

# Package ‘magic’

February 14, 2012

**Version** 1.5-1

**Date** 6 Jan 2009

**Title** create and investigate magic squares

**Author** Robin K. S. Hankin

**Depends** R (>= 2.4.0), abind

**Description** a collection of efficient, vectorized algorithms for the creation and investigation of magic squares and hypercubes, including a variety of functions for the manipulation and analysis of arbitrarily dimensioned arrays. The package includes methods for creating normal magic squares of any order greater than 2. The ultimate intention is for the package to be a computerized embodiment all magic square knowledge, including direct numerical verification of properties of magic squares (such as recent results on the determinant of odd-ordered semimagic squares). Some antimagic functionality is included. The package also serves as a rebuttal to the often-heard comment ‘I thought R was just for statistics’.

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

**License** GPL-2

**Repository** CRAN

**Date/Publication** 2011-12-27 10:06:13

## R topics documented:

magic-package . . . . .	3
adiag . . . . .	3
allsubhypercubes . . . . .	5
allsums . . . . .	6
apad . . . . .	8
apl . . . . .	9
aplus . . . . .	10

arev	11
arot	12
arow	13
as.standard	14
cilleruelo	16
circulant	17
cube2	18
diag.off	19
do.index	20
eq	21
fnsd	22
force.integer	23
Frankenstein	24
hadamard	24
hendricks	25
hudson	25
is.magic	26
is.magichypercube	29
is.ok	32
is.square.palindromic	33
latin	34
lozenge	36
magic	37
magic.2np1	38
magic.4n	39
magic.4np2	39
magic.8	40
magic.constant	41
magic.prime	42
magic.product	43
magiccube.2np1	44
magiccubes	45
magichypercube.4n	45
magicplot	46
minmax	47
notmagic.2n	48
nqueens	48
Ollerenshaw	50
panmagic.4	50
panmagic.8	51
perfectcube5	52
perfectcube6	52
process	53
recurse	53
sam	54
shift	55
strachey	56
subsums	57

<i>magic-package</i>	3
transf . . . . .	59
<b>Index</b>	<b>60</b>

---

<i>magic-package</i>	<i>Magic squares and hypercubes; arbitrary dimensioned array manipulation</i>
----------------------	---

---

### Description

A collection of efficient, vectorized algorithms for the creation and investigation of magic squares and hypercubes, including a variety of functions for the manipulation and analysis of arbitrarily dimensioned arrays.

The package includes methods for creating normal magic squares of any order greater than 2. The ultimate intention is for the package to be a computerized embodiment all magic square knowledge, including direct numerical verification of properties of magic squares (such as recent results on the determinant of odd-ordered semimagic squares).

### Author(s)

Robin K. S. Hankin <rksh1@cam.ac.uk>

### Examples

```
magic(5)

a <- magiccube.2np1(1)
adiag(1,a)
apad(a,2,1)
allsubhypercubes(a)
arev(a)
apltake(a,c(2,2))
arot(a)
arow(a,1)
```

---

<i>adiag</i>	<i>Binds arrays corner-to-corner</i>
--------------	--------------------------------------

---

### Description

Array generalization of blockdiag()

### Usage

```
adiag(... , pad=as.integer(0), do.dimnames=TRUE)
```

**Arguments**

...	Arrays to be binded together
pad	Value to pad array with; note default keeps integer status of arrays
do.dimnames	Boolean, with default TRUE meaning to return dimnames if possible. Set to FALSE if performance is an issue

**Details**

Binds any number of arrays together, corner-to-corner. Because the function is associative provided pad is of length 1, this page discusses the two array case.

If  $x = \text{adiag}(a, b)$  and  $\text{dim}(a) = c(a_1, \dots, a_d)$ ,  $\text{dim}(b) = c(b_1, \dots, b_d)$ ; then  $\text{all}(\text{dim}(x) == \text{dim}(a) + \text{dim}(b))$  and  $x[1:a_1, \dots, 1:a_d] = a$  and  $x[(a_1+1):(a_1+b_1), \dots, (a_d+1):(a_d+b_d)] = b$ .

Dimnames are preserved, if both arrays have non-null dimnames, and `do.dimnames` is TRUE.

Argument `pad` is usually a length-one vector, but any vector is acceptable; standard recycling is used. Be aware that the output array (of dimension  $\text{dim}(a) + \text{dim}(b)$ ) is filled with (copies of) `pad` *before* `a` and `b` are copied. This can be confusing.

**Value**

Returns an array of dimensions  $\text{dim}(a) + \text{dim}(b)$  as described above.

**Note**

In  $\text{adiag}(a, b)$ , if `a` is a length-one vector, it is coerced to an array of dimensions  $\text{rep}(1, \text{length}(\text{dim}(b)))$ ; likewise `b`. If both `a` and `b` are length-one vectors, return  $\text{diag}(c(a, b))$ .

If `a` and `b` are arrays, function  $\text{adiag}()$  requires  $\text{length}(\text{dim}(a)) == \text{length}(\text{dim}(b))$  (the function does not guess which dimensions have been dropped; see examples section). In particular, note that vectors are not coerced except if of length one.

$\text{adiag}()$  is used when padding magic hypercubes in the context of evaluating subarray sums.

**Author(s)**

Peter Wolf with some additions by Robin Hankin

**See Also**

[subsums](#), [apad](#)

**Examples**

```
a <- array( 1, c(2, 2))
b <- array(-1, c(2, 2))
adiag(a, b)

## dropped dimensions can count:

b2 <- b1 <- b
dim(a) <- c(2, 1, 2)
```

```

dim(b1) <- c(2,2,1)
dim(b2) <- c(1,2,2)

dim(adiag(a,b1))
dim(adiag(a,b2))

## dimnames are preserved if not null:

a <- matrix(1,2,2,dimnames=list(col=c("red","blue"),size=c("big","small")))
b <- 8
dim(b) <- c(1,1)
dimnames(b) <- list(col=c("green"),size=c("tiny"))
adiag(a,b) #dimnames preserved
adiag(a,8) #dimnames lost because second argument has none.

## non scalar values for pad can be confusing:
q <- matrix(0,3,3)
adiag(q,q,pad=1:4)

## following example should make the pattern clear:
adiag(q,q,pad=1:36)

# Now, a use for arrays with dimensions of zero extent:
z <- array(dim=c(0,3))
colnames(z) <- c("foo","bar","baz")

adiag(a,z) # Observe how this has
           # added no (ie zero) rows to "a" but
           # three extra columns filled with the pad value

adiag(a,t(z))
adiag(z,t(z)) # just the pad value

```

---

allsubhypercubes

*Subhypercubes of magic hypercubes*


---

### Description

Extracts all subhypercubes from an n-dimensional hypercube.

### Usage

```
allsubhypercubes(a)
```

### Arguments

a                   The magic hypercube whose subhypercubes are computed

**Value**

Returns a list, each element of which is a subhypercube. Note that major diagonals are also returned (as n-by-1 arrays).

The names of the list are the extracted subhypercubes. Consider `a <- magichypercube.4n(1,d=4)` (so `n=4`) and if `jj <- allsubhypercubes(a)`, consider `jj[9]`. The name of `jj[9]` is "n-i+1,i,i,"; its value is a square matrix. The columns of `jj[9]` may be recovered by `a[n-i+1,i,i,]` with  $i = 1 \dots n$  (**NB**: that is, `jj[[9]] == cbind(a[n-1+1,1,1,], a[n-2+1,2,2,], a[n-3+1,3,3,], a[n-4+1,4,4,])` where `n=4`).

The list does not include the whole array.

**Note**

This function is a dog's dinner. It's complicated, convoluted, and needs an absurd use of the `eval(parse(text=...))` construction. Basically it sucks big time.

BUT...I cannot for the life of me see a better way that gives the same results, without loops, on hypercubes of arbitrary dimension.

On my 256MB Linuxbox, `allsubhypercubes()` cannot cope with `d` as high as 5, for `n=4`. Heigh ho.

**Author(s)**

Robin K. S. Hankin

**See Also**

[is.perfect](#)

**Examples**

```
a <- magichypercube.4n(1,d=4)
allsubhypercubes(a)
```

---

allsums

*Row, column, and two diagonal sums of arrays*

---

**Description**

Returns all rowsums, all columnsums, and all (broken) diagonal sums of a putative magic square.

**Usage**

```
allsums(m, func=NULL)
```

**Arguments**

m	The square to be tested
func	Function, with default NULL interpreted as <code>assum()</code> , to be applied to the square rowwise, columnwise, and diagonalwise

**Value**

Returns a list of four elements. In the following, “sums” means “the result of applying `func()`”.

rowsums	All $n$ row sums
colsums	All $n$ column sums
majors	All $n$ broken major diagonals (northwest-southeast). First element is the long (unbroken) major diagonal, tested by <code>is.magic()</code>
minors	All $n$ broken minor diagonals (northeast-southwest). First element is the long (unbroken) minor diagonal.

**Note**

If `func()` returns a vector, then the `allsums()` returns a list whose columns are the result of applying `func()`. See third and fourth examples below.

Used by `is.magic()` et seq.

The major and minor diagonals would benefit from being recoded in C.

**Author(s)**

Robin K. S. Hankin

**See Also**

[is.magic](#), [is.semimagic](#), [is.panmagic](#)

**Examples**

```
allsums(magic(7))
allsums(magic(7),func=max)

allsums(magic(7),func=range)
allsums(magic(7),func=function(x){x[1:2]})

allsums(magic(7),sort)
# beware! compare apply(magic(7),1,sort) and apply(magic(7),2,sort)
```

apad

*Pad arrays***Description**

Multidimensional pad for arrays of arbitrary dimension

**Usage**

```
apad(a, l, e = NULL, method = "ext", post = TRUE)
```

**Arguments**

a	Array to be padded
l	Amount of padding to add. If a vector of length greater than one, it is interpreted as the extra extent of a along each of its dimensions (standard recycling is used). If of length one, interpret as the dimension to be padded, in which case the amount is given by argument l.
e	If l is of length one, the amount of padding to add to dimension l
method	String specifying the values of the padded elements. See details section.
post	Boolean, with default TRUE meaning to append to a and FALSE meaning to prepend.

**Details**

Argument `method` specifies the values of the padded elements. It can be either “ext”, “mirror”, or “rep”.

Specifying `ext` (the default) uses a padding value given by the “nearest” element of `a`, as measured by the Manhattan metric.

Specifying `mirror` fills the array with alternate mirror images of `a`; while `rep` fills it with unreflected copies of `a`.

**Note**

Function `apad()` does not work with arrays with dimensions of zero extent: what to pad it with? To pad with a particular value, use `adiag()`.

