructed.
`...`	additional arguments. If `object` is an `lm`, subsets of these arguments are passed to `gridExtra::grid.arrange` and to the **lattice** plotting routines; in particular, `nrow` and `ncol` can be used to control the number of rows and columns used.
`which`	a numeric vector used to select from 7 potential plots
`system`	which graphics system to use (initially) for plotting (**ggplot2** or **lattice**). A check box will allow on the fly change of plotting system.
`ask`	if TRUE, each plot will be displayed separately after the user responds to a prompt.
`multiplot`	if TRUE and `ask` == `FALSE`, all plots will be displayed together.
`par.settings`	**lattice** theme settings
`level`	a confidence level
`title`	title for plot
`rows`	rows to show. This may be a numeric vector, `TRUE` (for all rows), or a character vector of row names.
`id.n`	Number of id labels to display.
`id.size`	Size of id labels.
`id.color`	Color of id labels.
`id.nudge`	a numeric used to increase (>1) or decrease (<1) the amount that observation labels are nudged. Use a negative value to nudge down instead of up.
`add.smooth`	A logicial indicating whether a LOESS smooth should be added (where this makes sense to do). Currently ignored for lattice plots.
`smooth.color`, `smooth.size`, `smooth.alpha`	Color, size, and alpha used for LOESS curve. Currently ignored for lattice plots.
`span`	A positive number indicating the amount of smoothing. A larger number indicates more smoothing. See `stats::loess()` for details. Currently ignored for lattice plots.
`format`, `default`	default type of plot to create; one of `"scatter"`, `"jitter"`, `"boxplot"`, `"violin"`, `"histogram"`, `"density"`, `"frequency polygon"`, or `"map"`. Unique prefixes suffice.
`show`	a logical, if `TRUE`, the code will be displayed each time the plot is changed.
`data_text`	text representation of the data set. In typical use cases, the default value should suffice.
`ylab`	label for y-axis
`xlab`	label for x-axis
`order`	one of `"pval"`, `"diff"`, or `"asis"` determining the order of the `pair` factor, which determines the order in which the differences are displayed on the plot.
`data`	a data frame containing the variables that might be used in the plot.

Details

The method for models (lm and glm) is still a work in progress, but should be usable for relatively simple models. When the results for a logistic regression model created with `glm()` are satisfactory will depend on the format and structure of the data used to fit the model.

Due to a bug in RStudio 1.3, the method for data frames may not display the controls consistently. We have found that executing this code usually fixes the problem:

```
library(manipulate)
manipulate(plot(A), A = slider(1, 10))
```

Value

Nothing. Just for side effects.

Examples

```
lm( width ~ length * sex, data = KidsFeet) |>
  mplot(which = 1:3, id.n = 5)
lm( width ~ length * sex, data = KidsFeet) |>
  mplot(smooth.color = "blue", smooth.size = 1.2, smooth.alpha = 0.3, id.size = 3)
lm(width ~ length * sex, data = KidsFeet) |>
  mplot(rows = 2:3, which = 7)
## Not run:
mplot( HELPrct )
mplot( HELPrct, "histogram" )

## End(Not run)
lm(width ~ length * sex, data = KidsFeet) |>
  summary() |>
  mplot()

lm(width ~ length * sex, data = KidsFeet) |>
  summary() |>
  mplot(rows = c("sex", "length"))

lm(width ~ length * sex, data = KidsFeet) |>
  summary() |>
  mplot(rows = TRUE)
lm(age ~ substance, data = HELPrct) |>
  TukeyHSD() |>
  mplot()
lm(age ~ substance, data = HELPrct) |>
  TukeyHSD() |>
  mplot(system = "lattice")
```

`mUSMap` *Make a US map with* `ggplot2`

Description

`mUSMap` takes in one dataframe that includes information about different US states. It merges this dataframe with a dataframe that includes geographical coordinate information. Depending on the arguments passed, it returns this data or a ggplot object constructed with the data.

Usage

```
mUSMap(
  data = NULL,
  key,
  fill = NULL,
  plot = c("borders", "frame", "none"),
  style = c("compact", "real")
)
```

Arguments

`data`	A dataframe with US states as cases
`key`	The column name in the `data` that holds the unique names of each state
`fill`	A variable in the `data` used to specify the fill color of states in the map (note: if `fill` is not null, then `plot` cannot be set to "none")
`plot`	The plot desired for the output. `plot` = "none" returns the merged data that is the result of merging the `data` and the dataframe with the geographical coordinate information; `plot` = "frame" returns an empty (unplottable) ggplot object; `plot` = "border" (the default) returns a ggplot object with one geom_polygon layer that shows the borders of the states
`style`	The style in which to display the map. `compact` gives a polyconic projection with Alaska and Hawaii on the lower left corner; `real` gives the real size and position of all states without any projection.

Examples

```
USArrests2 <- USArrests |> tibble::rownames_to_column("state")
mUSMap(USArrests2, key="state", fill = "UrbanPop")
```

`Mustangs` *Mustang Prices*

Description

Mustang Prices

Usage

```
data(Mustangs)
```

Format

A data frame with 25 observations on the following 3 variables.

`Age`	age of vehicle in years
`Miles`	1000s of miles driven
`Price`	selling price in 1000s USD

Details

#' @docType data

A student collected data on the selling prices for a sample of used Mustang cars being offered for sale at an internet website.

Source

These data were used in a "resampling bake-off" hosted by Robin Lock.

`mWorldMap` *Make a world map with* `ggplot2`

Description

`mWorldMap` takes in one dataframe that includes information about different countries. It merges this dataframe with a dataframe that includes geographical coordinate information. Depending on the arguments passed, it returns this data or a ggplot object constructed with the data.

Usage

```
mWorldMap(
  data = NULL,
  key = NA,
  fill = NULL,
  plot = c("borders", "frame", "none")
)
```

Arguments

`data`	A dataframe with countries as cases
`key`	The column name in the `data` that holds the unique names of each country
`fill`	A variable in the `data` used to specify the fill color of countries in the map (note: if `fill` is not null, then `plot` cannot be set to "none")
`plot`	The plot desired for the output. `plot` = "none" returns the merged data that is the result of merging the `data` and the dataframe with the geographical coordinate information; `plot` = "frame" returns an empty (unplottable) ggplot object; `plot` = "border" (the default) returns a ggplot object with one geom_polygon layer that shows the borders of the countries

Examples

```
## Not run:
gdpData <- CIAdata("GDP")      # load some world data

mWorldMap(gdpData, key="country", fill="GDP")

gdpData <- gdpData |> mutate(GDP5 = ntiles(-GDP, 5, format="rank"))
mWorldMap(gdpData, key="country", fill="GDP5")

mWorldMap(gdpData, key="country", plot="frame") +
geom_point()

mergedData <- mWorldMap(gdpData, key="country", plot="none")

ggplot(mergedData, aes(x=long, y=lat, group=group, order=order)) +
geom_polygon(aes(fill=GDP5), color="gray70", size=.5) + guides(fill=FALSE)

## End(Not run)
```

`ntiles` *Create vector based on roughly equally sized groups*

Description

Create vector based on roughly equally sized groups

Usage

```
ntiles(
  x,
  n = 3,
  format = c("rank", "interval", "mean", "median", "center", "left", "right"),
  digits = 3
)
```

Arguments

`x`	a numeric vector
`n`	(approximate) number of quantiles
`format`	a specification of desired output format.
`digits`	desired number of digits for labeling of factors.

Value

a vector. The type of vector will depend on `format`.

Examples

```
if (require(mosaicData)) {
tally( ~ ntiles(age, 4), data=HELPrct)
tally( ~ ntiles(age, 4, format="center"), data=HELPrct)
tally( ~ ntiles(age, 4, format="interval"), data=HELPrct)
tally( ~ ntiles(age, 4, format="left"), data=HELPrct)
tally( ~ ntiles(age, 4, format="right"), data=HELPrct)
tally( ~ ntiles(age, 4, format="mean"), data=HELPrct)
tally( ~ ntiles(age, 4, format="median"), data=HELPrct)
bwplot( i2 ~ ntiles(age, n=5, format="interval"), data=HELPrct)
}
```

`orrr` *Odds Ratio and Relative Risk for 2 x 2 Contingency Tables*

Description

This function calculates the odds ratio and relative risk for a 2 x 2 contingency table and a confidence interval (default `conf.level` is 95 percent) for the each estimate. `x` should be a matrix, data frame or table. "Successes" should be located in column 1 of `x`, and the treatment of interest should be located in row 2. The odds ratio is calculated as (Odds row 2) / (Odds row 1). The confidence interval is calculated from the log(OR) and backtransformed.

Usage

```
orrr(
  x,
  conf.level = 0.95,
  verbose = !quiet,
  quiet = TRUE,
  digits = 3,
  relrisk = FALSE
)

oddsRatio(x, conf.level = 0.95, verbose = !quiet, quiet = TRUE, digits = 3)
```

```
relrisk(x, conf.level = 0.95, verbose = !quiet, quiet = TRUE, digits = 3)

## S3 method for class 'oddsRatio'
print(x, digits = 4, ...)

## S3 method for class 'relrisk'
print(x, digits = 4, ...)

## S3 method for class 'oddsRatio'
summary(object, digits = 4, ...)

## S3 method for class 'relrisk'
summary(object, digits = 4, ...)
```

Arguments

`x`	a 2 x 2 matrix, data frame, or table of counts
`conf.level`	the confidence interval level
`verbose`	a logical indicating whether verbose output should be displayed
`quiet`	a logical indicating whether verbose output should be suppressed
`digits`	number of digits to display
`relrisk`	a logical indicating whether the relative risk should be returned instead of the odds ratio
`...`	additional arguments
`object`	an R object to print or summarise. Here an object of class `"oddsRatio"` or `"relrisk"`.

Value

an odds ratio or relative risk. If `verpose` is true, more details and the confidence intervals are displayed.

Author(s)

Kevin Middleton (<`kmm@csusb.edu`>); modified by R Pruim.

