Package ‘rje’

June 10, 2020

Type Package

Title Miscellaneous Useful Functions for Statistics

Version 1.10.16

Description A series of functions in some way considered useful to the author. These include functions for subsetting tables and generating indices for arrays, conditioning and intervening in probability distributions, generating combinations and more...

Depends R (>= 2.0.0),

License GPL (>= 2)

LazyLoad yes

Imports knitr

Suggests testthat, rmarkdown

VignetteBuilder knitr

Author Robin Evans [aut, cre],
Mathias Drton [ctb]

Maintainer Robin Evans <evans@stats.ox.ac.uk>

RoxygenNote 7.0.2

NeedsCompilation yes

Repository CRAN

Date/Publication 2020-06-10 17:30:02 UTC

R topics documented:

and0 2
armijo 3
combinations 5
conditionMatrix 6
cubeHelix 8
designMatrix 10
Dirichlet 11
expit 12
fastHadamard 13

fsapply 14
greaterThan 15
indexBox 16
interventionMatrix 17
is.subset 19
is.wholenumber 20
last 20
marginTable 21
patternRepeat 22
powerSet 24
printPercentage 25
quickSort 26
rje 27
rowMins 28
rprobdist 29
schur 30
setmatch 30
subsetMatrix 32
subsetOrder 33
subtable 34

Index **36**

`and0` *Fast pairwise logical operators*

Description

Fast but loose implementations of AND and OR logical operators.

Usage

```
and0(x, y)
```

Arguments

`x, y`	logical or numerical vectors

Details

Returns pairwise application of logical operators AND and OR. Vectors are recycled as usual.

Value

A logical vector of length `max(length(x),length(y))` with entries `x[1] & x[2]` etc.; each entry of `x` or `y` is `TRUE` if it is non-zero.

Note

These functions should only be used with well understood vectors, and may not deal with unusual cases correctly.

Examples

```
and0(c(0,1,0), c(1,1,0))
## Not run:
set.seed(1234)
x = rbinom(5000, 1, 0.5)
y = rbinom(5000, 1, 0.5)

# 3 to 4 times improvement over `&`
system.time(for (i in 1:5000) and0(x,y))
system.time(for (i in 1:5000) x & y)

## End(Not run)
```

---

armijo *Generic functions to aid finding local minima given search direction*

Description

Allows use of an Armijo rule or coarse line search as part of minimisation (or maximisation) of a differentiable function of multiple arguments (via gradient descent or similar). Repeated application of one of these rules should (hopefully) lead to a local minimum.

Usage

```
armijo(
  fun,
  x,
  dx,
  beta = 3,
  sigma = 0.5,
  grad,
  maximise = FALSE,
  searchup = TRUE,
  adj.start = 1,
  ...
)

coarseLine(fun, x, dx, beta = 3, maximise = FALSE, ...)
```

Arguments

`fun`	a function whose first argument is a numeric vector
`x`	a starting value to be passed to `fun`
`dx`	numeric vector containing feasible direction for search; defaults to `-grad` for ordinary gradient descent
`beta`	numeric value (greater than 1) giving factor by which to adjust step size
`sigma`	numeric value (less than 1) giving steepness criterion for move
`grad`	numeric gradient of `f` at `x` (will be estimated if not provided)
`maximise`	logical: if set to `TRUE` search is for a maximum rather than a minimum.
`searchup`	logical: if set to `TRUE` method will try to find largest move satisfying Armijo criterion, rather than just accepting the first it sees
`adj.start`	an initial adjustment factor for the step size.
`...`	other arguments to be passed to `fun`

Details

`coarseLine` performs a stepwise search and tries to find the integer k minimising $f(x_k)$ where

$$x_k = x + \beta^k dx.$$

Note k may be negative. This is genearlly quicker and dirtier than the Armijo rule.

`armijo` implements an Armijo rule for moving, which is to say that

$$f(x_k) - f(x) < -\sigma\beta^k dx \cdot \nabla_x f.$$

This has better convergence guarantees than a simple line search, but may be slower in practice. See Bertsekas (1999) for theory underlying the Armijo rule.

Each of these rules should be applied repeatedly to achieve convergence (see example below).

Value

A list comprising

`best`	the value of the function at the final point of evaluation
`adj`	the constant in the step, i.e. β^n
`move`	the final move; i.e. $\beta^n dx$
`code`	an integer indicating the result of the function; 0 = returned OK, 1 = very small move suggested, may be at minimum already, 2 = failed to find minimum: function evaluated to `NA` or was always larger than $f(x)$ (direction might be infeasible), 3 = failed to find minimum: stepsize became too small or large without satisfying rule.

Functions

- `coarseLine`: Coarse line search

Author(s)

Robin Evans

References

Bertsekas, D.P. *Nonlinear programming*, 2nd Edition. Athena, 1999.

