Recreational mathematics with R: introducing the “magic” package

Creating and investigating magic squares with R

Robin K. S. Hankin

Preface

The R computer language (R Development Core Team, 2004) has been applied with a great deal of success to a wide variety of statistical, physical, and medical applications. Here, I show that R is an equally superb research tool in the field of recreational mathematics.

Recreational mathematics is easier to recognize than define, but seems to be characterized by requiring a bare minimum of ”raw material”: complex notation is not needed, and problems are readily communicated to the general public.

This is not to say that all problems of recreational mathematics are trivial: one could argue that much number theory is recreational in nature; yet attempts to prove Fermat’s Last Theorem, or the search for ever higher perfect numbers, have been the catalyst for the development of many fruitful new areas of mathematics.

The study of magic squares is also an example of nontrivial recreational mathematics as the basic concept is simple to grasp—yet there remain unsolved problems in the field whose study has revealed deep mathematical truths.

Here, I introduce the “magic” package, and show that R is an excellent environment for the creation and investigation of magic squares. I also show that one’s appreciation of magic squares may be enhanced through computer tools such as R, and that the act of translating ‘paper’ algorithms of the literature into R idiom can lead to new insight.

Magic squares

Magic squares have essentially zero practical use; their fascination—like much of pure mathematics—lies in the appeal of æsthetics and structure rather than immediate usefulness.

The following definitions are almost universal:

- A *semimagic square* is one all of whose row sums equal all its columnwise sums (i.e. 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.

(all squares are understood to be $n \times n$ and to be *normal*, that is, to comprise n^2 consecutive integers[1]). Functions `is.semimagic()`, `is.magic()`, and `is.panmagic()` test for these properties.

A good place to start is the simplest—and by far the most commonly encountered—magic square, *lo zhu*:

```
> magic(3)

     [,1] [,2] [,3]
[1,]    2    7    6
[2,]    9    5    1
[3,]    4    3    8
```

This magic square has been known since antiquity (legend has it that the square was revealed to humanity inscribed upon the shell of a divine turtle). More generally, if consecutive numbers of a magic square are joined by lines, a pleasing image is often obtained (figure 1, for example, shows a magic square of order 7; when viewed in this way, the algorithm for creating such a square should be immediately obvious).

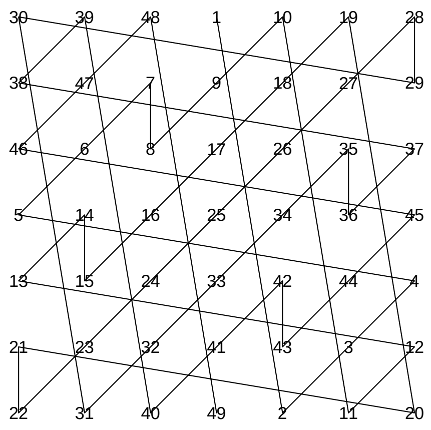


Figure 1: Magic square of order 7 in graphical form (obtained by `magicplot(magic.2np1(3))`)

[1]Most workers require the entries to start at 1, which is the convention here; but there are several instances where starting at 0 is far more convenient. In any case, if `x` is magic, then `x+n` is magic for any integer `n`.

Function `magic()` takes an integer argument n and returns a normal magic square of size $n \times n$. There are eight equivalent forms for *lo zhu* or indeed any magic square, achieved by rotating and reflecting the matrix (Benson and Jacoby, 1976); such equivalence is tested by `eq()` or `%eq%`. Of these eight forms, a magic square `a` is said to be in *Frénicle's standard form* if `a[1,1]`$\leq$`b[1,1]` whenever `a %eq% b`, and `a[1,2]<a[2,1]`. Function `is.standard()` tests for this, and function `as.standard()` places a magic square in standard form. Magic squares returned by `magic()` are always in standard form.

