Package ‘gbutils’

October 5, 2018

Type Package
Title Simulation of Real and Complex Numbers and Small Programming Utilities
Version 0.4-0
Date 2018-10-05
Depends methods
Imports stats, utils, Rpack (>= 0.9)
Suggests classGraph, graph, Rgraphviz
RdMacros Rdpack
Description 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. Plot density and distribution functions with automatic selection of suitable regions. 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://github.com/GeoBosh/gbutils
https://geobosh.github.io/gbutils/
BugReports https://github.com/GeoBosh/gbutils/issues
License GPL (>= 2)
Encoding UTF-8
RoxygenNote 6.1.0
NeedsCompilation no
Author Georgi N. Boshnakov [aut, cre]
Maintainer Georgi N. Boshnakov <georgi.boshnakov@manchester.ac.uk>
Repository CRAN
Date/Publication 2018-10-05 15:40:03 UTC
**R topics documented:**

- adjacencyOfClasses .................................................. 2
- cdf2quantile ............................................................. 6
- isargunnamed ............................................................. 7
- isNA ............................................................................. 8
- mintersect ................................................................. 10
- missing_arg ................................................................. 11
- myouter ................................................................. 12
- nposargs ................................................................. 13
- parse_text ................................................................. 14
- plotpdf ................................................................. 15
- pseudoInverse .......................................................... 16
- raw_history ............................................................. 17
- sim_complex ............................................................. 18
- sim_numbers ............................................................. 20

**Index**

- adjacencyOfClasses .................................................. 23

---

**adjacencyOfClasses**

*Adjacency graph of classes in packages*

**Description**

Get inheritance graph of classes in one or more packages.

**Usage**

```r
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 Details.
- **Wolfram** if TRUE, print a suitable graph expression to be run by Mathematica, see 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 \textit{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 "->" from a superclass to a subclass. Here is an example that shows that this package defines one class, which is a subclass of "list":

```r
adjacencyOfClasses("gbutils")
```

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

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

This illustrates the effect of argument "externalsubclasses":

```r
adjacencyOfClasses("gbutils", externalsubclasses = TRUE)
```

```r
##: $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 no edge, so the matrix contains one row:

```r
adjacencyOfClasses("gbutils", result = "matrixOfPairs")
```

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

##: $edges
```
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 superclass to the \(i\)th. In other words, the \(i\)th 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 \text{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 = \text{TRUE})
###: Graph[{objectPad,list}, {list -> objectPad}, VertexLabels -> Automatic]

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

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

Setting result = "adjacencyMatrix" in the last \(R\) command would export a graph based on its adjacency matrix.

Value

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

vertices a character vector of S4 class names,
adjacencyOfClasses

edges the edges of the graph, in the format controlled by argument results (not present when result is equal to "adjacencyMatrix"),

AM the adjacency matrix of the graph (present only when result is "adjacencyMatrix").

Author(s)

Georgi N. Boshnakov

References


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,


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") && requireNamespace("Rgraphviz")) withAutoprint(
  ## another package
  adjacencyOfClasses("graph")
  ac1 <- adjacencyOfClasses("graph", FALSE, "adjacencyMatrix")
  gr_ac1 <- graph::graphAM(adjMat = ac1$AM, edgemode = "directed")
  if(require("Rgraphviz"))
    plot(gr_ac1)
  ## more than one package
### Description

Numerically calculate a quantile from a distribution function.

### Usage

```r
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 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 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.
isargunnamed

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 precision 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(0.95, 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, isargunnamed returns FALSE.
**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 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

Author(s)

Georgi N. Boshnakov

See Also

isTRUE, is.na, identical

Examples

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

```r
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
```
myouter

Functions for some basic operations

Description

Small utility functions

Usage

myouter(x, y, fun)
shiftleft(x, k = 1)
shiftright(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 \(\text{fun}(x[i], y[j])\). 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.
shiftleft(x, k) rotates the vector \(x\) \(k\) positions to the left.

