Package 'permute'

July 23, 2025

Title Functions for Generating Restricted Permutations of Data

Version 0.9-8

Date 2025-06-25

Depends R (>= 3.6.0)

Imports stats

Suggests vegan (>= 2.0-0), testthat (>= 0.5), parallel, knitr, rmarkdown, bookdown, sessioninfo

Description A set of restricted permutation designs for freely exchangeable, line transects (time series), and spatial grid designs plus permutation of blocks (groups of samples) is provided. 'permute' also allows split-plot designs, in which the whole-plots or split-plots or both can be freely-exchangeable or one of the restricted designs. The 'permute' package is modelled after the permutation schemes of 'Canoco 3.1' (and later) by Cajo ter Braak.

License GPL-2

ByteCompile true

URL https://github.com/gavinsimpson/permute

BugReports https://github.com/gavinsimpson/permute/issues

Copyright see file COPYRIGHTS

VignetteBuilder knitr

NeedsCompilation no

Author Gavin L. Simpson [aut, cph, cre] (ORCID: <https://orcid.org/0000-0002-9084-8413>), R Core Team [cph], Douglas M. Bates [ctb], Jari Oksanen [ctb]

Maintainer Gavin L. Simpson <ucfagls@gmail.com>

Repository CRAN

Date/Publication 2025-06-25 14:00:02 UTC

Contents

`allPerms` *Complete enumeration of all possible permutations*

Description

`allPerms` is a utility function to return the set of permutations for a given R object and a specified permutation design.

Usage

```
allPerms(n, control = how(), check = TRUE)

## S3 method for class 'allPerms'
summary(object, ...)

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

as.allPerms(object, control)
```

Arguments

`n`	the number of observations or an 'object' from which the number of observations can be determined via `getNumObs`.
`control`	a list of control values describing properties of the permutation design, as returned by a call to how.
`check`	logical; should `allPerms` check the design? The default is to check, but this can be skipped, for example if a function checked the design earlier.
`object`	for `summary.allPerms`, an object of class `"allPerms"`. For `as.allPerms` a matrix or something that can be coerced to a matrix by `as.matrix`.

...	arguments to other methods.
x	an object of class "allPerms", as returned by allPerms.

Details

Function allPerms enumerates all possible permutations for the number of observations and the selected permutation scheme. It has print and summary methods. allPerms returns a matrix containing all possible permutations, possibly containing the observed ordering (if argument observed is TRUE). The rows of this matrix are the various permutations and the columns reflect the number of samples.

With free permutation designs, and restricted permutation schemes with large numbers of observations, there are a potentially huge number of possible permutations of the samples. It would be inefficient, not to mention incredibly time consuming, to enumerate them all. Storing all possible permutations would also become problematic in such cases. To control this and guard against trying to evaluate too large a number of permutations, if the number of possible permutations is larger than getMaxperm(control), allPerms exits with an error.

The as.matrix method sets the control and seed attributes to NULL and removes the "permutationMatrix" class, resulting in a standard matrix object.

Value

For allPerms, and object of class "allPerms", a matrix whose rows are the set of all possible permutations for the supplies number of observations and permutation scheme selected. The matrix has two additional attributes control and observed. Attribute control contains the argument control (possibly updated via check). Attribute observed contains argument observed.

Warning

If permuting the strata themselves, a balanced design is required (the same number of observations in each level of strata. This is common to all functions in the package.

Author(s)

Gavin Simpson

Examples

```
## allPerms can work with a vector
vec <- c(3,4,5)
allPerms(vec) ## free permutation

## enumerate all possible permutations for a more complicated
## design
fac <- gl(2,6)
ctrl <- how(within = Within(type = "grid", mirror = FALSE,
                            constant = TRUE, nrow = 3, ncol = 2),
            plots = Plots(strata = fac))
Nobs <- length(fac)
numPerms(seq_len(Nobs), control = ctrl) ## 6
(tmp <- allPerms(Nobs, control = update(ctrl, observed = TRUE)))
```

```
(tmp2 <- allPerms(Nobs, control = ctrl))

## turn on mirroring
##ctrl$within$mirror <- TRUE
ctrl <- update(ctrl, within = update(getWithin(ctrl), mirror = TRUE))
numPerms(seq_len(Nobs), control = ctrl)
(tmp3 <- allPerms(Nobs, control = update(ctrl, observed = TRUE)))
(tmp4 <- allPerms(Nobs, control = ctrl))

## prints out details of the permutation scheme as
## well as the matrix of permutations
summary(tmp3)
summary(tmp4)
```

`allUtils` *Utility functions for complete enumeration of all possible permutations*

Description

Utility functions to return the set of all permutations under different designs. For most practical applications, i.e. to combine designs permuting blocks and/or within blocks function `allPerms` will be required.