See Also

`chisq.test()`, `fisher.test()`

Examples

```
M1 <- matrix(c(14, 38, 51, 11), nrow = 2)
M1
oddsRatio(M1)

M2 <- matrix(c(18515, 18496, 1427, 1438), nrow = 2)
rownames(M2) <- c("Placebo", "Aspirin")
```

```
colnames(M2) <- c("No", "Yes")
M2
oddsRatio(M2)
oddsRatio(M2, verbose = TRUE)
relrisk(M2, verbose = TRUE)
if (require(mosaicData)) {
relrisk(tally(~ homeless + sex, data = HELPrct) )
do(3) * relrisk( tally( ~ homeless + shuffle(sex), data = HELPrct) )
}
```

`panel.levelcontourplot` *Lattice plot that draws a filled contour plot*

Description

Used within plotFun

Usage

```
panel.levelcontourplot(
  x,
  y,
  z,
  subscripts = 1,
  at,
  shrink,
  labels = TRUE,
  label.style = c("mixed", "flat", "align"),
  contour = FALSE,
  region = TRUE,
  col = add.line$col,
  lty = add.line$lty,
  lwd = add.line$lwd,
  border = "transparent",
  ...,
  col.regions = regions$col,
  filled = TRUE,
  alpha.regions = regions$alpha
)
```

Arguments

`x`	x on a grid
`y`	y on a grid
`z`	zvalues for the x and y
`subscripts`	which points to plot

`at`	cuts for the contours
`shrink`	what does this do?
`labels`	draw the contour labels
`label.style`	where to put the labels
`contour`	logical draw the contours
`region`	logical color the regions
`col`	color for contours
`lty`	type for contours
`lwd`	width for contour
`border`	type of border
`...`	dots additional arguments
`col.regions`	a vector of colors or a function (`topo.colors` by default) for generating such
`filled`	whether to fill the contours with color
`alpha.regions`	transparency of regions

`panel.lmbands` *show confidence and prediction bands on plots*

Description

show confidence and prediction bands on plots

Usage

```
panel.lmbands(
  x,
  y,
  interval = "confidence",
  level = 0.95,
  model = lm(y ~ x),
  band.col = c(conf = slcol[3], pred = slcol[2]),
  band.lty = c(conf = slty[3], pred = slty[2]),
  band.show = TRUE,
  fit.show = TRUE,
  band.alpha = 0.6,
  band.lwd = 1,
  npts = 100,
  ...
)
```

Arguments

`x, y`	numeric vectors
`interval`	a vector subset of `'confidence'` and `'prediction'`
`level`	confidence level
`model`	model to be used for generating bands
`band.col`	a vector of length 1 or 2 giving the color of bands
`band.lty`	a vector of length 1 or 2 giving the line type for bands
`band.show`	logical vector of length 1 or 2 indicating whether confidence and prediction bands should be shown
`fit.show`	logical indicating whether the model fit should be shown
`band.alpha`	a vector of length 1 or 2 alpha level for bands
`band.lwd`	a vector of length 1 or 2 giving line width for bands
`npts`	resolution parameter for bands (increase to get better resolution)
`...`	additional arguments

`panel.plotFun` *Panel function for plotting functions*

Description

Panel function for plotting functions

Usage

```
panel.plotFun(
  object,
  ...,
  type = "l",
  npts = NULL,
  zlab = NULL,
  filled = TRUE,
  levels = NULL,
  nlevels = 10,
  surface = FALSE,
  col.regions = topo.colors,
  lwd = trellis.par.get("superpose.line")$lwd,
  lty = trellis.par.get("superpose.line")$lty,
  alpha = NULL,
  discontinuity = NULL,
  discontinuities = NULL
)
```

Arguments

`object`	an object (e.g., a formula) describing a function
`...`	additional arguments, typically processed by `lattice` panel functions such as `lattice::panel.xyplot()` or `lattice::panel.levelplot()`. Frequently used arguments include `lwd` line width `lty` line type `col` a color
`type`	type of plot (`"l"` by default)
`npts`	an integer giving the number of points (in each dimension) to sample the function
`zlab`	label for z axis (when in surface-plot mode)
`filled`	fill with color between the contours (`TRUE` by default)
`levels`	levels at which to draw contours
`nlevels`	number of contours to draw (if `levels` not specified)
`surface`	a logical indicating whether to draw a surface plot rather than a contour plot
`col.regions`	a vector of colors or a function (`topo.colors` by default) for generating such
`lwd`	width of the line
`lty`	line type
`alpha`	number from 0 (transparent) to 1 (opaque) for the fill colors
`discontinuity`	a positive number determining how sensitive the plot is to potential discontinuity. Larger values result in less sensitivity. The default is 1. Use `discontinuity = Inf` to disable discontinuity detection. Discontinuity detection uses a crude numerical heuristic and may not give the desired results in all cases.
`discontinuities`	a vector of input values at which a function is discontinuous or `NULL` to use a heuristic to auto-detect.

See Also

plotFun

Examples

```
x <- runif(30,0,2*pi)
d <- data.frame( x = x,  y = sin(x) + rnorm(30,sd=.2) )
xyplot( y ~ x, data=d )
ladd(panel.plotFun( sin(x) ~ x, col='red' ) )
xyplot( y ~ x | rbinom(30,1,.5), data=d )
ladd(panel.plotFun( sin(x) ~ x, col='red', lty=2 ) )    # plots sin(x) in each panel
```

`panel.plotFun1` *Panel function for plotting functions*

Description

Panel function for plotting functions

Usage

```
panel.plotFun1(
  ..f..,
  ...,
  x,
  y,
  type = "l",
  lwd = trellis.par.get("superpose.line")$lwd,
  lty = trellis.par.get("superpose.line")$lty,
  col = trellis.par.get("superpose.line")$col,
  npts = NULL,
  zlab = NULL,
  filled = TRUE,
  levels = NULL,
  nlevels = 10,
  surface = FALSE,
  alpha = NULL,
  discontinuity = NULL,
  discontinuities = NULL
)
```

Arguments

`..f..`	an object (e.g., a formula) describing a function
`...`	additional arguments, typically processed by `lattice` panel functions such as `lattice::panel.xyplot()` or `lattice::panel.levelplot()`. Frequently used arguments include `lwd` line width `lty` line type `col` a color
`x, y`	ignored, but there for compatibility with other lattice panel functions
`type`	type of plot (`"l"` by default)
`lwd`	width of the line
`lty`	line type
`col`	a vector of colors
`npts`	an integer giving the number of points (in each dimension) to sample the function

`zlab`	label for z axis (when in surface-plot mode)
`filled`	fill with color between the contours (`TRUE` by default)
`levels`	levels at which to draw contours
`nlevels`	number of contours to draw (if `levels` not specified)
`surface`	a logical indicating whether to draw a surface plot rather than a contour plot
`alpha`	number from 0 (transparent) to 1 (opaque) for the fill colors
`discontinuity`	a positive number determining how sensitive the plot is to potential discontinuity. Larger values result in less sensitivity. The default is 1. Use `discontinuity = Inf` to disable discontinuity detection. Discontinuity detection uses a crude numerical heuristic and may not give the desired results in all cases.
`discontinuities`	a vector of input values at which a function is discontinuous or `NULL` to use a heuristic to auto-detect.

See Also

plotFun

Examples

```
x <- runif(30,0,2*pi)
d <- data.frame( x = x,  y = sin(x) + rnorm(30,sd=.2) )
xyplot( y ~ x, data=d )
ladd(panel.plotFun1( sin, col='red' ) )
xyplot( y ~ x | rbinom(30,1,.5), data=d )
ladd(panel.plotFun1( sin, col='red', lty=2 ) )    # plots sin(x) in each panel
```

`pdist` *Illustrated probability calculations from distributions*

Description

Illustrated probability calculations from distributions

Usage

```
pdist(
  dist = "norm",
  q,
  plot = TRUE,
  verbose = FALSE,
  invisible = FALSE,
  digits = 3L,
  xlim,
  ylim,
  resolution = 500L,
```

```
  return = c("values", "plot"),
  ...,
  refinements = list()
)

xpgamma(
  q,
  shape,
  rate = 1,
  scale = 1/rate,
  lower.tail = TRUE,
  log.p = FALSE,
  ...
)

xpt(q, df, ncp, lower.tail = TRUE, log.p = FALSE, ...)

xpchisq(q, df, ncp = 0, lower.tail = TRUE, log.p = FALSE, ...)

xpf(q, df1, df2, lower.tail = TRUE, log.p = FALSE, ...)

xpbinom(q, size, prob, lower.tail = TRUE, log.p = FALSE, ...)

xppois(q, lambda, lower.tail = TRUE, log.p = FALSE, ...)

xpgeom(q, prob, lower.tail = TRUE, log.p = FALSE, ...)

xpnbinom(q, size, prob, mu, lower.tail = TRUE, log.p = FALSE, ...)

xpbeta(q, shape1, shape2, ncp = 0, lower.tail = TRUE, log.p = FALSE, ...)
```

Arguments

`dist`	a character description of a distribution, for example `"norm"`, `"t"`, or `"chisq"`
`q`	a vector of quantiles
`plot`	a logical indicating whether a plot should be created
`verbose`	a logical
`invisible`	a logical
`digits`	the number of digits desired
`xlim`	x limits
`ylim`	y limits
`resolution`	Number of points used for detecting discreteness and generating plots. The default value of 5000 should work well except for discrete distributions that have many distinct values, especially if these values are not evenly spaced.
`return`	If `"plot"`, return a plot. If `"values"`, return a vector of numerical values.
`...`	Additional arguments, typically for fine tuning the plot.

`refinements`	A list of refinements to the plot. See `ggformula::gf_refine()`.
`shape, scale`	shape and scale parameters. Must be positive, `scale` strictly.
`rate`	an alternative way to specify the scale.
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \le x]$, otherwise, $P[X > x]$.
`log.p`	A logical indicating whether probabilities should be returned on the log scale.
`df`	degrees of freedom (> 0, maybe non-integer). `df = Inf` is allowed.
`ncp`	non-centrality parameter δ; currently except for `rt()`, only for `abs(ncp) <= 37.62`. If omitted, use the central t distribution.
`df1, df2`	degrees of freedom. `Inf` is allowed.
`size`	number of trials (zero or more).
`prob`	probability of success on each trial.
`lambda`	vector of (non-negative) means.
`mu`	alternative parametrization via mean: see 'Details'.
`shape1, shape2`	non-negative parameters of the Beta distribution.

Details

The most general function is `pdist` which can work with any distribution for which a p-function exists. As a convenience, wrappers are provided for several common distributions.

Value

A vector of probabilities; a plot is printed as a side effect.

See Also

`qdist()`, `xpnorm()`, `xqnorm()`.

Examples

```
pdist("norm", -2:2)
pdist("norm", seq(80,120, by = 10), mean = 100, sd = 10)
pdist("chisq", 2:4, df = 3)
pdist("f", 1, df1 = 2, df2 = 10)
pdist("gamma", 2, shape = 3, rate = 4)
```

`plotCumfreq` *Cumulative frequency plots*

Description

A high-level function for producing a cumulative frequency plot using `lattice` graphics.

Usage

```
plotCumfreq(x, data, ...)

## S3 method for class 'formula'
plotCumfreq(x, data = NULL, subscripts, ...)

## Default S3 method:
plotCumfreq(x, ...)

prepanel.cumfreq(x, ...)

panel.cumfreq(x, type = c("smooth", "step"), groups = NULL, ...)
```

Arguments

`x`	a formula or numeric vector
`data`	a data frame in which `x` is evaluated if `x` is a formula.
`...`	other lattice arguments
`subscripts`	as in lattice plots
`type`	smooth or step-function?
`groups`	grouping variable

Value

A plot of the empirical cumulative distribution function for sample values specified in `x`.

See Also

`histogram()`, `densityplot()`

Examples

```
plotCumfreq(~eruptions, faithful, xlab = 'duration of eruptions')
```

`plotDist` *Plots of Discrete and Continuous Distributions*

Description

Provides a simple way to generate plots of pdfs, probability mass functions, cdfs, probability histograms, and normal-quantile plots for distributions known to R.

