Package 'permutations'

February 7, 2020

Type Package

Title The Symmetric Group: Permutations of a Finite Set

Version 1.0-6

Imports magic,numbers,partitions (>= 1.9-17)

Maintainer Robin K. S. Hankin <hankin.robin@gmail.com>

Description Manipulates invertible functions from a finite set to itself. Can transform from word form to cycle form and back.

License GPL-2

Suggests knitr

VignetteBuilder knitr

URL https://github.com/RobinHankin/permutations.git

BugReports https://github.com/RobinHankin/permutations/issues

NeedsCompilation no

Author Robin K. S. Hankin [aut, cre] (<https://orcid.org/0000-0001-5982-0415>)

Repository CRAN

Date/Publication 2020-02-07 06:30:21 UTC

R topics documented:

`permutations-package` *The Symmetric Group: Permutations of a Finite Set*

Description

Manipulates invertible functions from a finite set to itself. Can transform from word form to cycle form and back.

Details

The DESCRIPTION file:

Package:	permutations
Type:	Package
Title:	The Symmetric Group: Permutations of a Finite Set
Version:	1.0-6
Imports:	magic,numbers,partitions (>= 1.9-17)
Authors@R:	person(given=c("Robin", "K. S."), family="Hankin", role = c("aut","cre"), email="hankin.robin@gmail.co
Maintainer:	Robin K. S. Hankin <hankin.robin@gmail.com>
Description:	Manipulates invertible functions from a finite set to itself. Can transform from word form to cycle form an
License:	GPL-2
Suggests:	knitr
VignetteBuilder:	knitr
URL:	https://github.com/RobinHankin/permutations.git
BugReports:	https://github.com/RobinHankin/permutations/issues

Author: Robin K. S. Hankin [aut, cre] (<https://orcid.org/0000-0001-5982-0415>)

Index of help topics:

```
Ops.permutation         Arithmetic Ops Group Methods for permutations
allperms                All permutations of a given size
as.function.permutation
                        Coerce a permutation to a function
c                       Concatenation of permutations
cayley                  Cayley tables for permutation groups
commutator              Group-theoretic commutator and group action
conjugate               Are two permutations conjugate?
cyclist                 details of cyclists
derangement             Tests for a permutation being a derangement
dodecahedron            The dodecahedron group
fbin                    The fundamental bijection
fixed                   Fixed elements
get1                    Retrieve particular cycles or components of
                        cycles
id                      The identity permutation
inverse                 Inverse of a permutation
length.word             Various vector-like utilities for permutation
                        objects.
megaminx                megaminx
megaminx_plotter        Plotting routine for megaminx sequences
nullperm                Null permutations
orbit                   Orbits of integers
perm_matrix             Permutation matrices
permorder               The order of a permutation
permutation             Functions to create and coerce word objects and
                        cycle objects
permutations-package    The Symmetric Group: Permutations of a Finite
                        Set
print.permutation       Print methods for permutation objects
rperm                   Random permutations
sgn                     Sign of a permutation
shape                   Shape of a permutation
size                    Gets or sets the size of a permutation
tidy                    Utilities to neaten permutation objects
valid                   Functions to validate permutations
```

Author(s)

NA

Maintainer: Robin K. S. Hankin <hankin.robin@gmail.com>

Examples

```
a <- rperm(10,5)
b <- rperm(10,5)

a*b

inverse(a)
```

---

allperms *All permutations of a given size*

Description

Returns all n factorial permutations of a set

Usage

```
allperms(n)
```

Arguments

n	The size of the set, integer

Details

The function is very basic (the idiom is `word(t(partitions::perms(n)))`) but is here for completeness.

Author(s)

Robin K. S. Hankin

Examples

```
as.cycle(allperms(5))
```

as.function.permutation
Coerce a permutation to a function

Description

Coerce a permutation to an executable function

Usage

```
## S3 method for class 'permutation'
as.function(x, ...)
```

Arguments

x	permutation
...	further arguments (currently ignored)

Note

Multiplication of permutations loses associativity when using functional notation; see examples

Author(s)

Robin K. S. Hankin

Examples

```
x <- cyc_len(3)
y <- cyc_len(5)

xfun <- as.function(x)
yfun <- as.function(y)

stopifnot(xfun(yfun(2)) == as.function(y*x)(2)) # note transposition of x & y

# written in postfix notation one has the very appealing form x(fg) = (xf)g

# it's vectorized:
as.function(rperm(10,9))(1)
as.function(as.cycle(1:9))(sample(9))
```

c — *Concatenation of permutations*

Description

Concatenate words or cycles together

Usage

```
## S3 method for class 'word'
c(...)
## S3 method for class 'cycle'
c(...)
## S3 method for class 'permutation'
rep(x, ...)
```

Arguments

`...`	In the methods for `c()`, objects to be concatenated. Must all be of the same type: either all `word`, or all `cycle`
`x`	In the method for `rep()`, a permutation object

Note

The methods for `c()` do not attempt to detect which type (word or cycle) you want as conversion is expensive.

Function `rep.permutation()` behaves like `base::rep()` and takes the same arguments, eg `times` and `each`.

Author(s)

Robin K. S. Hankin

See Also

`size`

Examples

```
x <- as.cycle(1:5)
y <- cycle(list(list(1:4,8:9),list(1:2)))


# concatenate cycles:
c(x,y)

# concatenate words:
c(rperm(5,3),rperm(6,9))   # size adjusted to maximum size of args
```

```
# repeat words:
rep(x, times=3)
```

`cayley` *Cayley tables for permutation groups*

Description

Produces a nice Cayley table for a subgroup of the symmetric group on n elements

Usage

```
cayley(x)
```

Arguments

`x`	A vector of permutations in cycle form

Details

Cayley's theorem states that every group G is isomorphic to a subgroup of the symmetric group acting on G. In this context it means that if we have a vector of permutations that comprise a group, then we can nicely represent its structure using a table.