The function works as expected with vectors, which are treated as one-dimensional arrays. See examples section.

Function `apad()` is distinct from `adiag()`, which takes two arrays and binds them together. Both functions create an array of the same dimensionality as their array arguments but with possibly larger extents. However, the functions differ in the values of the new array elements. Function `adiag()` uses a second array; function `apad()` takes the values from its primary array argument.

**Author(s)**

Robin K. S. Hankin

**See Also**[adiag](#)**Examples**

```

apad(1:10,4,method="mirror")

a <- matrix(1:30,5,6)

apad(a,c(4,4))
apad(a,c(4,4),post=FALSE)

apad(a,1,5)

apad(a,c(5,6),method="mirror")
apad(a,c(5,6),method="mirror",post=FALSE)

```

apl

*Replacements for APL functions take and drop***Description**

Replacements for APL functions take and drop

**Usage**

```

apldrop(a, b, give.indices=FALSE)
apldrop(a, b) <- value
apltake(a, b, give.indices=FALSE)
apltake(a, b) <- value

```

**Arguments**

a	Array
b	Vector of number of indices to take/drop. Length of b should not exceed length(dim(a)); if it does, an error is returned
give.indices	Boolean, with default FALSE meaning to return the appropriate subset of array a, and TRUE meaning to return the list of the selected elements in each of the dimensions. Setting to TRUE is not really intended for the end-user, but is used in the code of <code>apltake&lt;-()</code> and <code>apldrop&lt;-()</code>
value	elements to replace

**Details**

`apltake(a,b)` returns an array of the same dimensionality as `a`. Along dimension `i`, if `b[i]>0`, the first `b[i]` elements are retained; if `b[i]<0`, the last `b[i]` elements are retained.

`apldrop(a,b)` returns an array of the same dimensionality as `a`. Along dimension `i`, if `b[i]>0`, the first `b[i]` elements are dropped; if `b[i]<0`, the last `b[i]` elements are dropped.

These functions do not drop singleton dimensions. Use `drop()` if this is desired.

**Author(s)**

Robin K. S. Hankin

**Examples**

```
a <- magichypercube.4n(m=1)
apltake(a,c(2,3,2))
apldrop(a,c(1,1,2))

b <- matrix(1:30,5,6)
apldrop(b,c(1,-2)) <- -1

b <- matrix(1:110,10,11)
apltake(b,2) <- -1
apldrop(b,c(5,-7)) <- -2
b
```

---

aplus

*Generalized array addition*

---

**Description**

Given two arrays `a` and `b` with `length(dim(a))==length(dim(b))`, return a matrix with dimensions `pmax(dim(a),dim(b))` where “overlap” elements are `a+b`, and the other elements are either 0, `a`, or `b` according to location. See details section.

**Usage**

```
aplus(...)
```

**Arguments**

```
...          numeric or complex arrays
```

**Details**

The function takes any number of arguments (the binary operation is associative).

The operation of `aplus()` is understandable by examining the following **pseudocode**:

- `outa <- array(0, pmax(a, b))`
- `outb <- array(0, pmax(a, b))`
- `outa[1:dim(a)] <- a`
- `outb[1:dim(a)] <- b`
- `return(outa+outb)`

See how `outa` and `outb` are the correct size and the appropriate elements of each are populated with `a` and `b` respectively. Then the sum is returned.

**Author(s)**

Robin K. S. Hankin

**See Also**

[apad](#)

**Examples**

```
a <- matrix(1:10, 2, 5)
b <- matrix(1:9, 3, 3)
aplus(a, b, b)
```

---

arev

*Reverses some dimensions; a generalization of rev*

---

**Description**

A multidimensional generalization of `rev()`: given an array `a`, and a Boolean vector `swap`, return an array of the same shape as `a` but with dimensions corresponding to TRUE elements of `swap` reversed. If `swap` is not Boolean, it is interpreted as the dimensions along which to swap.

**Usage**

```
arev(a, swap = TRUE)
```

**Arguments**

<code>a</code>	Array to be reversed
<code>swap</code>	Vector of Boolean variables. If <code>swap[i]</code> is TRUE, then dimension <code>i</code> of array <code>a</code> is reversed. If <code>swap</code> is of length one, recycle to <code>length(dim(a))</code>

**Details**

If `swap` is not Boolean, it is equivalent to `1:n %in% swap` (where `n` is the number of dimensions). Thus multiple entries are ignored, as are entries greater than `n`.

If `a` is a vector, `rev(a)` is returned.

Function `arev()` handles zero-extent dimensions as expected.

Function `arev()` does not treat singleton dimensions specially, and is thus different from Octave's `flipdim()`, which (if supplied with no second argument) flips the first nonsingleton dimension. To reproduce this, use `arev(a, fnsd(a))`.

**Author(s)**

Robin K. S. Hankin

**See Also**

[ashift](#)

**Examples**

```
a <- matrix(1:42,6,7)
arev(a) #Note swap defaults to TRUE
```

```
b <- magichypercube.4n(1,d=4)
arev(b,c(TRUE,FALSE,TRUE,FALSE))
```

---

arot

*Rotates an array about two specified dimensions*

---

**Description**

Rotates an array about two specified dimensions by any number of 90 degree turns

**Usage**

```
arot(a, rights = 1,pair=1:2)
```

**Arguments**

<code>a</code>	The array to be rotated
<code>rights</code>	Integer; number of right angles to turn
<code>pair</code>	A two-element vector containing the dimensions to rotate with default meaning to rotate about the first two dimensions

**Note**

Function `arot()` is not exactly equivalent to octave's `rotdim()`; in `arot()` the order of the elements of `pair` matters because the rotation is clockwise when viewed in the `(pair[1],pair[2])` direction. Compare octave's `rotdim()` in which `pair` is replaced with `sort(pair)`.

Note also that the rotation is about the first two dimensions specified by `pair` but if `pair` has more than two elements then these dimensions are also permuted.

Also note that function `arot()` does not treat singleton dimensions specially.

**Author(s)**

Robin K. S. Hankin

**See Also**

[arev](#)

**Examples**

```
a <- array(1:16,rep(2,4))
arot(a)
arot(a,c(1,3))
```

---

arow

*Generalized row and col*


---

**Description**

Given an array, returns an array of the same size whose elements are sequentially numbered along the  $i^{\text{th}}$  dimension.

**Usage**

```
arow(a, i)
```

**Arguments**

a	array to be converted
i	Number of the dimension

**Value**

An integer matrix with the same dimensions as `a`, with element  $(n_1, n_2, \dots, n_d)$  being  $n_i$ .

**Note**

This function is equivalent to, but faster than, `function(a, i){do.index(a, function(x){x[i]})}`. However, it is much more complicated.

**Author(s)**

Robin K. S. Hankin

**Examples**

```
a <- array(0,c(3,3,2,2))
arow(a,2)
(arow(a,1)+arow(a,2)+arow(a,3)+arow(a,4))%2
```

---

as.standard

*Standard form for magic squares*


---

**Description**

Transforms a magic square or magic hypercube into Frenicle's standard form

**Usage**

```
as.standard(a, toroidal = FALSE, one_minus=FALSE)
is.standard(a, toroidal = FALSE, one_minus=FALSE)
```

**Arguments**

a	Magic square or hypercube (array) to be tested or transformed
toroidal	Boolean, with default FALSE meaning to use Frenicle's method, and TRUE meaning to use additional transformations appropriate to toroidal connectivity
one_minus	Boolean, with TRUE meaning to use the transformation $x \rightarrow n^2 + 1 - x$ if appropriate, and default FALSE meaning not to use this

**Details**

For a square, as.standard() transforms a magic square into Frenicle's standard form. The four numbers at each of the four corners are determined. First, the square is rotated so the smallest of the four is at the upper left. Then, element [1,2] is compared with element[2,1] and, if it is larger, the transpose is taken.

Thus all eight rotated and transposed versions of a magic square have the same standard form.

The square returned by magic() is in standard form.

For hypercubes, the algorithm is generalized. First, the hypercube is reflected so that a[1,1,...,1,1] is the smallest of the  $2^d$  corner elements (eg a[1,n,1,...,1,1]).

Next, aperm() is called so that

$$a[1,1,\dots,1,2] < a[1,1,\dots,2,1] < \dots < a[2,1,\dots,1,1].$$

Note that the inequalities are strict as hypercubes are assumed to be normal. As of version 1.3-1, as.standard() will accept arrays of any dimension (ie arrays a with minmax(dim(a))==FALSE will be handled sensibly).

An array with any dimension of extent zero is in standard form by definition; dimensions of length one are dropped.

If argument `toroidal` is TRUE, then the array `a` is translated using `ashift()` so that `a[1,1,...,1] == min(a)`. Such translations preserve the properties of semimagicness and pandiagonality (but not magicness or associativity).

It is easier (for me at least) to visualise this by considering two-dimensional arrays, tiling the plane with copies of `a`.

Next, the array is shifted so that `a[2,1,1,...,1] < a[dim(a)[1],1,1,...,1]` and `a[1,2,1,...,1] < a[1,dim(a)[2],1,...,1]` and so on.

Then `aperm()` is called as per the non-toroidal case above.

`is.standard()` returns TRUE if the magic square or hypercube is in standard form. `is.standard()` and `as.standard()` check for neither magicness nor normality (use `is.magic` and `is.normal` for this).

### Note

There does not appear to be a way to make the third letter of “Frenicle” have an acute accent, as it should do.

### Author(s)

Robin K. S. Hankin

### See Also

[magic.eq](#)

### Examples

```
is.standard(magic.2np1(4))
as.standard(magic.4n(3))

as.standard(magichypercube.4n(1,5))

##non-square arrays:
as.standard(magic(7)[1:3,])

## Toroidal transforms preserve pandiagonality:
is.pandiagonal(as.standard(hudson(11)))

## but not magicness:
is.magic(as.standard(magic(10),TRUE))
```

cilleruelo

*A class of multiplicative magic squares due to Cilleruelo and Luca***Description**

Cilleruelo and Luca give a class of multiplicative magic squares whose behaviour is interesting.

**Usage**

```
cilleruelo(n, m)
```

**Arguments**

`n, m` Arguments: usually integers

**Details**

$$\begin{pmatrix} (n+2)(m+0) & (n+3)(m+3) & (n+1)(m+2) & (n+0)(m+1) \\ (n+1)(m+1) & (n+0)(m+2) & (n+2)(m+3) & (n+3)(m+0) \\ (n+0)(m+3) & (n+1)(m+0) & (n+3)(m+1) & (n+2)(m+2) \\ (n+3)(m+2) & (n+2)(m+1) & (n+0)(m+0) & (n+1)(m+3) \end{pmatrix}$$

**Value**

Returns a  $4 \times 4$  matrix.