Usage

```
plotDist(
  dist,
  ...,
  xlim = NULL,
  ylim = NULL,
  add,
  under = FALSE,
  packets = NULL,
  rows = NULL,
  columns = NULL,
  kind = c("density", "cdf", "qq", "histogram"),
  xlab = "",
  ylab = "",
  breaks = NULL,
  type,
  resolution = 5000L,
  params = NULL
)
```

Arguments

`dist`	A string identifying the distribution. This should work with any distribution that has associated functions beginning with 'd', 'p', and 'q' (e.g, `dnorm()`, `pnorm()`, and `qnorm()`). `dist` should match the name of the distribution with the initial 'd', 'p', or 'q' removed.
`...`	other arguments passed along to lattice graphing routines
`xlim`	a numeric vector of length 2 or `NULL`, in which case the central 99.8 of the distribution is used.
`ylim`	a numeric vector of length 2 or `NULL`, in which case a heuristic is used to avoid chasing asymptotes in distributions like the F distributions with 1 numerator degree of freedom.
`add`	a logical indicating whether the plot should be added to the previous lattice plot. If missing, it will be set to match `under`.
`under`	a logical indicating whether adding should be done in a layer under or over the existing layers when `add = TRUE`.

`packets`, `rows`, `columns`	specification of which panels will be added to when `add` is `TRUE`. See `latticeExtra::layer()`.
`kind`	one of "density", "cdf", "qq", or "histogram" (or prefix of any of these)
`xlab`, `ylab`	as per other lattice functions
`breaks`	a vector of break points for bins of histograms, as in `histogram()`
`type`	passed along to various lattice graphing functions
`resolution`	number of points to sample when generating the plots
`params`	a list containing parameters for the distribution. If `NULL` (the default), this list is created from elements of `\dots` that are either unnamed or have names among the formals of the appropriate distribution function. See the examples.

Details

`plotDist()` determines whether the distribution is continuous or discrete by seeing if all the sampled quantiles are unique. A discrete random variable with many possible values could fool this algorithm and be considered continuous.

The plots are done referencing a data frame with variables `x` and `y` giving points on the graph of the pdf, pmf, or cdf for the distribution. This can be useful in conjunction with the `groups` argument. See the examples.

See Also

`ggformula::gf_dist()`

Examples

```
plotDist('norm')
plotDist('norm', type='h')
plotDist('norm', kind='cdf')
plotDist('exp',  kind='histogram')
plotDist('binom', params=list( 25, .25))       # explicit params
plotDist('binom', 25, .25)                     # params inferred
plotDist('norm', mean=100, sd=10, kind='cdf')  # params inferred
plotDist('binom', 25, .25, xlim=c(-1,26) )     # params inferred
plotDist('binom', params=list( 25, .25), kind='cdf')
plotDist('beta', params=list( 3, 10), kind='density')
plotDist('beta', params=list( 3, 10), kind='cdf')
plotDist( ”binom”, params=list(35,.25),
           groups= y < dbinom(qbinom(0.05, 35, .25), 35,.25) )
plotDist( ”binom”, params=list(35,.25),
           groups= y < dbinom(qbinom(0.05, 35, .25), 35,.25),
           kind='hist')
plotDist(”norm”, mean=10, sd=2, col=”blue”, type=”h”)
plotDist(”norm”, mean=12, sd=2, col=”red”, type=”h”, under=TRUE)
plotDist(”binom”, size=100, prob=.30) +
  plotDist(”norm”, mean=30, sd=sqrt(100 * .3 * .7))
plotDist(”chisq”, df=4, groups = x > 6, type=”h”)
plotDist(”f”, df1=1, df2 = 99)
if (require(mosaicData)) {
```

```
histogram( ~age|sex, data=HELPrct)
m <- mean( ~age|sex, data=HELPrct)
s <- sd(~age|sex, data=HELPrct)
plotDist( "norm", mean=m[1], sd=s[1], col="red", add=TRUE, packets=1)
plotDist( "norm", mean=m[2], sd=s[2], col="blue", under=TRUE, packets=2)
}
```

`plotFun` *Plotting mathematical expressions*

Description

Plots mathematical expressions in one and two variables.

Usage

```
plotFun(
  object,
  ...,
  plot = trellis.last.object(),
  add = NULL,
  under = FALSE,
  xlim = NULL,
  ylim = NULL,
  npts = NULL,
  ylab = NULL,
  xlab = NULL,
  zlab = NULL,
  filled = TRUE,
  levels = NULL,
  nlevels = 10,
  labels = TRUE,
  surface = FALSE,
  groups = NULL,
  col = trellis.par.get("superpose.line")$col,
  col.regions = topo.colors,
  type = "l",
  lwd = trellis.par.get("superpose.line")$lwd,
  lty = trellis.par.get("superpose.line")$lty,
  alpha = NULL,
  discontinuities = NULL,
  discontinuity = 1,
  interactive = rstudio_is_available()
)
```

Arguments

`object`	a mathematical expression or a function "of one variable" which will converted to something intuitively equivalent to `object(x) ~ x`. (See examples)
`...`	additional parameters, typically processed by `lattice` functions such as `lattice::xyplot()`, `lattice::levelplot()` or their panel functions. Frequently used parameters include `main` main title for plot `sub` subtitle for plot `lwd` line width `lty` line type `col` a color or a (small) integer indicating which color in the current color scheme is desired. Additionally, these arguments can be used to specify parameters for the function being plotted and to specify the plotting window with natural names. See the examples for such usage.
`plot`	a trellis object; by default, the most recently created trellis plot. When `add` is `TRUE`, the new function will be plotted into a layer added to this object.
`add`	if `TRUE`, then add a layer to an existing plot rather than creating a new plot. If `NULL`, this will be determined by the value of `under`.
`under`	if `TRUE`, then new layer is added beneath existing layers
`xlim`	limits for x axis (or use variable names, see examples)
`ylim`	limits for y axis (or use variable names, see examples)
`npts`	number of points for plotting.
`ylab`	label for y axis
`xlab`	label for x axis
`zlab`	label for z axis (when in surface-plot mode)
`filled`	fill with color between the contours (`TRUE` by default)
`levels`	levels at which to draw contours
`nlevels`	number of contours to draw (if `levels` not specified)
`labels`	if `FALSE`, don't label contours
`surface`	draw a surface plot rather than a contour plot
`groups`	grouping argument ala lattice graphics
`col`	vector of colors for line graphs and contours
`col.regions`	a vector of colors or a function (`topo.colors` by default) for generating such
`type`	type of plot (`"l"` by default)
`lwd`	vector of line widths for line graphs
`lty`	vector of line types for line graphs
`alpha`	number from 0 (transparent) to 1 (opaque) for the fill colors
`discontinuities`	a vector of input values at which a function is discontinuous or `NULL` to use a heuristic to auto-detect.

`discontinuity`	a positive number determining how sensitive the plot is to potential discontinuity. Larger values result in less sensitivity. The default is 1. Use `discontinuity = Inf` to disable discontinuity detection. Discontinuity detection uses a crude numerical heuristic and may not give the desired results in all cases.
`interactive`	a logical indicating whether the surface plot should be interactive.

Details

makes plots of mathematical expressions using the formula syntax. Will draw both line plots and contour/surface plots (for functions of two variables). In RStudio, the surface plot comes with sliders to set orientation. If the colors in filled surface plots are too blocky, increase `npts` beyond the default of 50, though `npts=300` is as much as you're likely to ever need. See examples for overplotting a constraint function on an objective function.

Value

a `trellis` object

Examples

```
plotFun( a*sin(x^2)~x, xlim=range(-5,5), a=2 )  # setting parameter value
plotFun( u^2 ~ u, ulim=c(-4,4) )                # limits in terms of u
# Note roles of ylim and y.lim in this example
plotFun( y^2 ~ y, ylim=c(-2,20), y.lim=c(-4,4) )
# Combining plot elements to show the solution to an inequality
plotFun( x^2 -3 ~ x, xlim=c(-4,4), grid=TRUE )
ladd( panel.abline(h=0,v=0,col='gray50') )
plotFun( (x^2 -3) * (x^2 > 3) ~ x, type='h', alpha=.1, lwd=4, col='lightblue', add=TRUE )
plotFun( sin(x) ~ x,
   groups=cut(x, findZeros(sin(x) ~ x, within=10)$x),
   col=c('blue','green'), lty=2, lwd=3, xlim=c(-10,10) )
plotFun( sin(x) ~ x,
   groups=cut(x, findZeros(sin(x) ~ x, within=10)$x),
   col=c(1,2), lty=2, lwd=3, xlim=c(-10,10) )
## plotFun( sin(2*pi*x/P)*exp(-k*t)~x+t, k=2, P=.3)
f <- rfun( ~ u & v )
plotFun( f(u=u,v=v) ~ u & v, u.lim=range(-3,3), v.lim=range(-3,3) )
plotFun( u^2 + v < 3 ~ u & v, add=TRUE, npts=200 )
if (require(mosaicData)) {
# display a linear model using a formula interface
model <- lm(wage ~ poly(exper,degree=2), data=CPS85)
fit <- makeFun(model)
xyplot(wage ~ exper, data=CPS85)
plotFun(fit(exper) ~ exper, add=TRUE, lwd=3, col="red")
# Can also just give fit since it is a "function of one variable"
plotFun(fit, add=TRUE, lwd=2, col='white')
}
# Attempts to find sensible axis limits by default
plotFun( sin(k*x)~x, k=0.01 )
# Plotting a linear model with multiple predictors.
mod <- lm(length ~ width * sex, data=KidsFeet)
```

```
fitted.length <- makeFun(mod)
xyplot(length ~ width, groups=sex, data=KidsFeet, auto.key=TRUE)
plotFun(fitted.length(width, sex="B") ~ width, add=TRUE, col=1)
plotFun(fitted.length(width, sex="G") ~ width, add=TRUE, col=2)
```

`plotModel` *Plot a regression model*

Description

Visualize a regression model amid the data that generated it.

Usage

```
plotModel(mod, ...)

## Default S3 method:
plotModel(mod, ...)

## S3 method for class 'parsedModel'
plotModel(
  mod,
  formula = NULL,
  ...,
  auto.key = NULL,
  drop = TRUE,
  max.levels = 9L,
  system = c("ggplot2", "lattice")
)
```

Arguments

`mod`	A model of type `lm()` or `glm()`
`...`	arguments passed to `xyplot()` or `rgl::plot3d`.
`formula`	a formula indicating how the variables are to be displayed. In the style of `lattice` and `ggformula`.
`auto.key`	If TRUE, automatically generate a key.
`drop`	If TRUE, unused factor levels are dropped from `interaction()`.
`max.levels`	currently unused
`system`	which of `ggplot2` or `lattice` to use for plotting

Details

The goal of this function is to assist with visualization of statistical models. Namely, to plot the model on top of the data from which the model was fit.

The primary plot type is a scatter plot. The x-axis can be assigned to one of the predictors in the model. Additional predictors are thought of as co-variates. The data and fitted curves are partitioned by these covariates. When the number of components to this partition is large, a random subset of the fitted curves is displayed to avoid visual clutter.

If the model was fit on one quantitative variable (e.g. SLR), then a scatter plot is drawn, and the model is realized as parallel or non-parallel lines, depending on whether interaction terms are present.

Eventually we hope to support 3-d visualizations of models with 2 quantitative predictors using the `rgl` package.

Currently, only linear regression models and generalized linear regression models are supported.

Value

A lattice or ggplot2 graphics object.

Caution

This is still underdevelopment. The API is subject to change, and some use cases may not work yet. Watch for improvements in subsequent versions of the package.