Examples

```
# minimisation of simple function of three variables
x = c(0,-1,4)
f = function(x) ((x[1]-3)^2 + sin(x[2])^2 + exp(x[3]) - x[3])

tol = .Machine$double.eps
mv = 1

while (mv > tol) {
  # or replace with coarseLine()
  out = armijo(f, x, sigma=0.1)
  x = out$x
  mv = sum(out$move^2)
}

# correct solution is c(3,0,0) (or c(3,k*pi,0) for any integer k)
x
```

---

combinations *Combinations of Integers*

Description

Returns a matrix containing each possible combination of one entry from vectors of the lengths provided.

Usage

```
combinations(p)
powerSetMat(n)
```

Arguments

p	vector of non-negative integers.
n	non-negative integer.

Details

Returns a matrix, each row being one possible combination of integers from the vectors $(0, 1, \ldots, p_i - 1)$, for i between 1 and `length(p)`.

Based on `bincombinations` from package `e1071`, which provides the binary case.

`powerSetMat` is just a wrapper for `combinations(rep(2,n))`.

Value

A matrix with number of columns equal to the length of `p`, and number of rows equal to $p_1 \times \cdots \times p_k$, each row corresponding to a different combination. Ordering is reverse-lexographic.

Author(s)

Robin Evans

Examples

```
combinations(c(2,3,3))

powerSetMat(3)
```

`conditionMatrix` *Find conditional probability table*

Description

Given a numeric array or matrix (of probabilities), calculates margins of some dimensions conditional on particular values of others.

Usage

```
conditionMatrix(
  x,
  variables,
  condition = NULL,
  condition.value = NULL,
  dim = NULL,
  incols = FALSE,
  undef = NaN
)

conditionTable(
  x,
  variables,
  condition = NULL,
```

```
  condition.value = NULL,
  undef = NaN,
  order = TRUE
)

conditionTable2(x, variables, condition, undef = NaN)
```

Arguments

x	A numeric array.
variables	An integer vector containing the margins of interest from x.
condition	An integer vector containing the dimensions of x to condition on.
condition.value	An integer vector or list of the same length as condition, containing the values to condition with. If NULL, then the full conditional distribution is returned.
dim	Integer vector containing dimensions of variables. Assumed all binary if not specified.
incols	Logical specifying whether not the distributions are stored as the columns in the matrix; assumed to be rows by default.
undef	if conditional probability is undefined, what should the value be given as
order	logical - if TRUE conditioned variables come last, if FALSE variables are in original order.

Details

conditionTable calculates the marginal distribution over the dimensions in variables for each specified value of the dimensions in condition. Single or multiple values of each dimension in condition may be specified in condition.value; in the case of multiple values, condition.value must be a list.

The sum over the dimensions in variables is normalized to 1 for each value of condition.

conditionTable2 is just a wrapper which returns the conditional distribution as an array of the same dimensions and ordering as the original x. Values are repeated as necessary.

conditionMatrix takes a matrix whose rows (or columns if incols = TRUE) each represent a separate multivariate probability distribution and finds the relevant conditional distribution in each case. These are then returned in the same format. The order of the variables under conditionMatrix is always as in the original distribution, unlike for conditionTable above.

The probabilities are assumed in reverse lexicographic order, as in a flattened R array: i.e. the first value changes fastest: (1,1,1), (2,1,1), (1,2,1), ..., (2,2,2).

condition.table and condition.table2 are identical to conditionTable and conditionTable2.

Value

conditionTable returns an array whose first length(variables) corresponds to the dimensions in variables, and the remainder (if any) to dimensions in condition with a corresponding entry in condition.value of length > 1.

conditionTable2 always returns an array of the same dimensions as x, with the variables in the same order.

Functions

- `conditionMatrix`: Conditioning in matrix of distributions
- `conditionTable2`: Conditioning whilst preserving all dimensions

Author(s)

Mathias Drton, Robin Evans

See Also

marginTable, margin.table, interventionTable

Examples

```
x = array(1:16, rep(2,4))
x = x/sum(x) # probability distribution on 4 binary variables x1, x2, x3, x4.

# distribution of x2, x3 given x1 = 1 and x4=2.
conditionTable(x, c(2,3), c(1,4), c(1,2))
# x2, x3 given x1 = 1,2 and x4 = 2.
conditionTable(x, c(2,3), c(1,4), list(1:2,2))

# complete conditional of x2, x3 given x1, x4
conditionTable(x, c(2,3), c(1,4))

# conditionTable2 leaves dimensions unchanged
tmp = conditionTable2(x, c(2,3), c(1,4))
aperm(tmp, c(2,3,1,4))

####
set.seed(2314)
# set of 10 2x2x2 probability distributions
x = rdirichlet(10, rep(1,8))

conditionMatrix(x, 3, 1)
conditionMatrix(x, 3, 1, 2)
```

`cubeHelix` *Cube Helix colour palette*

Description

Cube Helix is a colour scheme designed to be appropriate for screen display of intensity images. The scheme is intended to be monotonically increasing in brightness when displayed in greyscale. This might also provide improved visualisation for colour blindness sufferers.