**Author(s)**

Robin K. S. Hankin

**References**

Javier Cilleruelo and Florian Luca 2010, "On multiplicative magic squares", *The Electronic Journal of Combinatorics* vol 17, #N8

**Examples**

```
is.magic(cilleruelo(5,6))
is.magic(cilleruelo(5,6),func=prod)

f <- function(n){
  jj <-
    sapply(
      seq(from=5,len=n),
      function(i){cilleruelo(i,i-4)}
    )
  xM <- apply(jj,2,max)
```

```

xm <- apply(jj,2,min)

cbind(xM-xm , 5^(5/12)*xm^0.5 , 6*xm^0.5)
}

matplot(f(200),type='l',log='xy',xlab='n',ylab='')
legend(x="topleft",legend=c("xM-xm","5^(5/12).xm^(1/2)","6xm^(1/2)"),
       lty=1:3,col=1:3)

```

---

circulant

*Circulant matrices of any order*


---

### Description

Creates and tests for circulant matrices of any order

### Usage

```

circulant(vec)
is.circulant(m,dir=rep(1,length(dim(m))))

```

### Arguments

vec	In <code>circulant()</code> , vector of elements of the first row. If of length one, interpret as the order of the matrix and use <code>1:vec</code> .
m	In <code>is.circulant()</code> , matrix to be tested for circulantism
dir	In <code>is.circulant()</code> , the direction of the diagonal. In a matrix, the default value <code>(c(1,1))</code> traces the major diagonals

### Details

A matrix  $a$  is *circulant* if all major diagonals, including broken diagonals, are uniform; ie if  $a_{ij} = a_{kl}$  when  $i - j = k - l$  (modulo  $n$ ). The standard values to use give `1:n` for the top row.

In function `is.circulant()`, for arbitrary dimensional arrays, the default value for `dir` checks that `a[v]==a[v+rep(1,d)]==...==a[v+rep((n-1),d)]` for all  $v$  (that is, following lines parallel to the major diagonal); indices are passed through `process()`.

For general `dir`, function `is.circulant()` checks that `a[v]==a[v+dir]==a[v+2*dir]==...==a[v+(n-1)*d]` for all  $v$ .

A *Toeplitz* matrix is one in which  $a[i,j]=a[i',j']$  whenever  $|i-j|=|i'-j'|$ . See function `toeplitz()` of the `stats` package for details.

### Author(s)

Robin K. S. Hankin

**References**

Arthur T. Benjamin and K. Yasuda. *Magic “Squares” Indeed!*, American Mathematical Monthly, vol 106(2), pp152-156, Feb 1999

**Examples**

```

circulant(5)
circulant(2^(0:4))
is.circulant(circulant(5))

a <- outer(1:3,1:3,"+")%%3
is.circulant(a)
is.circulant(a,c(1,2))

is.circulant(array(c(1:4,4:1),rep(2,3)))

is.circulant(magic(5)%%5,c(1,-2))

```

---

cube2

*A pantriagonal magic cube*

---

**Description**

A pantriagonal magic cube of order 4 originally due to Hendricks

**Usage**

```
data(cube2)
```

**Details**

Meaning of “pantriagonal” currently unclear

**Source**

Hendricks

**Examples**

```

data(cube2)
is.magichypercube(cube2)
is.perfect(cube2)

```

---

diag.off	<i>Extracts broken diagonals</i>
----------	----------------------------------

---

## Description

Returns broken diagonals of a magic square

## Usage

```
diag.off(a, offset = 0, nw.se = TRUE)
```

## Arguments

a	Square matrix
offset	vertical offset
nw.se	Boolean variable with TRUE meaning trace diagonals along the northwest-southeast direction (point [1, n] to [n, 1]).

## Details

Useful when testing for panmagic squares. The first element is always the unbroken one (ie [1, 1] to [n, n] if nw.se is TRUE and [1, n] to [n, 1] if nw.se is FALSE).

## Author(s)

Robin K. S. Hankin

## See Also

[is.panmagic](#)

## Examples

```
diag.off(magic(10), nw.se=FALSE, offset=0)  
diag.off(magic(10), nw.se=FALSE, offset=1)
```

do.index

*Apply a function to array element indices*

---

**Description**

Given a function `f()` that takes a vector of indices, and an array of arbitrary dimensions, apply `f()` to the elements of `a`

**Usage**

```
do.index(a, f, ...)
```

**Arguments**

<code>a</code>	Array
<code>f</code>	Function that takes a vector argument of the same length as <code>dim(a)</code>
<code>...</code>	Further arguments supplied to <code>f()</code>

**Value**

Returns a matrix of the same dimensions as `a`

**Note**

Tamas Papp suggests the one-liner `function(a, f, ...) { array(apply(as.matrix(expand.grid(lapply(dim(a), seq(...), dim(a))))), dim(a))}` which is functionally identical to `do.index()`; but it is no faster than the version implemented in the package, and (IMO) is harder to read.

Further note that function `arow()` is much much faster than `do.index()`; it is often possible to rephrase a call to `do.index()` as a call to `arow()`; do this where possible unless the additional code opacity outweighs the speed savings.

**Author(s)**

Robin K. S. Hankin, with improvements by Gabor Grothendieck and Martin Maechler, via the R help list

**See Also**

[arow](#)

**Examples**

```
a <- array(0,c(2,3,4))
b <- array(rpois(60,1),c(3,4,5))

f1 <- function(x){sum(x)}
f2 <- function(x){sum((x-1)^2)}
f3 <- function(x){b[t(x)]}
```

```
f4 <- function(x){sum(x)%%2}
f5 <- function(x,u){x[u]}

do.index(a,f1) # should match arow(a,1)+arow(a,2)+arow(a,3)
do.index(a,f2)
do.index(a,f3) # same as apltake(b,dim(a))
do.index(a,f4) # Male/female toilets at NOC
do.index(a,f5,2) # same as arow(a,2)
```

---

 eq

---

*Comparison of two magic squares*


---

### Description

Compares two magic squares according to Frenicle's method. Mnemonic is the old Fortran ".GT." (for "Greater Than") comparison et seq.

To compare magic square a with magic square b, their elements are compared in rowwise order: a[1,1] is compared with b[1,1], then a[1,2] with b[1,2], up to a[n,n]. Consider the first element that is different, say [i, j]. Then a<b if a[i, j]<b[i, j].

The generalization to hypercubes is straightforward: comparisons are carried out natural order.

### Usage

```
eq(m1, m2)
ne(m1, m2)
gt(m1, m2)
lt(m1, m2)
ge(m1, m2)
le(m1, m2)
m1 %eq% m2
m1 %ne% m2
m1 %gt% m2
m1 %lt% m2
m1 %ge% m2
m1 %le% m2
```

### Arguments

m1	First magic square
m2	Second magic square

### Note

Rather clumsy function definition due to the degenerate case of testing two identical matrices (min(NULL) is undefined).

The two arguments are assumed to be matrices of the same size. If not, an error is given.

**Author(s)**

Robin K. S. Hankin

**See Also**

[as.standard](#)

**Examples**

```
magic(4) %eq% magic.4n(1)
eq(magic(4) , magic.4n(1))
```

---

fnsd

*First non-singleton dimension*

---

**Description**

Given an array, returns the first non-singleton dimension. Useful for emulating some of Matlab / Octave's multidimensional functions.

If n is supplied, return the first n nonsingleton dimensions.

**Usage**

```
fnsd(a,n)
```

**Arguments**

a	An array
n	Integer. Return the first n nonsingleton dimensions

**Value**

Returns an integer vector with elements in the range 1 to `length(dim(a))`.

**Note**

Treats zero-extent dimensions as singletons.

Case n=0 now treated sensibly (returns a zero-length vector).

**Author(s)**

Robin K. S. Hankin

**See Also**

[arev](#)

**Examples**

```
a <- array(1:24,c(1,1,1,1,2,1,3,4))
fnsd(a)
fnsd(a,2)
```

---

`force.integer`*Integerize array elements*

---

**Description**

Returns an elementwise `as.integer`-ed array. All magic squares should have integer elements.

**Usage**

```
force.integer(x)
```

**Arguments**

`x`                    Array to be converted

**Note**

Function `force.integer()` differs from `as.integer()` as the latter returns an integer vector, and the former returns an array whose elements are integer versions of `x`; see examples section below.

**Author(s)**

Robin K. S. Hankin

**Examples**

```
a <- matrix(rep(1,4),2,2)
force.integer(a)
as.integer(a)
```

---

Frankenstein	<i>A perfect magic cube due to Frankenstein</i>
--------------	---

---

**Description**

A perfect magic cube due to Frankenstein

**Usage**

```
data(Frankenstein)
```

**Examples**

```
data(Frankenstein)
is.perfect(Frankenstein)
```

---

hadamard	<i>Hadamard matrices</i>
----------	--------------------------

---

**Description**

Various functionality for Hadamard matrices

**Usage**

```
sylvester(k)
is.hadamard(m)
```

**Arguments**

k	Function <code>sylvester()</code> gives the k-th Sylvester matrix
m	matrix

**Details**

A *Hadamard matrix* is a square matrix whose entries are either +1 or -1 and whose rows are mutually orthogonal.

**Author(s)**

Robin K. S. Hankin

**References**

“Hadamard matrix.” *Wikipedia, The Free Encyclopedia*. 19 Jan 2009, 18:21 UTC. 20 Jan 2009  
[http://en.wikipedia.org/w/index.php?title=Hadamard\\_matrix&oldid=265119778](http://en.wikipedia.org/w/index.php?title=Hadamard_matrix&oldid=265119778)

**Examples**

```
is.hadamard(sylvester(4))
image(sylvester(5))
```

---

hendricks

*A perfect magic cube due to Hendricks*


---

**Description**

A perfect  $8 \times 8 \times 8$  magic cube due to Hendricks

**Usage**

```
data(hendricks)
```

**Source**

[http://members.shaw.ca/hdhcubes/cube\\_perfect.htm#Definitions](http://members.shaw.ca/hdhcubes/cube_perfect.htm#Definitions)

**Examples**

```
data(hendricks)
is.perfect(hendricks)
```

---

hudson

*Pandiagonal magic squares due to Hudson*


---

**Description**

Returns a regular pandiagonal magic square of order  $6m \pm 1$  using a method developed by Hudson.

