

Package ‘gbutils’

December 18, 2025

Type Package

Title Utilities for Simulation, Plots, Quantile Functions and Programming

Version 0.5.1

Depends methods

Imports stats, utils, Rdpack (>= 0.9)

Suggests testthat (>= 3.0.0), classGraph, graph, Rgraphviz

RdMacros Rdpack

Description Plot density and distribution functions with automatic selection of suitable regions. Numerically invert (compute quantiles) distribution functions. Simulate real and complex numbers from distributions of their magnitude and arguments. Optionally, the magnitudes and/or arguments may be fixed in almost arbitrary ways. Create polynomials from roots given in Cartesian or polar form. Small programming utilities: check if an object is identical to NA, count positional arguments in a call, set intersection of more than two sets, check if an argument is unnamed, compute the graph of S4 classes in packages.

URL <https://geobosh.github.io/gbutils/> (doc),
<https://CRAN.R-project.org/package=gbutils>

BugReports <https://github.com/GeoBosh/gbutils/issues>

License GPL (>= 2)

Encoding UTF-8

Collate parse_text.R sim_numbers.R isNA.R mintersect.R pad.R args.R
history.R cdf2qf.R S4utils.R pseudoInverse.R rpoly.R

Config/testthat/edition 3

NeedsCompilation no

Author Georgi N. Boshnakov [aut, cre] (ORCID:
<https://orcid.org/0000-0003-2839-346X>)

Maintainer Georgi N. Boshnakov <georgi.boshnakov@manchester.ac.uk>

Repository CRAN

Date/Publication 2025-12-18 17:10:17 UTC

Contents

adjacencyOfClasses	2
cdf2quantile	6
isargunamed	7
isNA	9
mintersect	10
missing_arg	11
myouter	12
nposargs	13
parse_text	14
plotpdf	15
pseudoInverse	17
raw_history	17
rpoly	18
sim_complex	20
sim_numbers	22
Index	25

adjacencyOfClasses	<i>Adjacency graph of classes in packages</i>
--------------------	---

Description

Get inheritance graph of classes in one or more packages.

Usage

```
adjacencyOfClasses(packages, externalSubclasses = FALSE,  
                    result = c("default", "matrixOfPairs", "adjacencyMatrix"),  
                    Wolfram = FALSE)
```

Arguments

- | | |
|--------------------|--|
| packages | names of one or more packages, a character vector. |
| externalSubclasses | if TRUE, exclude subtrees of classes not defined in any of the packages listed in argument packages. |
| result | format of the result, can be missing or one of "default", "matrixOfPairs", "adjacencyMatrix", see section 'Details'. |
| Wolfram | if TRUE, print a suitable graph expression to be run by Mathematica, see section 'Details'. |

Details

`adjacencyOfClasses` computes a graph representation of the dependencies of S4 classes defined in one or more packages (as specified by argument `package`) and returns a list. The contents of the list returned by `adjacencyOfClasses` depend on argument `result`. Partial matching is used for the value of argument `result`, e.g., "adj" is equivalent to "adjacencyMatrix".

If `externalSubclasses = FALSE`, the default, subclasses defined outside the requested packages are excluded. This is typically what the user will be looking for. To get a complete tree, set `externalSubclasses` to `TRUE`.

The S4 classes are represented by the vertices of the graph. Component "vertices" of the result gives them as a character vector. References below to the *i*th class or vertex correspond to the order in this vector. No attempt is made to arrange the vertices in a particular order. An empty list is returned if this vector is empty.

If `result` is missing or "default", the edges of the graph are represented by a character vector. Each edge is represented by a string with an arrow " \rightarrow " from a superclass to a subclass. Here is an example that shows that this package defines one class, which is a subclass of "list":

```
adjacencyOfClasses("gbutils")
##: $vertices
##: [1] "objectPad" "list"

##: $edges
##: [1] "list -> objectPad"
```

This illustrates the effect of argument "externalSubclasses":

```
adjacencyOfClasses("gbutils", externalSubclasses = TRUE)
##: $vertices
##: [1] "objectPad" "list"      "vector"

##: $edges
##: [1] "list -> objectPad" "vector -> list"
```

The edge, "vector -> list" was omitted in the previous example since this relationship is defined elsewhere. This resulted in class "vector" being dropped also from the vertices, since it is not defined in "gbutils" and none of the remaining edges contains it.