If the set `x` is not closed under multiplication and inversion (that is, if `x` is not a group) then the function may misbehave. No argument checking is performed, and in particular there is no check that the elements of `x` are unique, or even that they include an identity.

Value

A square matrix giving the group operation

Author(s)

Robin K. S. Hankin

Examples

```
## cyclic group of order 4:
cayley(as.cycle(1:4)^(0:3))

## Klein group:
K4 <- as.cycle(c("()","(12)(34)","(13)(24)","(14)(23)"))
names(K4) <- c("00","01","10","11")
```

```
cayley(K4)


## S3, the symmetric group on 3 elements:
S3 <- as.cycle(c(
    "()",
    "(12)(35)(46)", "(13)(26)(45)",
    "(14)(25)(36)", "(156)(243)", "(165)(234)"
))
names(S3) <- c("()","(ab)","(ac)","(bc)","(abc)","(acb)")
cayley(S3)


## Now an example from the onion package, the quaternion group:
## Not run:
 library(onion)
 a <- c(H1,-H1,Hi,-Hi,Hj,-Hj,Hk,-Hk)
 X <- word(sapply(1:8,function(k){sapply(1:8,function(l){which((a*a[k])[l]==a)})}))
 cayley(X)  # a bit verbose; rename the vector:
 names(X) <- letters[1:8]
 cayley(X)  # more compact

## End(Not run)
```

---

commutator *Group-theoretic commutator and group action*

Description

Group-theoretic commutator, defined as $[x,y]=x^{-1}y^{-1}xy$

Usage

```
commutator(x, y)
```

Arguments

x,y	Permutation objects, coerced to word

Author(s)

Robin K. S. Hankin

See Also

group_action

Examples

```
x <- rperm(10,7)
y <- rperm(10,8)
z <- rperm(10,9)

uu <-
commutator(commutator(x,y),z^x) *
commutator(commutator(z,x),y^z) *
commutator(commutator(y,z),x^y)

stopifnot(all(is.id(uu)))  # this is the  Hall-Witt identity
```

---

conjugate *Are two permutations conjugate?*

Description

Returns TRUE if two permutations are conjugate and FALSE otherwise.

Usage

```
are_conjugate(x, y)
are_conjugate_single(a,b)
```

Arguments

x,y,a,b	Objects of class permutation, coerced to cycle form

Details

Two permutations are conjugate if and only if they have the same shape. Function `are_conjugate()` is vectorized and user-friendly; function `are_conjugate_single()` is lower-level and operates only on length-one permutations.

The reason that `are_conjugate_single()` is a separate function and not bundled inside `are_conjugate()` is that dealing with the identity permutation is a pain in the arse.

Value

Returns a vector of Booleans

Note

The functionality detects conjugateness by comparing the shapes of two permutations; permutations are coerced to cycle form because function `shape()` does.

Author(s)

Robin K. S. Hankin

See Also

group_action,shape

Examples

```
are_conjugate(rperm(20,3),rperm(20,3))

rperm(20,3) %~% cycle(1:3)

z <- rperm(300,4)
stopifnot(all(are_conjugate(z,id)==is.id(z)))

data(megaminx)
stopifnot(all(are_conjugate(megaminx,megaminx^as.cycle(sample(129)))))
```

---

cyclist *details of cyclists*

Description

Various functionality to deal with cyclists

Usage

```
vec2cyclist_single(p)
vec2cyclist_single_cpp(p)
remove_length_one(x)
cyclist2word_single(cyc,n)
nicify_cyclist(x,rm1=TRUE, smallest_first=TRUE)
```

Arguments

p	Integer vector, interpreted as a word
x,cyc	A cyclist
n	In function `cycle2word_single()`, the size of the permutation to induce
rm1,smallest_first	In function `nicify_cyclist()`, Boolean, governing whether or not to remove length-1 cycles, and whether or not to place the smallest element in each cycle first (non-default values are used by `standard_cyclist()`)

Details

A *cyclist* is an object corresponding to a permutation P. It is a list with elements that are integer vectors corresponding to the cycles of P. This object is informally known as a cyclist, but there is no S3 class corresponding to it.

An object of S3 class *cycle* is a (possibly named) list of cyclists. NB: there is an unavoidable notational clash here. When considering a single permutation, "cycle" means group-theoretic cycle; when considering R objects, "cycle" means "an R object of class *cycle* whose elements are permutations written in cycle form".

The elements of a cyclist are the disjoint group-theoretic cycles. Note the redundancies inherent: firstly, because the cycles commute, their order is immaterial (and a list is ordered); and secondly, the cycles themselves are invariant under cyclic permutation. Heigh ho.

A cyclist may be poorly formed in a number of ways: the cycles may include repeats, or contain elements which are common to more than one cycle. Such problems are detected by `cycle.valid()`. Also, there are less serious problems: the cycles may include length-one cycles; the cycles may start with an element that is not the smallest. These issues are dealt with by `nicify_cyclist()`.

