Package 'permute'

November 6, 2026

Title Functions for Generating Restricted Permutations of Data

Version 0.10-0

Depends R (>= 3.6.0)

Imports stats

Suggests vegan (>= 2.0-0), testthat (>= 3.1.5), 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://gavinsimpson.github.io/permute/, https://github.com/gavinsimpson/permute

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

Copyright see file COPYRIGHTS

VignetteBuilder knitr

Language en-GB

Encoding UTF-8

RoxygenNote 8.0.0

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 2026-11-06 14:00:02 UTC

Contents

allPerms . . . 2
allUtils . . . 4
check . . . 5
get-methods . . . 8
how . . . 12
jackal . . . 16
nobs-methods . . . 17
numPerms . . . 18
partitions . . . 20
permutations . . . 22
set-methods . . . 23
shuffle . . . 25
shuffle-utils . . . 29
shuffleSet . . . 31

Index **35**

`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 base::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 base::print() and base::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) or is reported as Inf by numPerms(), 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.

With Plots(type = "partition"), the rows enumerate distinct assignments to the labelled groups in Plots$strata. If the group sizes are $n_1, \ldots, n_K$, there are $n!/\prod_k n_k!$ such assignments before optionally removing the observed assignment.

Value

For allPerms, an 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 does not apply to Plots(type = "partition"), which supports unequal group sizes.

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, such as combining designs that permute blocks and/or observations within blocks, `allPerms()` is required.

Usage

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

allSeries(n, nperms, mirror = FALSE)

allGrid(n, nperms, nr, nc, mirror, symmetric = FALSE)

allStrata(n, control)
```

Arguments

`n`	The number of observations.
`v`	Numeric vector of indices. The default is `seq_len(n)`.
`nperms`	Numeric; number of possible permutations.
`mirror`	Logical; should mirroring of permutations be allowed?

nr, nc	Integer; number of rows and columns of grid designs.
symmetric	Logical; for grid permutations, should simultaneous mirroring in both spatial directions be disallowed?
control	A list describing the permutation design, as returned by how().

Details

These utility functions are not designed for casual use. See allPerms() for further details.

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.

Usage

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

## 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?
num_perms	logical; should the number of permutations be computed during checks?
...	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. If num_perms is TRUE, check also 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

These functions provide abstracted access to components of permutation designs such as those returned by how(). Using them instead of directly indexing the underlying list allows that representation to evolve without breaking user code.

Usage

```
getAllperms(object, ...)

getBlocks(object, ...)

getComplete(object, ...)

getConstant(object, ...)

getCol(object, ...)

getDim(object, ...)

getMake(object, ...)
```

```
getMaxperm(object, ...)

getMinperm(object, ...)

getMirror(object, ...)

getSymmetric(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'
getSymmetric(object, which = c("plots", "within"), ...)

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

## S3 method for class 'Plots'
getSymmetric(object, ...)

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

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

## S3 method for class 'Within'
getSymmetric(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 on which to dispatch.
`...`	Arguments passed to other methods.
`which`	Character; the level of restriction for which information is required.
`drop`	Logical; should unused factor levels be dropped?

Details

`getHow()` is an alias for `getControl()`; specific `getControl()` methods are useful when debugging.

Value

The contents of the corresponding component of `object`.

Author(s)

Gavin Simpson

See Also

check() checks a permutation design described by how().

Examples

```
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,
  data = NULL
)

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

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

```
  nrow = NULL,
  symmetric = FALSE,
  data = 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), or a one-sided formula, the levels of which define the blocking structure.
nperm	numeric; the number of permutations.
complete	logical; should complete enumeration of all permutations be performed?
maxperm	numeric; the maximum number of permutations that allPerms() may enumerate.
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.
data	A data frame in which to evaluate formula values supplied to blocks or strata. When Plots() is called inside how(), supply data to how() rather than to Plots(). A standalone call to Plots() can take its own data argument.
type	character; the type of permutations required. One of "free", "series", "grid", "none", or, for Plots, "partition". See Details.
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.
symmetric	logical; for grid permutations, should simultaneous mirroring in both spatial directions be disallowed?
strata	A factor, an object that can be coerced to a factor via as.factor, or a one-sided formula specifying the strata for permutation. Multiple variables and nested terms in a formula are combined into a single factor representing their interaction.