If `result` is "matrixOfPairs", the edges of the graph are represented by a character matrix with two columns, where each row represents an edge from the element in the first column to the element in the second. In this example there is one edge, so the matrix contains one row:

```
adjacencyOfClasses("gbutils", result = "matrixOfPairs")
##: $vertices
##: [1] "objectPad" "list"

##: $edges
```

```
##:      [,1] [,2]
##: [1,] "list" "objectPad"
```

If result is "adjacencyMatrix", the adjacency matrix of the graph is in component "AM" of the returned list. Element (i, j) of this matrix is equal to one, if the j th class is a superclass to the i th. In other words, the j th column gives the superclasses of the i th class. Here the element in position (1,2) is non-zero, so "list" is the superclass of "objectPad":

```
adjacencyOfClasses("gbutils", result = "adjacencyMatrix")
##: $vertices
##: [1] "objectPad" "list"

##: $AM
##:      objectPad list
##: objectPad      0   1
##: list           0   0
```

Note that including the vertices in the result is not redundant, since some may not be in any edge. This can happen if a class does not have any superclasses and subclasses.

As described above the result is not converted to a graph object but it can be fed to functions provided by a number of R packages.

An additional option is to use argument Wolfram. If Wolfram is TRUE, a suitable Mathematica command is printed. It can be evaluated in a Mathematica session (e.g., by copy/paste) to produce a graphical representation of the graph and/or be manipulated further by it. This feature is a side effect, the return value of adjacencyOfClasses is as controlled by the other arguments. For example, the return value below is as without argument "Wolfram" but, in addition, the printed line defines a Wolfram language graph in terms of its vertices and edges:

```
adjacencyOfClasses("gbutils", Wolfram = TRUE)
##: Graph[{objectPad,list}, {list -> objectPad}, VertexLabels -> Automatic]

##: $vertices
##: [1] "objectPad" "list"

##: $edges
##: [1] "list -> objectPad"
```

Setting result = "adjacencyMatrix" in the last R command exports the graph in terms of its adjacency matrix:

```
adjacencyOfClasses("gbutils", Wolfram = TRUE, result = "adjacencyMatrix")
##: AdjacencyGraph[{objectPad,list}, {{0, 0},
##: {1, 0} }
##: , VertexLabels -> Automatic]
```

```
##: $vertices
##: [1] "objectPad" "list"

##: $AM
##:      objectPad list
##: objectPad      0   1
##: list           0   0
```

Value

a list with some of the following components (as described in section ‘Details’):

vertices	a character vector of S4 class names,
edges	the edges of the graph, in the format controlled by argument <code>results</code> (not present when <code>result</code> is equal to "adjacencyMatrix"),
AM	the adjacency matrix of the graph (present only when <code>result</code> is "adjacencyMatrix").

Author(s)

Georgi N. Boshnakov

References

Gentleman R, Whalen E, Huber W, Falcon S (2017). *graph: A package to handle graph data structures*. R package version 1.56.0.

Hansen KD, Gentry J, Long L, Gentleman R, Falcon S, Hahne F, Sarkar D (2017). *Rgraphviz: Provides plotting capabilities for R graph objects*. R package version 2.22.0.

Maechler M (2015). *classGraph: Construct Graphs of S4 Class Hierarchies*. (partly based on code from Robert Gentleman) R package version 0.7-5, <https://CRAN.R-project.org/package=classGraph>.

See Also

`?methods::classesToAM` which is used for the main computation here,

Maechler (2015) for a suite of related functions. Gentleman et al. (2017) for creation and manipulation of graphs, and Hansen et al. (2017) for visualisation of graphs.

Examples

```
adjacencyOfClasses("gbutils")
adjacencyOfClasses("gbutils", TRUE)

adjacencyOfClasses("gbutils", FALSE, "matrixOfPairs")
adjacencyOfClasses("gbutils", TRUE, "matrixOfPairs")

adjacencyOfClasses("gbutils", FALSE, "adjacencyMatrix")
adjacencyOfClasses("gbutils", TRUE, "adjacencyMatrix")
```

```
## as above, also represent the graph using the edges
adjacencyOfClasses("gbutils", Wolfram = TRUE)
adjacencyOfClasses("gbutils", TRUE, Wolfram = TRUE)

## here the graph is represented by the adjacency matrix:
adjacencyOfClasses("gbutils", FALSE, "adjacencyMatrix", Wolfram = TRUE)
adjacencyOfClasses("gbutils", TRUE, "adjacencyMatrix", Wolfram = TRUE)

if(requireNamespace("graph", quietly = TRUE) &&
    requireNamespace("Rgraphviz", quietly = TRUE)) withAutoprint({
  ## another package
  adjacencyOfClasses("graph")
  ac1 <- adjacencyOfClasses("graph", FALSE, "adjacencyMatrix")
  ## note the use of t() below
  gr_ac1 <- graph::graphAM(adjMat = t(ac1$AM), edgemode = "directed")
  if(require("Rgraphviz", quietly = TRUE, warn.conflicts = FALSE))
    plot(gr_ac1)
  ## more than one package
  ac2 <- adjacencyOfClasses(c("graph", "Rgraphviz"), FALSE, "adjacencyMatrix")
  gr_ac2 <- graph::graphAM(adjMat = t(ac2$AM), edgemode = "directed")
  if(require("Rgraphviz", quietly = TRUE))
    plot(gr_ac2)
})
```

cdf2quantile

*Calculate a quantile from a distribution function***Description**

Numerically calculate a quantile from a distribution function.

