Package ‘statnet.common’

May 24, 2023

Version 4.9.0
Date 2023-05-24
Title Common R Scripts and Utilities Used by the Statnet Project Software
Description Non-statistical utilities used by the software developed by the Statnet Project. They may also be of use to others.
Depends R (>= 3.5)
Imports utils, methods, coda, parallel, tools
BugReports https://github.com/statnet/statnet.common/issues
License GPL-3 + file LICENSE
URL https://statnet.org
Roxygen list(markdown = TRUE)
RoxygenNote 7.2.3
Encoding UTF-8
Suggests covr,
  rlang (>= 1.1.1),
  MASS,
  Matrix

R topics documented:

all_identical ................................................................. 3
as.control.list ............................................................. 3
attr ................................................................. 5
check.control.class .......................................................... 5
compress_rows ............................................................... 6
compress_rows.data.frame ................................................. 7
control.list.accessor ...................................................... 8
control.remap .............................................................. 9
default_options ............................................................ 10
deInf ................................................................. 10
deprecation-utilities ................................................... 11
R topics documented:

despaces ................................................................. 12
diff.control.list ...................................................... 12
empty_env ................................................................. 13
ERRVL ................................................................. 14
fixed.pval ................................................................. 15
forkTimeout .............................................................. 16
formula.utilities .................................................... 17
handle.controls ....................................................... 20
is.SPD ................................................................. 21
locate_function ....................................................... 21
logspace.utils .......................................................... 22
mcmc-utilities .......................................................... 24
message_print .......................................................... 25
NVL ................................................................. 26
once ................................................................. 28
opttest ................................................................. 29
order ................................................................. 30
paste.and ............................................................... 31
persistEval .............................................................. 32
print.control.list ...................................................... 33
set.control.class ...................................................... 34
simplify.simple .......................................................... 35
snctrl ................................................................. 36
snctrl_names ............................................................ 37
split.array .............................................................. 38
ssolve ................................................................. 39
statnet.cite ............................................................ 40
statnetStartupMessage ................................................. 41
sweep_cols.matrix .................................................... 42
term_list ............................................................... 43
trim_env ................................................................. 44
ult ................................................................. 45
unused_dots_warning .................................................. 46
unwhich ............................................................... 47
vector.namesmatch ................................................... 47
Welford ............................................................... 48
wmatrix ............................................................... 49
wmatrix_weights ...................................................... 52
xTAx ................................................................. 53

Index ..........................
Description

Test if all items in a vector or a list are identical.

Usage

all_identical(x)

Arguments

x

a vector or a list

Value

TRUE if all elements of x are identical to each other.

See Also

identical

Examples

stopifnot(!all_identical(1:3))

stopifnot(all_identical(list("a", "a", "a")))

Description

Convert to a control list.

Usage

as.control.list(x, ...)

## S3 method for class 'control.list'
as.control.list(x, ...)

## S3 method for class 'list'
as.control.list(x, FUN = NULL, unflat = TRUE, ...)

as.control.list
Arguments

x  An object, usually a list, to be converted to a control list.

... Additional arguments to methods.

FUN  Either a control.*() function or its name or suffix (to which "control." will be prepended); defaults to taking the nearest (in the call traceback) function that does not begin with "as.control.list", and prepending "control." to it. (This is typically the function that called as.control.list() in the first place.)

unflat Logical, indicating whether an attempt should be made to detect whether some of the arguments are appropriate for a lower-level control function and pass them down.

Value

a control.list object.

Methods (by class)

• as.control.list(control.list): Idempotent method for control lists.
• as.control.list(list): The method for plain lists, which runs them through FUN.

Examples

myfun <- function(..., control=control.myfun()){
  as.control.list(control)
}

control.myfun <- function(a=1, b=a+1){
  list(a=a,b=b)
}

myfun()
myfun(control = list(a=2))

myfun2 <- function(..., control=control.myfun2()){
  as.control.list(control)
}

control.myfun2 <- function(c=3, d=c+2, myfun=control.myfun()){
  list(c=c,d=d,myfun=myfun)
}

myfun2()
# Argument to control.myfun() (i.e., a) gets passed to it, and a
# warning is issued for unused argument e.
myfun2(control = list(c=3, a=2, e=3))
**attr**  
A wrapper for base::attr which defaults to exact matching.

**Description**  
A wrapper for base::attr which defaults to exact matching.

**Usage**  
attr(x, which, exact = TRUE)

**Arguments**  
x, which, exact as in base::attr, but with exact defaulting to TRUE in this implementation

**Value**  
as in base::attr

**Examples**  
x <- list()  
attr(x, "name") <- 10  
base::attr(x, "n")  
stopifnot(is.null(attr(x, "n")))  
base::attr(x, "n", exact = TRUE)

---

**check.control.class**  
Ensure that the class of the control list is one of those that can be used by the calling function

**Description**  
This function converts an ordinary list into a control list (if needed) and checks that the control list passed is appropriate for the function to be controlled.

**Usage**  
check.control.class(
  OKnames = as.character(ult(sys.calls(), 2)[[1L]]),
  myname = as.character(ult(sys.calls(), 2)[[1L]]),
  control = get("control", pos = parent.frame())
)
Arguments

OKnames: List of control function names which are acceptable.
mynames: Name of the calling function (used in the error message).
control: The control list or a list to be converted to a control list using control.mynames().
Defaults to the control variable in the calling function. See Details for detailed behavior.

Details

check.control.class() performs the check by looking up the class of the control argument (defaulting to the control variable in the calling function) and checking if it matches a list of acceptable given by OKnames.

Before performing any checks, the control argument (including the default) will be converted to a control list by calling as.control.list() on it with the first element of OKnames to construct the control function.

If control is missing, it will be assumed that the user wants to modify it in place, and a variable with that name in the parent environment will be overwritten.

Value

A valid control list for the function in which it is to be used. If control argument is missing, it will also overwrite the variable control in the calling environment with it.

Note

In earlier versions, OKnames and myname were autodetected. This capability has been deprecated and results in a warning issued once per session. They now need to be set explicitly.

See Also

set.control.class(), print.control.list(), as.control.list()
compress_rows.data.frame

Arguments

x a weighted matrix or data frame.
...
extra arguments for methods.

Value

For compress_rows A weighted matrix or data frame of the same type with duplicated rows removed and weights updated appropriately.

compress_rows.data.frame

"Compress" a data frame.

Description

compress_rows.data.frame "compresses" a data frame, returning unique rows and a tally of the number of times each row is repeated, as well as a permutation vector that can reconstruct the original data frame. decompress_rows.compressed_rows_df reconstructs the original data frame.

Usage

## S3 method for class 'data.frame'
compress_rows(x, ...)

## S3 method for class 'compressed_rows_df'
decompress_rows(x, ...)

Arguments

x For compress_rows.data.frame a data.frame to be compressed. For decompress_rows.compressed_rows_df a list as returned by compress_rows.data.frame.
...
Additional arguments, currently unused.

Value

For compress_rows.data.frame, a list with three elements:

rows Unique rows of x
frequencies A vector of the same length as the number or rows, giving the number of times the corresponding row is repeated
ordering A vector such that if c is the compressed data frame, c$rows[c$ordering,,drop=FALSE] equals the original data frame, except for row names
rownames Row names of x

For decompress_rows.compressed_rows_df, the original data frame.
See Also
data.frame

Examples

(x <- data.frame(V1=sample.int(3,30,replace=TRUE),
                 V2=sample.int(2,30,replace=TRUE),
                 V3=sample.int(4,30,replace=TRUE)))

(c <- compress_rows(x))

stopifnot(all(decompress_rows(c)==x))

---

cold.ist.accessor  Named element accessor for ergm control lists

Description

Utility method that overrides the standard `$` list accessor to disable partial matching for ergm control.list objects

Usage

## S3 method for class 'control.list'
object$name

Arguments

object  list-coarceable object with elements to be searched
name  literal character name of list element to search for and return

Details

Executes getElement instead of `$` so that element names must match exactly to be returned and partially matching names will not return the wrong object.

Value

Returns the named list element exactly matching name, or NULL if no matching elements found

Author(s)
Pavel N. Krivitsky

See Also

see getElement
control.remap

Overwrite control parameters of one configuration with another.

Description

Given a control.list, and two prefixes, from and to, overwrite the elements starting with to with the corresponding elements starting with from.

Usage

control.remap(control, from, to)

Arguments

control An object of class control.list.
from Prefix of the source of control parameters.
to Prefix of the destination of control parameters.

Value

An control.list object.

Author(s)

Pavel N. Krivitsky

See Also

print.control.list

Examples

(l <- set.control.class("test", list(a.x=1, a.y=2)))
control.remap(l, "a", "b")
default_options

Set options() according to a named list, skipping those already set.