Usage

```
allFree(n, v = seq_len(n))

allSeries(n, nperms, mirror = FALSE)

allGrid(n, nperms, nr, nc, mirror, constant)

allStrata(n, control)
```

Arguments

`n`	the number of observations.
`v`	numeric; vector of indices. Default is `1:n`.
`nperms`	numeric; number of possible permutations.
`mirror`	logical; mirroring of permutations allowed?
`nr`, `nc`	integer; number of rows and columns of grid designs.
`constant`	logical; same permutation within each block?
`control`	a list of control values describing properties of the permutation design, as returned by a call to `how`.

Details

These are utility functions and aren't designed for casual use. allPerms should be used instead.

Details on usage of these functions can be found in allPerms.

Value

A matrix of all possible permutations of n observations or of v, given the provided options.

Author(s)

Gavin Simpson

---

check *Utility functions for permutation schemes*

Description

check provides checking of permutation schemes for validity. permuplot produces a graphical representation of the selected permutation design.

Usage

```
check(object, control = how(), quietly = FALSE)

## S3 method for class 'check'
summary(object, ...)
```

Arguments

object	an R object. See Details for a complete description, especially for numPerms. For summary.check an object of class "check".
control	a list of control values describing properties of the permutation design, as returned by a call to how.
quietly	logical; should messages by suppressed?
...	arguments to other methods.

Details

check is a utility functions for working with the new permutation schemes available in shuffle.

check is used to check the current permutation schemes against the object to which it will be applied. It calculates the maximum number of possible permutations for the number of observations in object and the permutation scheme described by control. The returned object contains component control, an object of class "how" suitably modified if check identifies a problem.

The main problem is requesting more permutations than is possible with the number of observations and the permutation design. In such cases, nperm is reduced to equal the number of possible

permutations, and complete enumeration of all permutations is turned on (`control$complete` is set to `TRUE`).

Alternatively, if the number of possible permutations is low, and less than `control$minperm`, it is better to enumerate all possible permutations, and as such complete enumeration of all permutations is turned on (`control$complete` is set to `TRUE`). This guarantees that permutations are all unique and there are no duplicates.

Value

For `check` a list containing the maximum number of permutations possible and an object of class `"how"`.

Author(s)

Gavin L. Simpson

See Also

`shuffle` and `how`.

Examples

```
## only run this example if vegan is available
if (suppressPackageStartupMessages(require("vegan"))) {
    ## use example data from ?pyrifos in package vegan
    example(pyrifos)

    ## Demonstrate the maximum number of permutations for the pyrifos data
    ## under a series of permutation schemes

    ## no restrictions - lots of perms
    CONTROL <- how(within = Within(type = "free"))
    (check1 <- check(pyrifos, CONTROL))
    ## summary(check1)

    ## no strata but data are series with no mirroring, so 132 permutations
    CONTROL <- how(within = Within(type = "series", mirror = FALSE))
    check(pyrifos, CONTROL)

    ## no strata but data are series with mirroring, so 264 permutations
    CONTROL <- how(within = Within(type = "series", mirror = TRUE))
    check(pyrifos, control = CONTROL)

    ## unrestricted within strata
    check(pyrifos, control = how(plots = Plots(strata = ditch),
                   within = Within(type = "free")))

    ## time series within strata, no mirroring
    check(pyrifos,
          control = how(plots = Plots(strata = ditch),
          within = Within(type = "series", mirror = FALSE)))
```

```
    ## time series within strata, with mirroring
    check(pyrifos,
          control = how(plots = Plots(strata = ditch),
          within = Within(type = "series", mirror = TRUE)))

    ## time series within strata, no mirroring, same permutation
    ## within strata
    check(pyrifos,
          control = how(plots = Plots(strata = ditch),
          within = Within(type = "series", constant = TRUE)))

    ## time series within strata, with mirroring, same permutation
    ## within strata
    check(pyrifos,
          control = how(plots = Plots(strata = ditch),
          within = Within(type = "series", mirror = TRUE,
          constant = TRUE)))
    ## permute strata
    check(pyrifos, how(plots = Plots(strata = ditch, type = "free"),
                       within = Within(type = "none")))
}

## this should also also for arbitrary vectors
vec1 <- check(1:100)
vec2 <- check(1:100, how())
all.equal(vec1, vec2)
vec3 <- check(1:100, how(within = Within(type = "series")))
all.equal(100, vec3$n)
vec4 <- check(1:100, how(within = Within(type= "series", mirror = TRUE)))
all.equal(vec4$n, 200)

## enumerate all possible permutations
fac <- gl(2,6)
ctrl <- how(plots = Plots(strata = fac),
            within = Within(type = "grid", mirror = FALSE,
                            constant = TRUE, nrow = 3, ncol = 2))
check(1:12, ctrl)

numPerms(1:12, control = ctrl)
(tmp <- allPerms(12, control = update(ctrl, observed = TRUE)))
(tmp2 <- allPerms(12, control = ctrl))

## turn on mirroring
ctrl <- update(ctrl, within = update(getWithin(ctrl), mirror = TRUE))
numPerms(1:12, control = ctrl)
(tmp3 <- allPerms(12, control = update(ctrl, observed = TRUE)))
(tmp4 <- allPerms(12, control = ctrl))
## prints out details of the permutation scheme as
## well as the matrix of permutations
summary(tmp)
summary(tmp2)

## different numbers of observations per level of strata
```

```
fac <- factor(rep(1:3, times = c(3,2,2)))
## free permutations in levels of strata
numPerms(7, how(within = Within(type = "free"),
                plots = Plots(strata = fac, type = "none")))
allPerms(7, how(within = Within(type = "free"),
                plots = Plots(strata = fac)))
## series permutations in levels of strata
ctrl <- how(within = Within(type = "series"), plots = Plots(strata = fac))
numPerms(7, control = ctrl)
allPerms(7, control = ctrl)
```

