Package 'spatstat.utils'

September 20, 2025

Version 3.2-0

Date 2025-09-20

Title Utility Functions for 'spatstat'

Maintainer Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>

Depends R (>= 3.5.0), stats, graphics, grDevices, utils, methods

Suggests spatstat.model

Description Contains utility functions for the 'spatstat' family of packages which may also be useful for other purposes.

License GPL (>= 2)

URL http://spatstat.org/

NeedsCompilation yes

ByteCompile true

BugReports https://github.com/spatstat/spatstat.utils/issues

Author Adrian Baddeley [aut, cre] (ORCID: <https://orcid.org/0000-0001-9499-8382>), Rolf Turner [aut] (ORCID: <https://orcid.org/0000-0001-5521-5218>), Ege Rubak [aut] (ORCID: <https://orcid.org/0000-0002-6675-533X>)

Repository CRAN

Date/Publication 2025-09-20 05:10:03 UTC

Contents

`spatstat.utils-package`

The spatstat.utils Package

Description

The **spatstat.utils** package contains low-level utilities, written for the **spatstat** package, which may be useful in other packages as well.

Details

The functions in **spatstat.utils** were originally written as internal, undocumented, utility functions in the **spatstat** package.

Many of these functions could be useful to other programmers, so we have made them available in a separate package **spatstat.utils** and provided documentation.

The functionality contained in **spatstat.utils** includes:

Factorisation of integers Find prime numbers (`primesbelow`), factorise a composite number into its prime factors (`primefactors`), determine whether a number is prime (`is.prime`) or whether two numbers are relatively prime (`relatively.prime`), and find the least common multiple or greatest common divisor of two numbers (`least.common.multiple`, `greatest.common.divisor`).

Faster versions of basic R tools Faster versions of some basic R tools and idioms are provided. These are only faster in particular cases, but if you know that your data have a particular form, the acceleration can be substantial. See `ifelseAB`, `fave.order`, `revcumsum`, `tapplysum`.

Grammar Use the correct word in English to refer to an ordinal number (`ordinal`, `ordinalsuffix`) and the correct indefinite article (`articlebeforenumber`).

Tools for generating printed output The function `splat` is a replacement for `cat(paste(...))` which ensures that output stays inside the declared text margin (`getOption("width")`) and can also perform automatic indentation. There are useful functions to add or remove parentheses (`paren`, `unparen`) and to make comma-separated lists (`commasep`).

Handling intervals (ranges) of real numbers Simple functions handle an interval (range) of numerical values: `check.range`, `intersect.ranges`, `inside.range`, `check.in.range`, `prange`.

Handling a formula Tools for handling a formula in the R language include `lhs.of.formula`, `rhs.of.formula`, `variablesinformula`, `termsinformula`, `offsetsinformula`, `can.be.formula` and `identical.formulae`.

Polynomials There are tools for creating and manipulating symbolic expressions for polynomials, as they might appear in a formula (`sympoly`, `expand.polynom`).

Validating arguments There are many tools for validating an argument and generating a comprehensible error or warning message if the argument is not valid: `check.1.integer`, `check.nvector`, `check.named.vector`.

Passing arguments There are many tools for calling a function while passing only some of the arguments in a supplied list of arguments: `do.call.matched`, `do.call.without`, `resolve.defaults`.

Traced optimization `optimizeWithTrace` is a simple wrapper for the one-dimensional optimization routine `optimize`. It stores the values of the function argument each time it is called, stores the resulting function values, and returns them along with the optimal value.

Workarounds There are workarounds for known bugs or undesirable features in other software. `spatstatLocator` is a replacement for `locator` which works around a bug in the `RStudio` graphics interface. `cat.factor` concatenates several factors, merging the levels, to produce a new factor.

Licence

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <rolfturner@posteo.net> and Ege Rubak <rubak@math.aau.dk>.

---

articlebeforenumber *Indefinite Article Preceding A Number*

Description

Determines the indefinite article (*an* or *a*) which should precede a given number, if the number is read out in English.

Usage

```
articlebeforenumber(k, teenhundreds=FALSE)
```

Arguments

k	A single number.
teenhundreds	Logical value specifying that (for example) 1800 should be read as "eighteen hundred" instead of "one thousand eight hundred". See Details.

Details

This function applies the rule that, if the English language word or phrase for the number k begins with a vowel, then it should be preceded by *an*, and otherwise by *a*.

If teenhundreds=FALSE (the default), the numbers 1100 and 1800 will be read as 'one thousand one hundred' and 'one thousand eight hundred', and the indefinite article will be *a*. However if teenhundreds=TRUE, the numbers 1100 and 1800 be read as 'eleven hundred' and 'eighteen hundred' and the indefinite article will be *an*.

Value

One of the character strings "an" or "a".

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

See Also

ordinal

Examples

```
f <- function(k) cat(paste(articlebeforenumber(k),
                     paste0(k, "-fold"),
                     "increase\n"))
f(8)
f(18)
f(28)
```

`cat.factor` *Combine Several Factors*

Description

Combine (concatenate) several factor objects, to produce a factor.

Usage

```
cat.factor(...)
```

Arguments

`...`	Any number of arguments. Each argument should be a factor, or will be converted to a factor.

Details

The arguments `...` are concatenated as they would be using `c()` or `cat()`, except that factor levels are retained and merged correctly. See the Examples.

Value

A factor, whose length is the sum of the lengths of all arguments. The levels of the resulting factor are the union of the levels of the arguments.

Author(s)

Rolf Turner `<rolfturner@posteo.net>`.

See Also

`c`.

Examples

```
f <- factor(letters[1:3])
g <- factor(letters[3:5])
f
g
cat(f,g)
c(f,g)
cat.factor(f, g)
```

---

check.1.integer *Check Argument Type and Length*

Description

These utility functions check whether a given argument is a single value of the required type.

Usage

```
check.1.real(x, context = "", fatal = TRUE, warn=TRUE)
check.1.integer(x, context = "", fatal = TRUE, warn=TRUE)
check.1.string(x, context = "", fatal = TRUE, warn=TRUE)
```

Arguments

x	The argument to be checked.
context	Optional string describing the context in which the argument is checked.
fatal	Logical value indicating whether a fatal error should occur when x is not of the required type.
warn	Logical value indicating whether to issue a warning message if x is not of the required type.

Details

These functions check whether the argument x is a single atomic value of type numeric, integer or character.

If x does have the required length and type, the result of the function is the logical value TRUE.

Otherwise, if fatal=TRUE (the default) an error occurs, while if fatal=FALSE a warning is issued (if warn=TRUE) and the function returns the value FALSE.

Value

A logical value (or an error may occur).

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

See Also

check.named.vector

Examples

```
x <- pi
check.1.real(x)
check.1.integer(pi, fatal=FALSE, context="In your dreams,")
check.1.string(x, fatal=FALSE)
check.1.integer(x, fatal=FALSE, warn=FALSE)
```

---

check.anyvector *Check For Vector or Factor With Correct Length*

Description

This is a programmer's utility function to check whether the argument is a vector or factor of the correct length.

Usage

```
check.anyvector(v, npoints = NULL, fatal = TRUE, things = "data points",
                naok = FALSE, warn = FALSE, vname, oneok = FALSE)
```

Arguments

v	The argument to be checked.
npoints	The required length of v.
fatal	Logical value indicating whether to stop with an error message if v does not satisfy all requirements.
things	Character string describing what the entries of v should correspond to.
naok	Logical value indicating whether NA values are permitted.
warn	Logical value indicating whether to issue a warning if v does not satisfy all requirements.
vname	Character string giving the name of v to be used in messages.
oneok	Logical value indicating whether v is permitted to have length 1.