Usage

```
cubeHelix(n, start = 0.5, r = -1.5, hue = 1, gamma = 1)
```

Arguments

`n`	integer giving the number of colours in the scale
`start`	numeric: start gives the initial angle (in radians) of the helix
`r`	numeric: number of rotations of the helix over the scale; can be negative
`hue`	numeric controling the saturation of colour: 0 gives pure greyscale, defaults to 1
`gamma`	numeric which can be used to emphasise lower or higher intensity values, defaults to 1

Details

The function evaluates a helix which moves through the RGB "cube", beginning at black (0,0,0) and finishing at white (1,1,1). Evenly spaced points on this helix in the cube are returned as RGB colours. This provides a colour palette in which intensity increases monotonically, which makes for good transfer to greyscale displays or printouts. This also may have advantages for colour blindeness sufferers. See references for further details.

Value

Vector of RGB colours (strings) of length `n`.

Author(s)

Dave Green

Robin Evans

References

Green, D. A., 2011, A colour scheme for the display of astronomical intensity images. *Bulletin of the Astronomical Society of India*, 39, 289. http://adsabs.harvard.edu/abs/2011BASI...39..289G

See Dave Green's page at http://www.mrao.cam.ac.uk/~dag/CUBEHELIX/ for other details.

See Also

`rainbow` (for other colour palettes).

Examples

```
cubeHelix(21)

## Not run:
cols = cubeHelix(101)
```

```
plot.new()
plot.window(xlim=c(0,1), ylim=c(0,1))
axis(side=1)
for (i in 1:101) {
  rect((i-1)/101,0,(i+0.1)/101,1, col=cols[i], lwd=0)
}

## End(Not run)

## Not run:
require(grDevices)
# comparison with other palettes
n = 101
cols = cubeHelix(n)
heat = heat.colors(n)
rain = rainbow(n)
terr = terrain.colors(n)

plot.new()
plot.window(xlim=c(-0.5,1), ylim=c(0,4))
axis(side=1, at=c(0,1))
axis(side=2, at=1:4-0.5, labels=1:4, pos=0)
for (i in 1:n) {
  rect((i-1)/n,3,(i+0.1)/n,3.9, col=cols[i], lwd=0)
  rect((i-1)/n,2,(i+0.1)/n,2.9, col=heat[i], lwd=0)
  rect((i-1)/n,1,(i+0.1)/n,1.9, col=rain[i], lwd=0)
  rect((i-1)/n,0,(i+0.1)/n,0.9, col=terr[i], lwd=0)
}
legend(-0.6,4,legend=c("4. cube helix", "3. heat", "2. rainbow", "1. terrain"), box.lwd=0)

## End(Not run)
```

`designMatrix` *Orthogonal Design Matrix*

Description

Produces a matrix whose rows correspond to an orthogonal binary design matrix.

Usage

```
designMatrix(n)
```

Arguments

`n`	integer containing the number of elements in the set.

Value

An integer matrix of dimension 2^n by 2^n containing 1 and -1.

Note

The output matrix has orthogonal columns and is symmetric, so (up to a constant) is its own inverse. Operations with this matrix can be performed more efficiently using the fast Hadamard transform.

Author(s)

Robin Evans

See Also

`combinations`, `subsetMatrix`.

Examples

```
designMatrix(3)
```

`Dirichlet` *The Dirichlet Distribution*

Description

Density function and random generation for Dirichlet distribution with parameter vector `alpha`.

Usage

```
ddirichlet(x, alpha, log = FALSE, tol = 1e-10)
rdirichlet(n, alpha)
```

Arguments

`x`	vector (or matrix) of points in sample space.
`alpha`	vector of Dirichlet hyper parameters.
`log`	logical; if TRUE, natural logarithm of density is returned.
`tol`	tolerance of vectors not summing to 1 and negative values.
`n`	number of random variables to be generated.

Details

If x is a matrix, each row is taken to be a different point whose density is to be evaluated. If the number of columns in (or length of, in the alpha, the vector sum to 1.

The k-dimensional Dirichlet distribution has density

$$\frac{\Gamma\left(\sum_i \alpha_i\right)}{\prod_i \Gamma(\alpha_i)} \prod_{i=1}^{k} x_i^{\alpha_i - 1}$$

assuming that $x_i > 0$ and $\sum_i x_i = 1$, and zero otherwise.

If the sum of row entries in x differs from 1 by more than tol, is assumed to be

Value

rdirichlet returns a matrix, each row of which is an independent draw alpha.

ddirichlet returns a vector, each entry being the density of the corresponding row of x. If x is a vector, then the output will have length 1.

Author(s)

Robin Evans

References

http://en.wikipedia.org/wiki/Dirichlet_distribution

Examples

```
x = rdirichlet(10, c(1,2,3))
x

# Find densities at random points.
ddirichlet(x, c(1,2,3))
# Last column to be inferred.
ddirichlet(x[,c(1,2)], c(1,2,3))
ddirichlet(x, matrix(c(1,2,3), 10, 3, byrow=TRUE))
```

---

expit *Expit and Logit.*

Description

Functions to take the expit and logit of numerical vectors.

Usage

```
expit(x)
```

Arguments

`x`	vector of real numbers; for `logit` to return a sensible value these should be between 0 and 1.