Description

This function can be useful for setting default options, which do not override options set elsewhere.

Usage

default_options(...)

Arguments

... see options(): either a list of name=value pairs or a single unnamed argument giving a named list of options to set.

Value

The return value is same as that of options() (omitting options already set).

Examples

options(onesetting=1)

default_options(onesetting=2, anothersetting=3)
stopifnot(getOption("onesetting")==1) # Still 1.
stopifnot(getOption("anothersetting")==3)

default_options(list(yetanothersetting=5, anothersetting=4))
stopifnot(getOption("anothersetting")==3) # Still 3.
stopifnot(getOption("yetanothersetting")==5)

deInf

Truncate values of high magnitude in a vector.

Description

Truncate values of high magnitude in a vector.

Usage

deInf(x, replace = 1/.Machine$double.eps)

Arguments

x a numeric or integer vector.
replace a number or a string "maxint" or "intmax".
Value

Returns \( x \) with elements whose magnitudes exceed replace replaced replaced by \( \text{replace} \) (or its negation). If \( \text{replace} \) is "maxint" or "intmax", \( \text{.Machine}\$.integer.max \) is used instead.

NA and NAN values are preserved.

Description

\( \text{.Deprecate\_once} \) calls \( \text{.Deprecated()} \), passing all its arguments through, but only the first time it’s called.

\( \text{.Deprecate\_method} \) calls \( \text{.Deprecated()} \), but only if a method has been called by name, i.e., \( \text{METHOD} \). \( \text{CLASS} \). Like \( \text{.Deprecate\_once} \) it only issues a warning the first time.

Usage

\( \text{.Deprecate\_once}(...) \)

\( \text{.Deprecate\_method}(\text{generic}, \text{class}) \)

Arguments

\( \ldots \) arguments passed to \( \text{.Deprecated()} \).

\( \text{generic}, \text{class} \) strings giving the generic function name and class name of the function to be deprecated.

Examples

```r
## Not run:
options(warn=1) # Print warning immediately after the call.
f <- function(){
  \text{.Deprecate\_once}("new\_f")
}
f() # Deprecation warning
f() # No deprecation warning

## End(Not run)
## Not run:
options(warn=1) # Print warning immediately after the call.
summary.packageDescription <- function(object, ...){
  \text{.Deprecate\_method}("summary", "packageDescription")
  invisible(object)
}
summary(packageDescription("statnet.common")) # No warning.
summary.packageDescription(packageDescription("statnet.common")) # Warning.
```
summary.packageDescription(packageDescription("statnet.common")) # No warning.

## End(Not run)

despace A one-line function to strip whitespace from its argument.

Description

A one-line function to strip whitespace from its argument.

Usage

despace(s)

Arguments

s a character vector.

Examples

stopifnot(despace("\n \t ")=="")

diff.control.list Identify and the differences between two control lists.

Description

Identify and the differences between two control lists.

Usage

## S3 method for class 'control.list'
diff(x, y = eval(call(class(x)[[1L]])), ignore.environment = TRUE, ...)

## S3 method for class 'diff.control.list'
print(x, ..., indent = "")

Arguments

x a control.list

y a reference control.list; defaults to the default settings for x.

ignore.environment whether environment for environment-bearing parameters (such as formulas and functions) should be considered when comparing.

... Additional arguments to methods.

indent an argument for recursive calls, to facilitate indentation of nested lists.
empty_env

Value

An object of class diff.control.list: a named list with an element for each non-identical setting. The element is either itself a diff.control.list (if the setting is a control list) or a named list with elements x and y, containing x’s and y’s values of the parameter for that setting.

Methods (by generic)

• print(diff.control.list): A print method.

empty_env

Replace an object’s environment with a simple, static environment.

Description

Replace an object’s environment with a simple, static environment.

Usage

empty_env(object)
base_env(object)

Arguments

object An object with the environment()<- method.

Value

An object of the same type as object, with updated environment.

Examples

f <- y~x
environment(f) # GlobalEnv

environment(empty_env(f)) # EmptyEnv

environment(base_env(f)) # base package environment
ERRVL

Return the first argument passed (out of any number) that is not a try-error (result of try encountering an error.

Description

This function is inspired by NVL, and simply returns the first argument that is not a try-error, raising an error if all arguments are try-errors.

Usage

ERRVL(...)

Arguments

... Expressions to be tested; usually outputs of try.

Value

The first argument that is not a try-error. Stops with an error if all are.

Note

This function uses lazy evaluation, so, for example ERRVL(1, stop("Error!")) will never evaluate the stop call and will not produce an error, whereas ERRVL(try(solve(0)), stop("Error!")) would.

In addition, all expressions after the first may contain a , which is substituted with the try-error object returned by the previous expression.

See Also

try, inherits

Examples

print(ERRVL(1,2,3)) # 1
print(ERRVL(try(solve(0)),2,3)) # 2
print(ERRVL(1, stop("Error!"))) # No error

## Not run:
# Error:
print(ERRVL(try(solve(0), silent=TRUE),
      stop("Error!")))

# Error with an elaborate message:
print(ERRVL(try(solve(0), silent=TRUE),
           stop("Stopped with an error: ", .)))

## End(Not run)
fixed.pval

Format a p-value in fixed notation.

Description

This is a thin wrapper around `format.pval()` that guarantees fixed (not scientific) notation, links (by default) the eps argument to the digits argument and vice versa, and sets `nsmall` to equal digits.

Usage

```r
fixed.pval(
  pv,
  digits = max(1, getOption("digits") - 2),
  eps = 10^-digits,
  na.form = "NA",
  ...
)
```

Arguments

- `pv`, `digits`, `eps`, `na.form`, ...
  - `see format.pval()`.

Value

A character vector.

Examples

```r
pvs <- 10^((0:-12)/2)

# Jointly:
fpf <- fixed.pval(pvs, digits = 3)
fpf
format.pval(pvs, digits = 3) # compare

# Individually:
fpf <- sapply(pvs, fixed.pval, digits = 3)
fpf
sapply(pvs, format.pval, digits = 3) # compare

# Control eps:
fpf <- sapply(pvs, fixed.pval, eps = 1e-3)
fpf
```
Evaluate an \texttt{R} expression with a hard time limit by forking a process

**Description**

This function uses \texttt{parallel::mcparallel()}, so the time limit is not enforced on Windows. However, unlike functions using \texttt{setTimeLimit()}, the time limit is enforced even on native code.

**Usage**

```r
forkTimeout(
  expr,  
  timeout,  
  unsupported = c("warning", "error", "message", "silent"),  
  onTimeout = NULL  
)
```

**Arguments**

- **expr**: expression to be evaluated.
- **timeout**: number of seconds to wait for the expression to evaluate.
- **unsupported**: a character vector of length 1 specifying how to handle a platform that does not support \texttt{parallel::mcparallel()}.
  - "warning" or "message" Issue a warning or a message, respectively, then evaluate the expression without the time limit enforced.
  - "error" Stop with an error.
  - "silent" Evaluate the expression without the time limit enforced, without any notice.
  Partial matching is used.
- **onTimeout**: Value to be returned on time-out.

**Value**

Result of evaluating \texttt{expr} if completed, \texttt{onTimeout} otherwise.

**Note**

\texttt{onTimeout} can itself be an expression, so it is, for example, possible to stop with an error by passing \texttt{onTimeout=stop()}.

Note that this function is not completely transparent: side-effects may behave in unexpected ways. In particular, RNG state will not be updated.

**Examples**

```r
forkTimeout({Sys.sleep(1); TRUE}, 2) # TRUE
forkTimeout({Sys.sleep(1); TRUE}, 0.5) # NULL (except on Windows)
```
A suite of utilities for handling model formulas of the style used in Statnet packages.

Usage

append_rhs.formula(
  object = NULL,
  newterms,
  keep.onesided = FALSE,
  env = if (is.null(object)) NULL else environment(object)
)

append.rhs.formula(object, newterms, keep.onesided = FALSE)

filter_rhs.formula(object, newterms, keep.onesided = FALSE)

nonsimp_update.formula(object, new, ..., from.new = FALSE)

nonsimp.update.formula(object, new, ..., from.new = FALSE)

term.list.formula(rhs, sign = +1)

list_summands.call(object)

list_rhs.formula(object)

eval_lhs.formula(object)

Arguments

object formula object to be updated or evaluated
newterms a term_list object, or any list of terms (names or calls) to append to the formula, or a formula whose RHS terms will be used; its "sign" attribute vector can give the sign of each term (+1 or -1), and its "env" attribute vector will be used to set its environment, with the first available being used and subsequent ones producing a warning.
keep.onesided if the initial formula is one-sided, keep it whether to keep it one-sided or whether to make the initial formula the new LHS
env an environment for the new formula, if object is NULL
f a function whose first argument is the term and whose additional arguments are forwarded from ... that returns either TRUE or FALSE, for whether that term should be kept.
... Additional arguments. Currently unused.

new new formula to be used in updating

from.new logical or character vector of variable names. controls how environment of formula gets updated.

rhs, sign Arguments to the deprecated term.list.formula.