Author(s)

Ben Baumer, Galen Long, Randall Pruim

See Also

`plotPoints()`, `plotFun()`

Examples

```
require(mosaic)

mod <- lm( mpg ~ factor(cyl), data = mtcars)
plotModel(mod)

# SLR
mod <- lm( mpg ~ wt, data = mtcars)
plotModel(mod, pch = 19)

# parallel slopes
mod <- lm( mpg ~ wt + factor(cyl), data=mtcars)
plotModel(mod)

## Not run:
# multiple categorical vars
mod <- lm( mpg ~ wt + factor(cyl) + factor(vs) + factor(am), data = mtcars)
```

```
plotModel(mod)
plotModel(mod, mpg ~ am)

# interaction
mod <- lm( mpg ~ wt + factor(cyl) + wt:factor(cyl), data = mtcars)
plotModel(mod)

# polynomial terms
mod <- lm( mpg ~ wt + I(wt^2), data = mtcars)
plotModel(mod)

# GLM
mod <- glm(vs ~ wt, data=mtcars, family = 'binomial')
plotModel(mod)

# GLM with interaction
mod <- glm(vs ~ wt + factor(cyl), data=mtcars, family = 'binomial')
plotModel(mod)
# 3D model
mod <- lm( mpg ~ wt + hp, data = mtcars)
plotModel(mod)

# parallel planes
mod <- lm( mpg ~ wt + hp + factor(cyl) + factor(vs), data = mtcars)
plotModel(mod)

# interaction planes
mod <- lm( mpg ~ wt + hp + wt * factor(cyl), data = mtcars)
plotModel(mod)
plotModel(mod, system="g") + facet_wrap( ~ cyl )

## End(Not run)
```

`plotPoints` *Scatter plot of points*

Description

Make or add a scatter plot in a manner coordinated with `plotFun`.

Usage

```
plotPoints(
  x,
  data = parent.frame(),
  add = NULL,
  under = FALSE,
  panelfun = panel.xyplot,
  plotfun = xyplot,
  ...,
```

```
  plot = trellis.last.object()
)
```

Arguments

`x`	A formula specifying y ~ x or z ~ x&y
`data`	Data frame containing the variables to be plotted. If not specified, the variables will be looked up in the local environment
`add`	If `TRUE`, add points as a new layer to an existing plot. If `NULL`, the value of `under` will be used.
`under`	If `TRUE`, the new layer will be underneath existing layers.
`panelfun`	Lattice panel function to be used for adding. Set only if you want something other than a scatter plot. Mainly, this is intended to add new functionality through other functions.
`plotfun`	Lattice function to be used for initial plot creation. Set only if you want something other than a scatter plot. Mainly, this is intended to add new functionality through other functions.
`...`	additional arguments
`plot`	a trellis plot, by default the most recently created one. If `add` is `TRUE`, new points will be added as a new layer to `plot`.

Value

A trellis graphics object

See Also

`plotFun()`

Examples

```
if (require(mosaicData)) {
plotPoints( width ~ length, data=KidsFeet, groups=sex, pch=20)
f <- makeFun( lm( width ~ length * sex, data=KidsFeet))
plotFun( f(length=length,sex="G")~length, add=TRUE, col="pink")
plotFun( f(length=length,sex="B")~length, add=TRUE)
}
```

`project` *Projections*

Description

Compute projections onto the span of a vector or a model space, dot products, and vector lengths in Euclidean space.

Usage

```
project(x, ...)

## S4 method for signature 'formula'
project(x, u = NULL, data = parent.frame(2), coefficients = TRUE, ...)

## S4 method for signature 'numeric'
project(x, u = rep(1, length(x)), type = c("vector", "length", "coef"), ...)

## S4 method for signature 'matrix'
project(x, u, data = parent.frame())

vlength(x, ...)

dot(u, v)
```

Arguments

`x`	a numeric vector (all functions) or a formula (only for `project`). Left-hand sides of formulas should be a single quantity
`...`	additional arguments
`u`	a numeric vector
`data`	a data frame.
`coefficients`	For `project(y ~ x)` indicates whether the projection coefficents should be returned or the projection vector.
`type`	one of `"length"` or `"vector"` determining the type of the returned value
`v`	a numeric vector

Details

`project` (preferably pronounced "pro-JECT" as in "projection") does either of two related things: (1) Given two vectors as arguments, it will project the first onto the second, returning the point in the subspace of the second that is as close as possible to the first vector. (2) Given a formula as an argument, will work very much like `lm()`, constructing a model matrix from the right-hand side of the formula and projecting the vector on the left-hand side onto the subspace of that model matrix.

In (2), rather than returning the projected vector, `project()` returns the coefficients on each of the vectors in the model matrix. UNLIKE `lm()`, the intercept vector is NOT included by default. If you want an intercept vector, include `+1` in your formula.

Value

`project` returns the projection of `x` onto `u` (or its length if `u` and `v` are numeric vectors and `type == "length"`)

`vlength` returns the length of the vector (i.e., the square root of the sum of the squares of the components)

`dot` returns the dot product of `u` and `v`

See Also

`link{project}`

Examples

```
x1 <- c(1,0,0); x2 <- c(1,2,3); y1 <- c(3,4,5); y2 <- rnorm(3)
# projection onto the 1 vector gives the mean vector
mean(y2)
project(y2, 1)
# return the length of the vector, rather than the vector itself
project(y2, 1, type='length')
project(y1 ~ x1 + x2) -> pr; pr
# recover the projected vector
cbind(x1,x2) %*% pr -> v; v
project( y1 ~ x1 + x2, coefficients=FALSE )
dot( y1 - v, v ) # left over should be orthogonal to projection, so this should be ~ 0
if (require(mosaicData)) {
project(width~length+sex, data=KidsFeet)
}
vlength(rep(1,4))
if (require(mosaicData)) {
m <- lm( length ~ width, data=KidsFeet )
# These should be the same
vlength( m$effects )
vlength( KidsFeet$length)
# So should these
vlength( tail(m$effects, -2) )
sqrt(sum(resid(m)^2))
}
v <- c(1,1,1); w <- c(1,2,3)
u <- v / vlength(v)  # make a unit vector
# The following should be the same:
project(w,v, type="coef") * v
project(w,v)
# The following are equivalent
abs(dot( w, u ))
vlength( project( w, u) )
vlength( project( w, v) )
project( w, v, type='length' )
```

`prop.test` *Exact and Approximate Tests for Proportions*

Description

The mosaic `prop.test` provides wrapper functions around the function of the same name in **stats**. These wrappers provide an extended interface (including formulas). `prop.test` performs an approximate test of a simple null hypothesis about the probability of success in a Bernoulli or multinomial experiment from summarized data or from raw data.

Usage

```
prop.test(
  x,
  n,
  p = NULL,
  alternative = c("two.sided", "less", "greater"),
  conf.level = 0.95,
  data = NULL,
  success = NULL,
  ...
)
```

Arguments

`x`	count of successes, length 2 vector of success and failure counts, a formula, or a character, numeric, or factor vector containing raw data.
`n`	sample size (successes + failures) or a data frame (for the formula interface)
`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`	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.
`data`	a data frame (if missing, `n` may be a data frame)
`success`	level of variable to be considered success. All other levels are considered failure.
`...`	additional arguments (often ignored). When `x` is a formula, `groups` can be used to compare groups: `x = ~ var, groups=g` is equivalent to `x = var ~ g`. `na.rm` can be a logical or an integer vector of length 1 or 2 to indicate dimension along which NA's are removed before coputing the test. See the examples.

Details

```
conf.level = 0.95, ...)
```

This is a wrapper around `prop.test()` to simplify its use when the raw data are available, in which case an extended syntax for `prop.test` is provided.

Value

an `htest` object

Note

When `x` is a 0-1 vector, 0 is treated as failure and 1 as success. Similarly, for a logical vector `TRUE` is treated as success and `FALSE` as failure.

See Also

`binom.test()`, `stats::prop.test()`

Examples

```
# Several ways to get a confidence interval for the proportion of Old Faithful
# eruptions lasting more than 3 minutes.
prop.test( faithful$eruptions > 3 )
prop.test(97,272)
faithful$long <- faithful$eruptions > 3
prop.test( faithful$long )
prop.test( ~long , data = faithful )
prop.test( homeless ~ sex, data = HELPrct )
prop.test( ~ homeless | sex, data = HELPrct )
prop.test( ~ homeless, groups = sex, data = HELPrct )
prop.test(anysub ~ link, data = HELPrct, na.rm = TRUE)
prop.test(link ~ anysub, data = HELPrct, na.rm = 1)
prop.test(link ~ anysub, data = HELPrct, na.rm = TRUE)
```

`prop_test` *Internal function for testing proportion*

Description

This function is wrapped by `prop.test()`, which most users should use instead.

Usage

```
prop_test(
  x,
  n,
  p = NULL,
  alternative = c("two.sided", "less", "greater"),
  conf.level = 0.95,
  ...
)
```

Arguments

`x`	a vector, count, or formula.
`n`	a vector of counts of trials (not needed when `x` is a table or matrix).
`p`	a vector of probabilities of success (for the null hypothesis). The length must be the same as the number of groups specified by `x`.
`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.
`...`	additional arguments passed to methods.

`qdata` *The Data Distribution*

Description

Density, distribution function, quantile function, and random generation from data.

Usage

```
qdata(formula, p = seq(0, 1, 0.25), data = NULL, ...)

cdata(formula, p = 0.95, data = NULL, ...)

pdata(formula, q, data = NULL, ...)

rdata(formula, n, data = NULL, ...)

ddata(formula, q, data = NULL, ...)
```

Arguments

`formula`	a formula or a vector
`p`	a vector of probabilities
`data`	a data frame in which to evaluate `formula`
`...`	additional arguments passed to `quantile` or `sample`
`q`	a vector of quantiles
`n`	number of values to sample

Value

For `qdata`, a vector of quantiles

for `cdata`, a data frame giving upper and lower limits and the central proportion requested

For `pdata`, a vector of probabilities

For `rdata`, a vector of sampled values.

For `ddata`, a vector of probabilities (empirical densities)

Examples

```
data(penguins, package = "palmerpenguins")
qdata(flipper_length_mm ~ species, 0.5, data = penguins)
qdata( ~ flipper_length_mm, p = 0.5, groups = species, data = penguins)
qdata(penguins$flipper_length_mm, p = 0.5)
qdata( ~ flipper_length_mm, p = 0.5, data = penguins)
qdata( ~ flipper_length_mm, p = 0.5, groups = species, data = penguins)
data(penguins, package = 'palmerpenguins')
cdata(penguins$flipper_length_mm, 0.5)
cdata( ~ flipper_length_mm, 0.5, data = penguins)
cdata( ~ flipper_length_mm, 0.5, data = penguins)
cdata( ~ flipper_length_mm | species, data = penguins, p = .5)
data(penguins, package = 'palmerpenguins')
pdata(penguins$flipper_length_mm, 3:6)
pdata( ~ flipper_length_mm, 3:6, data = penguins)
data(penguins, package = 'palmerpenguins')
rdata(penguins$species, 10)
rdata( ~ species, n = 10, data = penguins)
rdata(flipper_length_mm ~ species,  n = 5, data = penguins)
data(penguins, package = 'palmerpenguins')
ddata(penguins$species, 'setosa')
ddata( ~ species, 'setosa', data = penguins)
```

`qdata_v` *The Data Distribution*

Description

Utility functions for density, distribution function, quantile function, and random generation from data.