Details

This function checks whether v is a vector or factor with length equal to npoints (or length equal to 1 if oneok=TRUE), not containing any NA values (unless naok=TRUE).

If these requirements are all satisfied, the result is the logical value TRUE.

If not, then if fatal=TRUE (the default), an error occurs; if fatal=FALSE, the result is the logical value FALSE with an attribute describing the requirement that was not satisfied.

Value

A logical value indicating whether all the requirements were satisfied. If FALSE, then this value has an attribute "whinge", a character string describing the requirements that were not satisfied.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

See Also

check.nvector, check.nmatrix, check.1.real, check.named.vector.

Examples

```
z <- factor(letters[1:10])
y <- z[1]
check.anyvector(z, 5, fatal=FALSE)
check.anyvector(y, 5, oneok=TRUE)
check.anyvector(42, 5, fatal=FALSE)
```

---

check.named.vector *Check Whether Object Has Required Components*

Description

These functions check whether the object x has components with the required names, and does not have any unexpected components.

Usage

```
check.named.vector(x, nam, context = "", namopt = character(0),
                   onError = c("fatal", "null"), xtitle=NULL)

check.named.list(x, nam, context = "", namopt = character(0),
                   onError = c("fatal", "null"), xtitle=NULL)

check.named.thing(x, nam, namopt = character(0),
                  xtitle = NULL, valid = TRUE, type = "object",
                  context = "", fatal = TRUE)
```

Arguments

x	The object to be checked.
nam	Vector of character strings giving the names of all the components which must be present.
namopt	Vector of character strings giving the names of components which may optionally be present.

context	Character string describing the context in which x is being checked, for use in error messages.
xtitle	Optional character string to be used when referring to x in error messages.
valid	Logical value indicating whether x belongs to the required class of objects.
type	Character string describing the required class of objects.
onError	Character string indicating what to do if x fails the checks.
fatal	Logical value indicating what to do if x fails the checks. If fatal=TRUE (the default), an error occurs.

Details

check.named.thing checks whether x has all the required components, in the sense that names(x) includes all the names in nam, and that every entry in names(x) belongs to either nam or namopt. If all these checks are true, the result is a zero-length character vector. Otherwise, if fatal=TRUE (the default), an error occurs; otherwise the result is a character vector describing the checks which failed.

check.named.vector checks whether x is a numeric vector and check.named.list checks whether x is a list. They then call check.named.thing to check whether all the required components are present. If all these checks are true, the result is a reordered version of x in which all the compulsory entries appear first. Otherwise, if onError="fatal" (the default) an error occurs; otherwise the result is NULL.

Value

check.named.vector returns a numeric vector or NULL.

check.named.list returns a list or NULL.

check.named.thing returns a character vector.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

See Also

check.1.integer

Examples

```
z <- list(a=1, b=2, e=42)
check.named.list(z, c("a", "b"), namopt=c("c", "d", "e"))
check.named.thing(z, c("a", "b"), namopt=c("c", "d", "e"))
zz <- unlist(z)
check.named.vector(zz, c("a", "b"), namopt=c("c", "d", "e"))
check.named.thing(z, c("b", "c"), namopt=c("d", "e"), fatal=FALSE)
```

check.nmatrix *Check for Numeric Matrix with Correct Dimensions*

Description

This is a programmer's utility function to check whether the argument is a numeric vector of the correct length.

Usage

```
check.nmatrix(m, npoints = NULL, fatal = TRUE, things = "data points",
              naok = FALSE, squarematrix = TRUE, matchto = "nrow",
              warn = FALSE, mname)
```

Arguments

m	The argument to be checked.
npoints	The required number of rows and/or columns for the matrix m.
fatal	Logical value indicating whether to stop with an error message if m does not satisfy all requirements.
things	Character string describing what the rows/columns of m should correspond to.
naok	Logical value indicating whether NA values are permitted.
squarematrix	Logical value indicating whether m must be a square matrix.
matchto	Character string (either "nrow" or "ncol") indicating whether it is the rows or the columns of m which must correspond to npoints.
warn	Logical value indicating whether to issue a warning if v does not satisfy all requirements.
mname	Optional character string giving the name of m for use in error messages and warnings.

Details

This programmer's utility function checks whether m is a numeric matrix of the correct dimensions, and checks for NA values. If matchto="nrow" (the default) then the number of rows of m must be equal to npoints. If matchto="ncol" then the number of columns of m must be equal to npoints. If squarematrix=TRUE (the default) then the numbers of rows and columns must be equal. If naok = FALSE (the default) then the entries of m must not include NA.

If these requirements are all satisfied, the result is the logical value TRUE.

If not, then if fatal=TRUE (the default), an error occurs; if fatal=FALSE, the result is the logical value FALSE with an attribute describing the requirement that was not satisfied.

Value

A logical value indicating whether all the requirements were satisfied.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

See Also

check.nvector

Examples

```
z <- matrix(1:16, 4, 4)
check.nmatrix(z, 4)
```

---

check.nvector *Check For Numeric Vector With Correct Length*

Description

This is a programmer's utility function to check whether the argument is a numeric vector of the correct length.

Usage

```
check.nvector(v, npoints = NULL, fatal = TRUE, things = "data points",
              naok = FALSE, warn = FALSE, vname, oneok = FALSE)
```

Arguments

v	The argument to be checked.
npoints	The required length of v.
fatal	Logical value indicating whether to stop with an error message if v does not satisfy all requirements.
things	Character string describing what the entries of v should correspond to.
naok	Logical value indicating whether NA values are permitted.
warn	Logical value indicating whether to issue a warning if v does not satisfy all requirements.
vname	Character string giving the name of v to be used in messages.
oneok	Logical value indicating whether v is permitted to have length 1.

Details

This function checks whether v is a numeric vector with length equal to npoints (or length equal to 1 if oneok=TRUE), not containing any NA values (unless naok=TRUE).

If these requirements are all satisfied, the result is the logical value TRUE.

If not, then if fatal=TRUE (the default), an error occurs; if fatal=FALSE, the result is the logical value FALSE with an attribute describing the requirement that was not satisfied.

Value

A logical value indicating whether all the requirements were satisfied. If FALSE, then this value has an attribute "whinge", a character string describing the requirements that were not satisfied.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

See Also

check.anyvector, check.nmatrix, check.1.real, check.named.vector.

Examples

```
z <- 1:10
check.nvector(z, 5, fatal=FALSE)
y <- 42
check.nvector(y, 5, fatal=FALSE, oneok=TRUE)
```

---

check.range *Utilities for Ranges of Values*

Description

These simple functions handle an interval or range of numerical values. check.range(r) checks whether r specifies a range of values, that is, whether r is a vector of length 2 with r[1] <= r[2]. intersect.ranges(r, s) finds the intersection of two ranges r and s. inside.range(x, r) returns a logical vector containing TRUE if the corresponding entry of x falls inside the range r, and FALSE if it does not. check.in.range(x, r) checks whether a single number x falls inside the specified range r. Finally prange(r) produces a character string that represents the range r.