Value

for myouter, a matrix, as described in ‘Details’
for shiftleft 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[i]`.

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]` is TRUE, it decreases the count by one, i.e. returns 1. This is to distinguish between `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

```r
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

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

Arguments

- `text` the text to parse, normally a character vector but can be anything that `parse` accepts for this argument.
- `...` additional arguments to be passed on to `parse`.
- `keep` required setting for option `keep.source`, see 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 opposite 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 $pdf(x)$ over the interval $(xmin, xmax)$ where $xmin$ and $xmax$ are the $lq$th and $uq$th 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

```r
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))
```

---

$pseudoInverse$ Compute a pseudo-inverse matrix
**Description**

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

**Usage**

```r
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**

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

---

**Description**

Get the command history.

**Usage**

```r
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**

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

## End(Not run)
```

---

**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**

```r
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

\texttt{sim\_real} simulates real numbers by simulating separately their absolute values and signs. \texttt{sim\_complex} simulates complex numbers by simulating separately their moduli and arguments.

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

Similarly, \texttt{sim\_complex} replaces NA's in argument \texttt{arg} with values simulated according to \texttt{arggen} and \texttt{argarg}.

Further, \texttt{sim\_real} replaces NA's in argument \texttt{sign} with a random sample of ones and minus ones, where the probability for the positive value is \texttt{signprob}.

Only NA entries in \texttt{abs}, \texttt{arg} and \texttt{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.

\texttt{abs} is combined with \texttt{arg} or \texttt{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 \texttt{sim\_real}, a vector of real numbers

for \texttt{sim\_complex}, a vector of complex numbers

Note

Currently the shape of the result for \texttt{sim\_real} is the same as that of argument \texttt{abs}. But \texttt{sim\_complex} always returns a vector. Probably this inconsistency should be removed.

Author(s)

Georgi N. Boshnakov

See Also

\texttt{sim\_numbers} which offers more flexible interface to these functions.

Examples

```r
## x[1] is fixed to 1, x[2] is negative with random magnitude:
x <- \texttt{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 <- \texttt{sim\_complex}(c(1,NA,NA,NA), c(0, pi/2, pi, -pi/2))

## without restrictions
\texttt{sim\_complex(rep(NA,4))}
\texttt{sim\_real(rep(NA,4))}

## moduli unrestricted; arguments restricted
```
**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**

```r
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` as 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
  - values: vector of values; it is of type numeric if all values are real and complex otherwise.
  - type: 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 pi.

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 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")
## }

sim_numbers(rep("r", 10))
Index

*Topic NA
  isNA, 8
*Topic S4classes
  adjacencyOfClasses, 2
*Topic distribution
  cdf2quantile, 6
  sim_complex, 18
  sim_numbers, 20
*Topic dplot
  plotpdf, 15
*Topic hplot
  plotpdf, 15
*Topic logic
  isNA, 8
*Topic manip
  isNA, 8
*Topic math
  pseudoInverse, 16
*Topic programming
  adjacencyOfClasses, 2
  isargunnamed, 7
  mintersect, 10
  missing_arg, 11
  myouter, 12
  nposargs, 13
  parse, 14, 15
  parse_text, 14
  plotpdf, 7, 15
  pseudoInverse, 16
  raw_history, 17
*Topic simulation
  sim_complex, 18
  sim_numbers, 20

adjacencyOfClasses, 2

cdf2quantile, 6, 16

identical, 9
is.na, 9
isargunnamed, 7
isNA, 8
isTRUE, 9

match.call, 8
mintersect, 10
missing_arg, 11
myouter, 12
nposargs, 13
parse, 14, 15
parse_text, 14
plotpdf, 7, 15
pseudoInverse, 16
raw_history, 17
shiftleft (myouter), 12
shiftright (myouter), 12
sim_complex, 18, 21
sim_numbers, 19, 20
sim_real, 21
sim_real (sim_complex), 18