- Function `nicify_cyclist()` takes a cyclist and puts it in a nice form but does not alter the permutation. It takes a cyclist and removes length-one cycles; then orders each cycle so that the smallest element appears first (that is, it changes `(523)` to `(235)`). It then orders the cycles by the smallest element.
- Function `remove_length_one()` takes a cyclist and removes length-one cycles from it.
- Function `vec2cyclist_single()` takes a vector of integers, interpreted as a word, and converts it into a cyclist. Length-one cycles are discarded.
- Function `vec2cyclist_single_cpp()` is a placeholder for a function that is not yet written.
- Function `cyclist2word_single()` takes a cyclist and returns a vector corresponding to a single word. This function is not intended for everyday use; function `cycle2word()` is much more user-friendly.
- Function `char2cyclist_single()` takes a character string like "`(342)(19)`" and turns it into a cyclist, in this case `list(c(3,4,2),c(1,9))`. This function returns a cyclist which is not necessarily canonicalized: it might have length-one cycles, and the cycles themselves might start with the wrong number or be incorrectly ordered. It attempts to deal with absence of commas in a sensible way, so "`(18,19)(2,5)`" is dealt with appropriately too. The function is insensitive to spaces. Also, one can give it an argument which does not correspond to a cycle object, eg `char2cyclist_single("(94)(32)(19)(1)")` (in which "9" is repeated). The function does not return an error, but to catch this kind of problem use `char2cycle()` which calls the validity checks.

  The user should use `char2cycle()` which executes validity checks and coerces to a cycle object.

Author(s)

Robin K. S. Hankin

See Also

`as.cycle`,`fbin`,`valid`

Examples

```
vec2cyclist_single(c(7,9,3,5,8,6,1,4,2))

char2cyclist_single("(342)(19)")

nicify_cyclist(list(c(4, 6), c(7), c(2, 5, 1), c(8, 3)))
nicify_cyclist(list(c(4, 6), c(7), c(2, 5, 1), c(8, 3)),rm1=TRUE)


cyclist2word_single(list(c(1,4,3),c(7,8)))
```

`derangement` *Tests for a permutation being a derangement*

Description

A *derangement* is a permutation which leaves no element fixed.

Usage

```
is.derangement(x)
```

Arguments

`x`	Object to be tested

Value

A vector of Booleans corresponding to whether the permutations are derangements or not.

Note

The identity permutation is problematic because it potentially has zero size.

The identity element is not a derangement, although the (zero-size) identity cycle and permutation both return `TRUE` under the natural R idiom `all(P != seq_len(size(P)))`.

Author(s)

Robin K. S. Hankin

See Also

`id`

Examples

```
is.derangement(rperm(16,4))
```

dodecahedron *The dodecahedron group*

Description

Permutations comprising the dodecahedron group on either its faces or its edges; also the full dodecahedron group

Details

The package provides a number of objects for investigating dodecahedral groups:

Object `dodecahedron_face` is a cycle object with 60 elements corresponding to the permutations of the faces of a dodecahedron, numbered 1-12 as in the megaminx net. Object `dodecahedron_edge` is the corresponding object for permuting the edges of a dodecahedron. The edges are indexed by the lower of the two adjoining facets on the megaminx net.

Objects `full_dodecahedron_face` and `full_dodecahedron_edge` give the 120 elements of the full dodecahedron group, that is, the dodecahedron group including reflections. NB: these objects are **not** isomorphic to S5.

Note

File `zzz_dodecahedron.R` is not really intended to be human-readable. The source file is in `inst/dodecahedron_group.py` and `inst/full_dodecahedron_group.py` which contain documented python source code.

Examples

```
permprod(dodecahedron_face)
```

fbin *The fundamental bijection*

Description

Stanley defines the *fundamental bijection* on page 30.

Given $w = (14)(2)(375)(6)$, Stanley writes it in standard form (specifically: each cycle is written with its largest element first; cycles are written in increasing order of their largest element). Thus we obtain $(2)(41)(6)(753)$.

Then we obtain w^* from w by writing it in standard form an erasing the parentheses (that is, viewing the numbers as a *word*); here $w^* = 2416753$.

Given this, w may be recoverd by inserting a left parenthesis preceding every left-to-right maximum, and right parentheses where appropriate.

Usage

```
standard(cyc,n=NULL)
standard_cyclist(x,n=NULL)
fbin_single(vec)
fbin(W)
fbin_inv(cyc)
```

Arguments

vec	In function fbin_single(), an integer vector
W	In functions fbin() and fbin_inv(), an object of class permutation, coerced to word and cycle form respectively
cyc	In functions fbin_single() and standard(), permutation object coerced to cycle form
n	In function standard() and standard_cyclist(), size of the partition to assume, with default NULL meaning to use the largest element of any cycle
x	In function standard_cyclist(), a cyclist

Details

The user-friendly functions are fbin() and fbin_inv() which perform Stanley's "fundamental bijection". Function fbin() takes a word object and returns a cycle; function fbin_inv() takes a cycle and returns a word.

The other functions are low-level helper functions that are not really intended for the user (except possibly standard(), which puts a cycle object in standard order in list form).

Author(s)

Robin K. S. Hankin

References

R. P. Stanley 2011 *Enumerative Combinatorics*

See Also

nicify_cyclist

Examples

```
# Stanley's example w:
standard(cycle(list(list(c(1,4),c(3,7,5)))))

w_hat <- c(2,4,1,6,7,5,3)

fbin(w_hat)
fbin_inv(fbin(w_hat))
```

```
x <- rperm(40,9)
stopifnot(all(fbin(fbin_inv(x))==x))
stopifnot(all(fbin_inv(fbin(x))==x))
```

---

fixed *Fixed elements*

Description

Finds which elements of a permutation object are fixed

Usage

```
## S3 method for class 'word'
fixed(x)
## S3 method for class 'cycle'
fixed(x)
```

Arguments

x	Object of class `word` or `cycle`

Value

Returns a Boolean vector corresponding to the fixed elements of a permutation.

Note

The function is vectorized; if given a vector of permutations, `fixed()` returns a Boolean vector showing which elements are fixed by *all* of the permutations.