Details

`logit` implements the usual logit function, which is

$$logit(x) = \log \frac{x}{1-x},$$

and `expit` its inverse:

$$expit(x) = \frac{e^x}{1+e^x}.$$

It is assumed that `logit(0) = -Inf` and `logit(1) = Inf`, and correspondingly for `expit`.

Value

A real vector corresponding to the expits or logits of `x`

Warning

Choosing very large (positive or negative) values to apply to `expit` may result in inaccurate inversion (see example below).

Author(s)

Robin Evans

Examples

```
x = c(5, -2, 0.1)
y = expit(x)
logit(y)

# Beware large values!
logit(expit(100))
```

`fastHadamard` *Compute fast Hadamard-transform of vector*

Description

Passes vector through Hadamard orthogonal design matrix. Also known as the Fast Walsh-Hadamard transform.

Usage

```
fastHadamard(x, pad = FALSE)
```

Arguments

x	vector of values to be transformed
pad	optional logical asking whether vector not of length 2^k should be padded with zeroes

Details

This is equivalent to multiplying by `designMatrix(log2(length(x)))` but should run much faster

Value

A vector of the same length as x

Author(s)

Robin Evans

See Also

`designMatrix`, `subsetMatrix`.

Examples

```
fastHadamard(1:8)
fastHadamard(1:5, pad=TRUE)
```

---

fsapply — *Fast and loose application of function over list.*

Description

Faster highly stripped down version of `sapply()`

Usage

```
fsapply(x, FUN)
```

Arguments

x	a vector (atomic or list) or an expression object.
FUN	the function to be applied to each element of `x`. In the case of functions like `+`, the function name must be backquoted or quoted.

Details

This is just a wrapper for `unlist(lapply(x,FUN))`, which will behave as `sapply` if `FUN` returns an atomic vector of length 1 each time.

Speed up over sapply is not dramatic, but can be useful in time critical code.

Value

A vector of results of applying `FUN` to `x`.

Warning

Very loose version of `sapply` which should really only by used if you're confident about how `FUN` is applied to each entry in `x`.

Author(s)

Robin Evans

Examples

```
x = list(1:1000)
tmp = fsapply(x, sin)

## Not run:
x = list()
set.seed(142313)
for (i in 1:1000) x[[i]] = rnorm(100)

system.time(for (i in 1:100) sapply(x, function(x) last(x)))
system.time(for (i in 1:100) fsapply(x, function(x) last(x)))

## End(Not run)
```

`greaterThan` *Comparing numerical values*

Description

Just a wrapper for comparing numerical values, for use with quicksort.

Usage

```
greaterThan(x, y)
```

Arguments

`x`	A numeric vector.
`y`	A numeric vector.

Details

Just returns `-1` if `x` is less than `y`, `1` if `x` is greater, and `0` if they are equal (according to `==`). The vectors wrap as usual if they are of different lengths.

Value

An integer vector.

Author(s)

Robin Evans

See Also

`` `<` `` for traditional Boolean operator.

Examples

```
greaterThan(4,6)

# Use in sorting algorithm.
quickSort(c(5,2,9,7,6), f=greaterThan)
order(c(5,2,9,7,6))
```

---

indexBox — *Get indices of adjacent entries in array*

Description

Determines the relative vector positions of entries which are adjacent in an array.

Usage

```
indexBox(upp, lwr, dim)
```

Arguments

`upp`	A vector of non-negative integers, giving the distance in the positive direction from the centre in each co-ordinate.
`lwr`	A vector of non-positive integers, giving the negative distance from the centre.
`dim`	integer vector of array dimensions.

Details

Given a particular cell in an array, which are the entries within (for example) 1 unit in any direction? This function gives the (relative) value of such indices. See examples.

Indices may be repeated if the range exceeds the size of the array in any dimension.

Value

An integer vector giving relative positions of the indices.

Author(s)

Robin Evans

See Also

arrayInd.

Examples

```
arr = array(1:144, dim=c(3,4,3,4))
arr[2,2,2,3]
# which are entries within 1 unit each each direction of 2,2,2,3?

inds = 89 + indexBox(1,-1,c(3,4,3,4))
inds = inds[inds > 0 & inds <= 144]
arrayInd(inds, c(3,4,3,4))

# what about just in second dimension?
inds = 89 + indexBox(c(0,1,0,0),c(0,-1,0,0),c(3,4,3,4))
inds = inds[inds > 0 & inds <= 144]
arrayInd(inds, c(3,4,3,4))
```

---

interventionMatrix   *Calculate interventional distributions.*

Description

Calculate interventional distributions from a probability table or matrix of multivariate probability distributions.

Usage

```
interventionMatrix(x, variables, condition, dim = NULL, incols = FALSE)

interventionTable(x, variables, condition)
```

Arguments

`x`	An array of probabilities.
`variables`	The margin for the intervention.
`condition`	The dimensions to be conditioned upon.
`dim`	Integer vector containing dimensions of variables. Assumed all binary if not specified.
`incols`	Logical specifying whether not the distributions are stored as the columns in the matrix; assumed to be rows by default.

Details

This just divides the joint distribution $p(x)$ by $p(v|c)$, where v is `variables` and c is `condition`.

