Introduction to R

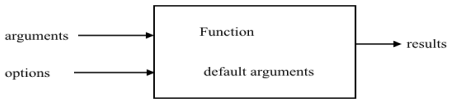
Free and powerful

Radu Trîmbiţaş

1 How R works

How R works

- When R is running, variables, data, functions, results, etc, are stored in the active memory of the computer in the form of *objects* which have a *name*.
- The user can do actions on these objects with *operators* (arithmetic, logical, comparison, . . .) and functions (which are themselves objects). The use of operators is relatively intuitive. An R function may be sketched as follows:



- The arguments can be objects (”data”, formulae, expressions, . . .), some of which could be defined by default in the function; these default values may be modified by the user by specifying options.
- An R function may require no argument: either all arguments are defined by default (and their values can be modified with the options), or no argument has been defined in the function.

How R works - continued

- All the actions of R are done on objects stored in the active memory of the computer: no temporary files are used (Figure 1). The readings and writings of files are used for input and output of data and results (graphics, . . .).

- The user executes the functions via some commands. The results are displayed directly on the screen, stored in an object, or written on the disk (particularly for graphics). Since the results are themselves objects, they can be considered as data and analysed as such.
- Data files can be read from the local disk or from a remote server through internet.
- The functions available to the user are stored in a library localised on the disk in a directory called R_HOME/library (R_HOME is the directory where R is installed).
- This directory contains *packages* of functions, which are themselves structured in directories. The package named `base` is in a way the core of R and contains the basic functions of the language, particularly, for reading and manipulating data. Each package has a directory called R with a file named like the package (for instance, for the package base, this is the file R_HOME/library/base/R/base). This file contains all the functions of the package.

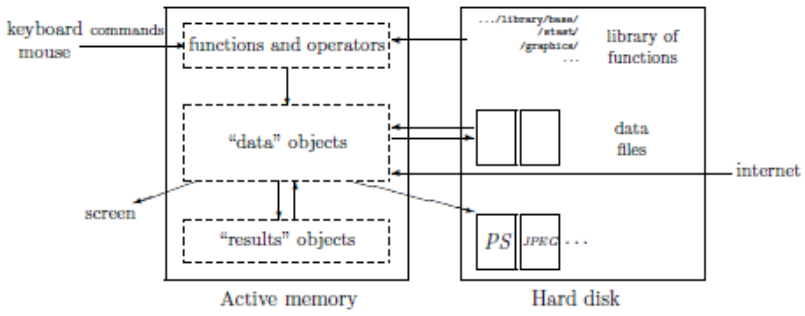


Figure 1: A schematic view of how R works.

2 R language essentials

2.1 Expressions, objects and functions

Expressions and objects

- The basic interaction mode in R is one of *expression evaluation*. The user enters an expression; the system evaluates it and prints the result. Some expressions are evaluated not for their result but for side effects such as putting up a graphics window or writing to a file.

- All R expressions return a value (possibly NULL), but sometimes it is "invisible" and not printed.
- Expressions typically involve variable references, operators such as +, and function calls, as well as some other items that have not been introduced yet.
- Expressions work on objects. This is an abstract term for anything that can be assigned to a variable. R contains several different types of objects. So far, we have almost exclusively seen numeric vectors, but several other types are introduced.

Functions and arguments

- Many things in R are done using *function calls*, commands that look like an application of a mathematical function of one or several variables; for example, `log(x)` or `plot(height, weight)`.
- The format is that a function name is followed by a set of parentheses containing one or more arguments. For instance, in `plot(height,weight)` the function name is `plot` and the arguments are `height` and `weight`.
- These are the *actual arguments*, which apply only to the current call. A function also has *formal arguments*, which get connected to actual arguments in the call.
- When you write `plot(height, weight)`, R assumes that the first argument corresponds to the x-variable and the second one to the y-variable. This is known as *positional matching*.
- The `plot` function is in fact an example of a function that has a large selection of arguments in order to be able to modify symbols, line widths, titles, axis type, and so forth. We used the alternative form of specifying arguments when setting the plot symbol to triangles with `plot(height, weight, pch=2)`.
- The `pch=2` form is known as a *named actual argument*, whose name can be matched against the formal arguments of the function and thereby allow keyword matching of arguments. The keyword `pch` was used to say that the argument is a specification of the plotting character.
- This type of function argument can be specified in arbitrary order. Thus, you can write `plot(y=weight,x=height)` and get the same plot as with `plot(x=height,y=weight)`.
- The two kinds of argument specification — positional and named — can be mixed in the same call.
- The formal arguments of a function are part of the function definition. The set of formal arguments to a function may be seen with `args`.