Usage

```
qdata_v(x, p = seq(0, 1, 0.25), na.rm = TRUE, ...)

qdata_f(x, ..., data = NULL, groups = NULL, na.rm = TRUE)

cdata_v(x, p = 0.95, na.rm = TRUE, ...)

cdata_f(x, ..., data = NULL, groups = NULL, na.rm = TRUE)

pdata_v(x, q, lower.tail = TRUE, ...)

pdata_f(x, ..., data = NULL, groups = NULL, na.rm = TRUE)

rdata_v(x, n, replace = TRUE, ...)

rdata_f(x, ..., data = NULL, groups = NULL, na.rm = TRUE)
```

```
ddata_v(x, q, ..., data = NULL, log = FALSE, na.rm = TRUE)

ddata_f(x, ..., data = NULL, groups = NULL, na.rm = TRUE)
```

Arguments

`x`	a vector containing the data
`p`	a vector of probabilities
`na.rm`	a logical indicating whether `NA`s should be removed before computing.
`...`	additional arguments passed to `quantile` or `sample`
`data`	a data frame in which to evaluate `formula`
`groups`	a grouping variable, typically the name of a variable in `data`
`q`	a vector of quantiles
`lower.tail`	a logical indicating whether to use the lower or upper tail probability
`n`	number of values to sample
`replace`	a logical indicating whether to sample with replacement
`log`	a logical indicating whether the result should be log transformed

See Also

`ddata()`, `pdata()`, `qdata()`, `rdata()`, `cdata()`

`qdist` *Illustrated quantile calculations from distributions*

Description

Illustrated quantile calculations from distributions

Usage

```
qdist(
  dist = "norm",
  p,
  plot = TRUE,
  verbose = FALSE,
  invisible = FALSE,
  resolution = 500L,
  digits = 3L,
  xlim,
  ylim,
  return = c("values", "plot"),
  refinements = list(),
```

```
  ...
)

xqgamma(
  p,
  shape,
  rate = 1,
  scale = 1/rate,
  lower.tail = TRUE,
  log.p = FALSE,
  ...
)

xqt(p, df, ncp, lower.tail = TRUE, log.p = FALSE, ...)

xqchisq(p, df, ncp = 0, lower.tail = TRUE, log.p = FALSE, ...)

xqf(p, df1, df2, lower.tail = TRUE, log.p = FALSE, ...)

xqbinom(p, size, prob, lower.tail = TRUE, log.p = FALSE, ...)

xqpois(p, lambda, lower.tail = TRUE, log.p = FALSE, ...)

xqgeom(p, prob, lower.tail = TRUE, log.p = FALSE, ...)

xqnbinom(p, size, prob, mu, lower.tail = TRUE, log.p = FALSE, ...)

xqbeta(p, shape1, shape2, ncp = 0, lower.tail = TRUE, log.p = FALSE, ...)
```

Arguments

`dist`	a character description of a distribution, for example `"norm"`, `"t"`, or `"chisq"`
`p`	a vector of probabilities
`plot`	a logical indicating whether a plot should be created
`verbose`	a logical
`invisible`	a logical
`resolution`	number of points used for detecting discreteness and generating plots. The default value of 5000 should work well except for discrete distributions that have many distinct values, especially if these values are not evenly spaced.
`digits`	the number of digits desired
`xlim`	x limits. By default, these are chosen to show the central 99.8\ of the distribution.
`ylim`	y limits
`return`	If `"plot"`, return a plot. If `"values"`, return a vector of numerical values.
`refinements`	A list of refinements to the plot. See `ggformula::gf_refine()`.

`...`	additional arguments, including parameters of the distribution and additional options for the plot. To help with name collisions (eg `size` for binomial distributions and `shape` for gamma distributions), argument names beginning `plot_` will be renamed to remove `plot_` and passed only to the plot. The unprefixed version will used as a parameter for the the distribution.
`shape, scale`	shape and scale parameters. Must be positive, `scale` strictly.
`rate`	an alternative way to specify the scale.
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \le x]$, otherwise, $P[X > x]$.
`log.p`	A logical indicating whether probabilities should be returned on the log scale.
`df`	degrees of freedom (> 0, maybe non-integer). `df = Inf` is allowed.
`ncp`	non-centrality parameter δ; currently except for `rt()`, only for `abs(ncp) <= 37.62`. If omitted, use the central t distribution.
`df1, df2`	degrees of freedom. `Inf` is allowed.
`size`	number of trials (zero or more).
`prob`	probability of success on each trial.
`lambda`	vector of (non-negative) means.
`mu`	alternative parametrization via mean: see 'Details'.
`shape1, shape2`	non-negative parameters of the Beta distribution.

Details

The most general function is `qdist` which can work with any distribution for which a q-function exists. As a convenience, wrappers are provided for several common distributions.

Value

a vector of quantiles; a plot is printed as a side effect

Examples

```
qdist("norm", seq(.1, .9, by = 0.10),
      title = "Deciles of a normal distribution", show.legend = FALSE,
      pattern = "rings")
xqnorm(seq(.2, .8, by = 0.20), mean = 100, sd = 10)
qdist("unif", .5)
xqgamma(.5, shape = 3, scale = 4)
xqgamma(.5, shape = 3, scale = 4, color = "black")
xqbeta(.5, shape1 = .9, shape2 = 1.4, dlwd = 1)
xqchisq(c(.25,.5,.75), df = 3)
xcbinom(c(0.80, 0.90), size = 1000, prob = 0.40)
# displayed as if continuous
xcbinom(c(0.80, 0.90), size = 5000, prob = 0.40)
xpbinom(c(480, 500, 520), size = 1000, prob = 0.48)
xpbinom(c(40, 60), size = 100, prob = 0.5)
xqpois(c(0.25, 0.5, 0.75), lambda = 12)
xcpois(0.50, lambda = 12)
xcpois(0.50, lambda = 12, refinements = list(scale_color_brewer(type = "qual", palette = 5)))
```

`rand` *Random Regressors*

Description

A utility function for producing random regressors with a specified number of degrees of freedom.

Usage

```
rand(df = 1, rdist = rnorm, args = list(), nrow, seed = NULL)
```

Arguments

`df`	degrees of freedom, i.e., number of random regressors
`rdist`	random distribution function for sampling
`args`	arguments for `rdist`
`nrow`	number of rows in resulting matrix. This can often be omitted in the context of functions like `lm` where it is inferred from the data frame, if one is provided.
`seed`	seed for random number generation

Value

A matrix of random variates with `df` columns. In its intended use, the number of rows will be selected to match the size of the data frame supplied to `lm`

Examples

```
rand(2,nrow=4)
rand(2,rdist=rpois, args=list(lambda=3), nrow=4)
summary(lm( waiting ~ eruptions + rand(1), faithful))
```

`read.file` *Read data files*

Description

A wrapper around various file reading functions.

Usage

```
read.file(
  file,
  header = T,
  na.strings = "NA",
  comment.char = NULL,
  filetype = c("default", "csv", "txt", "tsv", "fw", "rdata"),
  stringsAsFactors = FALSE,
  readr = FALSE,
  package = NULL,
  ...
)
```

Arguments

`file`	character: The name of the file which the data are to be read from. This may also be a complete URL or a path to a compressed 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. See `read.table()` for more details.
`header`	logical; For `.txt` and `.csv` files, this indicates whether the first line of the file includes variables names.
`na.strings`	character: strings that indicate missing data.
`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.
`filetype`	one of `"default"`, `"csv"`, `"txt"`, or `"rdata"` indicating the type of file being loaded. The default is to use the filename to guess the type of file.
`stringsAsFactors`	a logical indicating whether strings should be converted to factors. This has no affect when using `readr`.
`readr`	a logical indicating whether functions from the `readr` package should be used, if available.
`package`	if specified, files will be searched for among the documentation files provided by the package.
`...`	additional arguments passed on to `read.table()`, or `load()` or one of the functions in the `readr` package. Note that a message will indicate which underlying function is being used.

Details

Unless `filetype` is specified, `read.file` uses the (case insensitive) file extension to determine how to read data from the file. If `file` ends in `.rda` or `.rdata`, then `load()` is used to load the file. If `file` ends in `.csv`, then `readr::read_csv()` or `read.csv()` is used. Otherwise, `read.table()` is used.

Value

A data frame, unless `file` unless `filetype` is `"rdata"`, in which case arbitrary objects may be loaded and a character vector holding the names of the loaded objects is returned invisibly.

See Also

`read.csv()`, `read.table()`, `readr::read_table()`, `readr::read_csv()`, `load()`.

Examples

```
## Not run:
Dome <- read.file("http://www.mosaic-web.org/go/datasets/Dome.csv")

## End(Not run)
```

`relm` *Resample a Linear Model*

Description

Fit a new model to data created using `resample(model)`.

Usage

```
relm(model, ..., envir = environment(formula(model)))
```

Arguments

`model`	a linear model object produced using `lm()`.
`...`	additional arguments passed through to `resample()`.
`envir`	an environment in which to (re)evaluate the linear model.

See Also

`resample()`

Examples

```
mod <- lm(length ~ width, data = KidsFeet)
do(1) * mod
do(3) * relm(mod)
# use residual resampling to estimate standard error (very crude because so few replications)
Boot <- do(100) * relm(mod)
sd(~ width, data = Boot)
# standard error as produced by summary() for comparison
mod |> summary() |> coef()
```

`repeater-class` *Repeater objects*

Description

Repeater objects can be used with the `*` operator to repeat things multiple time using a different syntax and different output format from that used by, for example, `replicate()`.

Slots

- `n`**:** Object of class `"numeric"` indicating how many times to repeat something.
- `cull`**:** Object of class `"function"` that culls the output from each repetition.
- `mode`**:** Object of class `"character"` indicating the output mode ('default', 'data.frame', 'matrix', 'vector', or 'list'). For most purposes 'default' (the default) should suffice.
- `algorithm`**:** an algorithm number.
- `parallel`**:** a logical indicating whether to attempt parallel execution.

See Also

`do()`

`resample` *More Random Samples*

Description

These functions simplify and unify sampling in various ways.

Usage

```
resample(..., replace = TRUE)

deal(...)

shuffle(x, replace = FALSE, prob = NULL, groups = NULL, orig.ids = FALSE)

sample(x, size, replace = FALSE, ...)

## Default S3 method:
sample(
  x,
  size,
  replace = FALSE,
  prob = NULL,
```

```
  groups = NULL,
  orig.ids = FALSE,
  ...
)

## S3 method for class 'data.frame'
sample(
  x,
  size,
  replace = FALSE,
  prob = NULL,
  groups = NULL,
  orig.ids = TRUE,
  fixed = names(x),
  shuffled = c(),
  invisibly.return = NULL,
  ...
)

## S3 method for class 'matrix'
sample(
  x,
  size,
  replace = FALSE,
  prob = NULL,
  groups = NULL,
  orig.ids = FALSE,
  ...
)

## S3 method for class 'factor'
sample(
  x,
  size,
  replace = FALSE,
  prob = NULL,
  groups = NULL,
  orig.ids = FALSE,
  drop.unused.levels = FALSE,
  ...
)

## S3 method for class 'lm'
sample(
  x,
  size,
  replace = FALSE,
  prob = NULL,
```

```
  groups = NULL,
  orig.ids = FALSE,
  drop.unused.levels = FALSE,
  parametric = FALSE,
  transformation = NULL,
  ...
)
```

Arguments

`...`	additional arguments passed to `base::sample()` or `sample()`.
`replace`	Should sampling be with replacement?
`x`	Either a vector of one or more elements from which to choose, or a positive integer.
`prob`	A vector of probability weights for obtaining the elements of the vector being sampled.
`groups`	a vector (or variable in a data frame) specifying groups to sample within. This will be recycled if necessary.
`orig.ids`	a logical; should original ids be included in returned data frame?
`size`	a non-negative integer giving the number of items to choose.
`fixed`	a vector of column names. These variables are shuffled en masse, preserving associations among these columns.
`shuffled`	a vector of column names. these variables are reshuffled individually (within groups if `groups` is specified), breaking associations among these columns. examples.
`invisibly.return`	a logical, should return be invisible?
`drop.unused.levels`	a logical, should unused levels be dropped?
`parametric`	A logical indicating whether the resampling should be done parametrically.
`transformation`	NULL or a function providing a transformation to be applied to the synthetic responses. If NULL, an attempt it made to infer the appropriate transformation from the original call as recorded in `x`.