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.

Plots(type = "partition") randomly assigns observations to the labelled groups supplied in strata, retaining the observed number assigned to each group. Permutations that differ only by reordering observations carrying the same group label are omitted. If within is not supplied, how uses Within(type = "none") for this design. Supplying any other within-plot type is an error.

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.

For spatial grids, symmetric = TRUE preserves symmetric spatial autocovariance by preventing a permutation from being mirrored in both the row and column directions at the same time. It has no effect unless type = "grid" and mirror = TRUE.

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.

One-sided formulas can be used to define blocks and plot-level strata. A formula containing a single variable, such as ~ site, produces a factor from that variable. Variables participating in multiple model terms are combined into a single interaction factor. Consequently, ~ site + plot, ~ site:plot, and the nested forms ~ site / plot and ~ plot %in% site all describe groups formed from site and plot. Standard formula expansion and subtraction are honoured, so, for example, ~ . - unused uses all variables in data except unused.

Formulas are evaluated with standard model-frame semantics: variables are looked up first in data and then in the formula environment. Missing values are retained to preserve alignment with the observations, and unused factor levels and unobserved combinations are dropped. The formulas must be one-sided and contain at least one grouping term.

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)

## Random assignments to groups of fixed size
groups <- factor(c("a", "a", "a", "b", "b"))
(h2 <- how(plots = Plots(strata = groups, type = "partition")))
shuffle(length(groups), control = h2)

## Formula interfaces use a single data frame
dat <- data.frame(
    block = gl(2, 6),
    site = gl(3, 2, 12),
    plot = gl(2, 1, 12)
)
how(plots = Plots(strata = ~ site / plot), blocks = ~ block, data = dat)
getStrata(Plots(strata = ~ site / plot, data = dat))
```

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 2 variables:

`Length` A numeric vector containing mandible lengths in mm.

`Sex` A factor with levels `Male` and `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

`stats::nobs()` is a generic function that returns the number of observations from a model. permute provides methods for several other types of R object.

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 passed to other methods.

Details

These methods return the number of observations in numeric, integer, character, or factor vectors, matrices, and data frames.

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(), check = TRUE)
```

Arguments

object	any object handled by stats::nobs().
control	a list of control values describing properties of the permutation design, as returned by a call to how().
check	logical; should control be checked for problems?

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.

For Plots(type = "partition"), if the group sizes are $n_1, \ldots, n_K$, the number of distinct assignments is $n!/\prod_k n_k!$. With blocks, this quantity is calculated within each block and the results are multiplied.

Value

The (numeric) number of possible permutations of observations in `object`, or `Inf` if the count exceeds the range of a double.

Note

In general, mirroring `"series"` designs doubles the number of permutations and mirroring `"grid"` designs can quadruple it (within levels of strata if present). For grids with `symmetric = TRUE`, at most three orientations are included because simultaneous row and column mirroring is disallowed. Reflections of grid axes containing one or two cells are equivalent to toroidal shifts and do not add distinct permutations.

Mirroring does not double the number of series permutations when the series contains only two observations.

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).

Author(s)

Gavin Simpson

See Also

`shuffle()` and `how()`. Additional `stats::nobs()` methods are provided; 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))

## Distinct assignments to groups of sizes 3 and 2
groups <- factor(c("a", "a", "a", "b", "b"))
ctrl <- how(plots = Plots(groups, type = "partition"))
numPerms(length(groups), control = ctrl) ## 10
```

`partitions` *Generate random assignments to groups of fixed size*

Description

Generate all assignments, one random assignment, or a set of random assignments of observations to labelled groups whose sizes are fixed by a grouping factor.

Usage