Under certain causal assumptions this is the interventional distribution $p(x \mid do(v))$ (i.e. if the direct causes of v are precisely c.)

`intervention.table()` is identical to `interventionTable()`.

Value

A numerical array of the same dimension as x.

Functions

- `interventionMatrix`: Interventions in matrix of distributions

Author(s)

Robin Evans

References

Pearl, J., *Causality*, 2nd Edition. Cambridge University Press, 2009.

See Also

`conditionTable`, `marginTable`

Examples

```
set.seed(413)
# matrix of distributions
p = rdirichlet(10, rep(1,16))
interventionMatrix(p, 3, 2)

# take one in an array
ap = array(p[1,], rep(2,4))
interventionTable(ap, 3, 2)
```

is.subset *Check subset inclusion*

Description

Determines whether one vector contains all the elements of another.

Usage

```
is.subset(x, y)
```

Arguments

x	vector.
y	vector.

Details

Determines whether or not every element of `x` is also found in `y`. Returns `TRUE` if so, and `FALSE` if not.

Value

A logical of length 1.

Author(s)

Robin Evans

See Also

setmatch.

Examples

```
is.subset(1:2, 1:3)
is.subset(1:2, 2:3)
```

`is.wholenumber` *Determine whether number is integral or not.*

Description

Checks whether a numeric value is integral, up to machine or other specified prescision.

Usage

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

Arguments

`x`	numeric vector to be tested.
`tol`	The desired precision.

Value

A logical vector of the same length as `x`, containing the results of the test.

Author(s)

Robin Evans

Examples

```
x = c(0.5, 1, 2L, 1e-20)
is.wholenumber(x)
```

`last` *Last element of a vector or list*

Description

Returns the last element of a list or vector.

Usage

```
last(x)
```

Arguments

`x`	a list or vector.

Details

Designed to be faster than using `tail()` or `rev()`, and cleaner than writing `x[length(x)]`.

Value

An object of the same type as `x` of length 1 (or empty if `x` is empty).

Author(s)

Robin Evans

See Also

`tail`, `rev`.

Examples

```
last(1:10)
```

`marginTable` *Compute margin of a table faster*

Description

Computes the margin of a contingency table given as an array, by summing out over the dimensions not specified.

Usage

```
marginTable(x, margin = NULL, order = TRUE)
marginMatrix(x, margin, dim = NULL, incols = FALSE, order = FALSE)
```

Arguments

`x`	a numeric array
`margin`	integer vector giving margin to be calculated (1 for rows, etc.)
`order`	logical - should indices of output be ordered as in the vector `margin`? Defaults to `TRUE` for marginTable, `FALSE` for marginMatrix.
`dim`	Integer vector containing dimensions of variables. Assumed all binary if not specified.
`incols`	Logical specifying whether not the distributions are stored as the columns in the matrix; assumed to be rows by default.

Details

With order = TRUE this is the same as the base function margin.table(), but faster.

With order = FALSE the function is even faster, but the indices in the margin are returned in their original order, regardless of the way they are specified in margin.

propTable() returns a renormalized contingency table whose entries sum to 1. It is equivalent to prop.table(), but faster.

Value

The relevant marginal table. The class of x is copied to the output table, except in the summation case.

Note

Original functions are margin.table and prop.table.

Examples

```
m <- matrix(1:4, 2)
marginTable(m, 1)
marginTable(m, 2)

propTable(m, 2)

# 3-way example
m <- array(1:8, rep(2,3))
marginTable(m, c(2,3))
marginTable(m, c(3,2))
marginTable(m, c(3,2), order=FALSE)

#' set.seed(2314)
# set of 10 2x2x2 probability distributions
x = rdirichlet(10, rep(1,8))

marginMatrix(x, c(1,3))
marginMatrix(t(x), c(1,3), incols=TRUE)
```

---

patternRepeat *Complex repetitions*

Description

Recreate patterns for collapsed arrays

Usage

```
patternRepeat(x, which, n, careful = TRUE, keep.order = FALSE)
```

Arguments

x	A vector to be repeated.
which	Which indices of the implicit array are given in x.
n	Dimensions of implicit array.
careful	logical indicating whether to check vailidty of arguments, but therefore slow things down.
keep.order	logical indicating whether to respect the ordering of the entries in the vector which, in which case data are permuted before replication. In other words, does x change fastest in which[1], or in the minimal entry for which?

Details

These functions allow for the construction of complex repeating patterns corresponding to those obtained by unwrapping arrays. Consider an array with dimensions n; then for each value of the dimensions in which, this function returns a vector which places the corresponding entry of x into every place which would match this pattern when the full array is unwrapped.

For example, if a full 4-way array has dimensions 2*2*2*2 and we consider the margin of variables 2 and 4, then the function returns the pattern c(1,1,2,2,1,1,2,2,3,3,4,4,3,3,4,4). The entries 1,2,3,4 correspond to the patterns (0,0), (1,0), (0,1) and (1,1) for the 2nd and 4th indices.