Usage

```
cdf2quantile(p, cdf, interval = c(-3, 3), lower = min(interval),
             upper = max(interval), ...)
```

Arguments

p	a number in the interval (0,1).
cdf	cumulative distribution function, a function.
interval	interval in which to look for the root, see section ‘Details’.
lower	lower end point of the interval.
upper	upper end point of the interval.
...	any further arguments to be passed to the root finding function and the cdf, see section ‘Details’.

Details

The quantile, q , is computed numerically as the solution of the equation $cdf(q) - p = 0$.

Function `uniroot` is used to find the root. To request higher precision, set argument `tol`. Other arguments in `...` are passed on to `cdf`.

`uniroot` needs an interval where to look for the root. There is a default one, which is extended automatically if it does not contain the quantile. This assumes that argument `cdf` is an increasing function (as it should be).

To override the default interval, use argument `interval` (a vector of two numbers) or `lower` and/or `upper`. This may be necessary if the support of the distribution is not the whole real line and `cdf` does not cope with values outside the support of the distribution.

Value

the computed quantile as a number

Author(s)

Georgi N. Boshnakov

See Also

[plotpdf](#)

Examples

```
cdf2quantile(0.95, pnorm)
cdf2quantile(0.05, pexp) # support [0,Inf) is no problem for
cdf2quantile(0.05, plnorm) # for built-in distributions.

## default predication is about 4 digits after decimal point
cdf2quantile(0.95, pnorm, mean = 3, sd = 1)
cdf2quantile(0.05, pnorm, mean = 3, sd = 1)
qnorm(c(0.95, 0.05), mean = 3, sd = 1)

## request a higher precision:
cdf2quantile(0.05, pnorm, mean = 3, sd = 1, tol = 1e-8)
cdf2quantile(0.05, pnorm, mean = 3, sd = 1, tol = 1e-12)

## see also examples for plotpdf()
```

isargunnamed

Is an element of a list named?

Description

Check if a component of a list is not named.

Usage

```
isargunnamed(x, k)
```

Arguments

x	a list.
k	an integer, specifies a position in x.

Details

isargunnamed(x, k) returns TRUE if the k-th component of x is not named and FALSE otherwise.

Argument x is typically a list of arguments used in a call to a function, such as the one obtained by list(...) in the body of a function definition.

If k is not positive or larger than length(x), isargunnamed returns FALSE.

Value

TRUE or FALSE

Author(s)

Georgi N. Boshnakov

See Also

[match.call](#) in base package

Examples

```
li1 <- list(a=1, 2, 3)
isargunnamed(li1, 1)
isargunnamed(li1, 2)

## wholly unnamed list
li2 <- list(1, 2, 3)
isargunnamed(li2, 1)
isargunnamed(li2, 2)

## using in a function definition
f1 <- function(...){
  dots <- list(...)
  for(i in seq(along=dots))
    cat(i, isargunnamed(dots, i), "\n")
}

f1(1)
f1(a = 3, 4, c = 5)
f1(x = "a", y = "b")
```

`isNA`*Check if an object is NA*

Description

Check if an object is NA. Always return TRUE or FALSE, a logical vector of length one.

Usage

```
isNA(x)
```

Arguments

`x` any R object.

Details

`isNA` returns TRUE if the argument is a single NA, i.e. it is atomic, has length one, and represents an NA value. In any other case `isNA` returns FALSE.

`isNA` is suitable for use in conditional constructs since it always returns a single value which is never NA.

Note that `identical()` distinguishes different types of NA, i.e. `identical(x, NA)` is TRUE only if `x` is NA (logical).

Value

TRUE or FALSE

Note

The requirement that `x` is atomic means that `isNA(list(NA))` gives FALSE.

For comparison, `is.na(list(NA))` gives TRUE. The same holds for classed lists, such as `is.na(structure(list(NA), class = "myclass"))`.

Author(s)

Georgi N. Boshnakov

See Also

[isTRUE](#), [is.na](#), [identical](#)

Examples

```

v <- c(1, NA, 3)
isNA(v[2]) # TRUE

## a vector of two or more Na's is not isNA
isNA(rep(NA,3)) # FALSE

## a list containing NA is not isNA
isNA(list(NA)) # FALSE
## ... but
is.na(list(NA)) # TRUE

## identical() distinguishes different types of NA:
class(v) # "numeric", not "integer"

identical(v[2], NA)          # FALSE, NA on its own is "logical"
identical(v[2], NA_integer_) # FALSE
identical(v[2], NA_real_)    # TRUE

vi <- c(1L, NA_integer_, 3L)
isNA(vi[2]) # TRUE

class(vi) # "integer"
identical(vi[2], NA_integer_) # TRUE
identical(vi[2], NA_real_)    # FALSE

## is.na(NULL) would give a warning
isNA(NULL) # FALSE

## a length zero object is not NA, so isNA() returns FALSE:
isNA(logical(0)) # FALSE

## is.na() has a different remit and returns a 0-length vector:
is.na(logical(0)) # logical(0)

```

mintersect

Set intersection of arbitrary number of arguments

Description

Set intersection of arbitrary number of arguments.