**Usage**

```
hudson(n = NULL, a = NULL, b = NULL)
```

**Arguments**

n	Order of the square, $n = 6m \pm 1$ . If NULL, use the length of a
a	The first line of Hudson's <i>A</i> matrix. If NULL, use Hudson's value of $c(n-1, 0 : (n-2))$
b	The first line of Hudson's <i>B</i> matrix. If NULL, use Hudson's value of $c(2 : (n-1), n, 1)$ . Using default values for a and b gives an associative square

**Details**

Returns one member of a set of regular magic squares of order  $n = 6m \pm 1$ . The set is of size  $(n!)^2$ .

Note that  $n$  is not checked for being in the form  $6n \pm 1$ . If it is not the correct form, the square is magic but not necessarily normal.

**Author(s)**

Robin K. S. Hankin

**References**

C. B. Hudson, *On pandiagonal squares of order  $6t \pm 1$* , Mathematics magazine, March 1972, pp94-96

**See Also**

[recurse](#)

**Examples**

```
hudson(n=11)
magicplot(hudson(n=11))
is.associative(hudson(n=13))
hudson(a=(2*1:13)%%13 , b=(8*1:13)%%13)
all(replicate(10,is.magic(hudson(a=sample(13),b=sample(13))))))
```

---

is.magic

*Various tests for the magicness of a square*

---

**Description**

Returns TRUE if the square is magic, semimagic, panmagic, associative, normal. If argument `give.answers` is TRUE, also returns additional information about the sums.

**Usage**

```
is.magic(m, give.answers = FALSE, func=sum, boolean=FALSE)
is.panmagic(m, give.answers = FALSE, func=sum, boolean=FALSE)
is.pandiagonal(m, give.answers = FALSE, func=sum, boolean=FALSE)
is.semimagic(m, give.answers = FALSE, func=sum, boolean=FALSE)
is.semimagic.default(m)
is.associative(m)
is.normal(m)
is.sparse(m)
is.mostperfect(m,give.answers=FALSE)
is.2x2.correct(m,give.answers=FALSE)
is.bree.correct(m,give.answers=FALSE)
```

```

is.latin(m,give.answers=FALSE)
is.antimagic(m, give.answers = FALSE, func=sum)
is.totally.antimagic(m, give.answers = FALSE, func=sum)
is.sam(m)
is.stam(m)

```

### Arguments

m	The square to be tested
give.answers	Boolean, with TRUE meaning return additional information about the sums (see details)
func	A function that is evaluated for each row, column, and unbroken diagonal
boolean	Boolean, with TRUE meaning that the square is deemed magic, semimagic, etc, if all applications of func evaluate to TRUE. If boolean is FALSE, square m is magic etc if all applications of func are identical

### Details

- A *semimagic square* is one all of whose row sums equal all its columnwise sums (ie the magic constant).
- A *magic square* is a semimagic square with the sum of both unbroken diagonals equal to the magic constant.
- A *panmagic square* is a magic square all of whose broken diagonals sum to the magic constant. Ollerenshaw calls this a “pandiagonal” square.
- A *most-perfect square* has all 2-by-2 arrays anywhere within the square summing to  $2S$  where  $S = n^2 + 1$ ; and all pairs of integers  $n/2$  distant along the same major (NW-SE) diagonal sum to  $S$  (note that the  $S$  used here differs from Ollerenshaw’s because her squares are numbered starting at zero). The first condition is tested by `is.2x2.correct()` and the second by `is.bree.correct()`.  
All most-perfect squares are panmagic.
- A *normal square* is one that contains  $n^2$  consecutive integers (typically starting at 0 or 1).
- A *sparse matrix* is one whose nonzero entries are consecutive integers, starting at 1.
- An *associative square* (also *regular square*) is a magic square in which  $a_{i,j} + a_{n+1-i,n+1-j} = n^2 + 1$ . Note that an associative semimagic square is magic; see also `is.square.palindromic()`. The definition extends to magic hypercubes: a hypercube  $a$  is associative if  $a + \text{arev}(a)$  is constant.
- An *ultramagic square* is pandiagonal and associative.
- A *latin square* of size  $n \times n$  is one in which each column and each row comprises the integers 1 to  $n$  (not necessarily in that order). Function `is.latin()` is a wrapper for `is.latinhypercube()` because there is no natural way to present the extra information given when `give.answers` is TRUE in a manner consistent with the other functions documented here.
- An *antimagic square* is one whose row sums and column sums are consecutive integers; a *totally antimagic square* is one whose row sums, column sums, and two unbroken diagonals are consecutive integers. Observe that an antimagic square is not necessarily totally antimagic, and vice-versa.
- A square is *sam* (or *SAM*) if it is sparse and antimagic; it is *stam* (or *STAM*) if it is sparse and totally antimagic. See documentation at SAM.

**Value**

Returns TRUE if the square is semimagic, etc. and FALSE if not.

If `give.answers` is taken as an argument and is TRUE, return a list of at least five elements. The first element of the list is the answer: it is TRUE if the square is (semimagic, magic, panmagic) and FALSE otherwise. Elements 2-5 give the result of a call to `allsums()`, viz: rowwise and columnwise sums; and broken major (ie NW-SE) and minor (ie NE-SW) diagonal sums.

Function `is.bree.correct()` also returns the sums of elements distant  $n/2$  along a major diagonal (`diag.sums`); and function `is.2x2.correct()` returns the sum of each  $2 \times 2$  submatrix (`tbt.sums`); for other size windows use `subsums()` directly. Function `is.mostperfect()` returns both of these.

Function `is.semimagic.default()` returns TRUE if the argument is semimagic [with respect to `sum()`] using a faster method than `is.semimagic()`.

**Note**

Functions that take a `func` argument apply that function to each row, column, and diagonal as necessary. If `func` takes its default value of `sum()`, the sum will be returned; if `prod()`, the product will be returned, and so on. There are many choices for this argument that produce interesting tests; consider `func=max`, for example. With this, a “magic” square is one whose row, sum and (unbroken) diagonals have identical maxima. Thus `diag(5)` is magic with respect to `max()`, but `diag(6)` isn’t.

Argument `boolean` is designed for use with non-default values for the `func` argument; for example, a latin square is semimagic with respect to `func=function(x){all(diff(sort(x))==1)}`.

Function `is.magic()` is vectorized; if a list is supplied, the defaults are assumed.

**Author(s)**

Robin K. S. Hankin

**References**

<http://mathworld.wolfram.com/MagicSquare.html>

**See Also**

[minmax](#), [is.perfect](#), [is.semimagichypercube](#), [sam](#)

**Examples**

```
is.magic(magic(4))

is.magic(diag(7),func=max) # TRUE
is.magic(diag(8),func=max) # FALSE

stopifnot(is.magic(magic(3:8)))

is.panmagic(panmagic.4())
is.panmagic(panmagic.8())
```

```
data(Ollerenshaw)
is.mostperfect(Ollerenshaw)

proper.magic <- function(m){is.magic(m) & is.normal(m)}
proper.magic(magic(20))
```

---

is.magichypercube      *magic hypercubes*

---

## Description

Returns TRUE if a hypercube is semimagic, magic, perfect

## Usage

```
is.semimagichypercube(a, give.answers=FALSE, func=sum, boolean=FALSE)
is.diagonally.correct(a, give.answers = FALSE, func=sum, boolean=FALSE)
is.magichypercube(a, give.answers = FALSE, func=sum, boolean=FALSE)
is.perfect(a, give.answers = FALSE, func=sum, boolean=FALSE)
is.latinhypercube(a, give.answers=FALSE)
is.alicehypercube(a, ndim, give.answers=FALSE, func=sum, boolean=FALSE)
```

## Arguments

a	The hypercube (array) to be tested
give.answers	Boolean, with TRUE meaning to also return the sums
func	Function to be applied across each dimension
ndim	In is.alicehypercube(), dimensionality of subhypercube to take sums over. Thus ndim=1 corresponds to a conventional semimagichypercube and ndim=2 corresponds to each 2d subhypercube (ie each square) having the same sum
boolean	Boolean, with TRUE meaning that the hypercube is deemed magic, semimagic, etc, if all applications of func evaluate to TRUE. If boolean is FALSE, the hypercube is magic etc if all applications of func are identical

## Details

(Although apparently non-standard, here a hypercube is defined to have dimension  $d$  and order  $n$ —and thus has  $n^d$  elements).

- A *semimagic hypercube* has all “rook’s move” sums equal to the magic constant (that is, each  $\sum a[i_1, i_2, \dots, i_{r-1}, i_{r+1}, \dots, i_d]$  with  $1 \leq r \leq d$  is equal to the magic constant for all values of the  $i$ ’s). In is.semimagichypercube(), if give.answers is TRUE, the sums returned are in the form of an array of dimension  $c(\text{rep}(n, d-1), d)$ . The first  $d-1$  dimensions are the coordinates of the projection of the summed elements onto the surface hypercube. The last dimension indicates the dimension along which the sum was taken over.

Optional argument func, defaulting to sum(), indicates the function to be taken over each of the  $d$  dimensions. Currently requires func to return a scalar.

- A *Latin hypercube* is one in which each line of elements whose coordinates differ in only one dimension comprises the numbers 1 to  $n$  (or 0 to  $n - 1$ ), not necessarily in that order. Each integer thus appears  $n^{d-1}$  times.
- A *magic hypercube* is a semimagic hypercube with the additional requirement that all  $2^{d-1}$  long (ie extreme point-to-extreme point) diagonals sum correctly. Correct diagonal summation is tested by `is.diagonally.correct()`; by specifying a function other than `sum()`, criteria other than the diagonals returning the correct sum may be tested.
- An *Alice hypercube* is a different generalization of a semimagic square to higher dimensions. A semimagic hypercube has all (one-dimensional) lines summing correctly. An Alice hypercube has all  $ndim$ -dimensional hyperplanes summing correctly. Thus `is.alicehypercube(a, 1)` corresponds to a regular semimagic hypercube; `is.alicehypercube(a, 2)` corresponds to all 2d hyperplanes (ie squares) having the same sum.
- A *perfect magic hypercube* (use `is.perfect()`) is a magic hypercube with all nonbroken diagonals summing correctly. This is a seriously restrictive requirement for high dimensional hypercubes. As yet, this function does not take a `give.answers` argument.
- A *pandiagonal magic hypercube*, also *Nasik hypercube* (or sometimes just a *perfect hypercube*) is a semimagic hypercube with all diagonals, including broken diagonals, summing correctly. This is not implemented.

The terminology in this area is pretty confusing.

In `is.magichypercube()`, if argument `give.answers=TRUE` then a list is returned. The first element of this list is Boolean with TRUE if the array is a magic hypercube. The second element and third elements are answers from `is.semimagichypercube()` and `is.diagonally.correct()` respectively.