---

get-methods *Extractor functions to access components of a permutation design*

Description

Simple functions to allow abstracted access to components of a permutation design, for example as returned by how. Whilst many of these are very simple index opertations on a list, using these rather than directly accessing that list allows the internal representation of the permutation design to change without breaking code.

Usage

```
getAllperms(object, ...)
getBlocks(object, ...)
getComplete(object, ...)
getConstant(object, ...)
getCol(object, ...)
getDim(object, ...)
getMake(object, ...)
getMaxperm(object, ...)
getMinperm(object, ...)
getMirror(object, ...)
getNperm(object, ...)
getObserved(object, ...)
getPlots(object, ...)
getRow(object, ...)
getStrata(object, ...)
getType(object, ...)
getWithin(object, ...)
getControl(object, ...)
getHow(object, ...)

## S3 method for class 'how'
getAllperms(object, ...)

## S3 method for class 'how'
getBlocks(object, ...)
```

```
## S3 method for class 'how'
getCol(object, which = c("plots", "within"), ...)
## S3 method for class 'Plots'
getCol(object, ...)
## S3 method for class 'Within'
getCol(object, ...)

## S3 method for class 'how'
getComplete(object, ...)

## S3 method for class 'how'
getConstant(object, ...)
## S3 method for class 'Within'
getConstant(object, ...)

## S3 method for class 'how'
getDim(object, which = c("plots", "within"), ...)
## S3 method for class 'Plots'
getDim(object, ...)
## S3 method for class 'Within'
getDim(object, ...)

## S3 method for class 'how'
getMake(object, ...)

## S3 method for class 'how'
getMaxperm(object, ...)

## S3 method for class 'how'
getMinperm(object, ...)

## S3 method for class 'how'
getMirror(object, which = c("plots", "within"), ...)
## S3 method for class 'Plots'
getMirror(object, ...)
## S3 method for class 'Within'
getMirror(object, ...)

## S3 method for class 'how'
getNperm(object, ...)

## S3 method for class 'how'
getObserved(object, ...)

## S3 method for class 'how'
getPlots(object, ...)
```

```
## S3 method for class 'how'
getRow(object, which = c("plots", "within"), ...)
## S3 method for class 'Plots'
getRow(object, ...)
## S3 method for class 'Within'
getRow(object, ...)

## S3 method for class 'how'
getStrata(object, which = c("plots", "blocks"),
          drop = TRUE, ...)
## S3 method for class 'Plots'
getStrata(object, drop = TRUE, ...)

## S3 method for class 'how'
getType(object, which = c("plots", "within"), ...)
## S3 method for class 'Plots'
getType(object, ...)
## S3 method for class 'Within'
getType(object, ...)

## S3 method for class 'how'
getWithin(object, ...)

## S3 method for class 'allPerms'
getControl(object, ...)
```

Arguments

`object`	An R object to dispatch on.
`which`	character; which level of restriction to extract information for.
`drop`	logical; should un-used factor levels be dropped?
`...`	Arguments passed on to other methods.

Details

These are extractor functions for working with permutation design objects created by how. They should be used in preference to directly subsetting the permutation design in case the internal structure of object changes as **permute** is developed.

`getHow` is an alias for `getControl`; specific methods are implemented for `getControl` if you are debugging.

Value

These are simple extractor functions and return the contents of the corresponding components of `object`.

Author(s)

Gavin Simpson

See Also

check, a utility function for checking permutation scheme described by how.

Examples

```
## extract components from a "how" object
hh <- how()
getWithin(hh)
getNperm(hh)
```

---

how *How to define a permutation design?*

Description

Utility functions to describe unrestricted and restricted permutation designs for time series, line transects, spatial grids and blocking factors.