```
allPartitions(strata, control = how(), check = TRUE)

shufflePartition(strata, control = how())

shufflePartitionSet(
  strata,
  nset,
  control = how(),
  check = TRUE,
  quietly = FALSE
)
```

Arguments

`strata`	A factor, or an object coercible to a factor, containing the group membership of every observation.

control	An object of class "how". Any blocking factor and the permutation-count controls are retained; its plots and within components are replaced by the partition design.
check	Logical; should the permutation design be checked?
nset	The number of random assignments to generate. If missing, it is obtained from control.
quietly	Logical; should messages about complete enumeration be suppressed?

Details

The number assigned to each group is fixed by table(strata). If the group sizes are $n_1, \ldots, n_K$, the number of distinct assignments is

$$n! / \prod_{k=1}^{K} n_k!.$$

Each assignment is returned as a permutation of observation indices, so it can be used wherever output from shuffle(), shuffleSet(), or allPerms() is accepted. The relative order of observations originally belonging to the same group is retained. This selects one canonical index permutation for each distinct arrangement of the group labels and omits permutations that differ only by reordering observations carrying the same label.

These functions are convenience wrappers for a design constructed with Plots(strata = strata, type = "partition") and Within(type = "none").

Value

shufflePartition() returns an integer vector of length length(strata). allPartitions() and shufflePartitionSet() return permutation matrices with one assignment per row.

See Also

allPerms(), shuffle(), shuffleSet(), and how().

Examples

```
groups <- factor(c("a", "a", "a", "b", "b"))

## One of 5! / (3! 2!) = 10 assignments
set.seed(1)
(p <- shufflePartition(groups))
groups[p]

## A set of random assignments
shufflePartitionSet(groups, nset = 5, check = FALSE)

## Complete enumeration, excluding the observed assignment by default
allPartitions(groups)
```

permutations *Permute a vector of data*

Description

`permutations()` generates permutations from a specified design and applies them to the values in a supplied vector. Each row of the returned object is one permutation of `x`.

Usage

```
permutations(x, nset, control = how(), check = TRUE, quietly = FALSE)

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

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

Arguments

`x`	A non-empty, one-dimensional atomic vector to permute. Factors are supported and are returned using their character labels.
`nset`	Numeric; the number of permutations to generate. If missing, the number is taken from `control`.
`control`	An object of class `"how"` describing a valid permutation design.
`check`	Logical; should the permutation design be checked by `check()`?
`quietly`	Logical; should messages from checking the design be suppressed?
`...`	Arguments passed to other methods.

Details

By default, the permutation design is checked in the same way as for `shuffleSet()`. Consequently, the number of rows returned can differ from `nset` when the design requests complete enumeration or when there are few possible permutations. Set `check = FALSE` to always request exactly `nset` permutations, which need not be unique.

Value

An object of class `"permutations"`, inheriting from `"matrix"`. Rows are permutations and columns correspond to the elements of `x`. The object has `control` and `seed` attributes containing the checked design and the random-number seed at the start of permutation generation.

`as.matrix()` returns an ordinary matrix with the permutation metadata removed.

See Also

`shuffleSet()` for generating permutation indices and `how()` for specifying permutation designs.

Examples

```
set.seed(1)
permutations(letters[1:5], nset = 3, check = FALSE)

## Restricted permutations of data from a time series
control <- how(within = Within(type = "series"))
permutations(c(10, 20, 30, 40, 50), nset = 3,
             control = control, check = FALSE)
```

---

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

Description

These functions provide abstracted replacement of components in a permutation design such as one returned by `how()`. They also update the matched calls stored in the design so that `stats::update()` continues to work correctly.

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

setSymmetric(object) <- value

setConstant(object) <- value
```

Arguments

object	An R object on which to dispatch.
value	The replacement value or object.

Details

Use these functions instead of directly modifying the underlying list, so code does not depend on permute's internal representation.

Value

object, suitably modified.

Note

setStrata<- has methods for objects of class "how" and "Plots". The former sets the blocks component of the how() object, while 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() component with getPlots() or getWithin(), alter it, and replace it with setPlots<- or setWithin<-.

Author(s)

Gavin Simpson

See Also

check() checks a design described by how(). See get-methods for the corresponding extractor functions.

Examples