Examples

```
>args(plot)
function (x, y, ...)
NULL
> args(plot.default)
function (x, y = NULL, type = "p", xlim = NULL,
ylim = NULL, log = "", main = NULL, sub = NULL,
xlab = NULL, ylab = NULL, ann = par("ann"),
axes = TRUE, frame.plot = axes, panel.first = NULL,
panel.last = NULL, asp = NA, ...)
NULL
> args(ls)
function(name, pos=-1L, envir = as.environment(pos),
all.names = FALSE, pattern, sorted = TRUE)
NULL
```

2.2 Vectors

Vectors

Functions for vector creation: `c`, `seq`, `rep`.

`c` - "concatenate"

```
> c(42,57,12,39,1,3,4)
[1] 42 57 12 39 1 3 4
> x <- c("Huey", "Dewey", "Louie"); x
[1] "Huey"  "Dewey" "Louie"
```

You can also concatenate vectors of more than one element as in

```
> x <- c(1, 2, 3)
> y <- c(10, 20)
> c(x, y, 5)
[1]  1  2  3 10 20  5
```

It is also possible to assign names to the elements. This modifies the way the vector is printed and is often used for display purposes.

```
> x <- c(red="Huey", blue="Dewey", green="Louie")
> x
    red    blue   green
 "Huey" "Dewey" "Louie"
```

The names can be extracted or set using `names`:

```
> names(x)
[1] "red"   "blue"  "green"
```

All elements of a vector have the same type. If you concatenate vectors of different types, they will be converted to the least restrictive type:

```
> c(FALSE, 3)
[1] 0 3
> c(pi, "abc")
[1] "3.14159265358979" "abc"
> c(FALSE, "abc")
[1] "FALSE" "abc"
> c(1.2, 2, TRUE, "gaga")
[1] "1.2"  "2"    "TRUE" "gaga"
```

Vectors - sequences

The function `seq` (sequence), is used for equidistant series of numbers.

```
> seq(4,9)
[1] 4 5 6 7 8 9
> 4:9
[1] 4 5 6 7 8 9
```

If you want a sequence with a step, write

```
> seq(4,16,2)
[1]  4  6  8 10 12 14 16
```

Equivalent `seq(from=4, to=16, by=2)`.

The third function, `rep` (replicate), is used to generate repeated values. It is used in two variants, depending on whether the second argument is a vector or a single number:

```
> oops <- c(7,9,13)
> rep(oops,3)
[1]  7  9 13  7  9 13  7  9 13
> rep(oops,1:3)
[1]  7  9  9 13 13 13
```

The `rep` function is often used for things such as group codes: If it is known that the first 10 observations are men and the last 15 are women, you can use

```
> rep(1:2,c(10,15))
 [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

The special case where there are equally many replications of each value can be obtained using the each argument. E.g., `rep(1:2,each=10)` is the same as `rep(1:2,c(10,10))`.

Quoting and escaping sequences. Missing values

R allows vectors to contain a special `NA` value. This value is carried through in computations so that operations on `NA` yield `NA` as the result. `is.na`

```
> cat(c("Huey","Dewey","Louie"))
Huey Dewey Louie>
```

To get the system prompt onto the next line, you must include a newline character

```
> cat("Huey","Dewey","Louie", "\n")
Huey Dewey Louie
> cat("What is \"R\"?\n")
What is "R"?
```

Here, `\n` is an example of an *escape sequence*. It actually represents a single character, the linefeed (LF), but is represented as two. The backslash \is known as the *escape character*.

2.3 Matrices and arrays

Matrices and arrays

In R, the matrix notion is extended to elements of any type, so you could have, for instance, a matrix of character strings. Matrices and arrays are represented as vectors with dimensions:

```
> x <- 1:12
> dim(x) <- c(3,4)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

The `dim` assignment function sets or changes the *dimension attribute* of `x`, causing R to treat the vector of 12 numbers as a 3×4 matrix. Notice that the storage is column-major; that is, the elements of the first column are followed by those of the second, etc.