Usage

```
how(within = Within(), plots = Plots(), blocks = NULL,
    nperm = 199, complete = FALSE, maxperm = 9999,
    minperm = 5040, all.perms = NULL, make = TRUE,
    observed = FALSE)

Within(type = c("free","series","grid","none"),
       constant = FALSE, mirror = FALSE,
       ncol = NULL, nrow = NULL)

Plots(strata = NULL, type = c("none","free","series","grid"),
      mirror = FALSE, ncol = NULL, nrow = NULL)
```

Arguments

within, plots, blocks	Permutation designs for samples within the levels of plots (within), permutation of plots themselves, or for the definition of blocking structures which further restrict permutations (blocks). within and plots each require a named list as produced by Within and Plots respectively. blocks takes a factor (or an object coercible to a factor via as.factor), the levels of which define the blocking structure.
nperm	numeric; the number of permutations.
complete	logical; should complete enumeration of all permutations be performed?
type	character; the type of permutations required. One of "free", "series", "grid" or "none". See Details.
maxperm	numeric; the maximum number of permutations to perform. Currently unused.

minperm	numeric; the lower limit to the number of possible permutations at which complete enumeration is performed. When nperm is lower than minperm, sampling is performed from the set of complete permutations to avoid duplicate permutations. See argument complete and Details, below.
all.perms	an object of class allPerms, the result of a call to allPerms.
make	logical; should check generate all possible permutations? Useful if want to check permutation design but not produce the matrix of all permutations, or to circumvent the heuristics governing when complete enumeration is activated.
observed	logical; should the observed permutation be returned as part of the set of all permutations? Default is FALSE to facilitate usage in higher level functions.
constant	logical; should the same permutation be used within each level of strata? If FALSE a separate, possibly restricted, permutation is produced for each level of strata.
mirror	logical; should mirroring of sequences be allowed?
ncol, nrow	numeric; the number of columns and rows of samples in the spatial grid respectively.
strata	A factor, or an object that can be coerced to a factor via as.factor, specifying the strata for permutation.

Details

shuffle can generate permutations for a wide range of restricted permutation schemes. A small selection of the available combinations of options is provided in the Examples section below.

Argument type controls how samples are actually permuted; "free" indicates randomization, "series" indicates permutation via cyclic shifts (suitable for evenly-spaced line transect or time series data), "grid" indicates permutation via toroidal shifts (suitable for samples on a regular grid), and "none" indicates no permutation of samples. See the package vignette (browseVignettes("permute")) for additional information on each of these types of permutation.

Argument mirror determines whether grid or series permutations can be mirrored. Consider the sequence 1,2,3,4. The relationship between consecutive observations is preserved if we reverse the sequence to 4,3,2,1. If there is no inherent direction in your experimental design, mirrored permutations can be considered part of the Null model, and as such increase the number of possible permutations. The default is to not use mirroring so you must explicitly turn this on using mirror = TRUE in how.

To permute plots rather than the observations within plots (the levels of strata), use Within(type = "none") and Plots(type = foo), where foo is how you want the plots to be permuted. However, note that the number of observations within each plot **must** be equal!

For some experiments, such as BACI designs, one might wish to use the same permutation within each plot. This is controlled by argument constant. If constant = TRUE then the same permutation will be generated for each level of strata. The default is constant = FALSE.

Value

For how a list with components for each of the possible arguments.

Author(s)

Gavin Simpson

References

`shuffle()` is modelled after the permutation schemes of Canoco 3.1 (ter Braak, 1990); see also Besag & Clifford (1989).

Besag, J. and Clifford, P. (1989) Generalized Monte Carlo significance tests. *Biometrika* **76**; 633–642.

ter Braak, C. J. F. (1990). *Update notes: CANOCO version 3.1*. Wageningen: Agricultural Mathematics Group. (UR).

See Also

`shuffle` and `shuffleSet` for permuting from a design, and `check`, a utility function for checking permutation design described by how.

Examples

```
## Set up factors for the Plots and Blocks
plts <- gl(4, 10) ## 4 Plots of 10 samples each
blks <- gl(2, 20) ## 2 Blocks of 20 samples each

## permutation design
h1 <- how(within = Within(type = "series", mirror = TRUE),
          plots = Plots(strata = plts, type = "series"),
          blocks = blks)

## The design can be updated...
## ... remove the blocking:
update(h1, blocks = NULL)

## ... or switch the type of shuffling at a level:
#update(h1, plots = update(getPlots(h1), type = "none"))
plots2 <- update(getPlots(h1), type = "none")
update(h1, plots = plots2)
```

---

jackal *Mandible lengths of male and female golden jackals*

Description

Mandible lengths (in mm) for male and female golden jackals (*Canis aureus*) from a collection of specimens in the British Museum of Natural History, London, UK.

Usage

```
data(jackal)
```

Format

A data frame with 20 observations on the following 2 variables.