Value

append_rhs.formula each return an updated formula object; if object is NULL (the default), a one-sided formula containing only the terms in newterms will be returned.

nonsimp_update.formula each return an updated formula object

term_list.formula returns an object of type term_list; its "env" attribute is set to a list of NULLs, however.

list_rhs.formula returns an object of type term_list.

eval_lhs.formula an object of whatever type the LHS evaluates to.

Functions

- append_rhs.formula(): append_rhs.formula appends a list of terms to the RHS of a formula. If the formula is one-sided, the RHS becomes the LHS, if keep.onesided==FALSE (the default).
- append.rhs.formula(): append.rhs.formula has been renamed to append_rhs.formula.
- filter_rhs.formula(): filter_rhs.formula filters through the terms in the RHS of a formula, returning a formula without the terms for which function f(term, ...) is FALSE. Terms inside another term (e.g., parentheses or an operator other than + or -) will be unaffected.
- nonsimp_update.formula(): nonsimp_update.formula is a reimplementation of update.formula that does not simplify. Note that the resulting formula's environment is set as follows. If from.new==FALSE, it is set to that of object. Otherwise, a new sub-environment of object, containing, in addition, variables in new listed in from.new (if a character vector) or all of new (if TRUE).
- term.list.formula(): term.list.formula returns an object of type term_list that required the RHS call, rather than the formula itself.
- list_summands.call(): list_summands.call, given an unevaluated call or expression containing the sum of one or more terms, returns an object of class term_list with the terms being summed, handling + and - operators and parentheses, and keeping track of whether a term has a plus or a minus sign.
- list_rhs.formula(): list_rhs.formula returns an object of type term_list, containing terms in a given formula, handling + and - operators and parentheses, and keeping track of whether a term has a plus or a minus sign.
- eval_lhs.formula(): eval_lhs.formula extracts the LHS of a formula, evaluates it in the formula’s environment, and returns the result.
Examples

## append_rhs.formula

```r
(f1 <- append_rhs.formula(y~x,list(as.name("z1")),as.name("z2"))))
(f2 <- append_rhs.formula(~y,list(as.name("z"))))
(f3 <- append_rhs.formula(~y+x,structure(list(as.name("z")),sign=-1)))
(f4 <- append_rhs.formula(~y,list(as.name("z")),TRUE))
(f5 <- append_rhs.formula(y~x,~z1-z2))
(f6 <- append_rhs.formula(NULL,list(as.name("z"))))
(f7 <- append_rhs.formula(NULL,structure(list(as.name("z")),sign=-1)))
```

```r
te <- ~z2+z3
environment(te) <- new.env()
(f8 <- append_rhs.formula(NULL, te)) # OK
(f9 <- append_rhs.formula(y~x, te)) # Warning
(f10 <- append_rhs.formula(y~x, te, env=NULL)) # No warning, environment from te.
(f11 <- append_rhs.formula(te, ~z1)) # Warning, environment from te
```

## filter_rhs.formula

```r
(f1 <- filter_rhs.formula(~a-b+c, "a"))
(f2 <- filter_rhs.formula(~-a+b-c, "a"))
(f3 <- filter_rhs.formula(~a-b+c, "b")
(f4 <- filter_rhs.formula(~-a+b-c, "b")
(f5 <- filter_rhs.formula(~a-b+c, "c")
(f6 <- filter_rhs.formula(~-a+b-c, "c")
(f7 <- filter_rhs.formula(~c-a+b-c(a),
    function(x) (if(is.call(x)) x[[1]] else x)="c"))
```

```r
stopifnot(identical(list_rhs.formula(a~b),
    structure(list(b), sign=1, env=list(globalenv()), class="term_list")))
stopifnot(identical(list_rhs.formula(~b),
    structure(list(b), sign=1, env=list(globalenv()), class="term_list")))
stopifnot(identical(list_rhs.formula(~b+NULL),
    structure(list(b, NULL),
        sign=c(1,1), env=rep(list(globalenv()), 2), class="term_list")))
stopifnot(identical(list_rhs.formula(~-b+NULL),
    structure(list(b, NULL),
        sign=c(-1,1), env=rep(list(globalenv()), 2), class="term_list")))
stopifnot(identical(list_rhs.formula(~-b-NULL),
    structure(list(b, NULL),
        sign=c(-1,-1), env=rep(list(globalenv()), 2), class="term_list")))
stopifnot(identical(list_rhs.formula(~b-(NULL+c)),
    structure(list(b, NULL, c),
        sign=c(1,-1,-1), env=rep(list(globalenv()), 3), class="term_list")))
```

## eval_lhs.formula
(result <- eval_lhs.formula((2+2)-1))

stopifnot(identical(result,4))

---

**Description**

This function takes the arguments of its caller (whose name should be passed explicitly), plus any ... arguments and produces a control list based on the standard semantics of control.*() functions, including handling deprecated arguments, identifying undefined arguments, and handling arguments that should be passed through `match.arg()`.

**Usage**

`handle.controls(myname, ...)`

**Arguments**

- `myname` the name of the calling function.
- `...` the ... argument of the control function, if present.

**Details**

The function behaves based on the information it acquires from the calling function. Specifically,

- The values of formal arguments (except ..., if present) are taken from the environment of the calling function and stored in the list.
- If the calling function has a ... argument and defines an `old.controls` variable in its environment, then it remaps the names in ... to their new names based on `old.controls`. In addition, if the value is a list with two elements, action and message, the standard deprecation message will have message appended to it and then be called with `action()`.
- If the calling function has a `match.arg.pars` in its environment, the arguments in that list are processed through `match.arg()`.

**Value**

a list with formal arguments of the calling function.
**Description**

Test if the object is a matrix that is symmetric and positive definite

**Usage**

```r
is.SPD(x, tol = .Machine$double.eps)
```

**Arguments**

- `x` the object to be tested.
- `tol` the tolerance for the reciprocal condition number.

---

**locate_function**

Locate a function with a given name and return it and its environment.

**Description**

These functions first search the given environment, then search all loaded environments, including those where the function is not exported. If found, they return an unambiguous reference to the function.

**Usage**

```r
locate_function(name, env = globalenv(), ...)
locate_prefixed_function(
  name,
  prefix,
  errname,
  env = globalenv(),
  ...,
  call. = FALSE
)
```

**Arguments**

- `name` a character string giving the function’s name.
- `env` an environment where it should search first.
- `...` additional arguments to the warning and error warning messages. See Details.
- `prefix` a character string giving the prefix, so the searched-for function is `prefix.name`
errname: a character string; if given, if the function is not found an error is raised, with errname prepended to the error message.

call: a logical, whether the call (locate_prefixed_function) should be a part of the error message; defaults to FALSE (which is different from stop()’s default).

Details

If the initial search fails, a search using getAnywhere() is attempted, with exported ("visible") functions with the specified name preferred over those that are not. When multiple equally qualified functions are available, a warning is printed and an arbitrary one is returned.

Because getAnywhere() can be slow, past searches are cached.

Value

If the function is found, an unevaluated call of the form ENVNAME:::FUNNAME, which can then be used to call the function even if it is unexported. If the environment does not have a name, or is GlobalEnv, only FUNNAME is returned. Otherwise, NULL is returned.

Functions

• locate_function(): a low-level function returning the reference to the function named name, or NULL if not found.
• locate_prefixed_function(): a helper function that searches for a function of the form prefix.name and produces an informative error message if not found.

Examples

# Locate a random function in base.
locate_function(".row_names_info")

logspace.utils

Utilities for performing calculations on logarithmic scale.

Description

A small suite of functions to compute sums, means, and weighted means on logarithmic scale, minimizing loss of precision.

Usage

log_sum_exp(logx, use_ldouble = FALSE)

log_mean_exp(logx, use_ldouble = FALSE)

lweighted.mean(x, logw)
lweighted.var(x, logw)
lweighted.cov(x, y, logw)

Arguments

logx  Numeric vector of log(x), the natural logarithms of the values to be summed or averaged.
use_ldouble  Whether to use long double precision in the calculation. If TRUE, 's C built-in logspace_sum() is used. If FALSE, the package's own implementation based on it is used, using double precision, which is (on most systems) several times faster, at the cost of precision.
x, y  Numeric vectors or matrices of x and y, the (raw) values to be summed, averaged, or whose variances and covariances are to be calculated.
logw  Numeric vector of log(w), the natural logarithms of the weights.

Value

The functions return the equivalents of the R expressions given below, but faster and with less loss of precision.