A convenient way to create matrices is to use the `matrix` function:

```
> matrix(1:12,nrow=3,byrow=T)
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```

Notice how the `byrow=T` switch causes the matrix to be filled in a rowwise fashion rather than columnwise.

Useful functions that operate on matrices include `rownames`, `colnames`, and the transposition function `t` (notice the lowercase `t` as opposed to uppercase `T` for `TRUE`), which turns rows into columns and vice versa:

```
> x <- matrix(1:12,nrow=3,byrow=T)
> rownames(x) <- LETTERS[1:3]
> x
  [,1] [,2] [,3] [,4]
A    1    2    3    4
B    5    6    7    8
C    9   10   11   12
> t(x)
     A B  C
[1,] 1 5  9
[2,] 2 6 10
[3,] 3 7 11
[4,] 4 8 12
```

You can glue vectors together, columnwise or rowwise, using the `cbind` and `rbind` functions.

```
> cbind(A=1:4,B=5:8,C=9:12)
     A B  C
[1,] 1 5  9
[2,] 2 6 10
[3,] 3 7 11
[4,] 4 8 12
> rbind(A=1:4,B=5:8,C=9:12)
  [,1] [,2] [,3] [,4]
A    1    2    3    4
B    5    6    7    8
C    9   10   11   12
```

A more general way to store data is in an *array*. Arrays have multiple indices, and are created using the `array` function:

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

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

, , 2

     [,1] [,2] [,3] [,4]
```

```
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```

Notice that the dimensions were specified in a vector `c(3, 4, 2)`. When inserting data, the first index varies fastest; when it has run through its full range, the second index changes, etc.

2.4 Factors

Factors

- It is common in statistical data to have categorical variables, indicating some subdivision of data, such as social class, primary diagnosis, tumor stage, etc. Typically, these are input using a numeric code.
- Such variables should be specified as factors in R. This is a data structure that (among other things) makes it possible to assign meaningful names to the categories.
- There are analyses where it is essential for R to be able to distinguish between categorical codes and variables whose values have a direct numerical meaning.
- The terminology is that a factor has a set of levels. Internally, a k-level factor consists of two items: (a) a vector of integers between 1 and k and (b) a character vector of length k containing strings describing what the k levels are.

```
> pain <- c(0,3,2,2,1)
> fpain <- factor(pain,levels=0:3)
> levels(fpain) <- c("none","mild","medium","severe")
```

The first command creates a numeric vector `pain`, encoding the pain levels of five patients. We wish to treat this as a categorical variable, so we create a factor `fpain` from it using the function `factor`. This is called with one argument in addition to `pain`, namely `levels=0:3`, which indicates that the input coding uses the values $0-3$. The latter can in principle be left out since R by default uses the values in pain, suitably sorted, but it is a good habit to retain it; see below. The effect of the final line is that the level names are changed to the four specified character strings.

```
> fpain
[1] none   severe medium medium mild
Levels: none mild medium severe
> as.numeric(fpain)
[1] 1 4 3 3 2
> levels(fpain)
[1] "none"   "mild"   "medium" "severe"
```

The function `as.numeric` extracts the numerical coding as numbers 1:4 and levels extracts the names of the levels.

R also allows you to create a special kind of factor in which the levels are ordered. This is done using the `ordered` function, which works similarly to `factor`.

2.5 Lists

Lists

Lists are objects consisting of an ordered collection of objects known as its *components* (which coud be of different types). Here is a simple example:

```
> Lst<-list(name="Fred", wife="Mary", no.children=3,
+ child.ages=c(4,7,9))
```

Components are always numbered and may always be referred to as such. Individual components of `Lst` may be individually referred to as `Lst[[1]]`, `Lst[[2]]`, `Lst[[3]]` and `Lst[[4]]`.

Components of lists may also be named, and in this case the component may be referred to either by giving the component name as a character string in place of the number in double square brackets, or, more conveniently, by giving an expression of the form `name$component_name` for the same thing. Examples:

```
> Lst[1]
$name
[1] "Fred"
> Lst[[1]]
[1] "Fred"
> Lst[[4]][1]
[1] 4
> Lst$wife
[1] "Mary"
> Lst["wife"]
$wife
[1] "Mary"
> Lst[["wife"]]
[1] "Mary"
```

The length of a list (number of components at outer level) with `length(name)`.

2.6 Data frames

Data frames

A data frame corresponds to what other statistical packages call a data matrix or a data set. It is a list of vectors and/or factors of the same length that

are related across such that data in the same position come from the same experimental unit (subject, animal, etc.). In addition, it has a unique set of row names.