Usage

```
check.range(r, fatal = TRUE)

check.in.range(x, r, fatal = TRUE)

inside.range(x, r)

intersect.ranges(r, s, fatal = TRUE)

prange(r)
```

Arguments

r	A numeric vector of length 2 specifying the endpoints of a range of values.
x	Numeric vector of data.
s	A numeric vector of length 2 specifying the endpoints of a range of values.
fatal	Logical value indicating whether to stop with an error message if the data do not pass the check.

Details

`check.range` checks whether `r` specifies a range of values, that is, whether `r` is a vector of length 2 with `r[1] <= r[2]`. If so, the result is `TRUE`. If not, then if `fatal=TRUE`, an error occurs, while if `fatal=FALSE` the result is `FALSE`.

`intersect.ranges(r, s)` finds the intersection of two ranges `r` and `s`. If the intersection is non-empty, the result is a numeric vector of length 2. If the intersection is empty, then if `fatal=TRUE`, an error occurs, while if `fatal=FALSE` the result is `NULL`.

`inside.range(x, r)` returns a logical vector containing `TRUE` if the corresponding entry of `x` falls inside the range `r`, and `FALSE` if it does not.

`check.in.range(x, r)` checks whether a single number `x` falls inside the specified range `r`. If so, the result is `TRUE`. If not, then if `fatal=TRUE`, an error occurs, while if `fatal=FALSE` the result is `FALSE`.

Finally `prange(r)` produces a character string that represents the range `r`.

Value

The result of `check.range`, `check.in.range` and `inside.range`, is a logical value or logical vector. The result of `intersect.ranges` is a numerical vector of length 2, or `NULL`. The result of `prange` is a character string.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>

Examples

```
rr <- c(0, 2)
ss <- c(1, 3)
x <- seq(0.5, 3.5, by=1)
check.range(rr)
check.range(42, fatal=FALSE)
inside.range(x, rr)
intersect.ranges(rr, ss)
prange(rr)
```

commasep *List of Items Separated By Commas*

Description

Convert the elements of a vector into character strings and paste them together, separated by commas.

Usage

```
commasep(x, join = " and ", flatten = TRUE)
```

Arguments

x	Vector of items in the list.
join	The string to be used to separate the last two items in the list.
flatten	Logical value indicating whether to return a single character string (flatten=TRUE, the default) or a list (flatten=FALSE).

Value

A character string (if flatten=TRUE, the default) or a list of character strings.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

Examples

```
commasep(letters[1:4])
y <- commasep(sQuote(letters[1:4]))
cat(y, fill=TRUE)
```

do.call.matched *Call a Function, Passing Only Recognised Arguments*

Description

Call a specified function, using only those arguments which are known to be acceptable to the function.

Usage

```
do.call.matched(fun, arglist, funargs, extrargs = NULL,
                matchfirst = FALSE, sieve = FALSE, skipargs = NULL,
                envir=parent.frame())
```

Arguments

fun	A function, or a character string giving the name of a function, to be called.
arglist	A named list of arguments.
funargs	Character vector giving the names of arguments that are recognised by fun. Defaults to the names of the formal arguments of fun.
extrargs	Optional. Character vector giving the names of additional arguments that can be handled by fun.
skipargs	Optional. Character vector giving the names of arguments which should **not** be passed to fun.
matchfirst	Logical value indicating whether the first entry of arglist is permitted to have an empty name and should be matched to the first argument of fun.
sieve	Logical value indicating whether to return the un-used arguments as well as the result of the function call. See Details.
envir	An environment within which to evaluate the call, if any entries of arglist are quoted expressions.

Details

This function is a wrapper for do.call which avoids passing arguments that are unrecognised by fun.

In the simplest case do.call.matched(fun, arglist) is like do.call(fun, arglist), except that entries of arglist which do not match any formal argument of fun are removed. Extra argument names can be permitted using extrargs, and argument names can be forbidden using skipargs.

Value

If sieve=FALSE (the default), the result is the return value from fun.

If sieve=TRUE, the result is a list with entries result (the return value from fun) and otherargs (a list of the arguments that were not passed to fun).

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>

See Also

resolve.defaults, do.call.without.

do.call

Examples

```
f <- function(x=0,y=0, ...) { paste(x, y, ..., sep=", ") }
f()
do.call.matched(f, list(y=2))
do.call.matched(f, list(y=2, z=5), extrargs="z")
do.call.matched(f, list(y=2, z=5), extrargs="z", skipargs="y")
```

`do.call.without` *Call a Function, Omitting Certain Arguments*

Description

Call a specified function, omitting some arguments which are inappropriate to the function.

Usage

```
do.call.without(fun, ..., avoid, envir=parent.frame())
```

Arguments

`fun`	The function to be called. A function name, a character string giving the name of the function, or an expression that yields a function.
`...`	Any number of arguments.
`avoid`	Vector of character strings, giving the names of arguments that should *not* be passed to `fun`.
`envir`	An environment within which to evaluate the call, if any entries of `arglist` are quoted expressions.

Details

This is a simple mechanism for preventing some arguments from being passed in a function call. The arguments `...` are collected in a list. A argument is omitted if its name exactly matches one of the strings in `avoid`.

Value

The return value of `fun`.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

See Also

`do.call.matched` for a more complicated and flexible call.

Examples

```
do.call.without(paste, 1, 2, z=3, w=4, avoid="z")
```

evenly.spaced *Determine Whether a Vector is Evenly Spaced and Increasing*

Description

Determines whether the entries in a numeric vector are evenly spaced and increasing.

Usage

```
evenly.spaced(x, tol = 1e-07)
```

Arguments

x	Numeric vector.
tol	Relative tolerance.

Details

The result is TRUE if x is an increasing sequence in which the successive differences diff(x) are all equal to one another (within the specified relative tolerance), and FALSE otherwise.

Value

A single logical value.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

Examples

```
evenly.spaced(seq(0, 1, length=4))
```

exactCutBreaks *Determine Breakpoints for Cut*

Description

Computes the numerical breakpoints used by cut.default.

Usage

```
exactCutBreaks(x, breaks)
```

Arguments

x	Numeric vector which would be converted to a factor.
breaks	Either a numeric vector of breakpoints, or a single integer giving the number of intervals into which x will be cut.

Details

This function contains a copy of the code in cut.default which determines the numerical breakpoints used to convert x to a factor. It returns the breakpoints only.

The arguments x and breaks have the same interpretation as in cut.default. Only the range of x is used in the computation, so x could be replaced by range(x).

This function would normally be used when breaks is a single integer specifying the number of intervals for the cut operation. It returns the exact numerical values of the breakpoints which are determined, but not returned, by cut.default).

Value

Numeric vector.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

See Also

cut.default

Examples

```
exactCutBreaks(c(0,1), 4)
```

---

expand.polynom *Expand Symbolic Polynomials in a Formula*

Description

Create a formula representing a polynomial, or expand polynomials in an existing formula.

Usage

```
expand.polynom(f)
sympoly(x, y, n)
```

Arguments

f	A formula.
x, y	Variable names.
n	Integer specifying the degree of the polynomial. (If n is missing, y will be interpreted as the degree.)

Details

These functions expand a polynomial into its homogeneous terms and return a model formula.

sympoly(x, n) creates a formula whose right-hand side represents the polynomial of degree n in the variable x. Each homogeneous term x^k is a separate term in the formula.

sympoly(x, y, n) creates a formula representing the polynomial of degree n in the two variables x and y.