This function has two methods: `fixed.word()` and `fixed.cycle()`, neither of which coerce.

Author(s)

Robin K. S. Hankin

See Also

tidy

Examples

```
fixed(as.cycle(1:3)+as.cycle(8:9))   # elements 4,5,6,7 are fixed
fixed(id)


data(megaminx)
fixed(megaminx)
```

---

get1 *Retrieve particular cycles or components of cycles*

Description

Given an object of class `cycle`, function `get1()` returns a representative of each of the disjoint cycles in the object's elements. Function `get_cyc()` returns the cycle containing a specific element.

Usage

```
get1(x,drop=TRUE)
get_cyc(x,elt)
```

Arguments

`x`	permutation object (coerced to `cycle` class)
`drop`	In function `get1()`, argument `drop` controls the behaviour if `x` is length 1. If `drop` is `TRUE`, then a vector of representative elements is returned; if `FALSE`, then a list with one vector element is returned
`elt`	Length-one vector interpreted as a permutation object

Author(s)

Robin K. S. Hankin

Examples

```
data(megaminx)
get1(megaminx)
get1(megaminx[1])
get1(megaminx[1],drop=TRUE)

get_cyc(megaminx,11)
```

id — *The identity permutation*

Description

The *identity permutation* leaves every element fixed

Usage

```
is.id(x)
is.id_single_cycle(x)
## S3 method for class 'cycle'
is.id(x)
## S3 method for class 'list'
is.id(x)
## S3 method for class 'word'
is.id(x)
```

Arguments

`x`	Object to be tested

Details

The identity permutation is problematic because it potentially has zero size.

Value

The variable `id` is a *cycle* as this is more convenient than a zero-by-one matrix.

Function `is.id()` returns a Boolean with `TRUE` if the corresponding element is the identity, and `FALSE` otherwise. It dispatches to either `is.id.cycle()` or `is.id.word()` as appropriate.

Function `is.id.list()` tests a cyclist for identityness.

Note

The identity permutations documented here are distinct from the null permutations documented at `nullperm.Rd`.

Author(s)

Robin K. S. Hankin

See Also

`is.derangement`,`nullperm`

Examples

```
is.id(id)

as.word(id)  # weird

x <- rperm(10,4)
x[3] <- id
is.id(x*inverse(x))
```

`inverse` *Inverse of a permutation*

Description

Calculates the inverse of a permutation in either word or cycle form

Usage

```
inverse(x)
## S3 method for class 'word'
inverse(x)
## S3 method for class 'cycle'
inverse(x)
inverse_word_single(W)
inverse_cyclist_single(cyc)
```

Arguments

`x`	Object of class `permutation` to be inverted
`W`	In function `inverse_word_single()`, a vector corresponding to a permutation in word form (that is, one row of a `word` object)
`cyc`	In function `inverse_cyclist_single()`, a cyclist to be inverted

Details

The package provides methods to invert objects of class `word` (the R idiom is `W[W] <-seq_along(W)`) and also objects of class `cycle` (the idiom is `lapply(cyc,function(o){c(o[1],rev(o[-1]))})`).

The user should use `inverse()` directly, which dispatches to either `inverse.word()` or `inverse.cycle()` as appropriate.

Sometimes, using idiom such as `x^-1` or `id/x` gives neater code, although these may require coercion between word form and cycle form.

Value

Function `inverse()` returns an object of the same class as its argument.

Author(s)

Robin K. S. Hankin

See Also

cycle_power

Examples

```
x <- rperm(10,6)
inverse(x)

all(is.id(x*inverse(x)))  # should be TRUE

inverse(as.cycle(matrix(1:8,9,8)))
```

---

length *Various vector-like utilities for permutation objects.*

Description

Various vector-like utilities for permutation objects such as `length`, `names()`, etc

Usage

```
## S3 method for class 'word'
length(x)
## S3 replacement method for class 'permutation'
length(x) <- value
## S3 method for class 'word'
names(x)
## S3 replacement method for class 'word'
names(x) <- value
```

Arguments

x	permutation object
value	In function `names<-.word()`, the new names

Details

These functions have methods only for word objects; cycle objects use the methods for lists. It is easy to confuse the *length* of a permutation with its `size`.

It is not possible to set the length of a permutation; this is more trouble than it is worth.

Author(s)

Robin K. S. Hankin

See Also

size

Examples

```
x <- rperm(9,5)
names(x) <- letters[1:9]

data(megaminx)
length(megaminx)   # the megaminx group has 12 generators, one per face.
size(megaminx)     # the megaminx group is a subgroup of S_129.


names(megaminx) <- NULL   # prints more nicely.
megaminx
```

---

megaminx *megaminx*

Description

A set of generators for the megaminx group

Details

Each element of `megaminx` corresponds to a clockwise turn of 72 degrees. See the vignette for more details.

`megaminx[, 1]`	W	White
`megaminx[, 2]`	Pu	Purple
`megaminx[, 3]`	DY	Dark Yellow
`megaminx[, 4]`	DB	Dark Blue
`megaminx[, 5]`	R	Red
`megaminx[, 6]`	DG	Dark Green
`megaminx[, 7]`	LG	Light Green
`megaminx[, 8]`	O	Orange
`megaminx[, 9]`	LB	Light Blue
`megaminx[,10]`	LY	Light Yellow
`megaminx[,11]`	Pi	Pink
`megaminx[,12]`	Gy	Gray

Vector `megaminx_colours` shows what colour each facet has at START. Object `superflip` is a megaminx operation that flips each of the 30 edges.