```
>  intake.pre <- c(5260,5470,5640,6180,6390,
+ 6515,6805,7515,7515,8230,8770)
> intake.post <- c(3910,4220,3885,5160,5645,
+  4680,5265,5975,6790,6900,7335)
> d <- data.frame(intake.pre,intake.post)
> d
   intake.pre intake.post
1        5260        3910
2        5470        4220
3        5640        3885
4        6180        5160
5        6390        5645
6        6515        4680
7        6805        5265
8        7515        5975
9        7515        6790
10       8230        6900
11       8770        7335
```

As with lists, components (i.e., individual variables) can be accessed using the $ notation:

```
> d$intake.pre
 [1] 5260 5470 5640 6180 6390 6515 6805 7515
 [9] 7515 8230 8770
```

2.7 Indexing and selection

Indexing and selection

Indexing. Indexing an element of a vector

```
> intake.pre[5]
[1] 6390
```

Selection of a subvector of more elements, for instance elements 3, 5, 7

```
> intake.pre[c(3,5,7)]
[1] 5640 6390 6805
> v <- c(3,5,7)
> intake.pre[v]
[1] 5640 6390 6805
> intake.pre[1:5]
[1] 5260 5470 5640 6180 6390
```

A neat feature of R is the possibility of negative indexing. You can get all observations except nos. 3, 5, and 7 by writing

```
> intake.pre[-c(3,5,7)]
[1] 5260 5470 6180 6515 7515 7515 8230 8770
```

It is not possible to mix positive and negative indices. That would be highly ambiguous.

Conditional selection. In practice, you often need to extract data that satisfy certain criteria. This can be done simply by inserting a relational expression instead of the index.

```
> intake.post[intake.pre > 7000]
[1] 5975 6790 6900 7335
> intake.post[intake.pre > 7000 & intake.pre <= 8000]
[1] 5975 6790
```

The result of the logical expression is a logical vector

```
> intake.pre > 7000 & intake.pre <= 8000
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
 [9]  TRUE FALSE FALSE
```

Indexing with a logical vector implies that you pick out the values where the logical vector is TRUE, so in the preceding example we got the 8th and 9th values in `intake.post`.

If missing values (`NA`) appear in an indexing vector, then R will create the corresponding elements in the result but set the values to `NA`.

In addition to the relational and logical operators, there are a series of functions that return a logical value. A particularly important one is `is.na(x)`, which is used to find out which elements of `x` are recorded as missing (`NA`). Notice that there is a real need for `is.na` because you cannot make comparisons of the form `x==NA`. That simply gives `NA` as the result for any value of `x`. The result of a comparison with an unknown value is unknown!

Indexing and grouping in data frames

It is possible to extract variables from a data frame by typing, for example, `d$intake.post`. However, it is also possible to use a notation that uses the matrix-like structure directly:

```
> d <- data.frame(intake.pre,intake.post)
> d[5,1]
[1] 6390
> d[5,]
  intake.pre intake.post
5       6390        5645
```

`d[2]` is equivalent to `d[,2]`.

Other indexing techniques also apply, e.g. selection

```
> d[d$intake.pre>7000,]
   intake.pre intake.post
8        7515        5975
9        7515        6790
10       8230        6900
11       8770        7335
> #explain why
> sel <- d$intake.pre>7000
> sel
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
 [9] TRUE  TRUE  TRUE
> d[sel,]
   intake.pre intake.post
8        7515        5975
9        7515        6790
10       8230        6900
11       8770        7335
```

It is often convenient to look at the first few cases in a data set. This can be done with indexing. This is such a frequent occurrence that a convenience function called `head` exists. By default, it shows the first six lines. Similarly, `tail` shows the last part.