`Length` a numeric vector

`Sex` a factor with levels `Male` `Female`

Source

The data were manually transcribed from Manly (2007).

References

Higham, C.F.W., Kijngam, A., and Manly, B.F.J. (1980) An analysis of prehistoric canid remains from Thailand. *Journal of Archaeological Science* **7**:149-165.

Manly, B.F.J. (2007) *Randomization, bootstrap and Monte Carlo methods in biology. Third Edition*. Chapman & Hall/CRC, Boca Raton.

Examples

```
data(jackal)
str(jackal)

## boxplot of mandible length vs sex
plot(Length ~ Sex, data = jackal)
```

`nobs-methods` *Number of observations in a given object*

Description

`nobs` is a generic function to return the number of observations from a model. `shuffle` provides a few methods for other types of data object in R.

Usage

```
## S3 method for class 'numeric'
nobs(object, ...)

## S3 method for class 'integer'
nobs(object, ...)

## S3 method for class 'matrix'
nobs(object, ...)

## S3 method for class 'data.frame'
nobs(object, ...)
```

```
## S3 method for class 'character'
nobs(object, ...)

## S3 method for class 'factor'
nobs(object, ...)
```

Arguments

object	a data frame or matrix, or a numeric, integer, character, or factor vector.
...	arguments to other methods.

Details

Function nobs is a simple generic function to return the number of observations in a range of R model objects. Methods are provided to work with a variety of R objects.

Value

The (numeric) number of observations in object.

Author(s)

Gavin Simpson

Examples

```
set.seed(1)
## numeric vector
len <- sample(1:10, 1)
v <- as.numeric(sample(1:100, len))
len
obs <- nobs(v)
isTRUE(all.equal(len, obs))

## integer
len <- sample(1L:10L, 1)
obs <- nobs(len)
isTRUE(all.equal(len, obs))
```

---

numPerms *Number of possible permutations for a given object*

Description

numPerms calculates the maximum number of permutations possible under the current permutation scheme.

Usage

```
numPerms(object, control = how())
```

Arguments

`object`	any object handled by `nobs`.
`control`	a list of control values describing properties of the permutation design, as returned by a call to `how`.

Details

Function `numPerms` returns the number of permutations for the passed `object` and the selected permutation scheme. `object` can be one of a data frame, matrix, an object for which a scores method exists, or a numeric or integer vector. In the case of a numeric or integer vector, a vector of length 1 can be used and it will be expanded to a vector of length `object` (i.e., `1:object`) before computing the number of permutations. As such, `object` can be the number of observations not just the object containing the observations.

Value

The (numeric) number of possible permutations of observations in `object`.

Note

In general, mirroring `"series"` or `"grid"` designs doubles or quadruples, respectively, the number of permutations without mirroring (within levels of strata if present). This is **not** true in two special cases:

1. In `"grid"` designs where the number of columns is equal to 2, and
2. In `"series"` designs where the number of observations in a series is equal to 2.

For example, with 2 observations there are 2 permutations for `"series"` designs:

1. 1-2, and
2. 2-1.

If these two permutations were mirrored, we would have:

1. 2-1, and
2. 1-2.

It is immediately clear that this is the same set of permutations without mirroring (if one reorders the rows). A similar situation arises in `"grid"` designs where the number of **columns** per *grid* is equal to 2. Note that the number of rows per *grid* is not an issue here.

Author(s)

Gavin Simpson

See Also

shuffle and how. Additional nobs methods are provide, see nobs-methods.

Examples

```
## permutation design --- see ?how
ctrl <- how() ## defaults to freely exchangeable

## vector input
v <- 1:10
(obs <- nobs(v))
numPerms(v, control = ctrl)

## integer input
len <- length(v)
(obs <- nobs(len))
numPerms(len, control = ctrl)

## new design, objects are a time series
ctrl <- how(within = Within(type = "series"))
numPerms(v, control = ctrl)
## number of permutations possible drastically reduced...
## ...turn on mirroring
ctrl <- how(within = Within(type = "series", mirror = TRUE))
numPerms(v, control = ctrl)

## Try blocking --- 2 groups of 5
bl <- numPerms(v, control = how(blocks = gl(2,5)))
bl

## should be same as
pl <- numPerms(v, control = how(plots = Plots(strata = gl(2,5))))
pl
stopifnot(all.equal(bl, pl))
```

---

set-methods *Replacement functions to set components of a permutation design*

Description

Simple functions to allow abstracted replacement of components of a permutation design, for example as returned by how. In addition to performing replacement of components of the list returned by how, these replacement function also update the matched calls stored within the list to facilitate the use of update by users.