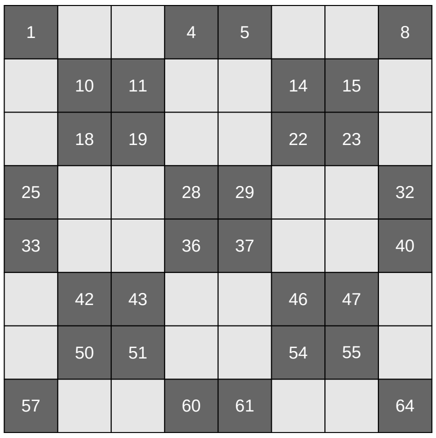
A typical (paper) algorithm for placing magic square `a` in standard form would be "rotate `a` until `a[1,1]<min(a[1,n],a[n,1],a[n,n])` then, if `a[1,2]>a[2,1]`, take the transpose". I shall show later that expressing such an algorithm in R leads to new insight when considering magic hypercubes.

A wide variety of algorithms exists for calculating magic squares. For a given order n, these algorithms generally depend on n modulo 4.

A typical paper algorithm for magic squares of order $n = 4m$ would go as follows.

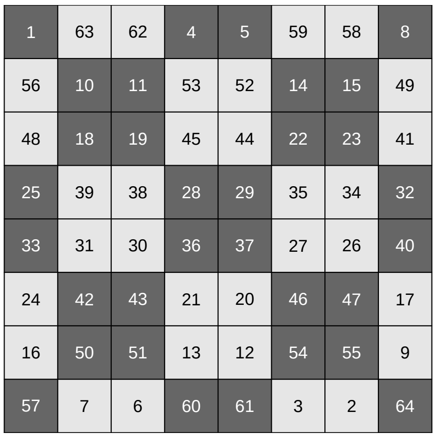
Algorithm 1: in a square of order $4m$, shade the long major diagonal. Then shade all major diagonals distant by a multiple of 4 cells from the long diagonal. Do the same with the minor diagonals. Then, starting with "1" at the top left corner and proceeding from left to right and top to bottom, count from 1 to n^2, filling in the shaded squares with the appropriate number and omitting the unshaded ones [figure 2]. Fill in the remaining (unshaded) squares in the same way, starting at the lower right corner, moving leftwards and upwards [figure 3].

Such paper algorithms are common in the literature but translating this one into code that uses R's vectorized tools effectively can lead to new insight. The magicness of such squares may be proved by considering the increasing and decreasing sequences separately.



1			4	5			8
	10	11			14	15	
	18	19			22	23	
25			28	29			32
33			36	37			40
	42	43			46	47	
	50	51			54	55	
57			60	61			64

Figure 2: Half-completed magic square of order 8



1	63	62	4	5	59	58	8
56	10	11	53	52	14	15	49
48	18	19	45	44	22	23	41
25	39	38	28	29	35	34	32
33	31	30	36	37	27	26	40
24	42	43	21	20	46	47	17
16	50	51	13	12	54	55	9
57	7	6	60	61	3	2	64

Figure 3: Magic square of order 8

The interesting part of the above paper algorithm lies in determining the pattern of shaded and unshaded squares[2]. As the reader may care to verify, parsing the algorithm into R idiom is not straightforward. An alternative, readily computed in R, would be to recognize that the repeating 4×4 cell `a[2:5,2:5]` is `kronecker(diag(2),matrix(1,2,2)) -> b` say, replicate it with `kronecker(matrix(1,3,3),b) -> g`; then trim off the border by selecting only the middle elements, in this case `g[2:9,2:9]`. Function `magic.4n()` implements the algorithm for general m.

[2]If `a <- matrix(1:(n*n),n,n)`, with `jj` a Boolean vector of length n^2 with `TRUE` corresponding to shaded squares, then with it is clear that `a[jj] <- rev(a[jj])` will return the above magic square.

Magic hypercubes

One of the great strengths of R is its ability to handle arbitrary dimensioned arrays in an efficient and elegant manner.