In patternRepeat() the argument x is repeated according to the pattern, while patternRepeat0() just returns the indexing pattern. So patternRepeat(x,which,n) is effectively equivalent to x[patternRepeat0(which,n)].

The length of x must be equal to prod(n[which]).

Value

Both return a vector of length prod(n); patternRepeat() one containing suitably repeated and ordered elements of x, for patternRepeat0() it is always the integers from 1 up to prod(n[which]).

Author(s)

Robin Evans

See Also

rep

Examples

```
patternRepeat(1:4, c(1,2), c(2,2,2))
c(array(1:4, c(2,2,2)))

patternRepeat0(c(1,3), c(2,2,2))
patternRepeat0(c(2,3), c(2,2,2))

patternRepeat0(c(3,1), c(2,2,2))
patternRepeat0(c(3,1), c(2,2,2), keep.order=TRUE)
```

```
patternRepeat(letters[1:4], c(1,3), c(2,2,2))
```

`powerSet` *Power Set*

Description

Produces the power set of a vector.

Usage

```
powerSet(x, m, rev = FALSE)

powerSetCond(x, y, m, rev = FALSE, sort = FALSE)
```

Arguments

`x`	vector of elements (the set).
`m`	maximum cardinality of subsets
`rev`	logical indicating whether to reverse the order of subsets.
`y`	set to condition on
`sort`	logical: should sets be sorted?

Details

Creates a list containing every subset of the elements of the vector `x`.

`powerSet` returns subsets up to size `m` (if this is specified). `powerSetCond` includes some non-empty subset of `x` in every set.

Value

A list of vectors of the same type as `x`.

With `rev = FALSE` (the default) the list is ordered such that all subsets containing the last element of `x` come after those which do not, and so on.

Functions

- `powerSetCond`: Add sets that can't be empty

Author(s)

Robin Evans

See Also

powerSetMat.

Examples

```
powerSet(1:3)
powerSet(letters[3:5], rev=TRUE)
powerSet(1:5, m=2)

powerSetCond(2:3, y=1)
```

---

printPercentage — *Print Percentage of Activity Completed to stdout*

Description

Prints percentage (or alternatively just a count) of loop or similar process which has been completed to the standard output.

Usage

```
printPercentage(i, n, dp = 0, first = 1, last = n, prev = i - 1)
```

Arguments

i	the number of iterations completed.
n	total number of iterations.
dp	number of decimal places to display.
first	number of the first iteration for which this percentage was displayed
last	number of the final iteration for which this percentage will be displayed
prev	number of the previous iteration for which this percentage was displayed

Details

`printPercentage` will use `cat` to print the proportion of loops which have been completed (i.e. `i/n`) to the standard output. In doing so it will erase the previous such percentage, except when `i = first`. A new line is added when `i = last`, assuming that the loop is finished.

Value

`NULL`

Warning

This will fail to work nicely if other information is printed to the standard output

Author(s)

Robin Evans

Examples

```
x = numeric(100)

for (i in 1:100) {
  x[i] = mean(rnorm(1e5))
  printPercentage(i,100)
}


i = 0
repeat {
  i = i+1
  if (runif(1) > 0.99) {
    break
  }
  printCount(i)
}
print("\n")
```

---

quickSort *Quicksort for Partial Orderings*

Description

Implements the quicksort algorithm for partial orderings based on pairwise comparisons.

Usage

```
quickSort(x, f = greaterThan, ..., random = TRUE)
```

Arguments

`x`	A list or vector of items to be sorted.
`f`	A function on two arguments for comparing elements of `x`. Returns `-1` if the first argument is less than the second, `1` for the reverse, and `0` if they are equal or incomparable.
`...`	other arguments to `f`
`random`	logical - should a random pivot be chosen? (this is recommended) Otherwise middle element is used.

Details

Implements the usual quicksort algorithm, but may return the same positions for items which are incomparable (or equal). Does not test the validity of `f` as a partial order.

If `x` is a numeric vector with distinct entries, this behaves just like `rank`.

Value

Returns an integer vector giving each element's position in the order (minimal element(s) is 1, etc).

Warning

Output may not be consistent for certain partial orderings (using random pivot), see example below. All results will be consistent with a total ordering which is itselft consistent with the true partial ordering.

`f` is not checked to see that it returns a legitimate partial order, so results may be meaningless if it is not.

Author(s)

Robin Evans

References

http://en.wikipedia.org/wiki/Quicksort.

See Also

`order`.

Examples

```
set.seed(1)
quickSort(powerSet(1:3), f=subsetOrder)
quickSort(powerSet(1:3), f=subsetOrder)
# slightly different answers, but both correposnding
# to a legitimate total ordering.
```

`rje` *Miscellaneous useful functions*

Description

A series of useful functions, some available in different forms in other packages, but which have been extended, sped up, or otherwise modified in some way considered useful to the author.