Usage

```
setBlocks(object) <- value
setPlots(object) <- value
```

```
setWithin(object) <- value
setStrata(object) <- value
setNperm(object) <- value
setAllperms(object) <- value
setMaxperm(object) <- value
setMinperm(object) <- value
setComplete(object) <- value
setMake(object) <- value
setObserved(object) <- value
setRow(object) <- value
setCol(object) <- value
setDim(object) <- value
setType(object) <- value
setMirror(object) <- value
setConstant(object) <- value
```

Arguments

`object`	An R object to dispatch on.
`value`	The replacement value/object.

Details

These are replacement functions for working with permutation design objects created by how. They should be used in preference to directly updating the permutation design in case the internal structure of object changes as **permute** is developed and because the matched call also needs to be updated to facilitate use of update on the how object.

Value

These replacement functions return `object` suitably modified.

Note

`setStrata<-` has methods for objects of class `"how"` and `"Plots"`. The former sets the `blocks` component of the how object, whilst the latter sets the `strata` component of the `Plots` object.

`setDim<-`, `setRow<-`, and `setCol<-` cannot be used on an object of class `"how"`. Instead, extract the `Plots` or `Within` components with `getPlots` or `getWithin` and alter those components, then use the resulting object to replace the `plots` or `within` components using `setPlots` or `setWithin`.

Author(s)

Gavin Simpson

See Also

check, a utility function for checking permutation scheme described by how. Comparable extractor functions are also available; see get-methods.

Examples

```
## extract components from a "how" object
hh <- how()
getNperm(hh)
setNperm(hh) <- 999
getNperm(hh)
```

---

shuffle *Unrestricted and restricted permutations*

Description

Unrestricted and restricted permutation designs for time series, line transects, spatial grids and blocking factors.

Usage

```
shuffle(n, control = how())

permute(i, n, control)
```

Arguments

n	numeric; the length of the returned vector of permuted values. Usually the number of observations under consideration. May also be any object that nobs knows about; see nobs-methods.
control	a list of control values describing properties of the permutation design, as returned by a call to how.
i	integer; row of control$all.perms to return.

Details

shuffle can generate permutations for a wide range of restricted permutation schemes. A small selection of the available combinations of options is provided in the Examples section below.

permute is a higher level utility function for use in a loop within a function implementing a permutation test. The main purpose of permute is to return the correct permutation in each iteration of the loop, either a random permutation from the current design or the next permutation from control$all.perms if it is not NULL and control$complete is TRUE.

Value

For shuffle a vector of length n containing a permutation of the observations 1, ..., n using the permutation scheme described by argument control.

For permute the ith permutation from the set of all permutations, or a random permutation from the design.

Author(s)

Gavin Simpson

References

shuffle() is modelled after the permutation schemes of Canoco 3.1 (ter Braak, 1990); see also Besag & Clifford (1989).

Besag, J. and Clifford, P. (1989) Generalized Monte Carlo significance tests. *Biometrika* **76**; 633–642.

ter Braak, C. J. F. (1990). *Update notes: CANOCO version 3.1*. Wageningen: Agricultural Mathematics Group. (UR).

See Also

check, a utility function for checking permutation scheme described by how.