Usage

```
mintersect(...)
```

Arguments

... arguments to be intersected, vectors of the same mode, see intersect.

Details

The base R function `intersect` is a binary operation. `mintersect` works with any positive number of arguments.

If called with one argument, `mintersect` returns it. This is unlike `intersect` which gives an error in this case.

Calling `mintersect` with no arguments is an error (as it is for `intersect`).

Value

a vector representing the intersection of the arguments

Author(s)

Georgi N. Boshnakov

Examples

```
mintersect(1:20, 3:18, 7:12)
mintersect(letters[1:20], letters[3:18], letters[7:12])
mintersect(1:4)
```

missing_arg

Check if an element of a pairlist is missing

Description

Check if an element of a pairlist is missing.

Usage

```
missing_arg(arg)
```

Arguments

`arg` the object to test.

Details

The argument passed to `missing_arg` is typically an element of a `pairlist` or the list produced by `alist()`. `missing_arg` returns `TRUE` if it is missing and `FALSE` otherwise.

Objects of type `pairlist` come up at R level almost exclusively as the formal arguments of functions. `missing_arg` can be useful when they are manipulated programmatically.

Value

`TRUE` or `FALSE`

Author(s)

Georgi N. Boshnakov

Examples

```
lmargs <- formals(lm)
class(lmargs) # pairlist
missing_arg(lmargs$data)
## which arguments of lm() have no (explicit) defaults?
sapply(lmargs, missing_arg)

## This gives an error:
## pairlist(x = 3, y = , z = 5)

## an example with alist()
pl2 <- alist(a = "", b = , c = 3)
class(pl2) # list
## this shows that 'b' is missing, 'a' and 'c' are not:
sapply(pl2, missing_arg) # FALSE TRUE FALSE
## superficially, 'b' is equal to the empty string:
pl2[[2]]
sapply(pl2, function(x) x == "") # TRUE TRUE FALSE

## with pairlist the results are the same:
g <- function(a = "", b, c = 3) NULL
a.g <- formals(g)
class(a.g) # pairlist
sapply(a.g, missing_arg) # FALSE TRUE FALSE
a.g[[2]]
sapply(a.g, function(x) x == "") # TRUE TRUE FALSE
```

myouter

*Functions for some basic operations***Description**

Small utility functions

Usage

```
myouter(x, y, fun)
shiftright(x, k = 1)
shiftleft(x, k = 1)
```

Arguments

x	a vector.
y	a vector.
k	a non-negative integer.
fun	a function, see ‘Details’.

Details

`myouter(x,y,fun)` computes the outer product of `x` and `y` using the function `fun`. The result is a matrix with (i,j) th element equal to `fun(x[i],y[j])`. It is not required for `fun` to be able to work with vector arguments. The function does the computations in R using a simple double loop. So, it is a convenience function, not a speed improving one.

`shiftright(x,k)` rotates the vector `x` `k` positions to the right.

`shiftright(x,k)` rotates the vector `x` `k` positions to the left.

Value

for `myouter`, a matrix, as described in ‘Details’

for `shiftright` and `shiftright`, a vector

Author(s)

Georgi N. Boshnakov

nposargs

Function to count the number of positional arguments used in a call

Description

Calculates the number of positional arguments used in a call.

Usage

```
nposargs(x, a = FALSE)
```

Arguments

`x` a call object, usually obtained from `sys.call()`.

`a` if `a[1]` is TRUE make a correction to distinguish `x[]` from `x[i]`, see details.

Details

`nposargs` is mainly for use in the body of function definitions, particularly for functions or methods that wish to mimic the behaviour of `"["`.

`nposargs` gives the number of positional arguments used in a call. It also takes into account empty arguments like those used in expressions like `x[1,]`.

Optionally, it makes a particular correction that is peculiar for `"["` - if there are no named arguments in the call and the count of the arguments is 2 and `a[1]=TRUE`, it decreases the count by one, i.e. returns 1. This is to distinguish between a `x[]` and `x[i]` which both would give 2 otherwise. I have forgotten the details but, roughly speaking, `x[i]` becomes `"["(x,i)` while `x[]` becomes `"["(x,)`, i.e. R puts the comma after `x` in any case.