Generalizing magic squares to magic hypercubes (Hendricks, 1973) is thus natural when working in R. The following definitions represent a general consensus, but are far from universal:

- A *semimagic hypercube* has all "rook's move" sums equal to the magic constant (that is, each $\sum_{i_r=1}^{n} a[i_1, i_2, \ldots, i_{r-1}, i_r, i_{r+1}, \ldots, i_d]$ with $1 \leq r \leq d$ is equal to the magic constant for all values of the other i's).

- 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.

- A *perfect magic hypercube* is a magic hypercube with all nonbroken diagonals summing correctly[3].

- A *pandiagonal hypercube* is a perfect magic hypercube with all broken diagonals summing correctly.

(a magic hypercube is understood to be of dimension `rep(n,d)` and normal). Functions `is.semimagichypercube()`, `is.magichypercube()` and `is.perfect(a)` test for the first three properties; the fourth is not yet implemented. Function `is.diagonally.correct()` tests for correct summation of the 2^d (sic) long diagonals.

Magic hypercubes of order $4n$

Consider algorithm 1 generalized to a d-dimensional hypercube. The appropriate generalization of the repeating cell of the 8×8 magic square discussed above is not immediately obvious when considering figure 2, but the R formalism (viz `kronecker(diag(2),matrix(1,2,2))` makes it clear that the appropriate generalization is to replace `matrix(1,2,2)` with `array(1,rep(2,d))`.

The appropriate generalization for `diag(2)` (call it `g`) is not so straightforward, but one might be guided by the following requirements:

- The dimension of `g` must match the first argument to `kronecker()`, viz `rep(2,d)`

- The number of 0s must be equal to the number of 1s: `sum(g==1)==sum(g==0)`

- The observation that `diag(2)` is equal to its transpose would generalize to requiring that `aperm(g,K)` be identical to `g` for any permutation `K`.

These lead to specifying that `g[i1,...,id]` should be zero if $(i_1, \ldots, i_d)$ contains an odd number of 2s and one otherwise.

One appropriate R idiom would be to define a function `dimension(a,p)` to be an integer matrix with the same dimensions as `a`, with element $(n_1, n_2, ..., n_d)$ being n_p, then if `jj` $= \sum_{i=1}^{d}$ `dimension(a,i)`, we can specify `g=jj*0` and then `g[jj%%2==1] <- 1`.

Another application of `kronecker()` gives a hypercube that is of extent $4m + 2$ in each of its `d` dimensions, and this may be trimmed off as above to give an array of dimensions `rep(4m,d)` using `do.call()` and `[<-`. The numbers may be filled in exactly as for the 2d case.

The resulting hypercube is magic, in the sense defined above[4], although it is not perfect; function `magichypercube.4n()` implements the algorithm. The ability to generate magic hypercubes of arbitrary dimension greater than one is apparently novel.

Standard form for hypercubes

Consider again the paper definition for Frénicle's standard form of a magic square `a`: it is rotated so that the smallest number appears at the top left; then if `a[1,2]<a[2,1]`, the transpose is taken.

When coding up such an algorithm in R with an eye to generalizing it to arbitrarily high dimensional hypercubes, it becomes apparent that "rotation" is not a natural operation. The generalization used in the package is directly suggested by R's array capabilities: it is a two-step process in which the first step is to maneuver the smallest possible element to position `[1,1,...,1]` using only operations that reverse the index of some (or all) dimensions. An example would be `a <- a[1:n,n:1,1:n,n:1]`.

The second step is to use function `aperm()` to ensure that the appropriate generalization of `a[1,2]<a[2,1]`, which would be

$$\texttt{a[1,1,...,1,2]} < \texttt{a[1,...,2,1]} < \ldots$$
$$\ldots < \texttt{a[1,2,...,1]} < \texttt{a[2,1,...,1]}$$

holds; the appropriate R idiom is `a <- aperm(a,order(-a[1+diag(d)]))`.