Details

These functions are wrappers around `sample()` providing different defaults and natural names.

Examples

```
# 100 Bernoulli trials -- no need for replace=TRUE
resample(0:1, 100)
tally(resample(0:1, 100))
if (require(mosaicData)) {
Small <- sample(KidsFeet, 10)
resample(Small)
```

```
tally(~ sex, data=resample(Small))
tally(~ sex, data=resample(Small))
# fixed marginals for sex
tally(~ sex, data=Small)
tally(~ sex, data=resample(Small, groups=sex))
# shuffled can be used to reshuffle some variables within groups
# orig.id shows where the values were in original data frame.
Small <- mutate(Small,
   id1 = paste(sex,1:10, sep=":"),
   id2 = paste(sex,1:10, sep=":"))
resample(Small, groups=sex, shuffled=c("id1","id2"))
}
deal(Cards, 13)    # A Bridge hand
shuffle(Cards)
model <- lm(width ~length * sex, data = KidsFeet)
KidsFeet |> head()
resample(model) |> head()
Boot <- do(500) * lm(width ~ length * sex, data = resample(KidsFeet))
df_stats(~ Intercept + length + sexG + length.sexG, data = Boot, sd)
head(Boot)
summary(coef(model))
```

`rescale` *Rescale*

Description

Rescale vectors or variables within data frames. This can be useful for comparing vectors that are on different scales, for example in parallel plots or heatmaps.

Usage

```
rescale(x, range, domain = NULL, ...)

## S3 method for class 'data.frame'
rescale(x, range = c(0, 1), domain = NULL, ...)

## S3 method for class 'factor'
rescale(x, range, domain = NULL, ...)

## S3 method for class 'numeric'
rescale(x, range = c(0, 1), domain = NULL, ...)

## Default S3 method:
rescale(x, range = c(0, 1), domain = NULL, ...)

## S3 method for class 'character'
rescale(x, range = c(0, 1), domain = NULL, ...)
```

Arguments

`x`	an R object to rescale
`range`	a numeric vector of length 2
`domain`	a numeric vector of length 2 or `NULL`
`...`	additional arguments

`rflip` *Tossing Coins*

Description

These functions simplify simulating coin tosses for those (students primarily) who are not yet familiar with the binomial distributions or just like this syntax and verbosity better.

Usage

```
rflip(
  n = 1,
  prob = 0.5,
  quiet = FALSE,
  verbose = !quiet,
  summarize = FALSE,
  summarise = summarize
)

## S3 method for class 'cointoss'
print(x, ...)

nflip(n = 1, prob = 0.5, ...)
```

Arguments

`n`	the number of coins to toss
`prob`	probability of heads on each toss
`quiet`	a logical. If `TRUE`, less verbose output is used.
`verbose`	a logical. If `TRUE`, more verbose output is used.
`summarize`	if `TRUE`, return a summary (as a data frame).
`summarise`	alternative spelling for `summarize`.
`x`	an object
`...`	additional arguments

Value

for `rflip`, a cointoss object

for `nflip`, a numeric vector

Examples

```
rflip(10)
rflip(10, prob = 1/6, quiet = TRUE)
rflip(10, prob = 1/6, summarize = TRUE)
do(5) * rflip(10)
as.numeric(rflip(10))
nflip(10)
```

`rfun` *Generate a natural-looking function*

Description

Produce a random function that is the sum of Gaussian random variables

`rpoly2` generates a random 2nd degree polynomial (as a function)

Usage

```
rfun(vars = ~x & y, seed = NULL, n = 0)

rpoly2(vars = ~x & y, seed = NULL)
```

Arguments

`vars`	a formula; the LHS is empty and the RHS indicates the variables used for input to the function (separated by &)
`seed`	seed for random number generator, passed to `set.seed()`.
`n`	the number of Gaussians. By default, this will be selected randomly.

Details

`rfun` is an easy way to generate a natural-looking but random function with ups and downs much as you might draw on paper. In two variables, it provides a good way to produce a random landscape that is smooth. Things happen in the domain -5 to 5. The function is pretty flat outside of that. Use `seed` to create a fixed function that will be the same for everybody

These functions are particularly useful for teaching calculus.

Value

a function with the appropriate number of inputs

a function defined by a 2nd degree polynomial with coefficients selected randomly according to a Unif(-1,1) distribution.

Examples

```
f <- rfun( ~ u & v)
plotFun(f(u,v)~u&v,u=range(-5,5),v=range(-5,5))
myfun <- rfun(~ u & v, seed=1959)
g <- rpoly2( ~ x&y&z, seed=1964)
plotFun(g(x,y,z=2)~x&y,xlim=range(-5,5),ylim=range(-5,5))
```

`rlatlon` *Sample longitude and latitude on a sphere*

Description

Randomly samples longitude and latitude on earth so that equal areas are (approximately) equally likely to be sampled. (Approximation assumes earth as a perfect sphere.)

Usage

```
rlatlon(...)

rlonlat(...)

rgeo(n = 1, latlim = c(-90, 90), lonlim = c(-180, 180), verbose = FALSE)

rgeo2(n = 1, latlim = c(-90, 90), lonlim = c(-180, 180), verbose = FALSE)
```

Arguments

`...`	arguments passed through to other functions
`n`	number of random locations
`latlim`, `lonlim`	range of latitudes and longitudes to sample within, only implemented for `rgeo`.
`verbose`	return verbose output that includes Euclidean coordinates on unit sphere as well as longitude and latitude.

Details

`rgeo` and `rgeo2` differ in the algorithms used to generate random positions. Each assumes a spherical globe. `rgeo` uses that fact that each of the x, y and z coordinates is uniformly distributed (but not independent of each other). Furthermore, the angle about the z-axis is uniformly distributed and independent of z. This provides a straightforward way to generate Euclidean coordinates using `runif`. These are then translated into latitude and longitude.

`rlatlon` is an alias for `rgeo` and `rlonlat` is too, expect that it reverses the order in which the latitude and longitude values are returned.

`rgeo2` samples points in a cube by independently sampling each coordinate. It then discards any point outside the sphere contained in the cube and projects the non-discarded points to the sphere. This method must oversample to allow for the discarded points.

Value

a data frame with variables `long` and `lat`. If `verbose` is TRUE, then x, y, and z coordinates are also included in the data frame.

See Also

`deg2rad()`, `googleMap()` and `latlon2xyz()`.

Examples

```
rgeo(4)
# sample from a region that contains the continental US
rgeo(4, latlim = c(25,50), lonlim = c(-65, -125))
rgeo2(4)
```

`rspin` *Simulate spinning a spinner*

Description

This is essentially `rmultinom` with a different interface.

Usage

```
rspin(n, probs, labels = 1:length(probs))
```

Arguments

`n`	number of spins of spinner
`probs`	a vector of probabilities. If the sum is not 1, the probabilities will be rescaled.
`labels`	a character vector of labels for the categories

Examples

```
rspin(20, prob=c(1,2,3), labels=c("Red", "Blue", "Green"))
do(2) * rspin(20, prob=c(1,2,3), labels=c("Red", "Blue", "Green"))
```

`rsquared` *Extract r-squared value*

Description

Attempts to extract an r-squared value from a model or model-like object.

Usage

```
rsquared(x, ...)
```

Arguments

`x`	an object
`...`	additional arguments

`rstudio_is_available` *Check whether RStudio is in use*

Description

This functions checks that RStudio is in use. It will likely be removed from this package once the versions of RStudio in popular use rely on the manipulate package on CRAN which will provide its own version.

Usage

```
rstudio_is_available()
```

Value

a logical

`set.rseed` *Set seed in parallel compatible way*

Description

When the parallel package is used, setting the RNG seed for reproducibility involves more than simply calling `set.seed()`. `set.rseed` takes care of the additional overhead.

Usage

```
set.rseed(seed)
```

Arguments

`seed`	seed for the random number generator

Details

If the `parallel` package is not on the search path, then `set.seed()` is called. If `parallel` is on the search path, then the RNG kind is set to `"L'Ecuyer-CMRG"`, the seed is set and `mc.reset.stream` is called.

Examples

```
# These should give identical results, even if the `parallel' package is loaded.
set.rseed(123); do(3) * resample(1:10, 2)
set.rseed(123); do(3) * resample(1:10, 2)
```

`Sleep` *Sleep and Memory*

Description

Sleep and Memory

Usage

```
data(Sleep)
```

Format

A data.frame with 24 observations on the following 2 variables.

`Group`	treatment group of the subject
`Words`	number of words recalled

Details

In an experiment on memory (Mednicj et al, 2008), students were given lists of 24 words to memorize. After hearing the words they were assigned at random to different groups. One group of 12 students took a nap for 1.5 hours while a second group of 12 students stayed awake and was given a caffeine pill. The data set records the number of words each participant was able to recall after the break.

Source

These data were used in a "resampling bake-off" hosted by Robin Lock.

`sp2df` *Transforms a shapefile into a dataframe*

Description

This function takes in a shapefile (formal class of `SpatialPolygonsDataFrame`) and transforms it into a dataframe

Usage

```
sp2df(map, ...)
```

Arguments

`map`	A map object of class `SpatialPolygonsDataFrame`
`...`	Other arguments, currently ignored

Value

A dataframe, in which the first 7 columns hold geographical information (ex: `long` and `lat`)

Examples

```
## Not run:
if(require(maptools)) {
  data(wrld_simpl)
  worldmap <- sp2df(wrld_simpl)
}

if ( require(ggplot2) && require(maptools) ) {
  data(wrld_simpl)
  World <- sp2df(wrld_simpl)
  World2 <- merge(World, Countries, by.x="NAME", by.y="maptools", all.y=FALSE)
  Mdata <- merge(Alcohol, World2, by.x="country", by.y="gapminder", all.y=FALSE)
  Mdata <- Mdata[order(Mdata$order),]
  qplot( x=long, y=lat, fill=ntiles(alcohol,5),
         data=subset(Mdata, year==2008), group = group,
```

```
                       geom="polygon")
}

## End(Not run)
```

`standardName` *Standardization of Geographic Names*

Description

Often different sources of geographical data will use different names for the same region. These utilities make it easier to merge data from different sources by converting names to standardized forms.