Examples

```
data(megaminx)
megaminx
megaminx^5  # should be the identity
inverse(megaminx)  # turn each face anticlockwise


megaminx_colours[permprod(megaminx)]  # risky but elegant...

W    # turn the White face one click clockwise (colour names as per the
     # table above)


megaminx_colours[as.word(W,129)]      # it is safer to ensure a size-129 word;
megaminx_colours[as.word(W)]          # but the shorter version will work


# Now some superflip stuff:

X <- W * Pu^(-1) * W * Pu^2 * DY^(-2)
Y <- LG^(-1) * DB^(-1) * LB * DG
Z <- Gy^(-2) * LB * LG^(-1) * Pi^(-1) * LY^(-1)


sjc3 <- (X^6)^Y * Z^9  # superflip (Jeremy Clark)


p1 <- (DG^2 * W^4 * DB^3 * W^3 * DB^2 * W^2 * DB^2 * R * W * R)^3
m1 <- p1^(Pi^3)

p2 <- (O^2 * LG^4 * DB^3 * LG^3 * DB^2 * LG^2 * DB^2 * DY * LG * DY)^3
m2 <- p2^(DB^2)

p3 <- (LB^2 * LY^4 * Gy * Pi^3 * LY * Gy^4)^3
m3 <- p3^LB

# m1,m2 are 32 moves, p3 is 20, total = 84

stopifnot(m1+m2+m3==sjc3)
```

`megaminx_plotter` *Plotting routine for megaminx sequences*

Description

Plots a coloured diagram of a dodecahedron net representing a megaminx

Usage

```
megaminx_plotter(megperm=id,offset=c(0,0),M=diag(2),setup=TRUE,...)
```

Arguments

megperm	Permutation to be plotted
offset,M	Offset and transformation matrix, see details
setup	Boolean, with default TRUE meaning to set up the plot with a plot() statement, and FALSE meaning to plot the points on a pre-existing canvas
...	Further arguments passed to polygon()

Details

Function megaminx_plotter() plots a coloured diagram of a dodecahedron net representing a megaminx. The argument may be specified as a sequence of turns that are applied to the megaminx from *START*.

The function uses rather complicated internal variables pentagons, triangles, and quads whose meaning and genesis is discussed in heavily-documented file inst/guide.R.

The diagram is centered so that the common vertex of triangles 28 and 82 is at $(0, 0)$.

Author(s)

Robin K. S. Hankin

Examples

```
data("megaminx")

megaminx_plotter()  # START
megaminx_plotter(W) # after turning the White face one click
megaminx_plotter(superflip)

size <- 0.95
o <- 290

## Not run:
pdf(file="fig1.pdf")
megaminx_plotter(M=size*diag(2),offset=c(-o,0),setup=TRUE)
megaminx_plotter(W,M=size*diag(2),offset=c(+o,0),setup=FALSE)
dev.off()

pdf(file="fig2.pdf")
p <- permprod(sample(megaminx,100,replace=TRUE))
megaminx_plotter(p,M=size*diag(2),offset=c(-o,0),setup=TRUE)
megaminx_plotter(superflip,M=size*diag(2),offset=c(+o,0),setup=FALSE)
```

```
dev.off()

## End(Not run)
```

---

nullperm *Null permutations*

Description

Null permutations are the equivalent of NULL

Usage

```
nullcycle
nullword
```

Format

Object nullcycle is an empty list coerced to class cycle, specfically cycle(list())

Object nullword is a zero-row matrix, coerced to word, specifically word(matrix(integer(0),0,0))

Details

These objects are here to deal with the case where a length-zero permutation is extracted. The behaviour of these null objects is not entirely consistent.

Note

The objects documented here are distinct from the identity permutation, id, documented separately.

See Also

id

Examples

```
rperm(10,4)[0]  # null word

as.cycle(1:5)[0]  # null cycle

data(megaminx)
c(NULL,megaminx)      # probably not what the user intended...
c(nullcycle,megaminx) # more useful.
c(id,megaminx)        # also useful.
```

Ops.permutation *Arithmetic Ops Group Methods for permutations*

Description

Allows arithmetic operators to be used for manipulation of permutation objects such as addition, multiplication, division, integer powers, etc.

Usage

```
## S3 method for class 'permutation'
Ops(e1, e2)
cycle_power(x,pow)
cycle_power_single(x,pow)
cycle_sum(e1,e2)
cycle_sum_single(c1,c2)
group_action(e1,e2)
word_equal(e1,e2)
word_prod(e1,e2)
word_prod_single(e1,e2)
permprod(x)
vps(vec,pow)
ccps(n,pow)
helper(e1,e2)
```

Arguments

x,e1,e2	Objects of class "permutation"
c1,c2	Objects of class cycle
pow	Integer vector of powers
vec	In function vps(), a vector of integers corresponding to a cycle
n	In function ccps(), the integer power to which cycle(seq_len(n)) is to be raised; may be positive or negative.

Details

The function Ops.permutation() passes binary arithmetic operators ("+", "*", "/", "^", and "==") to the appropriate specialist function.

Multiplication, as in a*b, is effectively word_prod(a,b); it coerces its arguments to word form (because a*b = b[a]).

Raising permutations to integer powers, as in a^n, is cycle_power(a,n); it coerces a to cycle form and returns a cycle. Negative and zero values of n operate as expected. Function cycle_power() is vectorized; it calls cycle_power_single(), which is not. This calls vps() ("Vector Power Single"), which checks for simple cases such as pow=0 or the identity permutation; and function vps() calls function ccps() which performs the actual number-theoretic manipulation to raise a cycle to a power.

Raising a permutation to the power of another permutation, as in a^b, is idiom for inverse(b)*a*b, sometimes known as *group action*; the notation is motivated by the identities x^(yz)=(x^y)^z and (xy)^z=x^z*y^z.