This generalization of Frénicle's standard form to arbitrary dimensional hypercubes appears to be new; it arises directly from the power and elegance of R's array handling techniques.

[3]This condition is quite restrictive; in the case of a tesseract, this would include subsets such as $\sum_{i=1}^{n} a[1, i, n-i+1, n]$ summing correctly.

[4]If I had a rigorous proof of this, the margin might be too narrow for it.

Conclusions

The R language is a natural environment for the investigation of magic squares and hypercubes; and the discipline of translating published algorithms into R idiom can yield new insight. These insights include a new generalization of Frénicle's standard form to hypercubes, and also what appears to be the first algorithm for generating magic hypercubes of any dimension,

Insofar as magic squares and hypercubes are worthy of attention, it is worth creating fast, efficient routines to carry out the "paper" algorithms of the literature. I hope that the `magic` package will continue to facilitate the study of these fascinating objects.

Acknowledgements

I would like to acknowledge the many stimulating and helpful comments made by the R-help list over the years.

Bibliography

W. H. Benson and O. Jacoby. *New recreations with magic squares*. Dover, 1976. 49

J. R. Hendricks. Magic tesseracts and N-dimensional magic hypercubes. *Journal of Recreational Mathematics*, 6(3):193–201, 1973. 50

R Development Core Team. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria, 2004. URL `http://www.R-project.org`. ISBN 3-900051-07-0. 48

Robin Hankin
Southampton Oceanography Centre
European Way
Southampton
United Kingdom
SO14 3ZH
`r.hankin@soc.soton.ac.uk`

Programmer's Niche

How Do You Spell That Number?

John Fox

Frank Duan recently posted a question to the `r-help` mailing list asking how to translate numbers into words. The program described in this column is a cleaned-up and slightly enhanced version of my response to his question. I found the problem to be an interesting puzzle, and the solution uses several programming techniques that demonstrate the flexibility of R, including its ability to manipulate character-string data and to employ recursive function calls.

One intriguing aspect of the problem is that it required me to raise into consciousness my subconscious knowledge about how numbers are spoken and written in English. I was much more aware of these conventions in the languages (French and Spanish) that I had studied as a non-native speaker. A bit later, I realized that there are variations among English-speaking countries in the manner in which numbers are spoken and written down. Because I was born in the United States and have lived most of my adult life in Canada, I'm terminally confused about English spelling and usage. Canadian conventions are an amalgam of American and British rules.

In any event, it didn't take much time to see that the numbers from *one* to *nineteen* are represented by individual words; the numbers from *twenty-one* to *ninety-nine* are formed as compound words, with components for the tens and units digits — with the exceptions of multiples of ten (*twenty*, *thirty*, etc.), which are single words. The *Chicago Manual of Style* tells me that these compound words should be hyphenated (but offered little additional useful advice about how numbers are to be written out). Numbers from 100 to 999 are written by tacking on (at the left) a phrase like "six hundred" — that is, composed of a number from *one* to *nine* plus the suffix *hundred* (and there is no hyphen). Above this point, additional terms are added at the left, representing multiples of powers of 1000. In American English (and in Canada), the first few powers of 1000 have the following names, to be used as suffixes:

1000^1	*thousand*
1000^2	*million*
1000^3	*billion*
1000^4	*trillion*

Thus, for example, the number 210,363,258 would be rendered "two hundred ten million, three hundred sixty-three thousand, two hundred fifty-eight." There really is no point in going beyond trillions, because double-precision numbers can represent integers exactly only to about 15 decimal digits, or hundreds of trillions. Of course, I could allow numbers to be specified optionally by arbitrarily long character strings of numerals (e.g., `"210363258347237492310"`), but I didn't see a real need to go higher than hundreds of trillions.

One approach to converting numbers to words would be to manipulate the numbers as integers, but