In `is.diagonally.correct()`, if argument `give.answers=TRUE`, the function also returns an array of dimension `c(q, rep(2, d))` (that is,  $q \times 2^d$  elements), where  $q$  is the length of `func()` applied to a long diagonal of `a` (if  $q = 1$ , the first dimension is dropped). If  $q = 1$ , then in dimension  $d$  having index 1 means `func()` is applied to elements of `a` with the  $d^{\text{th}}$  dimension running over  $1:n$ ; index 2 means to run over  $n:1$ . If  $q > 1$ , the index of the first dimension gives the index of `func()`, and subsequent dimensions have indices of 1 or 2 as above and are interpreted in the same way.

An example of a function for which these two are not identical is given below.

If `func=f` where `f` is a function returning a vector of length `i`, `is.diagonally.correct()` returns an array out of dimension `c(i, rep(2, d))`, with `out[, i_1, i_2, ..., i_d]` being `f(x)` where `x` is the appropriate long diagonal. Thus the  $2^d$  equalities `out[, i_1, i_2, ..., i_d]==out[, 3-i_1, 3-i_2, ..., 3-i_d]` hold if and only if `identical(f(x), f(rev(x)))` is TRUE for each long diagonal (a condition met, for example, by `sum()` but not by the identity function or `function(x){x[1]}`).

## Note

Note that not all subhypercubes are necessarily magic! (for example, consider a 5-dimensional magic hypercube `a`. The square `b` defined by `a[1, 1, 1, , ]` might not be magic: the diagonals of `b` are not covered by the definition of a magic hypercube). Some subhypercubes of a magic hypercube are not even semimagic: see below for an example.

Even in three dimensions, being perfect is pretty bad. Consider a  $5 \times 5 \times 5$  (ie three dimensional), cube. Say `a=magiccube.2np1(2)`. Then the square defined by `sapply(1:n, function(i){a[, i, 6-i]}, simplify=TRUE)`, which is a subhypercube of `a`, is not even semimagic: the rowsums are in-

correct (the colsums must sum correctly because a is magic). Note that the diagonals of this square are two of the “extreme point-to-point” diagonals of a.

A *pandiagonal magic hypercube* (or sometimes just a *perfect hypercube*) is semimagic and in addition the sums of all diagonals, including broken diagonals, are correct. This is one seriously bad-ass requirement. I reckon that is a total of  $\frac{1}{2}(3^d - 1) \cdot n^{d-1}$  correct summations. This is not coded up yet; I can’t see how to do it in anything like a vectorized manner.

### Author(s)

Robin K. S. Hankin

### References

- R. K. S. Hankin 2005. “Recreational mathematics with R: introducing the **magic** package”. R news, 5(1)
- Richards 1980. “Generalized magic cubes”. *Mathematics Magazine*, volume 53, number 2, (March).

### See Also

[is.magic](#), [allsubhypercubes](#), [hendricks](#)

### Examples

```
library(abind)
is.semimagihypercube(magiccube.2np1(1))
is.semimagihypercube(magihypercube.4n(1,d=4))

is.perfect(magihypercube.4n(1,d=4))

# Now try an array with minmax(dim(a))==FALSE:
a <- abind(magiccube.2np1(1),magiccube.2np1(1),along=2)
is.semimagihypercube(a,g=TRUE)$rook.sums

#argument boolean can be confusing:
zero <- array(0,rep(3,4))
is.semimagihypercube(a,func=is.ok,boolean=FALSE) # TRUE, because all
                                                    # applications of is.ok()
                                                    # return the same value!

a2 <- magihypercube.4n(m=1,d=4)
is.diagonally.correct(a2)
is.diagonally.correct(a2,g=TRUE)$diag.sums

## To extract corner elements (note func(1:n) != func(n:1)):
is.diagonally.correct(a2,func=function(x){x[1]},g=TRUE)$diag.sums

#Now for a subhypercube of a magic hypercube that is not semimagic:
is.magic(allsubhypercubes(magiccube.2np1(1))[[10]])
```

```

data(hendricks)
is.perfect(hendricks)

#note that Hendricks's magic cube also has many broken diagonals summing
#correctly:

a <- allsubhypercubes(hendricks)
ld <- function(a){length(dim(a))}

jj <- unlist(lapply(a,ld))
f <- function(i){is.perfect(a[[which(jj==2)[i]]])}
all(sapply(1:sum(jj==2),f))

#but this is NOT enough to ensure that it is pandiagonal (but I
#think hendricks is pandiagonal).

is.alicehypercube(magichypercube.4n(1,d=5),4,give.answers=TRUE)

```

---

is.ok

*does a vector have the sum required to be a row or column of a magic square?*

---

### Description

Returns TRUE if and only if `sum(vec)==magic.constant(n,d=d)`

### Usage

```
is.ok(vec, n=length(vec), d=2)
```

### Arguments

<code>vec</code>	Vector to be tested
<code>n</code>	Order of square or hypercube. Default assumes order is equal to length of <code>vec</code>
<code>d</code>	Dimension of square or hypercube. Default of 2 corresponds to a square

### Author(s)

Robin K. S. Hankin

### Examples

```
is.ok(magic(5)[1,])
```

---

is.square.palindromic *Is a square matrix square palindromic?*

---

## Description

Implementation of various properties presented in a paper by Arthur T. Benjamin and K. Yasuda

## Usage

```
is.square.palindromic(m, base=10, give.answers=FALSE)
is.centrosymmetric(m)
is.persymmetric(m)
```

## Arguments

m	The square to be tested
base	Base of number expansion, defaulting to 10; not relevant for the “sufficient” part of the test
give.answers	Boolean, with TRUE meaning to return additional information

## Details

The following tests apply to a general square matrix  $m$  of size  $n \times n$ .

- A *centrosymmetric* square is one in which  $a[i, j]=a[n+1-i, n+1-j]$ ; use `is.centrosymmetric()` to test for this (compare an *associative* square). Note that this definition extends naturally to hypercubes: a hypercube  $a$  is centrosymmetric if `all(a==arev(a))`.
- A *persymmetric square* is one in which  $a[i, j]=a[n+1-j, n+1-i]$ ; use `is.persymmetric()` to test for this.
- A matrix is *square palindromic* if it satisfies the rather complicated conditions set out by Benjamin and Yasuda (see refs).

## Value

These functions return a list of Boolean variables whose value depends on whether or not  $m$  has the property in question.

If argument `give.answers` takes the default value of FALSE, a Boolean value is returned that shows whether the sufficient conditions are met.

If argument `give.answers` is TRUE, a detailed list is given that shows the status of each individual test, both for the necessary and sufficient conditions. The value of the second element (named `necessary`) is the status of their Theorem 1 on page 154.

Note that the necessary conditions do not depend on the base  $b$  (technically, neither do the sufficient conditions, for being a square palindrome requires the sums to match for *every* base  $b$ . In this implementation, “sufficient” is defined only with respect to a particular base).

**Note**

Every associative square is square palindromic, according to Benjamin and Yasuda.

Function `is.square.palindromic()` does not yet take a `give.answers` argument as does, say, `is.magic()`.

**Author(s)**

Robin K. S. Hankin

**References**

Arthur T. Benjamin and K. Yasuda. *Magic “Squares” Indeed!*, American Mathematical Monthly, vol 106(2), pp152-156, Feb 1999

**Examples**

```
is.square.palindromic(magic(3))
is.persymmetric(matrix(c(1,0,0,1),2,2))

#now try a circulant:
a <- matrix(0,5,5)
is.square.palindromic(circulant(10)) #should be TRUE
```

---

latin

*Random latin squares*

---

**Description**

Various functionality for generating random latin squares

**Usage**

```
incidence(a)
is.incidence(a, include.improper)
is.incidence.improper(a)
unincidence(a)
inc_to_inc(a)
another_latin(a)
another_incidence(i)
rlatin(n,size=NULL,start=NULL,burnin=NULL)
```

**Arguments**

`a`                    A latin square  
`i`                      An incidence array  
`n,include.improper,size,start,burnin`  
                       Various control arguments; see details section

**Details**

- Function `incidence()` takes an integer array (specifically, a latin square) and returns the incidence array as per Jacobson and Matthew 1996
- Function `is.incidence()` tests for an array being an incidence array; if argument `include.improper` is TRUE, admit an improper array
- Function `is.incidence.improper()` tests for an array being an improper array
- Function `unincidence()` converts an incidence array to a latin square
- Function `another_latin()` takes a latin square and returns a different latin square
- Function `another_incidence()` takes an incidence array and returns a different incidence array
- Function `rlatin()` generates a (Markov) sequence of random latin squares, arranged in a 3D array. Argument `n` specifies how many to generate; argument `size` gives the size of latin squares generated; argument `start` gives the start latin square (it must be latin and is checked with `is.latin()`); argument `burnin` gives the burn-in value (number of Markov steps to discard).

Default value of NULL for argument `size` means to take the size of argument `start`; default value of NULL for argument `start` means to use `circulant(size)`

As a special case, if argument `size` and `start` both take the default value of NULL, then argument `n` is interpreted as the size of a single random latin square to be returned; the other arguments take their default values. This ensures that “`rlatin(n)`” returns a single random  $n \times n$  latin square.

From Jacobson and Matthew 1996, an  $n \times n$  latin square LS is equivalent to an  $n \times n \times n$  array A with entries 0 or 1; the dimensions of A are identified with the rows, columns and symbols of LS; a 1 appears in cell  $(r, c, s)$  of A iff the symbol  $s$  appears in row  $r$ , column  $s$  of LS. Jacobson and Matthew call this an *incidence cube*.

The notation is readily generalized to latin hypercubes and `incidence()` is dimensionally vectorized.

An *improper* incidence cube is an incidence cube that includes a single  $-1$  entry; all other entries must be 0 or 1; and all line sums must equal 1.

**Author(s)**

Robin K. S. Hankin

**References**

M. T. Jacobson and P. Matthews 1996. “Generating uniformly distributed random latin squares”. *Journal of Combinatorial Designs*, volume 4, No. 6, pp405–437

**See Also**

[is.magic](#)

**Examples**

```

rlatin(5)
rlatin(n=2, size=4, burnin=10)

# An example that allows one to optimize an objective function
# [here f()] over latin squares:
gr <- function(x){ another_latin(matrix(x,7,7)) }
set.seed(0)
index <- sample(49,20)
f <- function(x){ sum(x[index])}
jj <- optim(par=as.vector(latin(7)), fn=f, gr=gr, method="SANN", control=list(maxit=10))
best_latin <- matrix(jj$par,7,7)
print(best_latin)
print(f(best_latin))

#compare starting value:
f(circulant(7))

```

---

lozenge

*Conway's lozenge algorithm for magic squares*


---

**Description**

Uses John Conway's lozenge algorithm to produce magic squares of odd order.