Details

Package:	rje
Type:	Package
Version:	1.10.16
Date:	2020-03-17
License:	GPL (>= 2)
LazyLoad:	yes

Author(s)

Robin Evans

Mathias Drton

Maintainer: Robin Evans <evans@stats.ox.ac.uk>

`rowMins` *Row-wise minima and maxima*

Description

Row-wise minima and maxima

Usage

```
rowMins(x)
rowMaxs(x)
```

Arguments

`x`	a numeric (or logical) matrix or data frame

Details

The function coerces `x` to be a data frame and then uses `pmin` (`pmax`) on it. This is the same as `apply(x,1,min)` but generally faster if the number of rows is large.

Value

numeric vector of length `nrow(x)` giving the row-wise minima (or maxima) of `x`.

`rprobdist` *Generate a joint (or conditional) probability distribution*

Description

Wrapper functions to quickly generate discrete joint (or conditional) distributions using Dirichlets

Usage

```
rprobdist(dim, d, cond, alpha = 1)
```

Arguments

`dim`	the joint dimension of the probability table
`d`	number of dimensions
`cond`	optionally, vertices to condition upon
`alpha`	Dirichlet hyper parameter, defaults to 1 (flat density).

Details

`rprobdist` gives an array of dimension `dim` (recycled as necessary to have length `d`, if this is supplied) whose entries are probabilities drawn from a Dirichlet distribution whose parameter vector has entries equal to `alpha` (appropriately recycled).

Value

an array of appropriate dimensions

Side Effects

Uses as many gamma random variables as cells in the table, so will alter the random seed accordingly.

Author(s)

Robin Evans

Examples

```
rprobdist(2, 4)     # 2x2x2x2 table
rprobdist(c(2,3,2)) # 2x3x2 table

rprobdist(2, 4, alpha=1/16)     # using unit information prior

# get variables 2 and 4 conditioned upon
rprobdist(2, 4, cond=c(2,4), alpha=1/16)
```

schur *Obtain generalized Schur complement*

Description

Obtain generalized Schur complement

Usage

```
schur(M, x, y, z)
```

Arguments

`M`	symmetric positive definite matrix
`x, y, z`	indices of M to calculate with (see below)

Details

Calculates $M_{xy} - M_{xz}M^{zz}M_{zy}$, which (if M is a Gaussian covariance matrix) is the covariance between x and y after conditioning on z.

y defaults to equal x, and z to be the complement of $x \cup y$.

setmatch *Set Operations*

Description

Series of functions extending existing vector operations to lists of vectors.

Usage

```
setmatch(x, y, nomatch = NA_integer_)

setsetdiff(x, y)

setsetequal(x, y)

subsetmatch(x, y, nomatch = NA_integer_)

supersetmatch(x, y, nomatch = NA_integer_)
```

Arguments

`x`	list of vectors.
`y`	list of vectors.
`nomatch`	value to be returned in the case when no match is found. Note that it is coerced to integer.

Details

`setmatch` checks whether each vector in the list `x` is also contained in the list `y`, and if so returns position of the first such vector in `y`. The ordering of the elements of the vector is irrelevant, as they are considered to be sets.

`subsetmatch` is similar to `setmatch`, except vectors in `x` are searched to see if they are subsets of vectors in `y`. Similarly `supersetmatch` consideres if vectors in `x` are supersets of vectors in `y`.

`setsetdiff` is a setwise version of `setdiff`, and `setsetequal` a setwise version of `setequal`.

Value

`setmatch` and `subsetmatch` return a vector of integers of length the same as the list `x`.

`setsetdiff` returns a sublist `x`.

`setsetequal` returns a logical of length 1.

Functions

- `setsetdiff`: Setdiff for lists
- `setsetequal`: Test for equality of sets
- `subsetmatch`: Test for subsets
- `supersetmatch`: Test for supersets

Note

These functions are not recursive, in the sense that they cannot be used to test lists of lists. They also do not reduce to the vector case.

Author(s)

Robin Evans

See Also

`match`, `setequal`, `setdiff`

Examples

```
x = list(1:2, 1:3)
y = list(1:4, 1:3)
setmatch(x, y)
subsetmatch(x, y)
setsetdiff(x, y)

x = list(1:3, 1:2)
y = list(2:1, c(2,1,3))
setsetequal(x, y)
```

`subsetMatrix` *Matrix of Subset Indicators*

Description

Produces a matrix whose rows indicate what subsets of a set are included in which other subsets.

Usage

```
subsetMatrix(n)
```

Arguments

`n`	integer containing the number of elements in the set.

Details

This function returns a matrix, with each row and column corresponding to a subset of a hypothetical set of size `n`, ordered lexographically. The entry in row `i`, column `j` corresponds to whether or not the subset associated with `i` is a superset of that associated with `j`.

A 1 or -1 indicates that `i` is a superset of `j`, with the sign referring to the number of fewer elements in `j`. 0 indicates that `i` is not a superset of `j`.

Value

An integer matrix of dimension 2^n by 2^n.

Note

The inverse of the output matrix is just `abs(subsetMatrix(n))`.

Author(s)

Robin Evans

See Also

`combinations`, `powerSet`, `designMatrix`.

Examples

```
subsetMatrix(3)
```

subsetOrder *Compare sets for inclusion.*

Description

A wrapper for `is.subset` which returns set inclusions.