```
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, perms = NULL)

permutator(
  n,
  nset,
  control = how(),
  check = TRUE,
  quietly = FALSE,
  perms = NULL
)
```

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 `stats::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.
perms	a matrix of permutation indices, such as an object returned by `shuffleSet()` or `allPerms()`. Each row must contain the indices from 1 to the number of columns exactly once.
nset	numeric; the number of permutations to generate. If missing, the number is taken from `control`.
check	logical; should the permutation design be checked by `check()`?
quietly	logical; should messages from checking the design be suppressed?

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.

With `Plots(type = "partition")`, `shuffle` returns one canonical index permutation for a random assignment to the groups in `Plots$strata`. Group sizes and the relative order of observations originally carrying the same group label are retained.

`permute` is a higher level utility function for use in a loop within a function implementing a permutation test. It returns a requested row from `perms` when a permutation matrix is supplied. Otherwise, it returns either a random permutation from the current design or the requested permutation from `control$all.perms` when complete enumeration is enabled.

`permutator` returns a function that iterates over a fixed set of permutations. The set can be supplied through `perms`, or generated when `permutator` is called by forwarding `n`, `nset`, `control`, `check`, and `quietly` to `shuffleSet()`. Supply either `perms` or the generation arguments, but not both. Each call to the returned function yields the next row, and returns `NULL` once the set is exhausted.

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 `i`th row of `perms`, the `i`th permutation from the set of all permutations, or a random permutation from the design.

For `permutator`, a zero-argument function with `nperm` and `n` attributes giving the number of permutations and observations. The function returns successive permutation-index vectors, then `NULL` after exhaustion.

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)

## Reuse a generated permutation set
perm_set <- shuffleSet(6, nset = 3, check = FALSE)
permute(2, perms = perm_set)
nextPerm <- permutator(perms = perm_set)
nextPerm()
nextPerm()

## 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))
    }
    ## number of observations
    Nobs <- nobs(x)
    ## group names
    lev <- names(table(group))
    ## generate and retain the permutations once
    nextPerm <- permutator(Nobs, control = control)
    nperm <- attr(nextPerm, "nperm")
    ## vector to hold results, +1 because of observed t
    t.permu <- numeric(length = nperm + 1L)
    ## 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_len(nperm)) {
        ## return a permutation
        want <- nextPerm()
        ## 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])) / (nperm + 1L)
    ## 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,
  symmetric = FALSE
)

shuffleStrata(
  strata,
```

```
  type,
  mirror = FALSE,
  start = NULL,
  flip = NULL,
  nrow,
  ncol,
  start.row = NULL,
  start.col = NULL,
  symmetric = FALSE
)
```

Arguments

x	A vector of indices to permute.
size	The number of indices 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 of length one for shuffleSeries() or length two for shuffleGrid(); force mirroring of the permutation. For shuffleGrid(), the first element flips rows and the second flips columns.
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 not supplied, they are selected randomly.
symmetric	Logical; for grid permutations, should simultaneous mirroring in both spatial directions be disallowed?
strata	A factor containing the blocks to permute.
type	Character; the permutation type used to shuffle strata. One of "free", "grid", "series", or "partition".

Details

These are developer-level functions for generating permuted indices from restricted and unrestricted designs.

shuffleFree() is a lightweight wrapper around the code underlying base::sample(). It calls base::sample.int(x, size, replace = FALSE) without additional checks. Sampling is without replacement and without regard to prior probabilities. size can be one to draw a single index; in general use it is set equal to length(x).

With type = "partition", shuffleStrata() returns one canonical index permutation for a random arrangement of the labels in strata; indices carrying the same original label retain their relative order.

Value

An integer vector of permuted indices.

Author(s)