Permutation addition, as in a+b, is defined if the cycle representations of the addends are disjoint. The sum is defined as the permutation given by juxtaposing the cycles of a with those of b. Note that this operation is commutative. If a and b do not have disjoint cycle representations, an error is returned. This is useful if you want to guarantee that two permutations commute (NB: permutation a commutes with a^i for i any integer, and in particular a commutes with itself. But a+a returns an error: the operation checks for disjointness, not commutativity).

Permutation "division", as in a/b, is a*inverse(b). Note that a/b*c is evaluated left to right so is equivalent to a*inverse(b)*c. See note.

Function helper() sorts out recycling for binary functions, the behaviour of which is inherited from cbind(), which also handles the names of the returned permutation.

Value

None of these functions are really intended for the end user: use the ops as shown in the examples section.

Note

The class of the returned object is the appropriate one.

It would be nice to define a unary operator which inverted a permutation. I do not like "id/x" to represent a permutation inverse: the idiom introduces an utterly redundant object ("id"), and forces the use of a binary operator where a unary operator is needed.

The natural unary operator would be the exclamation mark, !x. However, redefining the exclamation mark to give permutation inverses, while possible, is not desirable because its precedence is too low. One would like !x*y to return inverse(x)*y but instead standard precendence rules means that it returns inverse(x*y). This caused such severe cognitive dissonance that I removed it.

There does not appear to be a way to define a new unary operator due to the construction of the parser.

Author(s)

Robin K. S. Hankin

Examples

```
x <- rperm(20,9) # word form
y <- rperm(20,9) # word form

x*y  # word form

x^5  # coerced to cycle form

x^as.cycle(1:5)  # group action; coerced to word.
```

```
x*inverse(x) == id  # all TRUE



# the 'sum' of two permutations is defined if their cycles are disjoint:
as.cycle(1:4) + as.cycle(7:9)

data(megaminx)
megaminx[1] + megaminx[7:12]
```

---

orbit *Orbits of integers*

Description

Finds the orbit of a given integer

Usage

```
orbit_single(c1,n1)
orbit(cyc,n)
```

Arguments

c1,n1	In (low-level) function orbit_single(), a cyclist and an integer vector respectively
cyc,n	In (vectorized) function orbit(), cyc is coerced to a cycle, and n is an integer vector

Value

Given a cyclist c1 and integer n1, function orbit_single() returns the single cycle containing integer n1. This is a low-level function, not intended for the end-user.

Function orbit() is the vectorized equivalent of orbit_single().

Author(s)

Robin K. S. Hankin

See Also

fixed

Examples

```
data(megaminx)
orbit(megaminx,13)


# orbit() is vectorized:
x <- cycle(list(list(a=1:2,4:6,8:10)))
orbit(x,1:10)
```

---

permorder *The order of a permutation*

Description

Returns the order of a permutation P: the smallest strictly positive integer n for which P^n is the identity.

Usage

```
permorder(x, singly = TRUE)
```

Arguments

x	Permutation, coerced to `cycle` form
singly	Boolean, with default `TRUE` meaning to return the order of each element of the vector, and `FALSE` meaning to return the order of the vector itself (that is, the smallest strictly positive integer for which `all(x^n==id)`).

Details

Coerces its argument to cycle form.

The order of the identity permutation is 1.

Note

Uses `mLCM()` from the `numbers` package.

Author(s)

Robin K. S. Hankin

See Also

`sgn`

Examples

```
x <- rperm(5,20)
permorder(x)
permorder(x,FALSE)

stopifnot(all(is.id(x^permorder(x))))
stopifnot(is.id(x^permorder(x,FALSE)))
```

---

permutation — *Functions to create and coerce word objects and cycle objects*

Description

Functions to create permutation objects. `permutation` is a virtual class.

Usage

```
word(M)
permutation(x)
is.permutation(x)
cycle(x)
is.word(x)
is.cycle(x)
as.word(x,n=NULL)
as.cycle(x)
cycle2word(x,n=NULL)
char2cycle(char)
cyc_len(n)
shift_cycle(n)
## S3 method for class 'word'
as.matrix(x,...)
```

Arguments

`M`	In function `word()`, a matrix with rows corresponding to permutations in word form
`x`	See details
`n`	In functions `as.word()` and `cycle2word()`, the size of the word to return; in function `cyc_len()`, the length of the cycle to return
`char`	In function `char2cycle()` a character vector which is coerced to a cycle object
`...`	Further arguments passed to `as.matrix()`

Details

Functions word() and cycle() are rather formal functions which make no attempt to coerce their arguments into sensible forms. The user should use permutation(), which detects the form of the input and dispatches to as.word() or as.cycle(), which are much more user-friendly.

Functions word() and cycle() are the only functions in the package which assign class word or cycle to an object.

A *word* is a matrix whose rows correspond to permutations in word format.

A *cycle* is a list whose elements correspond to permutations in cycle form. A cycle object comprises elements which are informally dubbed 'cyclists'. A cyclist is a list of integer vectors corresponding to the cycles of the permutation.

Function cycle2word() converts cycle objects to word objects.

Function shift_cycle() is a convenience wrapper for as.cycle(seq_len(n)); cyc_len() is a synonym.

It is a very common error (at least, it is for me) to use cycle() when you meant as.cycle().

The print method is sensitive to the value of option 'print_word_as_cycle', documented at print.Rd.

Function as.matrix.word() coerces a vector of permutations in word form to a matrix, each row of which is a word. To get a permutation matrix (that is, a square matrix of ones and zeros with exactly one entry of 1 in each row and each column), use perm_matrix().