Examples

```
set.seed(1234)

## unrestricted permutations
shuffle(20)

## observations represent a time series of line transect
CTRL <- how(within = Within(type = "series"))
shuffle(20, control = CTRL)

## observations represent a time series of line transect
## but with mirroring allowed
CTRL <- how(within = Within(type = "series", mirror = TRUE))
shuffle(20, control = CTRL)

## observations represent a spatial grid, 5rx4c
nr <- 5
nc <- 4
CTRL <- how(within = Within(type = "grid", ncol = nc, nrow = nr))
perms <- shuffle(20, control = CTRL)
## view the permutation as a grid
matrix(matrix(1:20, nrow = nr, ncol = nc)[perms],
       ncol = nc, nrow = nr)

## random permutations in presence of strata
plots <- Plots(strata = gl(4, 5))
CTRL <- how(plots = plots, within = Within(type = "free"))
shuffle(20, CTRL)
## as above but same random permutation within strata
CTRL <- how(plots = plots, within = Within(type = "free",
            constant = TRUE))
shuffle(20, CTRL)

## time series within each level of block
```

```
CTRL <- how(plots = plots, within = Within(type = "series"))
shuffle(20, CTRL)
## as above, but  with same permutation for each level
CTRL <- how(plots = plots, within = Within(type = "series",
            constant = TRUE))
shuffle(20, CTRL)

## spatial grids within each level of block, 4 x (5r x 5c)
nr <- 5
nc <- 5
nb <- 4 ## number of blocks
plots <- Plots(gl(nb, 25))
CTRL <- how(plots = plots,
            within = Within(type = "grid", ncol = nc, nrow = nr))
shuffle(100, CTRL)
## as above, but with same permutation for each level
CTRL <- how(plots = plots,
            within = Within(type = "grid", ncol = nc, nrow = nr,
                            constant = TRUE))
shuffle(100, CTRL)

## permuting levels of plots instead of observations
CTRL <- how(plots = Plots(gl(4, 5), type = "free"),
            within = Within(type = "none"))
shuffle(20, CTRL)
## permuting levels of plots instead of observations
## but plots represent a time series
CTRL <- how(plots = Plots(gl(4, 5), type = "series"),
            within = Within(type = "none"))
shuffle(20, CTRL)

## permuting levels of plots but plots represent a time series
## free permutation within plots
CTRL <- how(plots = Plots(gl(4, 5), type = "series"),
            within = Within(type = "free"))
shuffle(20, CTRL)

## permuting within blocks
grp <- gl(2, 10) # 2 groups of 10 samples each
CTRL <- how(blocks = grp)
shuffle(length(grp), control = CTRL)

## Simple function using permute() to assess significance
## of a t.test
pt.test <- function(x, group, control) {
    ## function to calculate t
    t.statistic <- function(x, y) {
        m <- length(x)
        n <- length(y)
        ## means and variances, but for speed
        xbar <- mean(x)
        ybar <- mean(y)
        xvar <- var(x)
```

```
        yvar <- var(y)
        pooled <- sqrt(((m-1)*xvar + (n-1)*yvar) / (m+n-2))
        (xbar - ybar) / (pooled * sqrt(1/m + 1/n))
    }
    ## check the control object
    #control <- check(x, control)$control ## FIXME
    ## number of observations
    Nobs <- nobs(x)
    ## group names
    lev <- names(table(group))
    ## vector to hold results, +1 because of observed t
    t.permu <- numeric(length = control$nperm) + 1
    ## calculate observed t
    t.permu[1] <- t.statistic(x[group == lev[1]], x[group == lev[2]])
    ## generate randomisation distribution of t
    for(i in seq_along(t.permu)) {
        ## return a permutation
        want <- permute(i, Nobs, control)
        ## calculate permuted t
        t.permu[i+1] <- t.statistic(x[want][group == lev[1]],
                                    x[want][group == lev[2]])
    }
    ## pval from permutation test
    pval <- sum(abs(t.permu) >= abs(t.permu[1])) / (control$nperm + 1)
    ## return value
    return(list(t.stat = t.permu[1], pval = pval))
}

## generate some data with slightly different means
set.seed(1234)
gr1 <- rnorm(20, mean = 9)
gr2 <- rnorm(20, mean = 10)
dat <- c(gr1, gr2)
## grouping variable
grp <- gl(2, 20, labels = paste("Group", 1:2))
## create the permutation design
control <- how(nperm = 999, within = Within(type = "free"))
## perform permutation t test
perm.val <- pt.test(dat, grp, control)
perm.val

## compare perm.val with the p-value from t.test()
t.test(dat ~ grp, var.equal = TRUE)
```

---

shuffle-utils *Utility functions for unrestricted and restricted permutations*

Description

Unrestricted and restricted permutations for time series, line transects, spatial grids and blocking factors.

Usage

```
shuffleFree(x, size)

shuffleSeries(x, mirror = FALSE, start = NULL, flip = NULL)

shuffleGrid(nrow, ncol, mirror = FALSE, start.row = NULL,
            start.col = NULL, flip = NULL)

shuffleStrata(strata, type, mirror = FALSE, start = NULL, flip = NULL,
              nrow, ncol, start.row = NULL, start.col = NULL)
```

Arguments

x	vector of indices to permute.
size	number of random permutations required
mirror	logical; should mirroring of sequences be allowed?
start	integer; the starting point for time series permutations. If missing, a random starting point is determined.
flip	logical, length 1 (shuffleSeries) or length 2 (shuffleGrid); force mirroring of permutation. This will always return the reverse of the computed permutation. For shuffleGrid, the first element pertains to flipping rows, the second to flipping columns of the grid.
nrow, ncol	numeric; the number of rows and columns in the grid.
start.row, start.col	numeric; the starting row and column for the shifted grid permutation. If non supplied, a random starting row and column will be selected.
strata	factor; the blocks to permute.
type	character; the type of permutation used to shuffle the strata. One of "free", "grid" or "series".

Details

These are developer-level functions for generating permuted indexes from one of several restricted and unrestricted designs.

shuffleFree is a wrapper to code underlying sample, but without the extra over head of sanity checks. It is defined as sample.int(x, size, replace = FALSE). You must arrange for the correct values to be supplied, where x is a vector of indices to sample from, and size is the number of indices to sample. Sampling is done without replacement and without regard to prior probabilities. Argument size is allowed so that one can draw a single observation at random from the indices x. In general use, size would be set equal to length{x}.

Value

A integer vector of permuted indices.

Author(s)

Gavin Simpson

See Also

check, a utility function for checking permutation scheme described by how. shuffle as a user-oriented wrapper to these functions.