Functions

• log_sum_exp(): log(sum(exp(logx)))
• log_mean_exp(): log(mean(exp(logx)))
• lweighted.mean(): weighted mean of x: sum(x*exp(logw))/sum(exp(logw)) for x scalar and colSums(x*exp(logw))/sum(exp(logw)) for x matrix
• lweighted.var(): weighted variance of x: crossprod(x-lweighted.mean(x,logw)*exp(logw/2))/sum(exp(logw))
• lweighted.cov(): weighted covariance between x and y: crossprod(x-lweighted.mean(x,logw)*exp(logw/2), y-lweighted.mean(y,logw)*exp(logw/2))/sum(exp(logw))

Author(s)

Pavel N. Krivitsky

Examples

x <- rnorm(1000)
stopifnot(all.equal(log_sum_exp(x), log(sum(exp(x))), check.attributes=FALSE))
stopifnot(all.equal(log_mean_exp(x), log(mean(exp(x))), check.attributes=FALSE))

logw <- rnorm(1000)
stopifnot(all.equal(m <- sum(x*exp(logw))/sum(exp(logw)), lweighted.mean(x, logw)))
stopifnot(all.equal(sum((x-m)^2*exp(logw))/sum(exp(logw)),
  lweighted.var(x, logw), check.attributes=FALSE))
x <- cbind(x, rnorm(1000))
stopifnot(all.equal(mx <- colSums(x*exp(logw))/sum(exp(logw)),
  lweighted.mean(x, logw), check.attributes=FALSE))
Utility operations for mcmc.list objects

Description

colMeans.mcmc.list is a "method" for (non-generic) colMeans applicable to mcmc.list objects.
sweep.mcmc.list is a "method" for (non-generic) sweep applicable to mcmc.list objects.
lapply.mcmc.list is a "method" for (non-generic) lapply applicable to mcmc.list objects.

Usage

colMeans.mcmc.list(x, ...)
sweep.mcmc.list(x, STATS, FUN = "-", check.margin = TRUE, ...)
lapply.mcmc.list(X, FUN, ...)

Arguments

x a mcmc.list object.
... additional arguments to colMeans or sweep.
STATS, FUN, check.margin
    See help for sweep.
X An mcmc.list object.

Value

colMeans.mcmc.list returns a vector with length equal to the number of mcmc chains in x with the mean value for each chain.
sweep.mcmc.list returns an appropriately modified version of x
lapply.mcmc.list returns an mcmc.list each of whose chains had been passed through FUN.
message_print

See Also

colMeans, mcmc.list
sweep
lapply

Examples

data(line, package="coda")
summary(line) # coda
colMeans.mcmc.list(line) # "Method"

data(line, package="coda")
colMeans.mcmc.list(line)[-1:3]
colMeans.mcmc.list(sweep.mcmc.list(line, 1:3))

data(line, package="coda")
colMeans.mcmc.list(line)[c(2,3,1)]
colMeans.mcmc.list(lapply.mcmc.list(line, `[^,],c(2,3,1)`))

message_print print objects to the message output.

Description

A thin wrapper around print that captures its output and prints it as a message, usually to STDERR.

Usage

message_print(..., messageArgs = NULL)

Arguments

... arguments to print.
messageArgs a list of arguments to be passed directly to message.

Examples

cat(1:5)

print(1:5)
message_print(1:5) # Looks the same (though may be in a different color on some frontends).

suppressMessages(print(1:5)) # Still prints
suppressMessages(message_print(1:5)) # Silenced
Convenience functions for handling NULL objects.

Description

Convenience functions for handling NULL objects.

Usage

NVL(...)
NVL2(test, notnull, null = NULL)
NVL3(test, notnull, null = NULL)
EVL(....)
EVL2(test, notnull, null = NULL)
EVL3(test, notnull, null = NULL)
NVL(x) <- value
EVL(x) <- value

Arguments

..., test expressions to be tested.
notnull expression to be returned if test is not NULL.
null expression to be returned if test is NULL.
x an object to be overwritten if NULL.
value new value for x.

Functions

• NVL(): Inspired by SQL function NVL, returns the first argument that is not NULL, or NULL if all arguments are NULL.
• NVL2(): Inspired by Oracle SQL function NVL2, returns the second argument if the first argument is not NULL and the third argument if the first argument is NULL. The third argument defaults to NULL, so NVL2(a, b) can serve as shorthand for (if(!is.null(a)) b).
• NVL3(): Inspired by Oracle SQL NVL2 function and magittr %>% operator, behaves as NVL2 but .s in the second argument are substituted with the first argument.
• EVL(): As NVL, but for any objects of length 0 (Empty) rather than just NULL. Note that if no non-zero-length arguments are given, NULL is returned.
• EVL2(): As NVL2, but for any objects of length 0 (Empty) rather than just NULL.
• **EVL3():** As NVL3, but for any objects of length 0 (Empty) rather than just NULL.

• **NVL(x) <- value:** Assigning to NVL overwrites its first argument if that argument is NULL. Note that it will always return the right-hand-side of the assignment (value), regardless of what x is.

• **EVL(x) <- value:** As assignment to NVL, but for any objects of length 0 (Empty) rather than just NULL.

**Note**
Whenever possible, these functions use lazy evaluation, so, for example `NVL(1, stop("Error!"))` will never evaluate the `stop` call and will not produce an error, whereas `NVL(NULL, stop("Error!"))` would.

**See Also**
`NULL`, `is.null`, `if`

**Examples**

```r
a <- NULL
da # NULL
NVL(a,0) # 0

b <- 1
b # 1
NVL(b,0) # 1

# Here, object x does not exist, but since b is not NULL, x is never evaluated, so the statement finishes.
NVL(b,x) # 1

# Also,
NVL(NULL,1,0) # 1
NVL(NULL,0,1) # 0
NVL(NULL,NULL,0) # 0
NVL(NULL,NULL,NULL) # NULL

NVL2(a, "not null!", "null!") # "null!"
NVL2(b, "not null!", "null!") # "not null!"

NVL3(a, "not null!", "null!") # "null!"
NVL3(b, .+1, "null!") # "null!"

NVL(NULL*2, 1) # numeric(0) is not NULL
EVL(NULL*2, 1) # 1

NVL(a) <- 2
a # 2
```
once

\[ \text{NVL}(b) \leftarrow 2 \]
\[ b \# \text{ still } 1 \]

---

**once**

*Evaluate a function once for a given input.*

---

**Description**

This is a `purrr`-style adverb that checks if a given function has already been called with a given configuration of arguments and skips it if it has.

**Usage**

\[
\text{once}(f, \text{expire}_\text{after} = \text{Inf}, \text{max}_\text{entries} = \text{Inf})
\]

**Arguments**

- **f**: A function to modify.
- **expire_after**: The number of seconds since it was added to the database before a particular configuration is "forgotten". This can be used to periodically remind the user without overwhelming them.
- **max_entries**: The number of distinct configurations to remember. If not \text{Inf}, *earliest-inserted* configurations will be removed from the database when capacity is exceeded. (This exact behavior may change in the future.)

**Details**

Each modified function instance returned by `once()` maintains a database of previous argument configurations. They are not in any way compressed, so this database may grow over time. Thus, this wrapper should be used with caution if arguments are large objects. This may be replaced with hashing in the future. In the meantime, you may want to set the `max_entries` argument to be safe.

Different instances of a modified function do not share databases, even if the function is the same. This means that if you, say, modify a function within another function, the modified function will call once per call to the outer function. Modified functions defined at package level count as the same "instance", however. See example.

**Note**

Because the function needs to test whether a particular configuration of arguments have already been used, do not rely on lazy evaluation behaviour.
Examples

```r
msg <- once(message)
msg("abc") # Prints.
msg("abc") # Silent.

msg <- once(message) # Starts over.
msg("abc") # Prints.

f <- function(){
  innermsg <- once(message)
  innermsg("efg") # Prints once per call to f().
  innermsg("efg") # Silent.
  msg("abcd") # Prints only the first time f() is called.
  msg("abcd") # Silent.
}
f() # Prints "efg" and "abcd".
f() # Prints only "efg".

msg3 <- once(message, max_entries=3)
msg3("a") # 1 remembered.
msg3("a") # Silent.
msg3("b") # 2 remembered.
msg3("a") # Silent.
msg3("c") # 3 remembered.
msg3("a") # Silent.
msg3("d") # "a" forgotten.
msg3("a") # Printed.

msg2s <- once(message, expire_after=2)
msg2s("abc") # Prints.
msg2s("abc") # Silent.
Sys.sleep(1)
msg2s("abc") # Silent after 1 sec.
Sys.sleep(1.1)
msg2s("abc") # Prints after 2.1 sec.
```

---

**opttest**

*Optionally test code depending on environment variable.*

---

**Description**

A convenience wrapper to run code based on whether an environment variable is defined.