If f is a formula containing a term of the form polynom(...) then expand.polynom(f) replaces this term by its expansion as a sum of homogeneous terms, as defined in the help for polynom.

Value

A formula.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <rolfturner@posteo.net> and Ege Rubak <rubak@math.aau.dk>.

See Also

polynom

Examples

```
sympoly(A, 4)
sympoly(A, B, 3)
expand.polynom(U ~ A + polynom(B, 2))
```

---

fastFindInterval *Find Intervals Containing Given Data*

Description

A faster alternative to findInterval for intervals which are equally-spaced.

Usage

```
fastFindInterval(x, b, labels = FALSE, reltol = 0.001, dig.lab = 3L,
                 left.open=TRUE)
```

Arguments

x	Data. Numeric vector of values that are to be classified.
b	Breakpoints. Numeric vector of increasing values that are the endpoints of the intervals.
labels	Logical value specifying whether to return a factor, whose levels are the string labels of the intervals.
reltol	Relative tolerance. A positive number.
dig.lab	Integer. Maximum number of digits to use in the labels for the intervals, when labels=TRUE.
left.open	Logical value specifying whether intervals are left-open and right-closed (left.open=TRUE, the default) or left-closed and right-open (left.open=FALSE).

Details

This is an alternative to findInterval(x, b, rightmost.closed=TRUE) which seems to be faster when b is equally spaced and the length of x is large.

If labels=FALSE (the default), the result is an integer vector giving, for each value x[i], the index j of the interval that contains x[i]:

- If left.open=TRUE (the default), the intervals are left-open and right-closed, except for the first interval. This means that x[i] belongs to the jth interval if b[j] < x[i] <= b[j+1] for j > 1 and b[1] <= x[i] <= b[2] for j=1.
- If left.open=FALSE, the intervals are left-closed and right-open, except for the last interval. This means that x[i] belongs to the jth interval if b[j] <= x[i] < b[j+1] for j < m and b[m] <= x[i] <= b[m+1] for j=m where m = length(b)-1 is the number of intervals.

If labels=TRUE, the result is a factor, and the levels are synthetic labels for the intervals, similar to those produced by findInterval.

Note that the default value of left.open is TRUE for fastFindInterval but FALSE for findInterval.

Value

Integer vector, or factor.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <rolfturner@posteo.net> and Ege Rubak <rubak@math.aau.dk>.

See Also

findInterval

Examples

```
x <- runif(10)
b <- seq(0, 1, by=0.2)
fastFindInterval(x, b, labels=TRUE)
```

geomseq *Geometric Sequence*

Description

Generate a geometric sequence between two endpoints. The sequence is equally spaced on a logarithmic scale.

Usage

```
geomseq(from, to, length.out)
```

Arguments

from	Starting value. A positive number.
to	Ending value. A positive number.
length.out	Number of elements in the sequence. A positive integer.

Details

This is a wrapper for seq.default which generates a geometric sequence between the two endpoints.

Value

Numeric vector.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <rolfturner@posteo.net> and Ege Rubak <rubak@math.aau.dk>.

See Also

seq.default

Examples

```
geomseq(1, 32, length.out=6)
```

harmonicmean *Harmonic Mean*

Description

Calculates the harmonic mean of a numeric vector, robustly handling special cases.

Usage

```
harmonicmean(x, na.rm = TRUE)
harmonicsum(x, na.rm = TRUE)
```

Arguments

x	Numeric vector.
na.rm	Logical value specifying whether to remove NA values before processing.

Details

The harmonic mean of a set of numbers is the reciprocal of the mean of the reciprocals of the numbers.

The function harmonicmean calculates the harmonic mean of x. The algorithm robustly handles special cases where some of the values in x are very small or are exactly equal to zero.

The function harmonicsum calculates the reciprocal of the *sum* of the reciprocals of the x values.

Value

A single numeric value.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

Examples

```
harmonicmean(1:3)
```

`ifelseAB` *Conditional Selection*

Description

These low-level functions provide faster alternatives to some uses of `ifelse`.

Usage

```
ifelseAB(test, a, b)
ifelseAX(test, a, x)
ifelseXB(test, x, b)
ifelseXY(test, x, y)
ifelseNegPos(test, x)
ifelse0NA(test)
ifelse1NA(test)
```

Arguments

`test`	A logical vector.
`a`	A single atomic value.
`b`	A single atomic value.
`x`	A vector of values, of the same length as `test`.
`y`	A vector of values, of the same length as `test`.

Details

These low-level functions provide faster alternatives to some uses of `ifelse`. They were developed by trial-and-error comparison of computation times of different expressions.

`ifelse0NA(test)` is equivalent to `ifelse(test, 0, NA)`.

`ifelse1NA(test)` is equivalent to `ifelse(test, 1, NA)`.

`ifelseAB(test, a, b)` is equivalent to `ifelse(test, a, b)` where `a` and `b` must be single values.

`ifelseAX(test, a, x)` is equivalent to `ifelse(test, a, x)` where `a` must be a single value, and `x` a vector of the same length as `test`.

`ifelseXB(test, x, b)` is equivalent to `ifelse(test, x, b)` where `b` must be a single value, and `x` a vector of the same length as `test`.

`ifelseXY(test, x, y)` is equivalent to `ifelse(test, x, y)` where `x` and `y` must be vectors of the same length as `test`.

`ifelseNegPos(test, x)` is equivalent to `ifelse(test, x, -x)` where `x` must be a vector of the same length as `test`.

Value

A vector of the same length as `test` containing values of the same type as `a,b,x,y`.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <rolfturner@posteo.net> and Ege Rubak <rubak@math.aau.dk>.

See Also

ifelse

Examples

```
x <- runif(4e5)
u <- (x < 0.5)
system.time(ifelse(u, 2, x))
system.time(ifelseAX(u, 2, x))
```

---

is.power *Recognise a Square, Cube, or Power of an Integer*

Description

Determine whether the given integer is a square number, a cube number, or a power of an integer.

Usage

```
is.square(n)
is.cube(n)
is.power(n)
```

Arguments

n	A single integer.

Details

is.square(n) returns TRUE if n is a square number, that is, n = m^2 for some integer m, and returns FALSE otherwise.

is.cube(n) returns TRUE if n is the cube of an integer, n = m^3 for some integer m, and returns FALSE otherwise.

is.power(n) returns TRUE if n is an integer power of an integer, n = m^k for some integers m and k, and returns FALSE otherwise.

These functions use the prime factorisation of n and may be more reliable than testing where sqrt(n) is an integer, etc.

Negative values of n are permitted.

Value

A single logical value.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

See Also

primefactors

Examples

```
is.square(9)
is.cube(9)
is.cube(27)
is.cube(-27)
is.power(27)
is.power(3^5)
```

---

methods.xypolygon *Calculations for Polygons in the Plane*

Description

Compute the area or boundary length of a polygon, determine whether a point falls inside a polygon, compute the area of overlap between two polygons, and related tasks.

Usage

```
verify.xypolygon(p, fatal = TRUE)
is.hole.xypolygon(polly)
Area.xypolygon(polly)
bdrylength.xypolygon(polly)
reverse.xypolygon(p, adjust=FALSE)
overlap.xypolygon(P, Q)
simplify.xypolygon(p, dmin)
inside.xypolygon(pts, polly, test01, method)
```

Arguments

p, polly, P, Q	Data representing a polygon. See Details.
dmin	Single numeric value giving the minimum permissible length of an edge in the simplified polygon.
fatal	Logical value indicating whether failure is a fatal error.
pts	Coordinates of points to be tested. A named list with entries x,y which are numeric vectors of coordinates.
adjust	Logical value indicating whether internal data should be adjusted. See Details.
test01, method	For developer use only.