Examples

```
set.seed(3)

## draw 1 value at random from the set 1:10
shuffleFree(1:10, 1)

## permute the series 1:10
x <- 1:10
shuffleSeries(x)                ## with random starting point
shuffleSeries(x, start = 5L)    ## known starting point
shuffleSeries(x, flip = TRUE)   ## random start, forced mirror
shuffleSeries(x, mirror = TRUE) ## random start, possibly mirror

## permute a grid of size 3x3
shuffleGrid(3, 3)                     ## random starting row/col
shuffleGrid(3, 3, start.row = 2,
            start.col = 3)            ## with known row/col
shuffleGrid(3, 3, flip = rep(TRUE, 2)) ## random start, forced mirror
```

---

shuffleSet *Generate a set of permutations from the specified design.*

Description

shuffleSet returns a set of nset permutations from the specified design. The main purpose of the function is to circumvent the overhead of repeatedly calling shuffle to generate a set of permutations.

Usage

```
shuffleSet(n, nset, control = how(), check = TRUE, quietly = FALSE)

## S3 method for class 'permutationMatrix'
as.matrix(x, ...)
```

Arguments

n	numeric; the number of observations in the sample set. May also be any object that nobs knows about; see nobs-methods.
nset	numeric; the number of permutations to generate for the set. Can be missing, the default, in which case nset is determined from control.
control	an object of class "how" describing a valid permutation design.
check	logical; should the design be checked for various problems via function check? The default is to check the design for the stated number of observations and update control accordingly. See Details.
quietly	logical; should messages by suppressed?
x	an object of class "permutationMatrix", as returned by shuffleSet.
...	arguments passed to other methods. For the as.matrix method only.

Details

shuffleSet is designed to generate a set of nset permutation indices over which a function can iterate as part of a permutation test. It is only slightly more efficient than calling shuffle nset times, but it is far more practical than the simpler function because a set of permutations can be worked on by applying a function to the rows of the returned object. This simplifies the function applied, and facilitates the use of parallel processing functions, thus enabling a larger number of permutations to be evaluated in reasonable time.

By default, shuffleSet will check the permutations design following a few simple heuristics. See check for details of these. Whether some of the heuristics are activiated or not can be controlled via how, essentialy via its argument minperm. In particular, if there are fewer than minperm permutations, shuffleSet will generate and return **all possible permutations**, which may differ from the number requested via argument nset.

The check argument to shuffleSet controls whether checking is performed in the permutation design. If you set check = FALSE then exactly nset permutations will be returned. However, do be aware that there is no guarantee that the set of permutations returned will be unique, especially so for designs and data sets where there are few possible permutations relative to the number requested.

The as.matrix method sets the control and seed attributes to NULL and removes the "permutationMatrix" class, resulting in a standard matrix object.

Value

Returns a matrix of permutations, where each row is a separate permutation. As such, the returned matrix has nset rows and n columns.

Author(s)

Gavin L. Simpson

References

shuffleSet() is modelled after the permutation schemes of Canoco 3.1 (ter Braak, 1990); see also Besag & Clifford (1989).

Besag, J. and Clifford, P. (1989) Generalized Monte Carlo significance tests. *Biometrika* **76**; 633–642.

ter Braak, C. J. F. (1990). *Update notes: CANOCO version 3.1*. Wageningen: Agricultural Mathematics Group. (UR).

See Also

See shuffle for generating a single permutation, and how for setting up permutation designs.

Examples

```
set.seed(1)
## simple random permutations, 5 permutations in set
shuffleSet(n = 10, nset = 5)

## series random permutations, 5 permutations in set
shuffleSet(10, 5, how(within = Within(type = "series")))

## series random permutations, 10 permutations in set,
## with possible mirroring
CTRL <- how(within = Within(type = "series", mirror = TRUE))
shuffleSet(10, 10, CTRL)

## Permuting strata
## 4 groups of 5 observations
CTRL <- how(within = Within(type = "none"),
            plots = Plots(strata = gl(4,5), type = "free"))
shuffleSet(20, 10, control = CTRL)

## 10 random permutations in presence of Plot-level strata
plotStrata <- Plots(strata = gl(4,5))
CTRL <- how(plots = plotStrata,
            within = Within(type = "free"))
numPerms(20, control = CTRL)
shuffleSet(20, 10, control = CTRL)
## as above but same random permutation within Plot-level strata
CTRL <- how(plots = plotStrata,
            within = Within(type = "free", constant = TRUE))
numPerms(20, control = CTRL)
shuffleSet(20, 10, CTRL) ## check this.

## time series within each level of Plot strata
CTRL <- how(plots = plotStrata,
            within = Within(type = "series"))
shuffleSet(20, 10, CTRL)
## as above, but  with same permutation for each Plot-level stratum
CTRL <- how(plots = plotStrata,
            within = Within(type = "series", constant = TRUE))
```

```
shuffleSet(20, 10, CTRL)
```

Index