Usage

```
standardName(
  x,
  standard,
  ignore.case = TRUE,
  returnAlternatives = FALSE,
  quiet = FALSE
)

standardCountry(
  x,
  ignore.case = TRUE,
  returnAlternatives = FALSE,
  quiet = FALSE
)

standardState(x, ignore.case = TRUE, returnAlternatives = FALSE, quiet = FALSE)
```

Arguments

`x`	A vector with the region names to standardize
`standard`	a named vector providing the map from non-standard names (names of vector) to standard names (values of vector)
`ignore.case`	a logical indicating whether case should be ignored when matching.
`returnAlternatives`	a logical indicating whether all alternatives should be returned in addition to the standard name.
`quiet`	a logical indicating whether warnings should be suppressed

Details

- **standardName** This is the most general standardizing function. In addition to `x`, this function requires another argument: `standard` - a named vector in which each name is a particular spelling of the region name in question and the corresponding value is the standardized version of that region name
- **standardCountry** This function will standardize the country names in `x` to the standard ISO_a3 country code format. If `returnAlternatives` is set to `TRUE`, this function will also return the the named vector used to standardize the country names
- **standardState** This function will standardize the US state names in `x` to the standard two-letter abbreviations. If `returnAlternatives` is set to `TRUE`, this function will also return the the named vector used to standardize the state names

In all three cases, any names not found in `standard` will be left unaltered. Unless supressed, a warning message will indicate the number of such cases, if there are any.

`statTally` *Tally test statistics*

Description

Tally test statistics from data and from multiple draws from a simulated null distribution

Usage

```
statTally(
  sample,
  rdata,
  FUN,
  direction = NULL,
  alternative = c("default", "two.sided", "less", "greater"),
  sig.level = 0.1,
  system = c("gg", "lattice"),
  shade = "navy",
  alpha = 0.1,
  binwidth = NULL,
  bins = NULL,
  fill = "gray80",
  color = "black",
  center = NULL,
  stemplot = dim(rdata)[direction] < 201,
  q = c(0.5, 0.9, 0.95, 0.99),
  fun = function(x) x,
  xlim,
  quiet = FALSE,
  ...
)
```

Arguments

`sample`	sample data
`rdata`	a matrix of randomly generated data under null hypothesis.
`FUN`	a function that computes the test statistic from a data set. The default value does nothing, making it easy to use this to tabulate precomputed statistics into a null distribution. See the examples.
`direction`	1 or 2 indicating whether samples in `rdata` are in rows (1) or columns (2).
`alternative`	one of `default`, `two.sided`, `less`, or `greater`
`sig.level`	significance threshold for `wilcox.test` used to detect lack of symmetry
`system`	graphics system to use for the plot
`shade`	a color to use for shading.
`alpha`	opacity of shading.
`binwidth`	bin width for histogram.
`bins`	number of bins for histogram.
`fill`	fill color for histogram.
`color`	border color for histogram.
`center`	center of null distribution
`stemplot`	indicates whether a stem plot should be displayed
`q`	quantiles of sampling distribution to display
`fun`	same as `FUN` so you don't have to remember if it should be capitalized
`xlim`	limits for the horizontal axis of the plot.
`quiet`	a logicial indicating whether the text output should be suppressed
`...`	additional arguments passed to `lattice::histogram()` or `ggplot2::geom_histogram()`

Value

A lattice or ggplot showing the sampling distribution.

As side effects, information about the empirical sampling distribution and (optionally) a stem plot are printed to the screen.

Examples

```
# is my spinner fair?
x <- c(10, 18, 9, 15)   # counts in four cells
rdata <- rmultinom(999, sum(x), prob = rep(.25, 4))
statTally(x, rdata, fun = max, binwidth = 1)  # unusual test statistic
statTally(x, rdata, fun = var, shade = "red", binwidth = 2)  # equivalent to chi-squared test
# Can also be used with test stats that are precomputed.
if (require(mosaicData)) {
D <- diffmean( age ~ sex, data = HELPrct); D
nullDist <- do(999) * diffmean( age ~ shuffle(sex), data = HELPrct)
statTally(D, nullDist)
statTally(D, nullDist, system = "lattice")
}
```

`surround` *Format strings for pretty output*

Description

Format strings for pretty output

Usage

```
surround(x, pre = " ", post = " ", width = 8, ...)
```

Arguments

`x`	a vector
`pre`	text to prepend onto string
`post`	text to postpend onto string
`width`	desired width of string
`...`	additional arguments passed to `format()`

Value

a vector of strings padded to the desired width

Examples

```
surround(rbinom(10,20,.5), " ", " ", width=4)
surround(rnorm(10), " ", " ", width=8, digits = 2, nsmall = 2)
```

`swap` *Swap values among columns of a data frame*

Description

Swap values among columns of a data frame

Usage

```
swap(data, which)
```

Arguments

`data`	a data frame
`which`	a formula or an integer or character vector specifying columns in `data`

Details

`swap` is not a particularly speedy function. It is intended primarily as an aid for teaching randomization for paired designs. Used this way, the number of randomizations should be kept modest (approximately 1000) unless you are very patient.

Examples

```
if (require(tidyr)) {
  Sleep2 <- sleep |> spread( key=group, val=extra )
  names(Sleep2) <- c("subject", "drug1", "drug2")
  swap(Sleep2, drug1 ~ drug2)
  mean( ~(drug1 - drug2), data=Sleep2)
  do(3) * mean( ~(drug1 - drug2), data=Sleep2 |> swap(drug1 ~ drug2) )
}
```

`theme.mosaic` *Lattice Theme*

Description

A theme for use with lattice graphics.

Usage

```
theme.mosaic(bw = FALSE, lty = if (bw) 1:7 else 1, lwd = 2, ...)

col.mosaic(bw = FALSE, lty = if (bw) 1:7 else 1, lwd = 2, ...)
```

Arguments

`bw`	whether color scheme should be "black and white"
`lty`	vector of line type codes
`lwd`	vector of line widths
`...`	additional named arguments passed to `trellis.par.set()`

Value

Returns a list that can be supplied as the `theme` to `trellis.par.set()`.

Note

These two functions are identical. `col.mosaic` is named similarly to `lattice::col.whitebg()`, but since more than just colors are set, `theme.mosaic` is a preferable name.

See Also

`trellis.par.set()`, `show.settings()`

Examples

```
trellis.par.set(theme=theme.mosaic())
show.settings()
trellis.par.set(theme=theme.mosaic(bw=TRUE))
show.settings()
```

`theme_map` *ggplot2 theme for maps*

Description

A very plain **ggplot2** theme that is good for maps.

Usage

```
theme_map(base_size = 12)
```

Arguments

`base_size`	the base font size for the theme.

Details

This theme is largely based on an example posted by Winston Chang at the **ggplot2** Google group forum.

`TukeyHSD.lm` *Additional interfaces to TukeyHSD*

Description

`TukeyHSD()` requires use of `aov()`. Since this is a hindrance for beginners, wrappers have been provided to remove this need.

Usage

```
## S3 method for class 'lm'
TukeyHSD(x, which, ordered = FALSE, conf.level = 0.95, ...)

## S3 method for class 'formula'
TukeyHSD(
  x,
  which,
  ordered = FALSE,
  conf.level = 0.95,
```

```
  data = parent.frame(),
  ...
)
```

Arguments

`x`	an object, for example of class `lm` or `formula`
`which`, `ordered`, `conf.level`, `...`	just as in `TukeyHSD()` from the base package
`data`	a data frame. NB: This does not come second in the argument list.

Examples

```
## These should all give the same results
if (require(mosaicData)) {
  model <- lm(age ~ substance, data=HELPrct)
  TukeyHSD(model)
  TukeyHSD( age ~ substance, data=HELPrct)
  TukeyHSD(aov(age ~ substance, data=HELPrct))
}
```

`t_test` *Student's t-Test*

Description

Performs one and two sample t-tests. The mosaic `t.test` provides wrapper functions around the function of the same name in **stats**. These wrappers provide an extended interface that allows for a more systematic use of the formula interface.

Usage

```
t_test(x, ...)

t.test(x, ...)

## S3 method for class 'formula'
t_test(formula, data, ..., groups = NULL)

## Default S3 method:
t_test(
  x,
  y = NULL,
  alternative = c("two.sided", "less", "greater"),
  mu = 0,
  paired = FALSE,
  var.equal = FALSE,
```

```
  conf.level = 0.95,
  ...
)
```

Arguments

`x`	a (non-empty) numeric vector of data values.
`...`	further arguments to be passed to or from methods.
`formula`	a formula of the form `lhs ~ rhs` where `lhs` is a numeric variable giving the data values and `rhs` either `1` for a one-sample or paired test or a factor with two levels giving the corresponding groups. If `lhs` is of class `"Pair"` and `rhs` is `1`, a paired test is done
`data`	an optional matrix or data frame (or similar: see `model.frame`) containing the variables in the formula `formula`. By default the variables are taken from `environment(formula)`.
`groups`	When `x` is a formula, `groups` can be used to compare groups: `x = ~ var, groups = g` is equivalent to `x = var ~ g`. See the examples.
`y`	an optional (non-empty) numeric vector of data values.
`alternative`	a character string specifying the alternative hypothesis, must be one of `"two.sided"` (default), `"greater"` or `"less"`. You can specify just the initial letter.
`mu`	a number indicating the true value of the mean (or difference in means if you are performing a two sample test).
`paired`	a logical indicating whether you want a paired t-test.
`var.equal`	a logical variable indicating whether to treat the two variances as being equal. If `TRUE` then the pooled variance is used to estimate the variance otherwise the Welch (or Satterthwaite) approximation to the degrees of freedom is used.
`conf.level`	confidence level of the interval.

Details

This is a wrapper around `stats::t.test()` from the **stats** package to extend the functionality of the formula interface. In particular, one can now use the formula interface for a 1-sample t-test. Before, the formula interface was only permitted for a 2-sample test. The type of formula that can be used for the 2-sample test has also be broadened. See the examples.

Value

an object of class `htest`

See Also

`prop.test()`, `binom.test()`, `stats::t.test()`

Examples

```
t.test(HELPrct$age)
# We can now do this with a formula
t.test(~ age, data = HELPrct)
# data = can be omitted, but it is better to use it
t.test(~ age, HELPrct)
# the original 2-sample formula
t.test(age ~ sex, data = HELPrct)
# alternative 2-sample formulas
t.test(~ age | sex, data = HELPrct)
t.test(~ age, groups = sex, data = HELPrct)
# 2-sample t from vectors
with(HELPrct, t.test(age[sex == "male"], age[sex == "female"]))
# just the means
mean(age ~ sex, data = HELPrct)
```

`update_ci` *Update confidence interval*

Description

Update the confidence interval portion of an object returned from `binom.test` using one of several alternative methods.

Usage

```
update_ci(
  object,
 method = c("clopper-pearson", "wald", "agresti-coull", "plus4", "score", "prop.test")
)
```

Arguments

`object`	An `"htest"` object produced by `binom.test()`
`method`	a method for computing a confidence interval for a proportion.

Value

an `"htest"` object with an updated confidence interval

See Also

`binom.test()`

`value` *Extract value from an object*

Description

Functions like `integrate()` and `nlm()` return objects that contain more information that simply the value of the integration or optimization. `value()` extracts the primary value from such objects. Currently implemented situations include the output from `integrate()`, `nlm()`, `cubature::adaptIntegrate()`, and `uniroot()`.

Usage

```
value(object, ...)

## S3 method for class 'integrate'
value(object, ...)

## Default S3 method:
value(object, ...)
```

Arguments

`object`	an object from which a "value" is to be extracted.
`...`	additional arguments (currently ignored).