```
> #indexing
> d[1:2,]
  intake.pre intake.post
1       5260        3910
2       5470        4220
> #head
> head(d)
  intake.pre intake.post
1       5260        3910
2       5470        4220
3       5640        3885
4       6180        5160
5       6390        5645
6       6515        4680
> #tail
> tail(d)
   intake.pre intake.post
6        6515        4680
7        6805        5265
8        7515        5975
9        7515        6790
10       8230        6900
11       8770        7335
```

The natural way of storing grouped data in a data frame is to have the data themselves in one vector and parallel to that have a factor telling which data are from which group. Consider, for instance, the following data set on energy expenditure for lean and obese women.

```
> energy
   expend stature
1    9.21   obese
2    7.53    lean
3    7.48    lean
4    8.08    lean
5    8.09    lean
6   10.15    lean
7    8.40    lean
8   10.88    lean
9    6.13    lean
10   7.90    lean
11  11.51   obese
12  12.79   obese
13   7.05    lean
14  11.85   obese
15   9.97   obese
16   7.48    lean
17   8.79   obese
18   9.69   obese
19   9.68   obese
20   7.58    lean
21   9.19   obese
22   8.11    lean
```

This is a convenient format since it generalizes easily to data classified by multiple criteria. However, sometimes it is desirable to have data in a separate vector for each group. Fortunately, it is easy to extract these from the data frame:

```
> exp.lean <- energy$expend[energy$stature=="lean"]
> exp.obese <- energy$expend[energy$stature=="obese"]
```

Alternatively, you can use the split function, which generates a list of vectors according to a grouping.

```
> l <- split(energy$expend, energy$stature)
> l
$lean
 [1]  7.53  7.48  8.08  8.09 10.15  8.40 10.88  6.13  7.90
[10]  7.05  7.48  7.58  8.11

$obese
[1]  9.21 11.51 12.79 11.85  9.97  8.79  9.69  9.68  9.19
```

2.8 Implicit loops

Implicit loops

A common application of loops is to apply a function to each element of a set of values or vectors and collect the results in a single structure. In R this is abstracted by the functions `lapply` and `sapply`. The former always returns a list (hence the l), whereas the latter tries to simplify (hence the s) the result to a vector or a matrix if possible.

```
> head(thuesen)
  blood.glucose short.velocity
1          15.3           1.76
2          10.8           1.34
3           8.1           1.27
4          19.5           1.47
5           7.2           1.27
6           5.3           1.49
> lapply(thuesen, mean, na.rm=T)
$blood.glucose
[1] 10.3

$short.velocity
[1] 1.325652

> sapply(thuesen, mean, na.rm=T)
 blood.glucose short.velocity
     10.300000       1.325652
```

Sometimes you just want to repeat something a number of times but still collect the results as a vector. Obviously, this makes sense only when the repeated computations actually give different results, the common case being simulation studies. This can be done using `sapply`, but there is a simplified version called `replicate`, in which you just have to give a count and the expression to evaluate:

```
> replicate(10,mean(rexp(20)))
 [1] 1.0280245 1.3731307 0.8057787 1.2005030 0.8069861
 [6] 0.8026956 0.8648251 0.8730785 0.7314418 1.2089620
```

A similar function, `apply`, allows you to apply a function to the rows or columns of a matrix (or over indices of a multidimensional array in general) as in

```
> m <- matrix(rnorm(12),4)
> m
            [,1]         [,2]       [,3]
[1,]  0.27791413 -0.008309014  1.7635520
[2,] -0.82308112  0.128855402  0.7625865
[3,] -0.06884093 -0.145875628  1.1114311
```

```
[4,] -1.16766233 -0.163910957 -0.9232070
> apply(m, 2, min)
[1] -1.167662 -0.163911 -0.923207
```

Also, the function `tapply` allows you to create tables (hence the t) of the value of a function on subgroups defined by its second argument, which can be a factor or a list of factors. In the latter case a cross-classified table is generated. (The grouping can also be defined by ordinary vectors. They will be converted to factors internally.)

```
> tapply(energy$expend, energy$stature, median)
 lean obese
 7.90  9.69
```

2.9 Sorting

Sorting

Command `sort` is trivial

```
> intake.post
 [1] 3910 4220 3885 5160 5645 4680 5265 5975 6790 6900
[11] 7335
> sort(intake.post)
 [1] 3885 3910 4220 4680 5160 5265 5645 5975 6790 6900
[11] 7335
```

However, sorting a single vector is not always what is required. Often you need to sort a series of variables according to the values of some other variables blood pressures sorted by sex and age, for instance. For this purpose, you must first compute an ordering of a variable.