**Usage**

```
lozenge(m)
```

**Arguments**

**m** magic square returned is of order  $n=2m+1$

**Author(s)**

Robin Hankin

**See Also**

[magic.4np2](#)

**Examples**

```

lozenge(4)
all(sapply(1:10,function(n){is.magic(lozenge(n))}))

```

---

magic	<i>Creates magic squares</i>
-------	------------------------------

---

**Description**

Creates normal magic squares of any order  $> 2$ . Uses the appropriate method depending on  $n$  modulo 4.

**Usage**

```
magic(n)
```

**Arguments**

n	Order of magic square. If a vector, return a list whose $i$ -th element is a magic square of order $n[i]$
---	---

**Details**

Calls either `magic.2np1()`, `magic.4n()`, or `magic.4np2()` depending on the value of  $n$ . Returns a magic square in standard format (compare the `magic.2np1()` et `seq`, which return the square as generated by the direct algorithm).

**Author(s)**

Robin K. S. Hankin

**References**

William H. Benson and Oswald Jacoby. *New recreations with magic squares*. Dover 1976.

**See Also**

[magic.2np1](#), [magic.prime](#), [magic.4np2](#), [magic.4n](#), [lozenge](#), [as.standard](#), [force.integer](#)

**Examples**

```
magic(6)
all(is.magic(magic(3:10)))

## The first eigenvalue of a magic square is equal to the magic constant:
eigen(magic(10),FALSE,TRUE)$values[1] - magic.constant(10)

## The sum of the eigenvalues of a magic square after the first is zero:
sum(eigen(magic(10),FALSE,TRUE)$values[2:10])
```

---

`magic.2np1`*Magic squares of odd order*

---

**Description**

Function to create magic squares of odd order

**Usage**

```
magic.2np1(m, ord.vec = c(-1, 1), break.vec = c(1, 0), start.point=NULL)
```

**Arguments**

<code>m</code>	creates a magic square of order $n = 2m + 1$
<code>ord.vec</code>	ordinary vector. Default value of <code>c(-1, 1)</code> corresponds to the usual northeast direction
<code>break.vec</code>	break vector. Default of <code>c(1, 0)</code> corresponds to the usual south direction
<code>start.point</code>	Starting position for the method (ie coordinates of unity). Default value of <code>NULL</code> corresponds to row 1, column <code>m</code>

**Author(s)**

Robin K. S. Hankin

**References**

Written up in loads of places. The method (at least with the default ordinary and break vectors) seems to have been known since at least the Renaissance.

Benson and Jacoby, and the Mathematica website, discuss the problem with nondefault ordinary and break vectors.

**See Also**

[magic](#), [magic.prime](#)

**Examples**

```
magic.2np1(1)
f <- function(n){is.magic(magic.2np1(n))}
all(sapply(1:20, f))

is.panmagic(magic.2np1(5, ord.vec=c(2, 1), break.vec=c(1, 3)))
```

---

 magic.4n

*Magic squares of order 4n*


---

**Description**

Produces an associative magic square of order  $4n$  using the delta-x method.

**Usage**

```
magic.4n(m)
```

**Arguments**

m                      Order  $n$  of the square is given by  $n = 4m$

**Author(s)**

Robin K. S. Hankin

**See Also**

[magic](#)

**Examples**

```
magic.4n(4)
is.magic(magic.4n(5))
```

---

 magic.4np2

*Magic squares of order 4n+2*


---

**Description**

Produces a magic square of order  $4n + 2$  using Conway's "LUX" method

**Usage**

```
magic.4np2(m)
```

**Arguments**

m                      returns a magic square of order  $n = 4m + 2$  for  $m \geq 1$ , using Conway's "LUX" construction

**Note**

I am not entirely happy with the method used: it's too complicated

**Author(s)**

Robin K. S. Hankin

**References**

<http://mathworld.wolfram.com/MagicSquare.html>

**See Also**

[magic](#)

**Examples**

```
magic.4np2(1)
is.magic(magic.4np2(3))
```

---

magic.8

*Regular magic squares of order 8*

---

**Description**

Returns all 90 regular magic squares of order 8

**Usage**

```
magic.8(...)
```

**Arguments**

... ignored

**Value**

Returns an array of dimensions  $c(8, 8, 90)$  of which each slice is an 8-by-8 magic square.

**Author(s)**

Robin K. S. Hankin

**References**

<http://www.grogon.com/magic/index.php>

**Examples**

```
## Not run:  
h <- magic.8()  
h[,1]  
is.magic(h[,4])  
  
## End(Not run)
```

---

magic.constant

*Magic constant of a magic square or hypercube*

---

**Description**

Returns the magic constant: that is, the common sum for all rows, columns and (broken) diagonals of a magic square or hypercube

**Usage**

```
magic.constant(n,d=2,start=1)
```

**Arguments**

n	Order of the square or hypercube
d	Dimension of hypercube, defaulting to d=2 (a square)
start	Start value. Common values are 0 and 1

**Details**

If n is an integer, interpret this as the order of the square or hypercube; return  $n(\text{start} + n^d - 1)/2$ .

If n is a square or hypercube, return the magic constant for a normal array (starting at 1) of the same dimensions as n.

**Author(s)**

Robin K. S. Hankin

**See Also**

[magic](#)

**Examples**

```
magic.constant(4)
```

---

`magic.prime`*Magic squares prime order*

---

**Description**

Produces magic squares of order using the standard method

**Usage**

```
magic.prime(n, i=2, j=3)
```

**Arguments**

n	The order of the square
i	row number of increment
j	column number of increment

**Details**

Claimed to work for prime order, but I've tried it (with the defaults for *i* and *j*) for many integers of the form  $6n + 1$  and  $6n - 1$  and found no exceptions; indeed, they all seem to be panmagic. It is not clear to me when the process works and when it doesn't.

**Author(s)**

Robin K. S. Hankin

**References**

<http://www.magic-squares.de/general/general.html>

**Examples**

```
magic.prime(7)
f <- function(n){is.magic(magic.prime(n))}
all(sapply(6*1:30+1, f))
all(sapply(6*1:30-1, f))

is.magic(magic.prime(9, i=2, j=4), give.answers=TRUE)
magic.prime(7, i=2, j=4)
```

---

magic.product	<i>Product of two magic squares</i>
---------------	-------------------------------------

---

### Description

Gives a magic square that is a product of two magic squares.

### Usage

```
magic.product(a, b, mat=NULL)
magic.product.fast(a, b)
```

### Arguments

a	First magic square; if a is an integer, use <code>magic(a)</code> .
b	as above
mat	Matrix, of same size as a, of integers treated as modulo 8. Default value of NULL equivalent to passing <code>a*0</code> . Each number from 0-7 corresponds to one of the 8 squares which have the same Frenicle's standard form, as generated by <code>transf()</code> . Then subsquares of the product square (ie tiles of the same size as b) are rotated and transposed appropriately according to their corresponding entry in mat. This is a lot easier to see than to describe (see examples section).

### Details

Function `magic.product.fast()` does not take a `mat` argument, and is equivalent to `magic.product()` with `mat` taking the default value of `NULL`. The improvement in speed is doubtful unless  $\text{order}(a) \gg \text{order}(b)$ , in which case there appears to be a substantial saving.

### Author(s)

Robin K. S. Hankin

### References

William H. Benson and Oswald Jacoby. New recreations with magic squares, Dover 1976 (and that paper in JRM)

### Examples

```
magic.product(magic(3),magic(4))
magic.product(3,4)

mat <- matrix(0,3,3)
a <- magic.product(3,4,mat=mat)
mat[1,1] <- 1
b <- magic.product(3,4,mat=mat)

a==b
```

---

`magiccube.2np1`*Magic cubes of order  $2n+1$* 

---

**Description**

Creates odd-order magic cubes

**Usage**

```
magiccube.2np1(m)
```

**Arguments**

```
m          n=2m+1
```

**Author(s)**

Robin K. S. Hankin

**References**

website

**See Also**

[magic](#)

**Examples**

```
#try with m=3, n=2*3+1=7:

m <-7
n <- 2*m+1

apply(magiccube.2np1(m),c(1,2),sum)
apply(magiccube.2np1(m),c(1,3),sum)
apply(magiccube.2np1(m),c(2,3),sum)

#major diagonal checks out:
sum(magiccube.2np1(m)[matrix(1:n,n,3)])

#now other diagonals:
b <- c(-1,1)
f <- function(dir,v){if(dir>0){return(v)}else{return(rev(v))}}
g <- function(jj){sum(magiccube.2np1(m)[sapply(jj,f,v=1:n)])}
apply(expand.grid(b,b,b),1,g) #each diagonal twice, once per direction.
```

---

`magiccubes`*Magic cubes of order 3*

---

**Description**

A list of four elements listing each fundamentally different magic cube of order 3

**Usage**

```
data(magiccubes)
```

**Source**

Originally discovered by Hendricks

**References**

[http://members.shaw.ca/hdhcubes/cube\\_perfect.htm](http://members.shaw.ca/hdhcubes/cube_perfect.htm)

**Examples**

```
data(magiccubes)
magiccubes$a1
sapply(magiccubes, is.magichypercube)
```

---

`magichypercube.4n`*Magic hypercubes of order 4n*

---

**Description**

Returns magic hypercubes of order  $4n$  and any dimension.

**Usage**

```
magichypercube.4n(m, d = 3)
```

**Arguments**

<code>m</code>	Magic hypercube produced of order $n = 4m$
<code>d</code>	Dimensionality of cube

**Details**

Uses a rather kludgy (but vectorized) method. I am not 100% sure that the method does in fact produce magic squares for all orders and dimensions.

**Author(s)**

Robin K. S. Hankin

**Examples**

```
magichypercube.4n(1,d=4)
magichypercube.4n(2,d=3)
```

---

magicplot

*Joins consecutive numbers of a magic square.*

---

**Description**

A nice way to graphically represent normal magic squares. Lines are plotted to join successive numbers from 1 to  $n^2$ . Many magic squares have attractive such plots.

**Usage**

```
magicplot(m, number = TRUE, do.circuit = FALSE, ...)
```

**Arguments**

m	Magic square to be plotted.
number	Boolean variable with TRUE meaning to include the numbers on the plot
do.circuit	Boolean variable with TRUE meaning to include the line joining $n^2$ to 1
...	Extra parameters passed to plot()

**Author(s)**

Robin K. S. Hankin

**Examples**

```
magicplot(magic.4n(2))
```

---

minmax	<i>are all elements of a vector identical?</i>
--------	--

---

### Description

Returns TRUE if and only if all elements of a vector are identical.