Details

In the **spatstat** family of packages, a polygon in the Euclidean plane is represented as a named list with the following entries:

`x,y` Numeric vectors giving the coordinates of the vertices. The vertices should be traversed in anti-clockwise order (unless the polygon is a hole, when they should be traversed in clockwise order) and the last vertex should **not** repeat the first vertex.

hole Optional. A logical value indicating whether the polygon is a hole.

area Optional. Single numeric value giving the area of the polygon (negative if it is a hole).

The function `verify.xypolygon` checks whether its argument satisfies this format. If so, it returns `TRUE`; if not, it returns `FALSE` or (if `fatal=TRUE`) generates a fatal error.

The other functions listed here perform basic calculations for polygons using elementary Cartesian analytic geometry in R.

`is.hole.xypolygon` determines whether a polygon is a hole or not.

`Area.xypolygon` computes the area of the polygon using the discrete Green's formula.

`bdrylength.xypolygon` calculates the total length of edges of the polygon.

`reverse.xypolygon` reverses the order of the coordinate vectors `x` and `y`. If `adjust=TRUE`, the other entries `hole` and `area` will be adjusted as well.

`overlap.xypolygon` computes the area of overlap between two polygons using the discrete Green's formula. It is slow compared to the code in the **polyclip** package.

`simplify.xypolygon` removes vertices of the polygon until every edge is longer than `dmin`.

`inside.xypolygon(pts, polly)` determines whether each point in `pts` lies inside the polygon `polly` and returns a logical vector.

Value

`verify.xypolygon` and `is.hole.xypolygon` return a single logical value.

`inside.xypolygon` returns a logical vector.

`Area.xypolygon`, `bdrylength.xypolygon` and `overlap.xypolygon` return a single numeric value.

`reverse.xypolygon` and `simplify.xypolygon` return another polygon object.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

Examples

```
p <- list(x=c(0,1,4,2), y=c(0,0,2,3))
is.hole.xypolygon(p)
Area.xypolygon(p)
bdrylength.xypolygon(p)
overlap.xypolygon(p, list(x=p$x+1, y=p$y+1))
reverse.xypolygon(p)

plot(c(0,5),c(0,3),type="n",xlab="x", ylab="y")
```

```
polygon(p)
polygon(simplify.xypolygon(p, 1.1), lty=3)

plot(c(0,5),c(0,3),type="n",xlab="x", ylab="y")
polygon(p)
xx <- runif(10, max=5)
yy <- runif(10, max=3)
points(xx, yy)
ok <- as.logical(inside.xypolygon(list(x=xx, y=yy), p))
points(xx[ok], yy[ok], pch=16)
```

`optimizeWithTrace` *One Dimensional Optimization with Tracing*

Description

Find the minimum or maximum of a function over an interval of real numbers, keeping track of the function arguments and function values that were evaluated.

Usage

```
optimizeWithTrace(f, interval, ...,
                  lower = min(interval), upper = max(interval))
```

Arguments

`f`	The function to be minimized or maximized.
`interval`	Numeric vector of length 2 containing the end-points of the interval to be searched.
`lower`, `upper`	The lower and upper endpoints of the interval to be searched.
`...`	Other arguments passed to `optimize`, including arguments to the function `f`.

Details

This is a simple wrapper for the optimization routine `optimize`. The function `f` will be optimized by computing its value at several locations in the interval, as described in the help for `optimize`. This wrapper function stores the locations and resulting function values, and returns them along with the result of the optimization.

Value

A list with components

- `minimum` (or `maximum`), the location in the search interval which yielded the optimum value;
- `objective`, the value of the function at this location;
- `x`, the sequence of locations in the interval that were considered (in the order considered);
- `y`, the function values corresponding to `x`.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <rolfturner@posteo.net> and Ege Rubak <rubak@math.aau.dk>.

See Also

optimize

Examples

```
f <- function (x, a) (x - a)^2
result <- optimizeWithTrace(f, c(0, 1), tol = 0.0001, a = 1/3)
result
curve(f(x, 1/3))
with(result, points(x, y, pch=16))
```

---

orderstats *Compute Order Statistics*

Description

Compute the k-th smallest value in a dataset, or find which entry in a dataset is the k-th smallest.

Usage

```
orderstats(x, k, decreasing = FALSE)
orderwhich(x, k, decreasing = FALSE)
```

Arguments

x	Data whose order statistics will be computed. A numeric vector.
k	Rank. An integer, or vector of integers.
decreasing	Logical value specifing whether a rank of 1 is assigned to the highest value (decreasing=TRUE) or the lowest value (decreasing=FALSE, the default).

Details

These are low-level functions for efficiently computing order statistics: orderstats(x, k) returns the k-th smallest value in x, and orderwhich(x, k) returns the *position* of the k-th smallest value in x.

Given a dataset of values $x_1, \ldots, x_n$, the *order statistic* of rank k is the k-th smallest value in the dataset. The order statistic of rank 1 is the smallest value, and the order statistic of rank n is the largest value. The order statistic of rank k is denoted $x_{[k]}$.

The full sequence of order statistics

$$x_{[1]} \leq x_{[2]} \leq \cdots \leq x_{[n]}$$

can simply be obtained by sorting the original values into increasing order.

The command `orderstats(x, k)` is equivalent to `sort(x)[k]`; it calculates the k-th smallest value in `x`.

The command `orderwhich(x, k)` is equivalent to `order(x)[k]`. It identifies the *position* of the k-th smallest value in `x`, that is, it returns the index `j` such that `x[j]` is the k-th smallest value in `x`.

The functions `orderstats` and `orderwhich` are more efficient than using `sort` and `order` when it is only desired to calculate a few of the order statistics (for example, only the smallest and second-smallest values in the dataset).

Value

`orderstats` returns a vector of the same kind as `x`, with the same length as `k`. `orderwhich` returns an integer vector with the same length as `k`.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

See Also

`sort`, `order`.

Examples

```
x <- runif(10)
orderstats(x, 2)
sort(x)[2]
orderwhich(x, 2:3)
order(x)[2:3]
```

---

ordinal *Ordinal Numbers*

Description

Returns the appropriate abbreviation in English for an ordinal number (for example `ordinal(5)` is `"5th"`).

Usage

```
ordinal(k)
ordinalsuffix(k)
```

Arguments

k	An integer or vector of integers.

Details

`ordinal(k)` returns a character string representing the kth ordinal number. `ordinalsuffix(k)` determines the appropriate suffix.

The suffix can be either `"st"` (abbreviating *first*), `"nd"` (abbreviating *second*), `"rd"` (abbreviating *third*) or `"th"` (for all other ordinal numbers `fourth`, `fifth`, etc).

Value

A character string or character vector of the same length as `k`.

Author(s)

Adrian Baddeley `<Adrian.Baddeley@curtin.edu.au>`.

See Also

`articlebeforenumber`

Examples

```
ordinal(1:7)
cat(paste("Happy", ordinal(21), "Birthday"), fill=TRUE)
```

---

orifnull *Specify a Default Value*

Description

Specify a value together with a default to be used when the first value is null.