Value

the number of positional arguments in the call

Note

I wrote this function (a long time ago) for use in methods for "[".

a[1] above is typically obtained by a call `missing(i)` somewhere at the beginning of the function. In my application I put the results of several such calls in a vector, hence the check for `a[1]` rather than `a`. For "[", we may set `a = c(missing(i), missing(j), missing(k))`.

Author(s)

Georgi N. Boshnakov

Examples

```
f <- function(x,y,z,...){
  call <- sys.call()
  nposargs(call)
}
f(a,b,c) # 3
f(a, , ) # 3
f(a, )   # 2
f(a)     # 1
f(, )    # 2
f(, a, ) # 3
f()      # 0
```

parse_text

Parse expressions residing in character vectors

Description

Parse expressions residing in character vectors. Similar to `parse()` but keeping or not the source is controlled by an argument rather than global options.

Usage

```
parse_text(text, ..., keep = TRUE)
```

Arguments

text	the text to parse, normally a character vector but can be anything that <code>parse</code> accepts for this argument.
...	additional arguments to be passed on to <code>parse</code> .
keep	required setting for option <code>keep.source</code> , see section 'Details'.

Details

This is like `parse(text=text, ...)`, except that whether or not the source is kept is controlled by argument `keep`, not by `options("keep.source")`.

`parse_text` sets `options("keep.source")` to `keep` (if they are different) before calling `parse` and restores it afterwards.

Value

an expression representing the parsed text, see [parse](#) for details

Note

The usual setting of option `"keep.source"` in interactive sessions is `TRUE`. However, in ‘R CMD check’ it is `FALSE`.

As a consequence, if the documentation of a package uses functions that depend on option `"keep.source"` being `TRUE`, then some examples may run fine when copied and pasted in an R session but (rightly) fail ‘R CMD check’.

The oposite may also happen, in that the documentation passes ‘R CMD check’ or Sweave files successfully build but some examples do not work when copied and pasted in an interactive session.

Author(s)

Georgi N. Boshnakov

See Also

[parse](#)

plotpdf

Plot a probability density function

Description

Plot a probability density function with x-axis limits determined by quantiles of the distribution. Quantiles are computed using a quantile function or cumulative distribution function, whichever is supplied.

Usage

```
plotpdf(pdf, qdf, cdf, lq = 0.01, uq = 0.99, ...)
```

Arguments

pdf	probability density to be plotted, a function.
qdf	quantile function to be used for computation of quantiles, a function.
cdf	cumulative distribution function to be used for computation of quantiles, a function. This argument is used if qdf is not given, see 'Details' section.
lq	lower quantile, used in the computation of the left limit.
uq	upper quantile, used in the computation of the right limit.
...	additional arguments to be passed on to the plot function.

Details

The function plots $\text{pdf}(x)$ over the interval (x_{\min}, x_{\max}) where x_{\min} and x_{\max} are the lqth and uqth quantiles, respectively, of the distribution. The quantile function, qdf , is used, if supplied. Otherwise the quantiles are computed numerically from the cdf .

Argument pdf is not required to be a pdf, it may be any function. For example, the same way of choosing the limits may be appropriate for a plot of the cdf , see the examples.

Similarly, qdf and cdf need not be related to pdf .

Author(s)

Georgi N. Boshnakov

See Also

[cdf2quantile](#)

Examples

```
pdf1 <- function(x) dnorm(x, mean = 100, sd = 5)
qdf1 <- function(x) qnorm(x, mean = 100, sd = 5)
cdf1 <- function(x) pnorm(x, mean = 100, sd = 5)

plot(pdf1) # needs to specify 'from' and 'to' args for meaningful plot
plotpdf(pdf1, qdf1) # using quantile function
plotpdf(pdf1, cdf = cdf1) # using cdf
plotpdf(pdf1, cdf = cdf1, lq = 0.001, uq = 0.999) # ... and non-default quantiles

plotpdf(cdf1, cdf = cdf1, lq = 0.001, uq = 0.999) # plot a cdf

## a mixture distribution:
pf1 <- function(x){
  0.25 * pnorm(x, mean = 3, sd = 0.2) + 0.75 * pnorm(x, mean = -1, sd = 0.5)
}
df1 <- function(x){
  0.25 * dnorm(x, mean = 3, sd = 0.2) + 0.75 * dnorm(x, mean = -1, sd = 0.5)
}
```



```
plotpdf(df1, cdf = pf1) # plot the pdf
plotpdf(pf1, cdf = pf1) # plot the cdf

c(cdf2quantile(0.05, pf1), cdf2quantile(0.95, pf1))
```

pseudoInverse	<i>Compute a pseudo-inverse matrix</i>
---------------	--

Description

Compute a pseudo-inverse matrix using singular value decomposition and setting very small singular values to zero.

Usage

```
pseudoInverse(a, tol = 1e-07)
```

Arguments

a	a matrix
tol	a number, the threshold for non-zero singular values.