**Usage**

```r
opttest(
  expr,
  testname = NULL,
)```
testvar = "ENABLE_statnet_TESTS",
yesvals = c("y", "yes", "t", "true", "1"),
lowercase = TRUE
)

Arguments

expr
An expression to be evaluated only if testvar is set to a non-empty value.
testname
Optional name of the test. If given, and the test is skipped, will print a message to that end, including the name of the test, and instructions on how to enable it.
testvar
Environment variable name. If set to one of the yesvals, expr is run. Otherwise, an optional message is printed.
yesvals
A character vector of strings considered affirmative values for testvar.
lowercase
Whether to convert the value of testvar to lower case before comparing it to yesvals.

Description

These function return a data frame sorted in lexicographic order or a permutation that will rearrange it into lexicographic order: first by the first column, ties broken by the second, remaining ties by the third, etc..

Usage

order(..., na.last = TRUE, decreasing = FALSE)

## Default S3 method:
order(..., na.last = TRUE, decreasing = FALSE)

## S3 method for class 'data.frame'
order(..., na.last = TRUE, decreasing = FALSE)

## S3 method for class 'matrix'
order(..., na.last = TRUE, decreasing = FALSE)

## S3 method for class 'data.frame'
sort(x, decreasing = FALSE, ...)

Implement the sort and order methods for data.frame and matrix, sorting it in lexicographic order.
Arguments

... Ignored for sort. For order, first argument is the data frame to be ordered. (This is needed for compatibility with order.)
na.last See order documentation.
decreasing Whether to sort in decreasing order.
x A data.frame to sort.

Value

For sort, a data frame, sorted lexicographically. For order, a permutation I (of a vector 1:nrow(x)) such that x[I,,drop=FALSE] equals x ordered lexicographically.

See Also
data.frame, sort, order, matrix

Examples

data(iris)
head(iris)
head(order(iris))
head(sort(iris))
stopifnot(identical(sort(iris),iris[order(iris),]))

paste.and Concatenates the elements of a vector (optionaly enclosing them in quotation marks or parentheses) adding appropriate punctuation and conjunctions.

Description

A vector x becomes "x[1]", "x[1] and x[2]", or "x[1], x[2], and x[3]", depending on the length of x.

Usage

paste.and(x, oq = "", cq = "", con = "and")

Arguments

x A vector.
oq Opening quotation symbol. (Defaults to none.)
cq Closing quotation symbol. (Defaults to none.)
con Conjunction to be used if length(x)>1. (Defaults to "and").
Value

A string with the output.

See Also

paste, cat

Examples

print(paste.(and(c())))
print(paste.(and(1)))
print(paste.(and(1:2)))
print(paste.(and(1:3)))
print(paste.(and(1:4,con='or')))
Arguments

expr  an expression to be retried; note the difference between `eval()` and `evalq()`.
retries  number of retries to make; defaults to "eval.retries" option, or 5.
beforeRetry  if given, an expression that will be evaluated before each retry if the initial attempt fails; it is evaluated in the same environment and with the same quoting semantics as `expr`, but its errors are not handled.
envir, enclos  see `eval()`.
verbose  Whether to output retries.

Value

Results of evaluating `expr`, including side-effects such as variable assignments, if successful in `retries` retries.

Note

If `expr` returns a "try-error" object (returned by `try()`), it will be treated as an error. This behavior may change in the future.

Examples

```r
x <- 0
persistEvalQ({if((x<-x+1)<3) stop("x < 3") else x},
beforeRetry = {cat("Will try incrementing...
")})

x <- 0
e <- quote(if((x<-x+1)<3) stop("x < 3") else x)
persistEval(e,
beforeRetry = quote(cat("Will try incrementing...
")))
```

print.control.list

Pretty print the control list

Description

This function prints the control list, including what it can control and the elements.

Usage

```r
## S3 method for class 'control.list'
print(x, ..., indent = "")
```

Arguments

x  A list generated by a `control.*` function.
...  Additional argument to print methods for individual settings.
indent  an argument for recursive calls, to facilitate indentation of nested lists.
set.control.class

Set the class of the control list

Description

This function sets the class of the control list, with the default being the name of the calling function.

Usage

```r
set.control.class(
  myname = as.character(ult(sys.calls(), 2)[[1L]]),
  control = get("control", pos = parent.frame())
)
```

Arguments

- `myname` Name of the class to set.
- `control` Control list. Defaults to the `control` variable in the calling function.

Value

The control list with class set.

Note

In earlier versions, `OKnames` and `myname` were autodetected. This capability has been deprecated and results in a warning issued once per session. They now need to be set explicitly.

See Also

`check.control.class()`, `print.control.list()`
simplify_simple

Convert a list to an atomic vector if it consists solely of atomic elements of length 1.

Description

This behaviour is not dissimilar to that of simplify2array(), but it offers more robust handling of empty or NULL elements and never promotes to a matrix or an array, making it suitable to be a column of a data.frame.

Usage

simplify_simple(
  x,
  toNA = c("null", "empty", "keep"),
  empty = c("keep", "unlist"),
  ...
)

Arguments

x an R list to be simplified.

toNA a character string indicating whether NULL entries (if "null") or 0-length entries including NULL (if "empty") should be replaced with NAs before attempting conversion; specifying keep or FALSE leaves them alone (typically preventing conversion).

empty a character string indicating how empty lists should be handled: either "keep", in which case they are unchanged or "unlist", in which cases they are unlisted (typically to NULL).

... additional arguments passed to unlist().

Value

an atomic vector or a list of the same length as x.

Examples

(x <- as.list(1:5))
stopifnot(identical(simplify_simple(x), 1:5))

x[3] <- list(NULL) # Put a NULL in place of 3.

x

x[3] <- list(NULL) # Put a NULL in place of 3.

x

stopifnot(identical(simplify_simple(x, FALSE), x)) # Can't be simplified without replacing the NULL.

stopifnot(identical(simplify_simple(x, c(1L,2L,NA,4L,5L))) # NULL replaced by NA and simplified.

x[[3]] <- integer(0)
x
stopifnot(identical(simplify_simple(x), x)) # A 0-length vector is not replaced by default,
stopifnot(identical(simplify_simple(x, "empty"), c(1L,2L,NA,4L,5L))) # but can be.

(x <- lapply(1:5, function(i) c(i,i+1L))) # Elements are vectors of equal length.
simplify2array(x) # simplify2array() creates a matrix,
stopifnot(identical(simplify_simple(x), x)) # but simplify_simple() returns a list.

---

**snctrl**  
*Statnet Control*

**Description**

A utility to facilitate argument completion of control lists.

**Usage**

`snctrl(...)`

**Arguments**

...  
The parameter list is updated dynamically as packages are loaded and unloaded.  
Their current list is given below.

**Details**

In and of itself, `snctrl` copies its named arguments into a list. However, its argument list is updated dynamically as packages are loaded, as are those of its reexports from other packages. This is done using an API provided by helper functions. (See API?snctrl.)

**Currently recognised control parameters**

This list is updated as packages are loaded and unloaded.

**Note**

You may see messages along the lines of

The following object is masked from 'package:PKG':  
`snctrl`

when loading packages. They are benign.
**snctrl_names**

**Helper functions used by packages to facilitate snctrl updating.**

---

**Description**

Helper functions used by packages to facilitate `snctrl` updating.

**Usage**

```r
snctrl_names()

update_snctrl(myname, arglists = NULL, callback = NULL)

collate_controls(x = NULL, ...)
```

**Arguments**

- `myname` Name of the package defining the arguments.
- `arglists` A named list of argument name-default pairs. If the list is not named, it is first passed through `collate_controls()`.
- `callback` A function with no arguments that updates the packages own copy of `snctrl()`.
- `x` Either a function, a list of functions, or an environment. If `x` is an environment, all functions starting with `dQuote(control.)` are obtained.
- `...` Additional functions or lists of functions.

**Format**

`UPDATE_MY_SCTRL_EXPR` is a quoted expression meant to be passed directly to `eval()`.

`COLLATE_ALL_MY_CONTROLS_EXPR` is a quoted expression meant to be passed directly to `eval()`.

**Value**

`update_snctrl()` has no return value and is used for its side-effects.

`collate_controls()` returns the combined list of name-default pairs of each function.

**Functions**

- `snctrl_names()`: Typeset the currently defined list of argument names by package and control function.
- `update_snctrl()`: Typically called from `.onLoad()`, Update the argument list of `snctrl()` to include additional argument names associated with the package, and set a callback for the package to update its own copy.
split.array

• `collate_controls()`: Obtain and concatenate the argument lists of specified functions or all functions starting with `dQuote(control.)` in the environment.

• `UPDATE_MY_SCTRL_EXPR`: A stored expression that, if evaluated, will create a callback function `update_my_snctrl()` that will update the client package’s copy of `snctrl()`.

• `COLLATE_ALL_MY_CONTROLS_EXPR`: A stored expression that, if evaluated on loading, will add arguments of the package’s `control.*()` functions to `snctrl()` and set the callback.

Examples

```r
## Not run:
# In the client package (outside any function):
eval(UPDATE_MY_SCTRL_EXPR)
## End(Not run)
## Not run:
# In the client package:
.onLoad <- function(libame, pkgname){
  # ... other code ...
  eval(statnet.common::COLLATE_ALL_MY_CONTROLS_EXPR)
  # ... other code ...
}
## End(Not run)
```

split.array

A `split()` method for `array` and `matrix` types on a margin.

Description

These methods split an `array` and `matrix` into a list of arrays or matrices with the same number of dimensions according to the specified margin.

Usage

```r
## S3 method for class 'array'
split(x, f, drop = FALSE, margin = NULL, ...)