Usage

```
a %orifnull% b
```

Arguments

`a`	Any kind of object or expression to be evaluated.
`b`	Default value to be used when `a` is `NULL`. Any kind of object or expression to be evaluated.

Details

The operator `%orifnull%` is designed to improve the readability of code.

`a %orifnull% b` is equivalent to `if(is.null(a)) a else b`.

That is, `a %orifnull% b` is equal to `a` provided `a` is not null, and otherwise the result is equal to `b`.

Expressions are evaluated only when necessary. If `a` is a language expression, it is first evaluated. Then if `is.null(a)` is `FALSE`, the result is `a`. Otherwise, `b` is evaluated, and the result is `b`. Note that `b` is not evaluated unless `a` is `NULL`.

The operator `%orifnull%` has higher precedence than the arithmetic operators `+`, `-`, `*`, `/` but lower precedence than `^`.

The operator is associative, and can be used repeatedly in an expression, so that a default value may have its own default. See the Examples.

Value

The result is `a` if `a` is not `NULL`, and otherwise the result is `b`.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <rolfturner@posteo.net> and Ege Rubak <rubak@math.aau.dk>

Examples

```
x <- 7
y <- 42
z <- w <- NULL
x %orifnull% y
z %orifnull% y
z %orifnull% x %orifnull% y
z %orifnull% w %orifnull% y
```

---

paren *Add or Remove Parentheses*

Description

Add or remove enclosing parentheses around a string.

Usage

```
paren(x, type = "(")
unparen(x)
```

Arguments

`x`	A character string, or vector of character strings.
`type`	Type of parentheses: either `"("`, `"["` or `"{"`.

Details

`paren(x)` adds enclosing parentheses to the beginning and end of the string `x`.

`unparen(x)` removes enclosing parentheses if they are present.

Value

A character string, or vector of character strings of the same length as `x`.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

See Also

`commasep`

Examples

```
paren("Hello world")
paren(42, "[")
paren(letters[1:10])
unparen(c("(yes)", "[no]", "{42}"))
```

---

percentage *Convert Fraction to Percentage*

Description

This is a programmer's utility which converts a fraction to a percentage and encodes the percentage as a character string.

Usage

```
percentage(x, digits = 3)
```

Arguments

`x`	Either a single number, or a logical vector.
`digits`	Number of digits accuracy.

Details

If `x` is a single number, it should be a fraction between 0 and 1. It will be converted to a percentage and then converted to a character string followed by the percentage symbol.

If `x` is a logical vector, the fraction of values which are `TRUE` will be computed, and used to determine the percentage.

Value

A character string.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>

Examples

```
percentage(1/3)
percentage(runif(20) > 0.2)
```

`primefactors` *Primes, Prime Factorization, Common Divisor*

Description

These functions find prime numbers, factorise a composite number into its prime factors, determine whether a number is prime, and find the least common multiple or greatest common divisor of two numbers.

Usage

```
primefactors(n, method=c("C", "interpreted"))
divisors(n)
is.prime(n)
relatively.prime(n, m)
least.common.multiple(n,m)
greatest.common.divisor(n,m)
primesbelow(nmax)
```

Arguments

`n, m`	Integers to be factorized.
`nmax`	Integer. Upper limit on prime numbers to be found.
`method`	Character string indicating the choice of algorithm. (Developer use only.)

Details

`is.prime(n)` returns `TRUE` if `n` is a prime number, and `FALSE` otherwise.

`primefactors(n)` factorises the integer `n` into its prime number factors, and returns an integer vector containing these factors. Some factors may be repeated.

`divisors(n)` finds all the integers which divide the integer `n`, and returns them as a sorted vector of integers (beginning with `1` and ending with `n`).

`relatively.prime(n, m)` returns `TRUE` if the integers `n` and `m` are relatively prime, that is, if they have no common factors.

`least.common.multiple` and `greatest.common.divisor` return the least common multiple or greatest common divisor of two integers `n` and `m`.

`primesbelow(nmax)` returns an integer vector containing all the prime numbers less than or equal to `nmax`.

Value

`is.prime` and `relatively.prime` return a logical value.

`least.common.multiple` and `greatest.common.divisor` return a single integer.

`primefactors` and `primesbelow` return an integer vector.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

Examples

```
is.prime(17)

is.prime(399137)

relatively.prime(2, 3)

primefactors(24) ## Note repeated factors

primefactors(713291035)

divisors(24)

greatest.common.divisor(60, 100)

least.common.multiple(10, 15)

primesbelow(20)
```

---

queueSpatstatLocator *Add Coordinates to a Queue for Use by Locator Function*

Description

Add the coordinates of a spatial location to a queue. The queue can be accessed by the `spatstatLocator` function in a non-interactive session.

Usage

```
queueSpatstatLocator(x, y)
```

Arguments

x, y	Numeric values, or vectors of the same length, containing spatial coordinates. Any data acceptable to xy.coords.

Details

The spatstatLocator function is a replacement for the locator function that can be used to test software which depends on user input.

When queueSpatstatLocator(x,y) is called, the coordinate data x,y are saved in a queue. The first-listed coordinate pair x[1], y[1] is at the front of the queue. Subsequently, when spatstatLocator is called, the coordinates are taken from the front of the queue and returned as if they had been clicked by the user.

This only works in a **non-**interactive session, that is, when interactive() returns FALSE.

Value

Integer (invisible). The length of the queue, after inclusion of the new points.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <rolfturner@posteo.net> and Ege Rubak <rubak@math.aau.dk>.

See Also

spatstatLocator

Examples

```
queueSpatstatLocator(0.5, 0.7)
queueSpatstatLocator(c(0.3, 0.4), c(0.2, 0.9))
if(!interactive()) {
   spatstatLocator(2)
   spatstatLocator(1)
}
```

RelevantNA *Missing Value, Zero-length Vector, or Zero Value of the Appropriate Type*

Description

Given any data x, these functions return the missing value NA, the empty vector, or the equivalent of the number 0, with the same type as x.

Usage

```
RelevantZero(x)
RelevantNA(x)
RelevantEmpty(x)

isRelevantZero(x)
```

Arguments

x	Data of any type.

Details

In the R system, missing values may have different types. For example, if an entry is missing from a numeric vector, it is a missing numeric value, not a missing logical value, and R distinguishes between these two types of missing values.

The function `RelevantNA` returns a missing value of the same type as the input `x` (as defined by `typeof`). Thus, `RelevantNA(3.2)` returns a missing numeric value and `RelevantNA(TRUE)` returns a missing logical value.

`RelevantEmpty(x)` returns a vector of length zero which has the same type as `x`. Thus, `RelevantEmpty(TRUE)` is equivalent to `logical(0)`.

`RelevantZero(x)` returns a single value, of the same type as `x`, that is equivalent to the number zero. For example, `RelevantZero(TRUE)` returns `FALSE`.

The function `isRelevantZero` tests whether `x` is a single zero value, by testing whether `x` is identical to `RelevantZero(x)`.

Value

`RelevantZero` and `RelevantNA` return a single value of the same type as `x`.

`RelevantEmpty` returns a zero-length vector of the same type as `x`.

`isRelevantZero` returns a single logical value.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

See Also

`typeof`

Examples

```
RelevantZero(42)
RelevantZero(TRUE)
RelevantZero("hello world")

RelevantNA(1:3)
typeof(RelevantNA(1:3))
```

```
typeof(RelevantNA("hello world"))
```

`resolve.defaults` *Determine Values of Variables Using Several Default Rules*

Description

Determine the values of variables by applying several different default rules in a given order.