Examples

```
integrate(sin, 0, 1) |> value()
nlm(cos, p = 0) |> value()
uniroot(cos, c(0, 2)) |> value()
```

`xchisq.test` *Augmented Chi-squared test*

Description

This augmented version of `chisq.test()` provides more verbose output.

Usage

```
xchisq.test(
  x,
  y = NULL,
  correct = TRUE,
  p = rep(1/length(x), length(x)),
```

```
  rescale.p = FALSE,
  simulate.p.value = FALSE,
  B = 2000,
  data = environment(x)
)
```

Arguments

`x, y, correct, p, rescale.p, simulate.p.value, B`	as in `chisq.test()`, but `x` may also be a formula, in which case `x` is replaced by `tally(x, data)` prior to the call to `chisq.test()`.
`data`	a data frame for use when `x` is a formula.

See Also

`chisq.test()`

Examples

```
# Physicians' Health Study data
phs <- cbind(c(104,189),c(10933,10845))
rownames(phs) <- c("aspirin","placebo")
colnames(phs) <- c("heart attack","no heart attack")
phs
xchisq.test(phs)
xchisq.test(sex ~ substance, data = HELPrct)
```

`xhistogramBreaks` *Augmented histograms*

Description

The **mosaic** package adds some additional functionality to `lattice::histogram()`, making it simpler to obtain certain common histogram adornments. This is done be resetting the default panel and prepanel functions used by histogram.

Usage

```
xhistogramBreaks(x, center = NULL, width = NULL, nint, ...)

prepanel.xhistogram(x, breaks = xhistogramBreaks, ...)

panel.xhistogram(
  x,
  dcol = trellis.par.get("plot.line")$col,
  dalpha = 1,
  dlwd = 2,
  gcol = trellis.par.get("add.line")$col,
```

```
  glwd = 2,
  fcol = trellis.par.get("superpose.polygon")$col,
  dmath = dnorm,
  verbose = FALSE,
  dn = 100,
  args = NULL,
  labels = FALSE,
  density = NULL,
  under = FALSE,
  fit = NULL,
  start = NULL,
  type = "density",
  v,
  h,
  groups = NULL,
  center = NULL,
  width = NULL,
  breaks,
  nint = round(1.5 * log2(length(x)) + 1),
  stripes = c("vertical", "horizontal", "none"),
  alpha = 1,
  ...
)
```

Arguments

`x`	a formula or a numeric vector
`center`	center of one of the bins
`width`	width of the bins
`nint`	approximate number of bins
`...`	additional arguments passed from `lattice::histogram()` to the panel function; by default when the **mosaic** package has been loaded this will be `panel.xhistogram()`.
`breaks`	break points for histogram bins, a function for computing such, or a method `hist()` knows about given as a character string. When using the **mosaic** package defaults, `xhistogramBreaks()` is used.
`dcol`	color of density curve
`dalpha`	alpha for density curve
`dlwd, glwd`	like `lwd` but affecting the density line and guide lines, respectively
`gcol`	color of guidelines
`fcol`	fill colors for histogram rectangles when using `groups`. (Use `col`, which is passed through to the histogram panel function, when not using `groups`.)
`dmath`	density function for density curve overlay
`verbose`	be verbose?
`dn`	number of points to sample from density curve
`args`	a list of additional arguments for `dmath`

`labels`	should counts/densities/percents be displayed or each bin?
`density`	a logical indicating whether to overlay a density curve
`under`	a logical indicating whether the density layers should be under or over other layers of the plot.
`fit`	a character string describing the distribution to fit. Known distributions include `"exponential"`, `"normal"`, `"lognormal"` , `"poisson"`, `"beta"`, `"geometric"`, `"t"`, `"weibull"`, `"cauchy"`, `"gamma"`, `"chisq"`, and `"chi-squared"`
`start`	numeric value passed to `MASS::fitdistr()`
`type`	one of `'density'`, `'count'`, or `'percent'`
`h, v`	a vector of values for additional horizontal and vertical lines
`groups`	as per `lattice::histogram()`
`stripes`	one of `"vertical"`, `"horizontal"`, or `"none"`, indicating how bins should be striped when `groups` is not `NULL`
`alpha`	transparency level
`panel`	a panel function

Details

The primary additional functionality added to `histogram()` are the arguments `width` and `center` which provide a simple way of describing equal-sized bins, and `fit` which can be used to overlay the density curve for one of several distributions. The `groups` argument can be used to color the bins. The primary use for this is to shade tails of histograms, but there may be other uses as well.

Value

`xhistogramBreaks` returns a vector of break points

Note

Versions of **lattice** since 0.20-21 support setting custom defaults for `breaks`, `panel`, and `prepanel` used by `histogram()`, so `xhistogram()` is no longer needed. As a result, `xhistogram()` (which was required in earlier versions of **mosaic** is no longer needed and has been removed.

See Also

`lattice::histogram()`, `mosaicLatticeOptions()`, and `restoreLatticeOptions()`.

Examples

```
if (require(mosaicData)) {
histogram(~age | substance, HELPrct, v=35, fit='normal')
histogram(~age, HELPrct, labels=TRUE, type='count')
histogram(~age, HELPrct, groups=cut(age, seq(10,80,by=10)))
histogram(~age, HELPrct, groups=sex, stripes='horizontal')
histogram(~racegrp, HELPrct, groups=substance,auto.key=TRUE)
xhistogramBreaks(1:10, center=5, width=1)
xhistogramBreaks(1:10, center=5, width=2)
```

```
xhistogramBreaks(0:10, center=15, width=3)
xhistogramBreaks(1:100, center=50, width=3)
xhistogramBreaks(0:10, center=5, nint=5)
}
```

`xpnorm` *Augmented versions of pnorm and qnorm*

Description

These functions behave similarly to the functions with the initial `x` removed from their names but add more verbose output and graphics.

Usage

```
xpnorm(
  q,
  mean = 0,
  sd = 1,
  plot = TRUE,
  verbose = TRUE,
  invisible = FALSE,
  digits = 4,
  lower.tail = TRUE,
  log.p = FALSE,
  xlim = mean + c(-4, 4) * sd,
  ylim = c(0, 1.4 * dnorm(mean, mean, sd)),
  manipulate = FALSE,
  ...,
  return = c("value", "plot")
)

xqnorm(
  p,
  mean = 0,
  sd = 1,
  plot = TRUE,
  verbose = TRUE,
  digits = getOption("digits"),
  lower.tail = TRUE,
  log.p = FALSE,
  xlim,
  ylim,
  invisible = FALSE,
  ...,
  return = c("value", "plot"),
  pattern = c("stripes", "rings")
```

```
)

xcnorm(
  p,
  mean = 0,
  sd = 1,
  plot = TRUE,
  verbose = TRUE,
  digits = getOption("digits"),
  lower.tail = TRUE,
  log.p = FALSE,
  xlim,
  ylim,
  invisible = FALSE,
  ...,
  return = c("value", "plot"),
  pattern = "rings"
)
```

Arguments

`q`	quantile
`mean, sd`	parameters of normal distribution.
`plot`	logical. If TRUE, show an illustrative plot.
`verbose`	logical. If TRUE, display verbose output.
`invisible`	logical. If TRUE, return value invisibly.
`digits`	number of digits to display in output.
`lower.tail`	logical. If FALSE, use upper tail probabilities.
`log.p`	logical. If TRUE, uses the log of probabilities.
`xlim, ylim`	limits for plotting.
`manipulate`	logical. If TRUE and in RStudio, then sliders are added for interactivity.
`...`	additional arguments.
`return`	If `"plot"`, return a plot. If `"values"`, return a vector of numerical values.
`p`	probability
`pattern`	One of `"stripes"` or `"rings"`. In the latter case, pairs of regions (from inside to outside) are grouped together for coloring and probability calculation.

See Also

`histogram()`, `chisq.test()`, `pnorm()`, `qnorm()`, `qqmath()`, and `plot()`.

Examples

```
xpnorm(650, 500, 100)
xqnorm(.75, 500, 100)
```

```
xpnorm(-3:3, return = "plot", system = "gg") |>
  gf_labs(title = "My Plot", x = "") |>
  gf_theme(theme_bw())

## Not run:
if (rstudio_is_available() & require(manipulate)) {
  manipulate(xpnorm(score, 500, 100, verbose = verbose),
    score = slider(200, 800),
    verbose = checkbox(TRUE, label = "Verbose Output")
  )
}

## End(Not run)
```

`xqqmath` *Augmented version of* `qqmath`

Description

Augmented version of `qqmath`

Usage

```
xqqmath(x, data = NULL, panel = "panel.xqqmath", ...)

panel.xqqmath(
  x,
  qqmathline = !(fitline || idline),
  idline = FALSE,
  fitline = NULL,
  slope = NULL,
  intercept = NULL,
  overlines = FALSE,
  groups = NULL,
  ...,
  col.line = trellis.par.get("add.line")$col,
  pch = 16,
  lwd = 2,
  lty = 2
)
```

Arguments

`x, data, panel, ...`	as in `lattice::qqmath()`
`qqmathline`	a logical: should line be displayed passing through first and third quartiles?
`idline`	a logical; should the line y=x be added to the plot?

`fitline`	a logical; should a fitted line be added to plot? Such a line will use `slope` and `intercept` if provided, else the standard deviation and mean of the data. If `slope` is specified, the line will be added unless `fitline` is `FALSE`.
`slope`	slope for added line
`intercept`	intercept for added line
`overlines`	a logical: should lines be on top of qq plot?
`groups, pch, lwd, lty`	as in lattice plots
`col.line`	color to use for added lines

Value

a trellis object

Examples

```
x <- rnorm(100)
xqqmath( ~ x)                  # with quartile line
xqqmath( ~ x, fitline = TRUE)  # with fitted line
xqqmath( ~ x, idline = TRUE)   # with y = x
x <- rexp(100, rate = 10)
xqqmath( ~ x, distribution = qexp)     # with quartile line
xqqmath( ~ x, distribution = qexp, slope = 1/10)
xqqmath( ~ x, distribution = qexp, slope = mean(x))
```

`xyz2latlon` *Convert back and forth between latitude/longitude and XYZ-space*

Description

Convert back and forth between latitude/longitude and XYZ-space

Usage

```
xyz2latlon(x, y, z)

latlon2xyz(latitude, longitude)

lonlat2xyz(longitude, latitude)
```

Arguments

`x, y, z`	numeric vectors
`latitude, longitude`	vectors of latitude and longitude values

Value

a matrix each row of which describes the latitudes and longitudes

a matrix each row of which contains the x, y, and z coordinates of a point on a unit sphere

See Also

`deg2rad()`, `googleMap()`, and `rgeo()`.

Examples

```
xyz2latlon(1, 1, 1)     # point may be on sphere of any radius
xyz2latlon(0, 0, 0)     # this produces a NaN for latitude
latlon2xyz(30, 45)
lonlat2xyz(45, 30)
```

`zscore` *Compute z-scores*

Description

Compute z-scores

Usage

```
zscore(x, na.rm = getOption("na.rm", FALSE))
```

Arguments

`x`	a numeric vector
`na.rm`	a logical indicating whether missing values should be removed

Examples

```
data(penguins, package = "palmerpenguins")
penguins |>
  group_by(species) |>
  mutate(zbill_length_mm = zscore(bill_length_mm, na.rm = TRUE)) |>
  head()
```

Index