## S3 method for class 'matrix'
split(x, f, drop = FALSE, margin = NULL, ...)
```

Arguments

- **x**
  A `matrix` or an `array`.

- **f, drop**
  See help for `split()`. Note that drop here is **not** for array dimensions: these are always preserved.

- **margin**
  Which margin of the array to split along. NULL splits as `split.default`, dropping dimensions.

- **...**
  Additional arguments to `split()`.
ssolve

Examples

```r
x <- diag(5)
f <- rep(1:2, c(2,3))
split(x, f, margin=1) # Split rows.
split(x, f, margin=2) # Split columns.

# This is similar to how data frames are split:
stopifnot(identical(split(x, f, margin=1),
  lapply(lapply(split(as.data.frame(x), f), as.matrix), unname)))
```

---

ssolve

Wrappers around matrix algebra functions that pre-scale their arguments

---

Description

Covariance matrices of variables with very different orders of magnitude can have very large ratios between their greatest and their least eigenvalues, causing them to appear to the algorithms to be near-singular when they are actually very much SPD. These functions first scale the matrix’s rows and/or columns by its diagonal elements and then undo the scaling on the result.

Usage

```r
ssolve(a, b, ..., snnd = TRUE)
sginv(X, ..., snnd = TRUE)
srcond(x, ..., snnd = TRUE)
snearPD(x, ...)
xTAx_ssolve(x, A, ...)
xTAx_qrssolve(x, A, tol = 1e-07, ...)
sandwich_ssolve(A, B, ...)
```

Arguments

- **snnd**: assume that the matrix is symmetric non-negative definite (SNND). If it’s "obvious" that it’s not (e.g., negative diagonal elements), an error is raised.
- **x, a, b, X, A, B, tol, ...**: corresponding arguments of the wrapped functions.
Details

ssolve(), sginv(), and snearPD() wrap solve(), MASS::ginv(), and Matrix::nearPD(), respectively. srcond() returns the reciprocal condition number of rcond() net of the above scaling. xTAx_ssolve, xTAx_qrssolve, and sandwich_ssolve wrap the corresponding statnet.common functions.

Examples

```r
x <- rnorm(2, sd=c(1,1e12))
x <- c(x, sum(x))
A <- matrix(c(1, 0, 1,
            0, 1e24, 1e24,
            1, 1e24, 1e24), 3, 3)
stopifnot(all.equal(
  xTAx_qrssolve(x,A),
  structure(drop(x%*%sginv(A)%*%x), rank = 2L, nullity = 1L))
)
x <- rnorm(2, sd=c(1,1e12))
x <- c(x, rnorm(1, sd=1e12))
A <- matrix(c(1, 0, 1,
            0, 1e24, 1e24,
            1, 1e24, 1e24), 3, 3)
stopifnot(try(xTAx_qrssolve(x,A), silent=TRUE) ==
  "Error in xTAx_qrssolve(x, A) : x is not in the span of A\n")
```

Description

These functions automate citation generation for Statnet Project packages. They no longer appear to work with CRAN and are thus deprecated.

Usage

```r
statnet.cite.head(pkg)
statnet.cite.foot(pkg)
statnet.cite.pkg(pkg)
```

Arguments

```r
pkg    Name of the package whose citation is being generated.
```
statnetStartupMessage

Value

For `statnet.cite.head` and `statnet.cite.foot`, an object of type `citationHeader` and `citationFooter`, respectively, understood by the `citation` function, with package name substituted into the template.

For `statnet.cite.pkg`, an object of class `bibentry` containing a 'software manual' citation for the package constructed from the current version and author information in the `DESCRIPTION` and a template.

See Also

`citation`, `citHeader`, `citFooter`, `bibentry`

Examples

```r
## Not run:
statnet.cite.head("statnet.common")
statnet.cite.pkg("statnet.common")
statnet.cite.foot("statnet.common")
## End(Not run)
```

---

**statnetStartupMessage**  
*Construct a "standard" startup message to be printed when the package is loaded.*

Description

This function uses information returned by `packageDescription()` to construct a standard package startup message according to the policy of the Statnet Project.

Usage

```r
statnetStartupMessage(pkgname, friends = c(), nofriends = c())
```

Arguments

- **pkgname**  
  Name of the package whose information is used.

- **friends**, **nofriends**  
  No longer used.

Value

A string containing the startup message, to be passed to the `packageStartupMessage()` call or `NULL`, if policy prescribes printing default startup message. (Thus, if `statnetStartupMessage()` returns `NULL`, the calling package should not call `packageStartupMessage()` at all.)
Note

Earlier versions of this function printed a more expansive message. This may change again as the Statnet Project policy evolves.

See Also

packageDescription(), packageStartupMessage()

Examples

## Not run:
.onAttach <- function(lib, pkg){
  sm <- statnetStartupMessage("ergm")
  if(!is.null(sm)) packageStartupMessage(sm)
}
## End(Not run)

sweep_cols.matrix  

Subtract a elements of a vector from respective columns of a matrix

Description

An optimized function equivalent to sweep(x, 2, STATS) for a matrix x.

Usage

sweep_cols.matrix(x, STATS, disable_checks = FALSE)

Arguments

x

a numeric matrix;

STATS

a numeric vector whose length equals to the number of columns of x.

disable_checks

if TRUE, do not check that x is a numeric matrix and its number of columns matches the length of STATS; set in production code for a significant speed-up.

Value

A matrix of the same attributes as x.

Examples

x <- matrix(runif(1000), ncol=4)
s <- 1:4
stopifnot(all.equal(sweep_cols.matrix(x, s), sweep(x, 2, s)))
A helper class for list of terms in an formula

Description

Typically generated by `list_rhs.formula()`, it contains, in addition to a list of `call()` or similar objects, attributes "sign" and "env", containing, respectively a vector of signs that the terms had in the original formula and a list of environments of the formula from which the term has been extracted. Indexing and concatenation methods preserve these.

Usage

term_list(x, sign = +1, env = NULL)

as.term_list(x, ...)

## S3 method for class 'term_list'
as.term_list(x, ...)

## Default S3 method:
as.term_list(x, sign = +1, env = NULL, ...)

## S3 method for class 'term_list'
c(x, ...)

## S3 method for class 'term_list'
x[i, ...]

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

Arguments

x a list of terms or a term; a term_list

sign a vector specifying the signs associated with each term (-1 and +1)

env a list specifying the environments, or NULL

... additional arguments to methods

i list index

See Also

`list_rhs.formula()`, `list_summands.call()`
Examples

e1 <- new.env()
f1 <- a-b+c
environment(f1) <- e1
f2 <- ~-NULL+1

(l1 <- list_rhs.formula(f1))
(l2 <- list_rhs.formula(f2))

(l <- c(l1,l2))

(l <- c(l2[1], l1[2], l1[1], l1[1], l2[2]))

trim_env

Make a copy of an environment with just the selected objects.

Description

Make a copy of an environment with just the selected objects.

Usage

trim_env(object, keep = NULL, ...)

## S3 method for class 'environment'
trim_env(object, keep = NULL, ...)

## Default S3 method:
trim_env(object, keep = NULL, ...)

Arguments

object An environment or an object with environment() and environment()<- methods.

keep A character vector giving names of variables in the environment (including its ancestors) to copy over, defaulting to dropping all. Variables that cannot be resolved are silently ignored.

... Additional arguments, passed on to lower-level methods.

Value

An object of the same type as object, with updated environment.
Methods (by class)

- `trim_env(environment)`: A method for environment objects.
- `trim_env(default)`: Default method, for objects such as `formula` and `function` that have `environment()` and `environment()<-` methods.

---

`ult`  
*Extract or replace the ultimate (last) element of a vector or a list, or an element counting from the end.*

**Description**

Extract or replace the ultimate (last) element of a vector or a list, or an element counting from the end.

**Usage**

\[ \text{ult}(x, i = 1L) \]

\[ \text{ult}(x, i = 1L) <- \text{value} \]

**Arguments**

- **x**: a vector or a list.
- **i**: index from the end of the list to extract or replace (where 1 is the last element, 2 is the penultimate element, etc.).
- **value**: Replacement value for the ith element from the end.

**Value**

An element of x.

**Note**

Due to the way in which assigning to a function is implemented in R, `ult(x) <- e` may be less efficient than `x[[\text{length}(x)]] <- e`.

**Examples**

```r
x <- 1:5
(last <- ult(x))
(penultimate <- ult(x, 2)) # 2nd last.