Value

Returns a cycle object or a word object

Author(s)

Robin K. S. Hankin

See Also

cyclist

Examples

```
word(matrix(1:8,7,8))    # default print method displays cycle form


cycle(list(list(c(1,8,2),c(3,6)),list(1:2, 4:8)))

char2cycle(c("(1,4)(6,7)","(3,4,2)(8,19)", "(56)","(12345)(78)","(78)"))

jj <- c(4,2,3,1)

as.word(jj)
as.cycle(jj)

as.cycle(1:2)*as.cycle(1:8) == as.cycle(1:8)*as.cycle(1:2)  # FALSE!
```

```
x <- rperm(10,7)
y <- rperm(10,7)
as.cycle(commutator(x,y))

cycle(sapply(seq_len(9),cyc_len))
```

`perm_matrix` *Permutation matrices*

Description

Given a permutation, coerce to word form and return the corresponding permutation matrix

Usage

```
perm_matrix(p)
is.perm_matrix(M)
pm_to_perm(M)
```

Arguments

`p`	Permutation, coerced to word form, of length 1
`M`	Permutation matrix

Details

Given a permutation `p` of size s, function `perm_matrix()` returns a square matrix with s rows and s columns. Entries are either 0 or 1; each row and each column has exactly one entry of 1 and the rest zero.

Row and column names of the permutation matrix are integers; this makes the printed version more compact.

Function `pm_to_perm()` takes a permutation matrix and returns the equivalent permutation in word form.

Note

Given a word `p` with size `s`, the idiom for `perm_matrix()` boils down to

```
M <- diag(s)
M[p,]
```

This is used explicitly in the `representations` vignette. There is another way:

```
M <- diag(s)
M[cbind(seq_len(s),p)] <- 1
M
```

which might be useful sometime.

Author(s)

Robin K. S. Hankin

See Also

permutation

Examples

```
perm_matrix(rperm(1,9))


p1 <- rperm(1,40)
M1 <- perm_matrix(p1)
p2 <- rperm(1,40)
M2 <- perm_matrix(p2)

stopifnot(is.perm_matrix(M1))

stopifnot(all(solve(M1) == perm_matrix(inverse(p1))))
stopifnot(all(M1 %*% M2 == perm_matrix(p1*p2)))


stopifnot(p1 == pm_to_perm(perm_matrix(p1)))

data("megaminx")
image(perm_matrix(permprod(megaminx)),asp=1,axes=FALSE)
```

---

print *Print methods for permutation objects*

Description

Print methods for permutation objects with matrix-like printing for words and bracket notation for cycle objects.

Usage

```
## S3 method for class 'cycle'
print(x, ...)
## S3 method for class 'word'
print(x, h = getOption("print_word_as_cycle"), ...)
as.character_cyclist(y,comma=TRUE)
```

Arguments

x	Object of class `permutation` with `word` objects dispatched to `print.word()` and `cycle` objects dispatched to `print.cycle()`
h	Boolean, with default `TRUE` meaning to coerce words to cycle form before printing. See details
...	Further arguments (currently ignored)
y,comma	In `as.character.cyclist()`, argument `y` is a list of cycles (a cyclist); and `comma` is Boolean, specifying whether to include a comma in the output

Details

Printing of word objects is controlled by `options("print_word_as_cycle")`. The default behaviour is to coerce a word to cycle form and print that, with a notice that the object itself was coerced from word.

If `options("print_word_as_cycle")` is `FALSE`, then objects of class word are printed as a matrix with rows being the permutations and fixed points indicated with a dot.

Function `as.character_cyclist()` is an internal function used by `print.cycle()`, and is not really designed for the end-user. It takes a cyclist and returns a character string.

Value

Returns its argument invisibly, after printing it.

Author(s)

Robin K. S. Hankin

See Also

`nicify_cyclist`

Examples

```
# generate a permutation in *word* form:
x <- rperm(4,9)

# default behaviour is to print in cycle form irregardless:
x
```

```
# change default using options():
options(print_word_as_cycle=FALSE)

# objects in word form now printed using matrix notation:
x

# printing of cycle form objects not altered:
as.cycle(x)

# restore default:
options(print_word_as_cycle=TRUE)

as.character_cyclist(list(1:4,10:11,20:33))  # x a cyclist;
as.character_cyclist(list(c(1,5,4),c(2,2)))  # does not check for consistency
as.character_cyclist(list(c(1,5,4),c(2,9)),comma=FALSE)
```

---

rperm *Random permutations*

Description

Create a word object of random permutations

Usage

```
rperm(n,r,moved=NA)
```

Arguments

n	Number of permutations to create
r	Size of permutations
moved	Integer specifying how many elements can move (that is, how many elements do not map to themselves), with default NA meaning to choose a permutation at random. This is useful if you want a permutation that has a compact cycle representation

Value

Returns an object of class word

Note

Argument moved specifies a *maximum* number of elements that do not map to themselves; the actual number of non-fixed elements might be lower (as some elements might map to themselves).

Author(s)

Robin K. S. Hankin

See Also

size

Examples

```
rperm(30,9)
as.cycle(rperm(30,9))


rperm(10,9,2)
```

---

sgn *Sign of a permutation*

Description

The sign of a permutation is ± 1 depending on whether it is even or odd

Usage

```
sgn(x)
is.even(x)
is.odd(x)
```

Arguments

x	permutation object

Details

Coerces to cycle form

Author(s)

Robin K. S. Hankin

See Also

shape

Examples

```

sgn(id)  # always problematic

sgn(rperm(10,5))

x <- rperm(40,6)
y <- rperm(40,6)
```

```
stopifnot(all(sgn(x*y) == sgn(x)*sgn(y)))   # sgn() is a homomorphism


z <- as.cycle(rperm(20,9,5))
z[is.even(z)]
z[is.odd(z)]
```

---

shape *Shape of a permutation*

Description

Returns the shape of a permutation. If given a word, it coerces to cycle form.