### Usage

```
minmax(x, tol=1e-6)
```

### Arguments

x	Vector to be tested
tol	Relative tolerance allowed

### Details

If x is an integer, exact equality is required. If real or complex, a relative tolerance of tol is required. Note that functions such as `is.magic()` and `is.semimagichypercube()` use the default value for tol. To change this, define a new Boolean function that tests the sum to the required tolerance, and set boolean to TRUE

### Author(s)

Robin K. S. Hankin

### See Also

`is.magic()`

### Examples

```
data(Ollerenshaw)
minmax(subsums(Ollerenshaw,2)) #should be TRUE, as per is.2x2.correct()
```

---

`notmagic.2n`*An unmagic square*

---

**Description**

Returns a square of order  $n = 2m$  that has been claimed to be magic, but isn't.

**Usage**`notmagic.2n(m)`**Arguments**

`m` Order of square is  $n = 2m$

**Note**

This took me a whole evening to code up. And I was quite pleased with the final vectorized form: it matches Andrews's (8 by 8) example square exactly. What a crock

**Author(s)**

Robin K. S. Hankin

**References**

"Magic Squares and Cubes", Andrews, (book)

**Examples**

```
notmagic.2n(4)
is.magic(notmagic.2n(4))
is.semimagic(notmagic.2n(4))
```

---

`nqueens`*N queens problem*

---

**Description**

Solves the N queens problem for any n-by-n board.

**Usage**

```
bernhardsson(n)
bernhardssonA(n)
bernhardssonB(n)
```

**Arguments**

n                      Size of the chessboard

**Details**

Uses a direct transcript of Bo Bernhardsson's method.

All solutions (up to reflection and translation) for the 8-by-8 case given in the examples.

**Author(s)**

Robin K. S. Hankin

**References**

- Bo Bernhardsson 1991. "Explicit solutions to the n-queens problem for all  $n$ ". *SIGART Bull.*, 2(2):7
- Weisstein, Eric W. "Queens Problem" From *MathWorld—A Wolfram Web Resource* <http://mathworld.wolfram.com/QueensProblem.html>

**Examples**

```
bernhardsson(7)

a <-
  matrix(
    c(3,6,2,7,1,4,8,5,
      2,6,8,3,1,4,7,5,
      6,3,7,2,4,8,1,5,
      3,6,8,2,4,1,7,5,
      4,8,1,3,6,2,7,5,
      7,2,6,3,1,4,8,5,
      2,6,1,7,4,8,3,5,
      1,6,8,3,7,4,2,5,
      1,5,8,6,3,7,2,4,
      2,4,6,8,3,1,7,5,
      6,3,1,8,4,2,7,5,
      4,6,8,2,7,1,3,5)
    ,8,12)

out <- array(0L,c(8,8,12))
for(i in 1:12){
  out[cbind(seq_len(8),a[,i],i)] <- 1L
}
```

---

Ollerenshaw

*A most perfect square due to Ollerenshaw*

---

**Description**

A 12-by-12 most perfect square due to Ollerenshaw

**Usage**

```
data(Ollerenshaw)
```

**Source**

“Most perfect pandiagonal magic squares”, K. Ollerenshaw and D. Bree, 1998, Institute of Mathematics and its applications

**Examples**

```
data(Ollerenshaw)
is.mostperfect(Ollerenshaw)
```

---

panmagic.4

*Panmagic squares of order 4*

---

**Description**

Creates all fundamentally different panmagic squares of order 4.

**Usage**

```
panmagic.4(vals = 2^(0:3))
```

**Arguments**

**vals** a length four vector giving the values which are combined in each of the  $2^4$  possible ways. Thus `vals=2*sample(0:3)` always gives a normal square (0-15 in binary).

**Author(s)**

Robin K. S. Hankin

**References**

<http://www.grogon.com/magic/index.php>

**Examples**

```
panmagic.4()  
panmagic.4(2^c(1,3,2,0))  
panmagic.4(10^(0:3))
```

---

panmagic.8

*Panmagic squares of order 8*

---

**Description**

Produces each of a wide class of order 8 panmagic squares

**Usage**

```
panmagic.8(chosen = 1:6, vals = 2^(0:5))
```

**Arguments**

chosen	Which of the magic carpets are used in combination
vals	The values combined to produce the magic square. Choosing 0:5 gives a normal magic square.

**Note**

Not all choices for chosen give normal magic squares. There seems to be no clear pattern. See website in references for details.

**Author(s)**

Robin K. S. Hankin

**References**

<http://www.grogono.com/magic/index.php>

**See Also**

[panmagic.4](#)

**Examples**

```
is.panmagic(panmagic.8(chosen=2:7))  
is.normal(panmagic.8(chosen=2:7))  
is.normal(panmagic.8(chosen=c(1,2,3,6,7,8)))
```

```
#to see the twelve basis magic carpets, set argument 'chosen' to each  
#integer from 1 to 12 in turn, with vals=1:
```

```
panmagic.8(chosen=1,vals=1)-1  
image(panmagic.8(chosen=12,vals=1))
```

---

perfectcube5	<i>A perfect magic cube of order 5</i>
--------------	--

---

**Description**

A perfect cube of order 5, due to Trump and Boyer

**Usage**

```
data(perfectcube5)
```

**Examples**

```
data(perfectcube5)  
is.perfect(perfectcube5)
```

---

perfectcube6	<i>A perfect cube of order 6</i>
--------------	----------------------------------

---

**Description**

A perfect cube of order 6 originally due to Trump

**Usage**

```
data(perfectcube6)
```

**Examples**

```
data(perfectcube6)  
is.perfect(perfectcube6)  
is.magichypercube(perfectcube6[2:5,2:5,2:5])
```

---

process	<i>Force index arrays into range</i>
---------	--------------------------------------

---

**Description**

Forces an (integer) array to have entries in the range 1-n, by (i) taking the entries modulo n, then (ii) setting zero elements to n. Useful for modifying index arrays into a form suitable for use with magic squares.

**Usage**

```
process(x, n)
```

**Arguments**

x	Index array to be processed
n	Modulo of arithmetic to be used

**Author(s)**

Robin K. S. Hankin

**Examples**

```
# extract the broken diagonal of magic.2np1(4) that passes
# through element [1,5]:

a <- magic.2np1(4)
index <- t(c(1,5)+rbind(1:9,1:9))
a[process(index,9)]
```

---

recurse	<i>Recursively apply a permutation</i>
---------	--

---

**Description**

Recursively apply a permutation to a vector an arbitrary number of times. Negative times mean apply the inverse permutation.

**Usage**

```
recurse(perm, i, start = seq_along(perm))
```

**Arguments**

perm	Permutation (integers 1 to length(start) in some order)
start	Start vector to be permuted
i	Number of times to apply the permutation. i=0 gives start by definition and negative values use the inverse permutation

**Author(s)**

Robin K. S. Hankin

**See Also**

[hudson](#)

**Examples**

```
n <- 15
noquote(recurse(start=letters[1:n],perm=shift(1:n),i=0))
noquote(recurse(start=letters[1:n],perm=shift(1:n),i=1))
noquote(recurse(start=letters[1:n],perm=shift(1:n),i=2))

noquote(recurse(start=letters[1:n],perm=sample(n),i=1))
noquote(recurse(start=letters[1:n],perm=sample(n),i=2))
```

---

sam

*Sparse antimagic squares*

---

**Description**

Produces an antimagic square of order  $m$  using Gray and MacDougall's method.

**Usage**

```
sam(m, u, A=NULL, B=A)
```

**Arguments**

m	Order of the magic square (not "n": the terminology follows Gray and MacDougall)
u	See details section
A,B	Start latin squares, with default NULL meaning to use <code>circulant(m)</code>

**Details**

In Gray's terminology, `sam(m,n)` produces a  $SAM(2m, 2u + 1, 0)$ .

The method is not vectorized.

To test for these properties, use functions such as `is.anti-magic()`, documented under `is.magic.Rd`.

**Author(s)**

Robin K. S. Hankin

**References**

I. D. Gray and J. A. MacDougall 2006. "Sparse anti-magic squares and vertex-magic labelings of bipartite graphs", *Discrete Mathematics*, volume 306, pp2878-2892

**See Also**

[magic](#), [is.magic](#)

**Examples**

```
sam(6,2)

jj <- matrix(c(
  5, 2, 3, 4, 1,
  3, 5, 4, 1, 2,
  2, 3, 1, 5, 4,
  4, 1, 2, 3, 5,
  1, 4, 5, 2, 3),5,5)

is.sam(sam(5,2,B=jj))
```

---

shift

*Shift origin of arrays and vectors*

---

**Description**

Shift origin of arrays and vectors.

**Usage**

```
shift(x, i=1)
ashift(a, v=rep(1,length(dim(a))))
```

**Arguments**

x	Vector to be shifted
i	Number of places elements to be shifted, with default value of 1 meaning to put the last element first, followed by the first element, then the second, etc
a	Array to be shifted
v	Vector of numbers to be shifted in each dimension, with default value corresponding to <code>shift()</code> ing each dimension by 1 unit. If the length of <code>v</code> is less than <code>length(dim(a))</code> , it is padded with zeroes (thus a scalar value of <code>i</code> indicates that the first dimension is to be shifted by <code>i</code> units)

**Details**

Function `shift(x,n)` returns  $P^n(x)$  where  $P$  is the permutation  $(n, 1, 2, \dots, n-1)$ .

Function `ashift` is the array generalization of this: the  $n^{\text{th}}$  dimension is shifted by `v[n]`. In other words, `ashift(a,v)=a[shift(1:(dim(a)[1]),v[1]),...,shift(1:(dim(a)[n]),v[n])]`. It is named by analogy with `abind()` and `aperm()`.

This function is here because a shifted semimagic square or hypercube is semimagic and a shifted pandiagonal square or hypercube is pandiagonal (note that a shifted magic square is not necessarily magic, and a shifted perfect hypercube is not necessarily perfect).

**Author(s)**

Robin K. S. Hankin

**Examples**

```
shift(1:10,3)
m <- matrix(1:100,10,10)
ashift(m,c(1,1))
ashift(m,c(0,1)) #note columns shifted by 1, rows unchanged.
ashift(m,dim(m)) #m unchanged (Mnemonic).
```

---

strachey

*Strachey's algorithm for magic squares*

---

**Description**

Uses Strachey's algorithm to produce magic squares of singly-even order.