Details

The singular value decomposition of a is computed and singular values smaller than tol are set to zero. The result is formed using the standard formula.

Value

a matrix

Examples

```
##---- Should be DIRECTLY executable !! ----
```

raw_history	<i>Get the command history</i>
-------------	--------------------------------

Description

Get the command history.

Usage

```
raw_history()
```

Details

The command history is saved to a temporary file with `savehistory` and read back into a character vector.

Value

a character vector

Author(s)

Georgi N. Boshnakov

Examples

```
## Not run:
hist <- raw_history()
length(hist)

## End(Not run)
```

rpoly

Polynomials with real coefficients

Description

Compute the coefficients of a polynomial with real coefficients, given its real zeroes (roots) and one representative for each complex pair. If complex numbers are given in polar form, there is an option to specify the complex arguments as multiples of π .

Usage

```
rpoly(x = numeric(0), arg = numeric(0), real = numeric(0), argpi = FALSE,
      monic = TRUE)
```

Arguments

<code>x</code>	if complex, the roots (including the real ones), otherwise the moduli of the complex roots of the polynomial. In both cases only one representative for each complex pair should be included.
<code>arg</code>	the complex arguments corresponding to the moduli in <code>x</code> . This argument is not needed when <code>x</code> is complex.
<code>real</code>	the real roots of the polynomial. This argument is not needed when <code>x</code> is complex.
<code>argpi</code>	if TRUE, then <code>arg</code> represents the complex arguments as a multiple of π , see section ‘Details’. The default is FALSE.
<code>monic</code>	if TRUE, the default, the coefficient of the highest term of the polynomial is set to 1. if FALSE, the constant term is one.

Details

The complex zeroes of polynomials with real coefficients come in complex conjugated pairs. Only one representative from each pair should be supplied to rpoly. The other is added automatically. Of course, all real roots should be supplied, if any.

If x is complex, it should contain all real roots and one representative for each complex pair.

Otherwise, if x is not complex, it contains the moduli of the numbers and arg contains the complex arguments. The two should be of equal length.

With the default FALSE for argpi, the k-th root of the polynomial is $x[k] \cdot \cos(\arg[k]) + i \cdot x[k] \cdot \sin(\arg[k])$. If argpi is TRUE it is $x[k] \cdot \cos(\pi \cdot \arg[k]) + i \cdot x[k] \cdot \sin(\pi \cdot \arg[k])$.

By default, a monic polynomial (the coefficient of the highest order term is 1) is created but if monic is FALSE, the constant term of the polynomial is set to 1.

The options for argpi = TRUE and/or monic = FALSE are convenient in some applications, e.g., time series analysis and digital signal processing.

Value

a real vector containing the coefficients of the polynomial.

Note

When argpi is TRUE, $\cos(\pi a)$ is computed using $\text{cospi}(a)$. So this may differ slightly from the equivalent result obtained with argpi = FALSE and $b = \pi \cdot a$, which is computed as $\cos(b) = \cos(\pi \cdot a)$, see the example.

Author(s)

Georgi N. Boshnakov

See Also

[sim_numbers](#)

Examples

```
## z-1
rpoly(real = 1)

## roots 1, i, -i; p3(z) = (z-1)(z-i)(z+i)
p3 <- rpoly(c(1, 1i))
p3
polyroot(p3)

## using polar for the complex roots (i = e^(i pi/2))
p3a <- rpoly(1, pi/2, real = 1)
p3a
## mathematically, p3a is the same as p3
## but the numerical calculation here gives a slight discrepancy
p3a == p3
p3a - p3
```

```
## using argpi = TRUE is somewhat more precise:
p3b <- rpoly(1, 1/2, real = 1, argpi = TRUE)
p3b
p3b == p3
p3b - p3
## indeed, in this case the results for p3b and p3 are identical:
identical(p3b, p3)

## two ways to expand (z - 2*exp(i*pi/4))(z - 2*exp(-i*pi/4))
rpoly(2, pi/4)
rpoly(2, 1/4, argpi = TRUE)

## set the constant term to 1; can be used, say, for AR models
rpoly(2, pi/4, monic = FALSE)
rpoly(2, 1/4, argpi = TRUE, monic = FALSE)
```

sim_complex

Simulate real or complex numbers using polar form

Description

Simulate complex numbers with given distributions for the modulus and the argument and real numbers with given distributions for the absolute value and the sign. Some of the values may be partially or fully specified.

Usage

```
sim_complex(abs, arg, absgen = "runif", absarg = list(0, 1),
            arggen = runif, argarg = list(-pi, pi), ...)

sim_real(abs, sign, signprob = 0.5, absgen = "runif",
          absarg = list(0, 1), ...)
```

Arguments

abs	vector of absolute values.
sign	vector of signs (1 or -1).
signprob	probability for a positive sign.
arg	vector of arguments (of complex numbers).
absgen	generator for the absolute values, a function or a character string naming a function.
absarg	arguments for absgen.
arggen	generator for the arguments of the complex numbers, a function or a string naming a function.
argarg	arguments for arggen.
...	not used, simplifies the call from sim_numbers.