Usage

```
shape(x, drop = TRUE,id1=TRUE)
shape_cyclist(cyc,id1=TRUE)
padshape(x, drop = TRUE, n=NULL)
shapepart(x)
shapepart_cyclist(cyc,n=NULL)
```

Arguments

x	Object of class `cycle` (if not, coerced)
cyc	A cyclist
n	Integer governing the size of the partition assumed, with default `NULL` meaning to use the largest element
drop	Boolean, with default `TRUE` meaning to unlist if possible
id1	Boolean, with default `TRUE` in function `shape_cyclist()` meaning that the shape of the identity is "1" and `FALSE` meaning that the shape is `NULL`

Value

Function `shape()` returns a list with elements representing the lengths of the component cycles.

Function `shapepart()` returns an object of class `partition` showing the permutation as a set partition of disjoint cycles.

Note

Function `shape()` returns the lengths of the cycles in the order returned by `nicify_cyclist()`, so not necessarily in increasing or decreasing order.

Author(s)

Robin K. S. Hankin

See Also

size

Examples

```
shape(rperm(10,9)) # coerced to cycle



data(megaminx)

shape(megaminx)
jj <- megaminx*megaminx[1]

stopifnot(identical(shape(jj),shape(tidy(jj))))  #tidy() does not change shape

shapepart(rperm(10,5))

shape_cyclist(list(1:4,8:9))
shapepart_cyclist(list(1:4,8:9))
```

---

size — *Gets or sets the size of a permutation*

Description

The ‘size’ of a permutation is the cardinality of the set for which it is a bijection.

Usage

```
size(x)
addcols(M,n)
## S3 method for class 'word'
size(x)
## S3 method for class 'cycle'
size(x)
## S3 replacement method for class 'word'
size(x) <- value
## S3 replacement method for class 'cycle'
size(x) <- value
```

Arguments

`x`	A permutation object
`M`	A matrix that may be coerced to a word
`n,value`	the size to set to, an integer

Details

For a `word` object, the *size* is equal to the number of columns. For a `cycle` object, it is equal to the largest element of any cycle.

Function `addcols()` is a low-level function that operates on, and returns, a matrix. It just adds columns to the right of `M`, with values equal to their column numbers, thus corresponding to fixed elements. The resulting matrix has `n` columns. This function cannot remove columns, so if `n<ncol(M)` an error is returned.

Setting functions cannot decrease the size of a permutation; use `trim()` for this.

It is meaningless to change the size of a `cycle` object. Trying to do so will result in an error. But you can coerce cycle objects to `word` form, and change the size of that.

Author(s)

Robin K. S. Hankin

See Also

`fixed`

Examples

```
x <- rperm(10,8)
size(x)
size(x) <- 15

size(as.cycle(1:5) + as.cycle(100:101))

size(id)
```

---

tidy *Utilities to neaten permutation objects*

Description

Various utilities to neaten word objects by removing fixed elements

Usage

```
tidy(x)
trim(x)
```

Arguments

x   Object of class word, or in the case of tidy(), coerced to class word

Details

Function trim() takes a word and, starting from the right, strips off columns corresponding to fixed elements until it finds a non-fixed element. This makes no sense for cycle objects; if x is of class cycle, an error is returned.

Function tidy() is more aggressive. This firstly removes *all* fixed elements, then renames the non-fixed ones to match the new column numbers. The map is an isomorphism (sic) with respect to composition.

Value

Returns an object of class word

Note

Results in empty (that is, zero-column) words if a vector of identity permutations is given

Author(s)

Robin K. S. Hankin

See Also

fixed,size,nicify_cyclist

Examples

```
tidy(as.cycle(5:3)+as.cycle(7:9))
as.cycle(tidy(c(as.cycle(1:2),as.cycle(6:7))))


nicify_cyclist(list(c(4,6), c(7), c(2,5,1), c(8,3)))

data(megaminx)
tidy(megaminx)  # has 120 columns, not 129
stopifnot(all(unique(sort(unlist(as.cycle(tidy(megaminx)),recursive=TRUE)))==1:120))

jj <- megaminx*megaminx[1]
stopifnot(identical(shape(jj),shape(tidy(jj))))  #tidy() does not change shape
```

valid *Functions to validate permutations*

Description

Functions to validate permutation objects: if valid, return TRUE and if not valid, generate a warning() and return FALSE.

Function singleword.valid() takes an integer vector, interpreted as a word, and checks that it is a permutation of seq_len(max(x)).

Function cycle.valid() takes a cyclist and checks for disjoint cycles of strictly positive integers with no repeats.

Usage

```
singleword_valid(w)
cyclist_valid(x)
```

Arguments

x	In function cycle_valid(), a cyclist
w	In function singleword_valid(), an integer vector

Value

Returns either TRUE, or stops with an informative error message

Author(s)

Robin K. S. Hankin

See Also

cyclist

Examples

```
singleword_valid(sample(1:9))       # TRUE
singleword_valid(c(3L,4L,2L,1L))    # TRUE
singleword_valid(c(3,4,2,1))        # FALSE (not integer)
singleword_valid(c(3L,3L,2L,1L))    # FALSE (3 repeated)

cyclist_valid(list(c(1,8,2),c(3,6))) # TRUE
cyclist_valid(list(c(1,8,2),c(3,6))) # FALSE ('8' is repeated)
cyclist_valid(list(c(1,8,1),c(3,6))) # FALSE ('1' is repeated)
cyclist_valid(list(c(0,8,2),c(3,6))) # FALSE (zero element)
```

Index