Usage

```
subsetOrder(x, y)
```

Arguments

`x`	A vector.
`y`	A vector of the same type as `x`.

Details

If `x` is a subset of `y`, returns `-1`, for the reverse returns `1`. If sets are equal or incomparable, it returns `0`.

Value

A single integer, 0, -1 or 1.

Author(s)

Robin Evans

See Also

`is.subset`.

Examples

```
subsetOrder(2:4, 1:4)
subsetOrder(2:4, 3:5)
```

subtable *Subset an array*

Description

More flexible calls of [on an array.

Usage

```
subtable(x, variables, levels, drop = TRUE)

subarray(x, levels, drop = TRUE)

subtable(x, variables, levels) <- value

subarray(x, levels) <- value
```

Arguments

x	An array.
variables	An integer vector containing the dimensions of x to subset.
levels	A list or vector containing values to retain.
drop	Logical indicating whether dimensions with only 1 retained should be dropped. Defaults to TRUE.
value	Value to assign to entries in table.

Details

Essentially just allows more flexible calls of [on an array.

subarray requires the values for each dimension should be specified, so for a $2 \times 2 \times 2$ array x, subarray(x,list(1,2,1:2)) is just x[1,2,1:2].

subtable allows unspecified dimensions to be retained automatically. Thus, for example subtable(x,c(2,3),list(1,1:2) is x[,1,1:2].

Value

Returns an array of dimension sapply(value,length) if drop=TRUE, otherwise *specified* dimensions of size 1 are dropped. Dimensions which are unspecified in subtable are never dropped.

Functions

- subarray: Flexible subsetting
- subtable<-: Assignment in a table
- subarray<-: Assignment in an array

Author(s)

Mathias Drton, Robin Evans

See Also

Extract

Examples

```
x = array(1:8, rep(2,3))
subarray(x, c(2,1,2)) == x[2,1,2]

x[2,1:2,2,drop=FALSE]
subarray(x, list(2,1:2,2), drop=FALSE)

subtable(x, c(2,3), list(1, 1:2))
```

Index

∗Topic **IO**
    printPercentage, 25
∗Topic **arith**
    combinations, 5
    designMatrix, 10
    expit, 12
    fastHadamard, 13
    greaterThan, 15
    interventionMatrix, 17
    is.subset, 19
    is.wholenumber, 20
    powerSet, 24
    quickSort, 26
    rje, 27
    setmatch, 30
    subsetMatrix, 32
    subsetOrder, 33
∗Topic **array**
    conditionMatrix, 6
    indexBox, 16
    marginTable, 21
    patternRepeat, 22
    rje, 27
    subtable, 34
∗Topic **color**
    cubeHelix, 8
∗Topic **distribution**
    Dirichlet, 11
    rprobdist, 29
∗Topic **iteration**
    printPercentage, 25
∗Topic **list**
    fsapply, 14
∗Topic **manip**
    last, 20
∗Topic **optimize**
    armijo, 3
    quickSort, 26
    rje, 27
∗Topic **package**
    rje, 27
∗Topic **print**
    printPercentage, 25
%subof% (is.subset), 19

and0, 2
armijo, 3
arrayInd, *17*

coarseLine (armijo), 3
combinations, 5, *11*, *32*
condition.table (conditionMatrix), 6
condition.table2 (conditionMatrix), 6
conditionMatrix, 6
conditionTable, *18*
conditionTable (conditionMatrix), 6
conditionTable2 (conditionMatrix), 6
cubeHelix, 8

ddirichlet (Dirichlet), 11
designMatrix, 10, *14*, *32*
Dirichlet, 11

expit, 12
Extract, *35*

fastHadamard, 13
fsapply, 14

greaterThan, 15

indexBox, 16
intervention.table (interventionMatrix), 17
interventionMatrix, 17
interventionTable, *8*
interventionTable (interventionMatrix), 17
is.subset, 19, *33*
is.wholenumber, 20

last, 20
logit (expit), 12

margin.table, *8*, *22*
marginMatrix (marginTable), 21
marginTable, *8*, *18*, 21
match, *31*

or0 (and0), 2
order, *27*

patternRepeat, 22
patternRepeat0 (patternRepeat), 22
powerSet, 24, *32*
powerSetCond (powerSet), 24
powerSetMat, *25*
powerSetMat (combinations), 5
printCount (printPercentage), 25
printPercentage, 25
prop.table, *22*
propTable (marginTable), 21

quickSort, 26

rainbow, *9*
rank, *27*
rdirichlet (Dirichlet), 11
rep, *23*
rev, *21*
rje, 27
rje-package (rje), 27
rowMaxs (rowMins), 28
rowMins, 28
rprobdist, 29

schur, 30
setdiff, *31*
setequal, *31*
setmatch, *19*, 30
setsetdiff (setmatch), 30
setsetequal (setmatch), 30
subarray (subtable), 34
subarray<- (subtable), 34
subsetmatch (setmatch), 30
subsetMatrix, *11*, *14*, 32
subsetOrder, 33
subtable, 34
subtable<- (subtable), 34
supersetmatch (setmatch), 30

tail, *21*