(ult(x) <- 6)
(ult(x, 2) <- 7) # 2nd last.

x
```
An error handler for `rlang::check_dots_used()` that issues a warning that only lists argument names.

**Description**

This handler parses the error message produced by `rlang::check_dots_used()`, extracting the names of the unused arguments, and formats them into a more gentle warning message. It relies on `rlang` maintaining its current format.

**Usage**

```r
unused_dots_warning(e)
```

**Arguments**

- `e` a `condition` object, typically not passed by the end-user; see example below.

**Examples**

```r
g <- function(b=NULL, ...){
  invisible(force(b))
}
f <- function(...){
  rlang::check_dots_used(error = unused_dots_warning)
  g(....)
}

f() # OK
f(b=2) # OK
f(a=1, b=2, c=3) # Warning about a and c but not about b
```
unwhich

Construct a logical vector with TRUE in specified positions.

Description
This function is basically an inverse of which.

Usage
unwhich(which, n)

Arguments
which a numeric vector of indices to set to TRUE.
n total length of the output vector.

Value
A logical vector of length n whose elements listed in which are set to TRUE, and whose other elements are set to FALSE.

Examples
x <- as.logical(rbinom(10,1,0.5))
stopifnot(all(x == unwhich(which(x), 10)))

vector.namesmatch reorder vector v into order determined by matching the names of its elements to a vector of names

Description
A helper function to reorder vector v (if named) into order specified by matching its names to the argument names

Usage
vector.namesmatch(v, names, errname = NULL)

Arguments
v a vector (or list) with named elements, to be reordered
names a character vector of element names, corresponding to names of v, specifying desired ordering of v
errename optional, name to be reported in any error messages. default to deparse(substitute(v))
Details

does some checking of appropriateness of arguments, and reorders v by matching its names to character vector names

Value

returns v, with elements reordered

Note

earlier versions of this function did not order as advertised

Examples

test<-list(c=1,b=2,a=3)
vector.namesmatch(test,names=c('a','c','b'))

Description

A simple class for keeping track of the running mean and the sum of squared deviations from the mean for a vector.

Usage

Welford(dn, means, vars)

## S3 method for class 'Welford'
update(object, newdata, ...)

Arguments

dn, means, vars initialization of the Welford object: if means and vars are given, they are treated as the running means and variances, and dn is their associated sample size, and if not, dn is the dimension of the vector (with sample size 0).
object a Welford object.
newdata either a numeric vector of length d, a numeric matrix with d columns for a group update, or another Welford object with the same d.
... additional arguments to methods.
Value

an object of type `Welford`: a list with four elements:

1. n: Running number of observations
2. means: Running mean for each variable
3. SSDs: Running sum of squared deviations from the mean for each variable
4. vars: Running variance of each variable

Methods (by generic)

- `update(Welford)`: Update a Welford object with new data.

Examples

```r
X <- matrix(rnorm(200), 20, 10)
w0 <- Welford(10)

w <- update(w0, X)
stopifnot(isTRUE(all.equal(w$means, colMeans(X))))
stopifnot(isTRUE(all.equal(w$vars, apply(X,2,var))))

w <- update(w0, X[1:12,])
w <- update(w, X[13:20,])
stopifnot(isTRUE(all.equal(w$means, colMeans(X))))
stopifnot(isTRUE(all.equal(w$vars, apply(X,2,var))))

w <- update(w0, X[1:12,])
w <- update(w, X[13:20,])
stopifnot(isTRUE(all.equal(w$means, colMeans(X))))
stopifnot(isTRUE(all.equal(w$vars, apply(X,2,var))))
```

wmatrix

A data matrix with row weights

Description

A representation of a numeric matrix with row weights, represented on either linear (`linwmatrix`) or logarithmic (`logwmatrix`) scale.

Usage

```r
logwmatrix(
  data = NA,
  nrow = 1,
  ncol = 1,
  byrow = FALSE,
  dimnames = NULL,
)```
```r
w = NULL

linwmatrix(
  data = NA,
  nrow = 1,
  ncol = 1,
  byrow = FALSE,
  dimnames = NULL,
  w = NULL
)

is.wmatrix(x)

is.logwmatrix(x)

is.linwmatrix(x)

as.linwmatrix(x, ...)

as.logwmatrix(x, ...)

## S3 method for class 'linwmatrix'
as.linwmatrix(x, ...)

## S3 method for class 'logwmatrix'
as.logwmatrix(x, ...)

## S3 method for class 'linwmatrix'
as.logwmatrix(x, ...)

## S3 method for class 'matrix'
as.linwmatrix(x, w = NULL, ...)

## S3 method for class 'matrix'
as.logwmatrix(x, w = NULL, ...)

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

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

## S3 method for class 'linwmatrix'
print(x, ...)
```
## S3 method for class 'logwmatrix'
compress_rows(x, ...)

## S3 method for class 'linwmatrix'
compress_rows(x, ...)

## S3 method for class 'wmatrix'
decompress_rows(x, target.nrows = NULL, ...)

## S3 method for class 'wmatrix'
x[i, j, ..., drop = FALSE]

## S3 replacement method for class 'wmatrix'
x[i, j, ...] <- value

### Arguments
- **data, nrow, ncol, byrow, dimnames** passed to `matrix`.
- **w** row weights on the appropriate scale.
- **x** an object to be coerced or tested.
- **...** extra arguments, currently unused.
- **target.nrows** the approximate number of rows the uncompressed matrix should have; if not achievable exactly while respecting proportionality, a matrix with a slightly different number of rows will be constructed.
- **i, j, value** rows and columns and values for extraction or replacement; as `matrix`.
- **drop** Used for consistency with the generic. Ignored, and always treated as `FALSE`.

### Value
An object of class `linwmatrix/logwmatrix` and `wmatrix`, which is a `matrix` but also has an attribute `w` containing row weights on the linear or the natural-log-transformed scale.

### Note
Note that `wmatrix` itself is an "abstract" class: you cannot instantiate it.

Note that at this time, `wmatrix` is designed as, first and foremost, as class for storing compressed data matrices, so most methods that operate on matrices may not handle the weights correctly and may even cause them to be lost.

### See Also
- `rowweights`, `lrowweights`, `compress_rows`
Examples

```r
(m <- matrix(1:3, 2, 3, byrow=TRUE))
(m <- rbind(m, 3*m, 2*m, m))
(mlog <- as.logwmatrix(m))
(mlin <- as.linwmatrix(m))
(cmlog <- compress_rows(mlog))
(cmlin <- compress_rows(mlin))

stopifnot(all.equal(as.linwmatrix(cmlog), cmlin))

cmlog[2,] <- 1:3
(cmlog <- compress_rows(cmlog))
stopifnot(sum(rowweights(cmlog)) == nrow(m))

(m3 <- matrix(c(1:3, (1:3)*2, (1:3)*3), 3, 3, byrow=TRUE))
(rowweights(m3) <- c(4, 2, 2))

stopifnot(all.equal(compress_rows(as.logwmatrix(m)), as.logwmatrix(m3), check.attributes=FALSE))
stopifnot(all.equal(rowweights(compress_rows(as.logwmatrix(m))), rowweights(as.logwmatrix(m3)), check.attributes=FALSE))
```

---

**wmatrix_weights**

*Set or extract weighted matrix row weights*

**Description**

Set or extract weighted matrix row weights

**Usage**

```r
rowweights(x, ...)

# S3 method for class 'linwmatrix'
rowweights(x, ...)

# S3 method for class 'logwmatrix'
rowweights(x, ...)

lrowweights(x, ...)

# S3 method for class 'logwmatrix'
lrowweights(x, ...)

# S3 method for class 'linwmatrix'
lrowweights(x, ...)

rowweights(x, ...) <- value
```
### Arguments

- **x**: a `linwmatrix`, a `logwmatrix`, or a `matrix`; a `matrix` is coerced to a weighted matrix of an appropriate type.
- **...**: extra arguments for methods.
- **value**: weights to set, on the appropriate scale.
- **update**: if `TRUE` (the default), the old weights are updated with the new weights (i.e., corresponding weights are multiplied on linear scale or added on log scale); otherwise, they are overwritten.