Gavin Simpson

See Also

`check()` checks a permutation design, `how()` describes a design, and `shuffle()` is the user-oriented wrapper around 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
shuffleGrid(3, 3, mirror = TRUE,
            symmetric = TRUE)         ## never mirror both directions
```

`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 stats::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 activated or not can be controlled via how(), essentially 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.

For Plots(type = "partition"), rows represent distinct random assignments to groups of fixed size. Independently sampled rows may repeat, but permutations that differ only by reordering observations carrying the same original group label have the same canonical representation.

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

Value

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

∗ **datasets**
jackal, 16
∗ **design**
check, 5
partitions, 20
shuffle, 25
shuffle-utils, 29
shuffleSet, 31
∗ **htest**
shuffle, 25
shuffle-utils, 29
shuffleSet, 31
∗ **methods**
check, 5
get-methods, 8
set-methods, 23
∗ **utilities**
check, 5
∗ **utils**
get-methods, 8
how, 12
set-methods, 23

allFree *(*allUtils*)*, 4
allGrid *(*allUtils*)*, 4
allPartitions *(*partitions*)*, 20
allPerms, 2
allPerms(), *4, 5*, *13*, *21*, *25*
allSeries *(*allUtils*)*, 4
allStrata *(*allUtils*)*, 4
allUtils, 4
as.allPerms *(*allPerms*)*, 2
as.matrix.allPerms *(*allPerms*)*, 2
as.matrix.permutationMatrix *(*shuffleSet*)*, 31
as.matrix.permutations *(*permutations*)*, 22

base::as.matrix(), *3*
base::print(), *3*
base::sample(), *30*
base::summary(), *3*

check, 5
check(), *12*, *15*, *22*, *24–26*, *31*, *32*

get-methods, 8, *24*
getAllperms *(*get-methods*)*, 8
getBlocks *(*get-methods*)*, 8
getCol *(*get-methods*)*, 8
getComplete *(*get-methods*)*, 8
getConstant *(*get-methods*)*, 8
getControl *(*get-methods*)*, 8
getDim *(*get-methods*)*, 8
getHow *(*get-methods*)*, 8
getMake *(*get-methods*)*, 8
getMaxperm *(*get-methods*)*, 8
getMinperm *(*get-methods*)*, 8
getMirror *(*get-methods*)*, 8
getNperm *(*get-methods*)*, 8
getObserved *(*get-methods*)*, 8
getPlots *(*get-methods*)*, 8
getPlots(), *24*
getRow *(*get-methods*)*, 8
getStrata *(*get-methods*)*, 8
getSymmetric *(*get-methods*)*, 8
getType *(*get-methods*)*, 8
getWithin *(*get-methods*)*, 8
getWithin(), *24*

how, 12
how(), *2*, *5*, *6*, *8*, *12*, *18*, *19*, *21–24*, *26*, *31–33*

jackal, 16

nobs-methods, 17, *19*, *25*, *32*
nobs.character *(*nobs-methods*)*, 17
nobs.data.frame *(*nobs-methods*)*, 17
nobs.factor *(*nobs-methods*)*, 17
nobs.integer *(*nobs-methods*)*, 17
nobs.matrix *(*nobs-methods*)*, 17

nobs.numeric *(nobs-methods)*, 17
numPerms, 18
numPerms(), *3*, *5*

partitions, 20
permutations, 22
permutator *(shuffle)*, 25
permute *(shuffle)*, 25
Plots *(how)*, 12
Plots(), *24*
print.allPerms *(allPerms)*, 2
print.check *(check)*, 5
print.how *(how)*, 12
print.permutations *(permutations)*, 22
print.summary.allPerms *(allPerms)*, 2
print.summary.check *(check)*, 5

set-methods, 23
setAllperms<- *(set-methods)*, 23
setBlocks<- *(set-methods)*, 23
setCol<- *(set-methods)*, 23
setComplete<- *(set-methods)*, 23
setConstant<- *(set-methods)*, 23
setDim<- *(set-methods)*, 23
setMake<- *(set-methods)*, 23
setMaxperm<- *(set-methods)*, 23
setMinperm<- *(set-methods)*, 23
setMirror<- *(set-methods)*, 23
setNperm<- *(set-methods)*, 23
setObserved<- *(set-methods)*, 23
setPlots<- *(set-methods)*, 23
setRow<- *(set-methods)*, 23
setStrata<- *(set-methods)*, 23
setSymmetric<- *(set-methods)*, 23
setType<- *(set-methods)*, 23
setWithin<- *(set-methods)*, 23
shuffle, 25
shuffle(), *6*, *15*, *19*, *21*, *31–33*
shuffle-utils, 29
shuffleFree *(shuffle-utils)*, 29
shuffleGrid *(shuffle-utils)*, 29
shufflePartition *(partitions)*, 20
shufflePartitionSet *(partitions)*, 20
shuffleSeries *(shuffle-utils)*, 29
shuffleSet, 31
shuffleSet(), *15*, *21*, *22*, *25*, *26*
shuffleStrata *(shuffle-utils)*, 29
stats::nobs(), *17–19*, *25*, *32*
stats::update(), *23*

summary.allPerms *(allPerms)*, 2
summary.check *(check)*, 5

Within *(how)*, 12
Within(), *24*