Details

`sim_real` simulates real numbers by simulating separately their absolute values and signs. `sim_complex` simulates complex numbers by simulating separately their moduli and arguments.

Both functions replace NA's in argument `abs` with values simulated by the function specified by `absgen`. Arguments for `absgen` are specified by the (possibly named) list `absarg`.

Similarly, `sim_complex` replaces NA's in argument `arg` with values simulated according to `arggen` and `argarg`.

Further, `sim_real` replaces NA's in argument `sign` with a random sample of ones and minus ones, where the probability for the positive value is `signprob`.

Only NA entries in `abs`, `arg` and `sign` are filled with simulated values, the remaining entries are left unchanged. This means that some (and even all) values may be specified partially or completely.

`abs` is combined with `arg` or `sign` to create the result. These arguments are expected to be of matching shape and length but this is not enforced and the usual recycling rules will apply if this is not the case (not recommended to rely on this).

The default range for the (complex) argument is $(-\pi, \pi)$.

Value

for `sim_real`, a vector of real numbers

for `sim_complex`, a vector of complex numbers

Note

Currently the shape of the result for `sim_real` is the same as that of argument `abs`. But `sim_complex` always returns a vector. Probably this inconsistency should be removed.

Author(s)

Georgi N. Boshnakov

See Also

[sim_numbers](#) which offers more flexible interface to these functions.

Examples

```
## x[1] is fixed to 1, x[2] is negative with random magnitude:
x <- sim_real(c(1,NA,NA,NA), c(1, -1, NA, NA))

## z[1] fixed to 1, the remaining elements of z
## have random magnitude and fixed arguments:
z <- sim_complex(c(1,NA,NA,NA), c(0, pi/2, pi, -pi/2))

## without restrictions
sim_complex(rep(NA,4))
sim_real(rep(NA,4))

## moduli unrestricted; arguments restricted
```

```
sim_complex(rep(NA,4), c(0, pi/2, pi, -pi/2))
```

sim_numbers

Simulation based on polar form specifications

Description

Simulate real and complex numbers from polar form specifications. The numbers may be partially or fully specified. The distributions of absolute values and arguments/signs are specified independently.

Usage

```
sim_numbers(type = rep(as.character(NA), length(abs)),
            abs = rep(as.numeric(NA), length(type)),
            sign = rep(as.numeric(NA), length(type)), values = NULL, ...)
```

Arguments

type	character vector specifying the types of the eigenvalues, see Details.
abs	vector of absolute values (moduli).
sign	vector of signs (for reals) and arguments (for complex numbers), see Details for interpretation.
values	values, see details.
...	additional arguments to be passed to sim_real and sim_complex.

Details

sim_numbers simulates a vector of real and complex numbers with given distributions of their polar parts. It is possible also to fix some of the numbers or one of their polar parts. The length of the simulated vector is inferred from the length of type or abs, so one of them must be provided. sim_numbers is a flexible front-end for sim_real and sim_complex.

sim_numbers generates a vector of values with types specified by argument type and/or inferred from argument values. The recommended way to use sim_numbers is to provide argument type.

type[i] specifies the type of the i-th element of the result: real (type[i]=="r"), complex (type[i]=="c") or representing a complex conjugate pair (type[i]=="cp"). If values is provided, the imaginary parts of its non-NA elements are used to fill NA elements of type ("r" if zero, "cp" otherwise).

Some (or even all) values may be fixed or partially fixed with the help of arguments abs, sign and values. A non-missing value of values[i] fixes the i-th element of the result to that value. Similarly abs[i] fixes the modulus and sign[i] fixes the sign/argument of the i-th element. If values[i] is not NA, then it takes precedence and abs[i] and sign[i] are ignored.

For real numbers sign is the sign with possible values 1 (positive) or -1 (negative). For complex numbers, sign is the argument and is in the interval $(-\pi, \pi)$.

If values is supplied, then NA entries in type are replaced by "r" or "cp" depending on whether or not the imaginary parts of the corresponding entries in values are equal to zero. A check is done

for consistency when both `type[i]` and `values[i]` are non-missing. Generally, `values` is meant to be used for values that are fixed and available directly in Cartesian form, to avoid having to fill the corresponding entries of `abs` and `sign`.

NA entries of `abs` and `sign` are filled with simulated values, the remaining entries are considered fixed and left unchanged. The default generator is uniform (0,1) for `abs`, uniform $(-\pi, \pi)$ for the argument of complex values, and 1 or -1 with $p=1/2$ for the sign of real values.