### Value

For the accessor functions, the row weights or the row log-weights; otherwise, a weighted matrix with modified weights. The type of weight (linear or logarithmic) is converted to the required type and the type of weighting of the matrix is preserved.

---

**Common quadratic forms**

**Description**

Common quadratic forms
Usage

\( \text{xT}A\text{x}(x, A) \)

\( \text{xAxT}(x, A) \)

\( \text{xT}A\text{x}\_\text{solve}(x, A, \ldots) \)

\( \text{xT}A\text{x}\_\text{qr}\text{solve}(x, A, \text{tol} = 1e\text{-}07, \ldots) \)

\( \text{sandwich}\_\text{solve}(A, B, \ldots) \)

Arguments

- \( x \): a vector
- \( A \): a square matrix
- \( \ldots \): additional arguments to subroutines
- \( \text{tol} \): tolerance argument passed to the relevant subroutine
- \( B \): a square matrix

Details

These are somewhat inspired by emulator::quad.form.inv() and others.

Functions

- \( \text{xT}A\text{x}() \): Evaluate \( x'Ax \) for vector \( x \) and square matrix \( A \).
- \( \text{xAxT}() \): Evaluate \( xAx' \) for vector \( x \) and square matrix \( A \).
- \( \text{xT}A\text{x}\_\text{solve}() \): Evaluate \( x'A^{-1}x \) for vector \( x \) and invertible matrix \( A \) using \text{solve}().
- \( \text{xT}A\text{x}\_\text{qr}\text{solve}() \): Evaluate \( x'A^{-1}x \) for vector \( x \) and matrix \( A \) using QR decomposition and confirming that \( x \) is in the span of \( A \) if \( A \) is singular; returns \text{rank} and \text{nullity} as attributes just in case subsequent calculations (e.g., hypothesis test degrees of freedom) are affected.
- \( \text{sandwich}\_\text{solve}() \): Evaluate \( A^{-1}B(A')^{-1} \) for \( B \) a square matrix and \( A \) invertible.
Index

* arith
  logspace.utils, 22
* datasets
  snctrl_names, 37
* debugging
  opttest, 29
* environment
  opttest, 29
* manip
  compress_rows.data.frame, 7
  order, 30
* utilities
  check.control.class, 5
  control.remap, 9
  ERRVL, 14
  NVL, 26
  opttest, 29
  paste.and, 31
  print.control.list, 33
  set.control.class, 34
  statnet.cite, 40
  statnetStartupMessage, 41
  .Deprecate_method
    (deprecation-utilities), 11
  .Deprecate_once
    (deprecation-utilities), 11
  .Deprecated(), 11
  .onLoad(), 37
  [.term_list(term_list), 43
  [.wmatrix (wmatrix), 49
  [<=-wmatrix (wmatrix), 49
  $, 8
  $.control.list (control.list.accessor), 8
  all_identical, 3
  append.rhs.formula (formula.utilities), 17
  append_rhs.formula (formula.utilities), 17
  array, 38
  as.control.list, 3
  as.control.list(), 6
  as.linwmatrix (wmatrix), 49
  as.logwmatrix (wmatrix), 49
  as.term_list (term_list), 43
  attr, 5

  base_env (empty_env), 13
  bibentry, 41
  c.term_list (term_list), 43
  call(), 43
  check.control.class, 5, 34
  check.control.class(), 34
  citation, 41
  COLLATE_ALL_MY_CONTROLS_EXPR
    (snctrl_names), 37
  collate_controls (snctrl_names), 37
  collate_controls(), 37
  colMeans, 24, 25
  colMeans.mcmc.list (mcmc-utilities), 24
  compress_rows, 6, 51
  compress_rows.data.frame, 7
  compress_rows.linwmatrix (wmatrix), 49
  compress_rows.logwmatrix (wmatrix), 49
  condition, 46
  control.list.accessor, 8
  control.remap, 9

  data.frame, 7, 8, 30, 31, 35
  decompress_rows (compress_rows), 6
  decompress_rows.compressed_rows_df
    (compress_rows.data.frame), 7
  decompress_rows.wmatrix (wmatrix), 49
  default_options, 10
  deInf, 10
  despace-utilities, 11
  deprecation-utilities, 11
  despace, 12
  diff.control.list, 12
empty_env, 13
environment, 21, 44
environment(), 44, 45
ERRVL, 14
eval(), 32, 33, 37
eval_lhs.formula(formula.utilities), 17
evalq(), 32, 33
EVL (NVL), 26
EVL2 (NVL), 26
EVL3 (NVL), 26
EVL<- (NVL), 26
filter_rhs.formula(formula.utilities), 17
fixed.pval, 15
forkTimeout, 16
format.pval(), 15
formula, 45
formula.utilities, 17
function, 45
getAnywhere(), 22
ggetElement, 8
handle.controls, 20
identical, 3
if, 27
inherits, 14
is.linwmatrix (wmatrix), 49
is.logwmatrix (wmatrix), 49
is.null, 27
is.SPD, 21
is.wmatrix (wmatrix), 49
lapply, 24, 25
lapply.mcmc.list (mcmc-utilities), 24
linwmatrix, 53
linwmatrix (wmatrix), 49
list, 4, 7, 35
list_rhs.formula(formula.utilities), 17
list_rhs.formula(), 43
list_summands.call (formula.utilities), 17
list_summands.call(), 43
locate_function, 21
locate_prefixed_function
  (locate_function), 21
log_mean_exp (logspace.utils), 22
log_sum_exp (logspace.utils), 22
logspace.utils, 22
logwmatrix, 53
logwmatrix (wmatrix), 49
lrowweights, 51
lrowweights (wmatrix_weights), 52
lrowweights<- (wmatrix_weights), 52
lweighted.cov (logspace.utils), 22
lweighted.mean (logspace.utils), 22
lweighted.var (logspace.utils), 22
MASS::ginv(), 40
match.arg(), 40
matrix, 30, 31, 38, 51, 53
Matrix::nearPD(), 40
mcmc-utilities, 24
mcmc.list, 24, 25
message, 25
message_print, 25
nonsimp.update.formula
  (formula.utilities), 17
nonsimp_update.formula
  (formula.utilities), 17
NULL, 26, 27
NVL, 14, 26
NVL2 (NVL), 26
NVL3 (NVL), 26
NVL<- (NVL), 26
once, 28
options(), 10
opttest, 29
order, 30, 30, 31
packageDescription(), 41, 42
packageStartupMessage(), 41, 42
parallel::mcparallel(), 16
paste.and, 31
persistEval, 32
persistEvalQ (persistEval), 32
print, 25
print.control.list, 9, 33
print.control.list(), 6, 34
print.diff.control.list
  (diff.control.list), 12
print.linwmatrix (wmatrix), 49
print.logwmatrix (wmatrix), 49
print.term_list (term_list), 43
INDEX

print.wmatrix (wmatrix), 49
rcond(), 40
rlang::check_dots_used(), 46
rowweights, 51
rowweights (wmatrix_weights), 52
rowweights<-(wmatrix_weights), 52
sandwich_solve (xTAx), 53
sandwich_ssolve (ssolve), 39
set.control.class, 34, 34
set.control.class(), 6
setTimeLimit(), 16
sginv (ssolve), 39
simplify2array(), 35
simplify_simple, 35
snctrl, 36, 37
snctrl(), 37, 38
snctrl-API (snctrl_names), 37
snctrl_names, 37
sneapD (ssolve), 39
solve(), 40, 54
sort, 30, 31
sort.data.frame (order), 30
split(), 38
split.array, 38
split.default, 38
split.matrix (split.array), 38
srcond (ssolve), 39
ssolve, 39
statnet.cite, 40
statnetStartupMessage, 41
statnetStartupMessage(), 41
stop, 14, 27
stop(), 22
sweep, 24, 25
sweep.mcmc.list (mcmc-utilities), 24
sweep.cols.matrix, 42
term.list.formula (formula.utilities), 17
term_list, 17, 18, 43
trim_env, 44
try, 14
try(), 33
ult, 45
ult<-(ult), 45
unlist(), 35
unused_dots_warning, 46
unwhich, 47
update.formula, 18
update.Welford (Welford), 48
UPDATE_MY_SCTRL_EXPR (snctrl_names), 37
update_snctrl (snctrl_names), 37
vector.namesmatch, 47
Welford, 48
which, 47
wmatrix, 49
wmatrix_weights, 52
xAXT (xTAx), 53
xTAx, 53
xTAx_qrsolve (xTAx), 53
xTAx_qrssolve (ssolve), 39
xTAx_solve (xTAx), 53
xTAx_ssolve (ssolve), 39