**Usage**

```
strachey(m, square=magic.2np1(m))
```

**Arguments**

m	magic square produced of order $n=2m+1$
square	magic square of order $2m+1$ needed for Strachey's method. Default value gives the standard construction, but the method will work with any odd order magic square

**Details**

Strachey's method essentially places four identical magic squares of order  $2m + 1$  together to form one of  $n = 4m + 2$ . Then  $0, n^2/4, n^2/2, 3n^2/4$  is added to each square; and finally, certain squares are swapped from the top subsquare to the bottom subsquare.

See the final example for an illustration of how this works, using a zero matrix as the submatrix. Observe how some 75s are swapped with some 0s, and some 50s with some 25s.

**Author(s)**

Robin K. S. Hankin

**See Also**

[magic.4np2,lozenge](#)

**Examples**

```
strachey(3)
strachey(2,square=magic(5))

strachey(2,square=magic(5)) %eq% strachey(2,square=t(magic(5)))
#should be FALSE

#Show which numbers have been swapped:
strachey(2,square=matrix(0,5,5))

#It's still magic, but not normal:
is.magic(strachey(2,square=matrix(0,5,5)))
```

---

subsums

*Sums of submatrices*


---

**Description**

Returns the sums of submatrices of an array; multidimensional moving window averaging

**Usage**

```
subsums(a,p,func="sum",wrap=TRUE, pad=0)
```

**Arguments**

a	Array to be analysed
p	Argument specifying the subarrays to be summed. If a vector of length greater than one, it is assumed to be of length $d = \text{length}(\text{dim}(a))$ , and is interpreted to be the dimensions of the subarrays, with the size of the window's $n^{\text{th}}$ dimension being $a[n]$ . If the length of $p$ is one, recycling is used. If not a vector, is assumed to be a matrix with $d$ columns, each row representing the coordinates of the elements to be summed. See examples.
func	Function to be applied over the elements of the moving window. Default value of <code>sum</code> gives the sum as used in <code>is.2x2.correct()</code> ; other choices might be <code>mean</code> , <code>prod</code> , or <code>max</code> . If <code>sum=""</code> , return the array of dimension $c(\text{dim}(a), \text{prod}(p))$ where each hyperplane is a shifted version of $a$ .
wrap	Boolean, with default value of <code>TRUE</code> meaning to view array $a$ as a $n$ -dimensional torus. Thus, if $x = \text{subsums}(a, p, \text{wrap} = \text{TRUE})$ , and if $\text{dim}(a) = c(a_1, \dots, a_d)$ then $x[a_1, \dots, a_d]$ is the sum of all corner elements of $a$ . If <code>FALSE</code> , do not wrap $a$ and return an array of dimension $\text{dim}(a) + p - 1$ .
pad	If <code>wrap</code> is <code>TRUE</code> , <code>pad</code> is the value used to pad the array with. Use a "neutral" value here; for example, if <code>func = sum</code> , then use <code>0</code> ; if <code>max</code> , use <code>-\infty</code> .

**Details**

The offset is specified so that  $\text{allsums}(a, v)[1, 1, \dots, 1] = \text{sum}(a[1:v[1], 1:v[2], \dots, 1:v[n]])$ , where  $n = \text{length}(\text{dim}(a))$ .

Function `subsums()` is used in `is.2x2.correct()` and `is.diagonally.correct()`.

**Author(s)**

Robin K. S. Hankin

**Examples**

```
data(Ollerenshaw)
subsums(Ollerenshaw, c(2, 2))
subsums(Ollerenshaw[, 1:10], c(2, 2))
subsums(Ollerenshaw, matrix(c(0, 6), 2, 2)) # effectively, is.bree.correct()

# multidimensional example.
a <- array(1, c(3, 4, 2))
subsums(a, 2) # note that p=2 is equivalent to p=c(2, 2, 2);
# all elements should be identical

subsums(a, 2, wrap=FALSE) #note "middle" elements > "outer" elements

#Example of nondefault function:
x <- matrix(1:42, 6, 7)
subsums(x, 2, func="max", pad=Inf, wrap=TRUE)
subsums(x, 2, func="max", pad=Inf, wrap=FALSE)
```

---

transf	<i>Frenicle's equivalent magic squares</i>
--------	--

---

**Description**

For a given magic square, returns one of the eight squares whose Frenicle's standard form is the same.

**Usage**

```
transf(a, i)
```

**Arguments**

a	Magic square
i	Integer, considered modulo 8. Specifying 0-7 gives a different magic square

**Author(s)**

Robin K. S. Hankin

**See Also**

[is.standard](#)

**Examples**

```
a <- magic(3)
identical(transf(a,0),a)

transf(a,1)
transf(a,2)

transf(a,1) %eq% transf(a,7)
```

# Index

## \*Topic **array**

- adiag, 3
- allsubhypercubes, 5
- allsums, 6
- apad, 8
- apl, 9
- aplus, 10
- arev, 11
- arot, 12
- arow, 13
- as.standard, 14
- circulant, 17
- diag.off, 19
- eq, 21
- fnsd, 22
- force.integer, 23
- hadamard, 24
- hendricks, 25
- hudson, 25
- is.magic, 26
- is.magichypercube, 29
- is.ok, 32
- is.square.palindromic, 33
- latin, 34
- lozenge, 36
- magic, 37
- magic.2np1, 38
- magic.4n, 39
- magic.4np2, 39
- magic.8, 40
- magic.constant, 41
- magic.prime, 42
- magic.product, 43
- magiccube.2np1, 44
- magichypercube.4n, 45
- magicplot, 46
- minmax, 47
- notmagic.2n, 48
- nqueens, 48

- panmagic.4, 50
- panmagic.8, 51
- process, 53
- recurse, 53
- sam, 54
- shift, 55
- strachey, 56
- subsums, 57
- transf, 59

## \*Topic **datasets**

- cube2, 18
- Frankenstein, 24
- hendricks, 25
- magiccubes, 45
- Ollerenshaw, 50
- perfectcube5, 52
- perfectcube6, 52

## \*Topic **math**

- do.index, 20

## \*Topic **package**

- magic-package, 3

- %eq% (eq), 21
- %ge% (eq), 21
- %gt% (eq), 21
- %le% (eq), 21
- %lt% (eq), 21
- %ne% (eq), 21

- adiag, 3, 9
- allsubhypercubes, 5, 31
- allsums, 6
- another\_incidence (latin), 34
- another\_latin (latin), 34
- apad, 4, 8, 11
- apl, 9
- apldrop (apl), 9
- apldrop<- (apl), 9
- apltake (apl), 9
- apltake<- (apl), 9
- aplus, 10

- arev, [11](#), [13](#), [22](#)
- arot, [12](#)
- arow, [13](#), [20](#)
- as.standard, [14](#), [22](#), [37](#)
- ashift, [12](#)
- ashift (shift), [55](#)
  
- bernhardsson (nqueens), [48](#)
- bernhardssonA (nqueens), [48](#)
- bernhardssonB (nqueens), [48](#)
  
- cilleruelo, [16](#)
- circulant, [17](#)
- cube2, [18](#)
  
- diag.off, [19](#)
- do.index, [20](#)
  
- eq, [15](#), [21](#)
  
- fnsd, [22](#)
- force.integer, [23](#), [37](#)
- Frankenstein, [24](#)
  
- ge (eq), [21](#)
- gt (eq), [21](#)
  
- hadamard, [24](#)
- hendricks, [25](#), [31](#)
- hudson, [25](#), [54](#)
  
- inc\_to\_inc (latin), [34](#)
- incidence (latin), [34](#)
- is.2x2.correct (is.magic), [26](#)
- is.alicehypercube (is.magichypercube), [29](#)
- is.antimagic (is.magic), [26](#)
- is.associative (is.magic), [26](#)
- is.bree.correct (is.magic), [26](#)
- is.centrosymmetric  
(is.square.palindromic), [33](#)
- is.circulant (circulant), [17](#)
- is.diagonally.correct  
(is.magichypercube), [29](#)
- is.hadamard (hadamard), [24](#)
- is.incidence (latin), [34](#)
- is.latin (is.magic), [26](#)
- is.latinhypercube (is.magichypercube), [29](#)
- is.magic, [7](#), [15](#), [26](#), [31](#), [35](#), [55](#)
- is.magichypercube, [29](#)
- is.mostperfect (is.magic), [26](#)
- is.nasik (is.magichypercube), [29](#)
- is.normal, [15](#)
- is.normal (is.magic), [26](#)
- is.ok, [32](#)
- is.pandiagonal (is.magic), [26](#)
- is.panmagic, [7](#), [19](#)
- is.panmagic (is.magic), [26](#)
- is.perfect, [6](#), [28](#)
- is.perfect (is.magichypercube), [29](#)
- is.persymmetric  
(is.square.palindromic), [33](#)
- is.regular (is.magic), [26](#)
- is.sam (is.magic), [26](#)
- is.semimagic, [7](#)
- is.semimagic (is.magic), [26](#)
- is.semimagichypercube, [28](#)
- is.semimagichypercube  
(is.magichypercube), [29](#)
- is.sparse (is.magic), [26](#)
- is.square.palindromic, [33](#)
- is.stam (is.magic), [26](#)
- is.standard, [59](#)
- is.standard (as.standard), [14](#)
- is.totally.antimagic (is.magic), [26](#)
- is.ultramagic (is.magic), [26](#)
  
- latin, [34](#)
- le (eq), [21](#)
- lozenge, [36](#), [37](#), [57](#)
- lt (eq), [21](#)
  
- magic, [15](#), [37](#), [38–41](#), [44](#), [55](#)
- magic-package, [3](#)
- magic.2np1, [37](#), [38](#)
- magic.4n, [37](#), [39](#)
- magic.4np2, [36](#), [37](#), [39](#), [57](#)
- magic.8, [40](#)
- magic.constant, [41](#)
- magic.prime, [37](#), [38](#), [42](#)
- magic.product, [43](#)
- magiccube.2np1, [44](#)
- magiccubes, [45](#)
- magichypercube.4n, [45](#)
- magicplot, [46](#)
- minmax, [28](#), [47](#)
  
- ne (eq), [21](#)

notmagic.2n, 48  
nqueens, 48

Ollerenshaw, 50

panmagic.4, 50, 51  
panmagic.8, 51  
perfectcube5, 52  
perfectcube6, 52  
process, 53

recurse, 26, 53  
rlatin(latin), 34

sam, 28, 54  
shift, 55  
strachey, 56  
subsums, 4, 57  
sylvester (hadamard), 24

take (apl), 9  
transf, 59

unincidence (latin), 34