To specify a different generator for the moduli and absolute values, use argument `absgen`, giving it a function or the name of a function. The arguments for this function can be specified by `absarg` (as a list). Similarly, the generator for arguments of complex numbers can be specified by `arggen` and `argarg`. Finally, the probability for the real numbers to be positive is given by `signprob`. These arguments are not in the signature of the function since they are passed on directly (via "...") to the underlying `sim_complex` and `sim_real`, see their documentation and the examples below for further details.

Value

a list with components:

<code>values</code>	vector of values; it is of type <code>numeric</code> if all values are real and complex otherwise.
<code>type</code>	a character vector of the types as above

Note

Values of type "cp" (complex pairs) are represented by one element, the complex conjugate elements are NOT generated. (todo: maybe add an argument to control this)

The convention for the sign of a real eigenvalue is 1 and -1, not 0 and π .

The checks for consistency between `type` and `values` are not complete and only straightforward use is recommended.

The current defaults for the arguments, see the signature above, require that at least one of `type` and `abs` is not missing.

Author(s)

Georgi N. Boshnakov

See Also

[sim_real](#), [sim_complex](#)

Examples

```
## one real number and one complex conjugated pair
## (maybe to specify a cubic polynomial through its roots)
sim_numbers(type = c("r", "cp"))

## here the real value is fixed to have modulus 1, leaving the sign unspecified
sim_numbers(type = c("r", "cp"), abs = c(1, NA))
```

```

## now the real value is fixed to 1,
##     the complex pair has argument  $+\pi/2$ , and free modulus:
sim_numbers(type = c("r", "cp"), abs = c(1, NA), sign = c(0, pi/2))

## using argument 'values' to fix some values;
## here the the third value is fixed:
sim_numbers(type = c("r", "cp", "r"), values = c(NA,NA,3)) # type[3] = "r"
sim_numbers(type = c("r", "cp", "cp"), values = c(NA,NA,3i)) # type[3] = "cp"

## type[3] can be left NA since it can be inferred from values[3]:
sim_numbers(type = c("r", "cp", NA), values = c(NA,NA,3)) # type[3] = "r"
sim_numbers(type = c("r", "cp", NA), values = c(NA,NA,3i)) # type[3] = "cp"

## it is an error to have a mismatch between args `type' and value:
## Not run:
sim_numbers(type = c("r", "cp", "cp"), values = c(NA,NA,3))
sim_numbers(type = c("r", "cp", "r"), values = c(NA,NA,3i))

## End(Not run)

## simulate 10 reals with the default generators
sim_numbers(rep("r", 10))

## simulate modulus from Rayleigh distribution
##
rR <- function(n, sigma = 1) sigma * sqrt(-2*log(runif(n)))
sim_numbers(type = c("cp", "cp"), absgen = rR, absarg = list())

# test the the components are  $N(0,1)$ 
## (not run to save time for CRAN check)
## \dontrun{
## v <- sim_numbers(type = rep("cp", 10000), absgen = "rR",
##                   absarg = list(sigma = 1))
## ks.test(Re(v$values), "pnorm")
## ks.test(Im(v$values), "pnorm")
## }

```


Index

- * **NA**
 - isNA, [9](#)
- * **S4classes**
 - adjacencyOfClasses, [2](#)
- * **distribution**
 - cdf2quantile, [6](#)
 - sim_complex, [20](#)
 - sim_numbers, [22](#)
- * **dplot**
 - plotpdf, [15](#)
- * **hplot**
 - plotpdf, [15](#)
- * **logic**
 - isNA, [9](#)
- * **manip**
 - isNA, [9](#)
- * **math**
 - pseudoInverse, [17](#)
- * **programming**
 - adjacencyOfClasses, [2](#)
 - isargunnamed, [7](#)
 - mintersect, [10](#)
 - missing_arg, [11](#)
 - myouter, [12](#)
 - nposargs, [13](#)
 - parse_text, [14](#)
 - raw_history, [17](#)
- * **simulation**
 - sim_complex, [20](#)
 - sim_numbers, [22](#)

adjacencyOfClasses, [2](#)

cdf2quantile, [6](#), [16](#)

identical, [9](#)

is.na, [9](#)

isargunnamed, [7](#)

isNA, [9](#)

isTRUE, [9](#)

match.call, [8](#)

mintersect, [10](#)

missing_arg, [11](#)

myouter, [12](#)

nposargs, [13](#)

parse, [15](#)

parse_text, [14](#)

plotpdf, [7](#), [15](#)

pseudoInverse, [17](#)

raw_history, [17](#)

rpoly, [18](#)

shiftleft(myouter), [12](#)

shiftright(myouter), [12](#)

sim_complex, [20](#), [23](#)

sim_numbers, [19](#), [21](#), [22](#)

sim_real, [23](#)

sim_real(sim_complex), [20](#)