Usage

```
resolve.defaults(..., .MatchNull = TRUE, .StripNull = FALSE)

resolve.1.default(.A, ...)
```

Arguments

`...`	Several lists of `name=value` pairs.
`.MatchNull`	Logical value. If `TRUE` (the default), an entry of the form `name=NULL` will be treated as assigning the value `NULL` to the variable `name`. If `FALSE`, such entries will be ignored.
`.StripNull`	Logical value indicating whether entries of the form `name=NULL` should be removed from the result.
`.A`	Either a character string giving the name of the variable to be extracted, or a list consisting of one `name=value` pair giving the variable name and its fallback default value.

Details

These functions determine the values of variables by applying a series of default rules, in the order specified.

Each of the arguments `...` should be a list of `name=value` pairs giving a value for a variable `name`. Each list could represent a set of arguments given by the user, or a rule assigning default values to some variables. Lists that appear earlier in the sequence of arguments `...` take precedence.

The arguments `...` will be concatenated into a single list. The earliest occurrence of each `name` is then used to determine the final value of the variable `name`.

The function `resolve.defaults` returns a list of `name=value` pairs for all variables encountered. It is commonly used to decide the values of arguments to be passed to another function using `do.call`.

The function `resolve.1.default` returns the value of the specified variable as determined by `resolve.defaults`. It is commonly used inside a function to determine the value of an argument.

Value

The result of `resolve.defaults` is a list of `name=value` pairs.

The result of `resolve.1.default` can be any kind of value.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>

See Also

do.call

Examples

```
user <- list(day="Friday")
ruleA <- list(month="Jan", gravity=NULL)
ruleB <- list(day="Tuesday", month="May", gravity=42)
resolve.defaults(user, ruleA, ruleB)
resolve.defaults(user, ruleA, ruleB, .StripNull=TRUE)
resolve.defaults(user, ruleA, ruleB, .MatchNull=FALSE)

resolve.1.default("month", user, ruleA, ruleB)
```

---

revcumsum *Reverse Cumulative Sum*

Description

Returns a vector of cumulative sums of the input values, running in reverse order. That is, the ith entry in the output is the sum of entries i to n in the input, where n is the length of the input.

Usage

```
revcumsum(x)
```

Arguments

x	A numeric, logical or complex vector.

Details

This low-level utility function is a faster alternative to rev(cumsum(rev(x))) under certain conditions. It computes the reverse cumulative sum of the entries of x. If y <- revcumsum(x), then y[i] = sum(x[i:n]) where n = length(x).

This function should not be used if x could contain NA values: this would lead to an error.

Value

A vector of the same length and type as x (except that if x is logical then the result is an integer vector).

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

See Also

cumsum.

Examples

```
revcumsum(1:5)
rev(cumsum(rev(1:5)))
x <- runif(1e6)
system.time(rev(cumsum(rev(x))))
system.time(revcumsum(x))
```

---

simplenumber *Simple Rational Number*

Description

Given a numeric value, try to express it as a simple rational number.

Usage

```
simplenumber(x, unit = "", multiply = "*", tol = .Machine$double.eps)
```

Arguments

x	A single numeric value.
unit	Optional. Character string giving the name of the unit in which x is expressed. Typically an irrational number such as pi. See Examples.
multiply	Optional. Character string representing multiplication.
tol	Numerical tolerance.

Details

The code tries to express x as an integer x=n, or as the reciprocal of an integer x=1/n, or as a simple rational number x = m/n, where m,n are small integers.

Value

A character string representing the simple number, or NULL if not successful.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

Examples

```
simplenumber(0.3)
simplenumber(0.33333333333333333333333)
x <- pi * 2/3
simplenumber(x/pi, "pi")
```

spatstat.utils-deprecated *Deprecated Functions of spatstat.utils Package*

Description

Deprecated functions of the spatstat.utils package.

Usage

```
equispaced(z, reltol=0.001)
```

Arguments

z	Numeric vector.
reltol	Numeric value for relative tolerance.

Details

equispaced has been replaced by evenly.spaced.

spatstatLocator *Graphical Input*

Description

This is an alternative to the locator function. It contains a workaround for a bug that occurs in RStudio.

Usage

```
spatstatLocator(n, type = c("p", "l", "o", "n"), ...,
                snap.step=NULL, snap.origin=c(0,0))
```

Arguments

n	Optional. Maximum number of points to locate.
type	Character specifying how to plot the locations. If "p" or "o" the points are plotted; if "l" or "o" they are joined by lines.
...	Additional graphics parameters used to plot the locations.
snap.step	Optional. Spatial coordinates will be rounded to the nearest multiple of snap.step. A positive number specifying the step length, or a vector of 2 positive numbers specifying step lengths for the x and y coordinates.
snap.origin	Optional. Numeric vector of length 2. Coordinates of the origin that will be used when rounding coordinates.

Details

This is a replacement/workaround for the locator function in some versions of **RStudio** which do not seem to recognise the option type="p".

See locator for a description of the behaviour.

If snap.step is given, then the coordinates of the selected locations will be rounded to the nearest multiple of snap.step.

Value

A list containing components x and y which are vectors giving the coordinates of the identified points in the user coordinate system, i.e., the one specified by par("usr").

Software Testing

Programmers may like to know that code which depends on spatstatLocator can be tested in a non-interactive session, if the coordinates are previously queued using queueSpatstatLocator.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <rolfturner@posteo.net> and Ege Rubak <rubak@math.aau.dk>.

See Also

locator.

queueSpatstatLocator

Examples

```
if(interactive()) locator(1, type="p")
```

---

splat *Print Text Within Margins*

Description

Prints a given character string or strings inside the text margin specified by options("width"). Indents the text if required.

Usage

```
splat(..., indent = 0)
```

Arguments

...	Character strings, or other arguments acceptable to paste.
indent	Optional. Indentation of the text. Either an integer specifying the number of character positions by which the text should be indented, or a character string whose length determines the indentation.

Details

splat stands for 'split cat'.

The command splat(...) is like cat(paste(...)) except that the output will be split into lines that can be printed within the current text margin specified by getOption("width").

The arguments ... are first combined into a character vector using paste. Then they are split into words separated by white space. A newline will be inserted whenever the next word does not fit in the available text area. (Words will not be broken, so the text margin could be exceeded if any word is longer than getOption("width")).

If any argument is a vector, each element of the vector is treated as a separate line. Existing newline characters in ... are also respected.

Value

Null.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <rolfturner@posteo.net> and Ege Rubak <rubak@math.aau.dk>.

Examples

```
op <- options(width=20)
splat("There is more than one way to skin a cat.")
splat("There is more than one", "way to skin a cat.", indent=5)

options(width=10)
splat("The value of pi is", pi)
splat("The value of pi is", signif(pi))
options(op)
```

---

taperoff *Taper Functions*

Description

Computes a function that tapers smoothly from 0 to 1.

Usage

```
taperoff(x, zeropoint = 0, onepoint = 1,
         type = c("smooth", "cosine", "Gaussian"))
```

Arguments

x	Function argument. A number or a numeric vector.
zeropoint	Value of x that should return a function value of 0.
onepoint	Value of x that should return a function value of 1.
type	Character string (partially matched) specifying which taper function to use.

Details

A taper is a mathematical function that exhibits a gradual transition between the values 0 and 1.

By default, the function value $f(x)$ is equal to 0 if $x \leq 0$, is equal to 1 if $x \geq 1$, and lies between 0 and 1 when $0 < x < 1$.

If type="cosine", the function is the cosine taper $f(x) = (1 - \cos(\pi x))/2$.

If type="smooth" the function is the smooth partition of unity $f(x) = \theta(x)/(\theta(x) + \theta(1-x))$ where $\theta(x) = \exp(-1/x)$.

If type="Gaussian" the function is the cumulative distribution function of the Gaussian (normal) distribution with mean $1/2$ and standard deviation $1/6$.

If zeropoint and onepoint are specified, then the function value is equal to 0 when x=zeropoint, equal to 1 when x=onepoint, and lies between 0 and 1 when x lies between zeropoint and onepoint.

Value

A numeric vector of the same length as x.

Author(s)

Adrian Baddeley

Examples

```
curve(taperoff(x, type="smooth"))
curve(taperoff(x, type="cosine"), add=TRUE, col="green")
curve(taperoff(x, type="Gaussian"), add=TRUE, col="blue")
```

`tapplysum` *Sum By Factor Level*

Description

A faster equivalent of `tapply(FUN=sum)`.

Usage

```
tapplysum(x, flist, do.names = FALSE, na.rm = TRUE)
```

Arguments

`x`	Vector of numeric or complex values.
`flist`	A list of `factor`s of the same length as `x`.
`do.names`	Logical value indicating whether to attach names to the result.
`na.rm`	Logical value indicating whether to remove `NA` values before computing the sums.

Details

This function is designed to be a faster alternative to the idiom `y <- tapply(x, flist, sum); y[is.na(y)] <- 0`. The result `y` is a vector, matrix or array of dimension equal to the number of factors in `flist`. Each position in `y` represents one of the possible combinations of the factor levels. The resulting value in this position is the sum of all entries of `x` where the factors in `flist` take this particular combination of values. The sum is zero if this combination does not occur.

Currently this is implemented for the cases where `flist` has length 1, 2 or 3 (resulting in a vector, matrix or 3D array, respectively). For other cases we fall back on `tapply`.

Value

A numeric vector, matrix or array.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au> and Tilman Davies.

See Also

`tapply`, `table`

Examples

```
x <- 1:12
a <- factor(rep(LETTERS[1:2], each=6))
b <- factor(rep(letters[1:4], times=3))
ff <- list(a, b)
tapply(x, ff, sum)
tapplysum(x, ff, do.names=TRUE)
tapplysum(x + 2i, ff, do.names=TRUE)
```

`termsinformula` *Manipulate Formulae*

Description

Operations for manipulating formulae.

Usage

```
termsinformula(x)
variablesinformula(x)
offsetsinformula(x)
lhs.of.formula(x)
rhs.of.formula(x, tilde=TRUE)
lhs.of.formula(x) <- value
rhs.of.formula(x) <- value
can.be.formula(x)
identical.formulae(x,y)
```

Arguments

`x, y`	Formulae, or character strings representing formulae.
`tilde`	Logical value indicating whether to retain the tilde.
`value`	Symbol or expression in the R language. See Examples.

Details

`variablesinformula(x)` returns a character vector of the names of all variables which appear in the formula `x`.

`termsinformula(x)` returns a character vector of all terms in the formula `x` (after expansion of interaction terms).

`offsetsinformula(x)` returns a character vector of all offset terms in the formula.

`rhs.of.formula(x)` returns the right-hand side of the formula as another formula (that is, it removes the left-hand side) provided `tilde=TRUE` (the default). If `tilde=FALSE`, then the right-hand side is returned as a language object.

`lhs.of.formula(x)` returns the left-hand side of the formula as a symbol or language object, or `NULL` if the formula has no left-hand side.

`lhs.of.formula(x) <- value` and `rhs.of.formula(x) <- value` change the formula `x` by replacing the left or right hand side of the formula by `value`.

`can.be.formula(x)` returns `TRUE` if `x` is a formula or a character string that can be parsed as a formula, and returns `FALSE` otherwise.

`identical.formulae(x,y)` returns `TRUE` if `x` and `y` are identical formulae (ignoring their environments).

Value

`variablesinformula`, `termsinformula` and `offsetsinformula` return a character vector.

`rhs.of.formula` returns a formula. `lhs.of.formula` returns a symbol or language object, or `NULL`.

`can.be.formula` and `identical.formulae` return a logical value.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <rolfturner@posteo.net> and Ege Rubak <rubak@math.aau.dk>.

Examples

```
f <- (y ~ x + z*w + offset(h))
lhs.of.formula(f)
rhs.of.formula(f)
variablesinformula(f)
termsinformula(f)
offsetsinformula(f)
g <- f
environment(g) <- new.env()
identical(f,g)
identical.formulae(f,g)
lhs.of.formula(f) <- quote(mork) # or as.name("mork")
f
rhs.of.formula(f) <- quote(x+y+z) # or parse(text="x+y+z")[[1]]
f
```

---

verbalogic *Verbal Logic*

Description

Perform the specified logical operation on the character vector `x`, recognising the special strings `"TRUE"` and `"FALSE"` and treating other strings as logical variables.

Usage

```
verbalogic(x, op = "and")
```

Arguments

x	Character vector.
op	Logical operation: one of the character strings "and", "or" or "not".

Details

This function performs simple logical operations on character strings that represent human-readable statements.

The character vector x may contain any strings: the special strings "TRUE" and "FALSE" are treated as the logical values TRUE and FALSE, while all other strings are treated as if they were logical variables.

If op="and", the result is a single string, logically equivalent to x[1] && x[2] && ... && x[n]. First, any entries of x equal to "TRUE" are removed. The result is "FALSE" if any of the entries of x is "FALSE"; otherwise it is equivalent to paste(x, collapse=" and ").

If op="or", the result is a single string, logically equivalent to x[1] || x[2] || ... || x[n]. First, any entries of x equal to "FALSE" are removed. The result is "TRUE" if any of the entries of x is "TRUE"; otherwise it is equivalent to paste(x, collapse=" or ").

If op="not", the result is a character vector y such that y[i] is the logical negation of x[i].

The code does not understand English grammar and cannot expand logical expressions.

Value

A character string.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

Examples

```
x <- c("The sky is blue", "my name is not Einstein", "FALSE")
verbalogic(x, "and")
verbalogic(x, "or")
verbalogic(x, "not")
```

---

which.min.fair *Where is the Minimum or Maximum*

Description

Determines the index of the minimum or maximum of a vector. If there are multiple entries which achieve the minimum or maximum, one of the indices is selected at random.

Usage

```
which.min.fair(x)
which.max.fair(x)
```

Arguments

x	numeric, logical, integer or double vector.

Details

These functions are alternatives to the standard R functions `which.min` and `which.max`.

The standard functions `which.min` and `which.max` find the index of the **first** entry in the vector `x` which achieves the minimum or maximum value. This can cause a bias in some simulation experiments.

The functions `which.min.fair` and `which.max.fair` identify all entries of the vector `x` which achieve the minimum or maximum respectively, and **select one of them at random**.

Value

A single integer (or `integer(0)` if all entries of `x` are `NA` or `NaN`).

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>

See Also

`which.min`

Examples

```
z <- c(20, 40, 20, 10, 40, 20, 10, 20, 40)
replicate(5, which.max(z))
replicate(5, which.max.fair(z))
replicate(5, which.min.fair(z))
```

Index