```
> order(intake$post)
 [1]  3  1  2  6  4  7  5  8  9 10 11
```

The point is that, by indexing with this vector, other variables can be sorted by the same criterion.

```
> order(intake$post)->o
> intake$post[o]
 [1] 3885 3910 4220 4680 5160 5265 5645 5975 6790 6900
[11] 7335
> intake$pre[o]
 [1] 5640 5260 5470 6515 6180 6805 6390 7515 7515 8230
[11] 8770
```

It is of course also possible to sort the entire data frame `intake`

```
> intake.sorted <- intake[o,]
> intake.sorted
```

```
    pre post
3  5640 3885
1  5260 3910
2  5470 4220
6  6515 4680
4  6180 5160
7  6805 5265
5  6390 5645
8  7515 5975
9  7515 6790
10 8230 6900
11 8770 7335
```

Sorting by several criteria is done simply by having several arguments to `order`; for instance, `order(sex,age)` will give a main division into men and women, and within each sex an ordering by age. The second variable is used when the order cannot be decided from the first variable. Sorting in reverse order can be handled by, for example, changing the sign of the variable.

3 Operators, Matrices and Linear Algebra

3.1 Operators

Operators

Operators				
Arithmetic		Comparison	Logical	
+	addition	<	!x	not
-	subtraction	>	x&y	and
*	multiplication	<=	x&&y	and f.
/	division	>=	x|y	or
^	power	==	x||y	or f.
%%	modulo	!=	xor(x,y)	exclusive or
%/%	integer division			

The following characters are also operators for R: `$`, `@`, `[`, `[[`, `:`, `?`, `<-`, `<<-`, `=`, `::`. A table of operators describing precedence rules can be found with `?Syntax`.

3.2 Matrix facilities

Outer Product

If a and b are two numeric arrays, their *outer product* is an array whose dimension vector is obtained by concatenating their two dimension vectors (order is important), and whose data vector is got by forming all possible products of elements of the data vector of a with those of b. The outer product is formed by the special operator `%o%`:

```
> a<-1:3
> b<-4:6
> ab<-a%o%b
> ab
     [,1] [,2] [,3]
[1,]    4    5    6
[2,]    8   10   12
[3,]   12   15   18
```

An alternative is `ab<-outer(a,b,"*")`. The multiplication function can be replaced by an arbitrary function of two variables. For example if we wished to evaluate the function $f(x, y) = cos(y)/(1 + x^2)$ over a regular grid of values with x- and y-coordinates defined by the R vectors `x` and `y` respectively, we could proceed as follows:

```
> f <- function(x, y) cos(y)/(1 + x^2)
> z <- outer(x, y, f)
```

3.3 Matrix facilities

Matrix facilities

`t(X)` - transpose of X
`nrow(A)`
`ncol(A)`
`%*%` - matrix multiplication
If x is a vector, then `x %*% A %*% x` is a quadratic form.

The function `crossprod()` forms crossproducts, meaning that `crossprod(X,y)` is the same as `t(X) %*% y` but the operation is more efficient. If the second argument to `crossprod()` is omitted it is taken to be the same as the first.

The meaning of `diag()` depends on its argument. `diag(v)`, where v is a vector, gives a diagonal matrix with elements of the vector as the diagonal entries. On the other hand `diag(M)`, where `M` is a matrix, gives the vector of main diagonal entries of `M`. This is the same convention as that used for `diag()` in MATLAB. Also, somewhat confusingly, if `k` is a single numeric value then `diag(k)` is the `k` by `k` identity matrix!

To solve the system $Ax = b$ use `solve(A,b)`. This is mathematically equivalent to $x = A^{-1}b$. The inverse of A^{-1} of A can be computed by `solve(A)`. The function `eig` computes eigenvalues and eigenvectors of a matrix, `det` the determinant and `svd` the singular eigenvalue decomposition. Examples:

```
> A<-matrix(1:4,2,2)
> A
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> b<-c(4,6)
> x<-solve(A,b); x
```

```
[1] 1 1
> X<-solve(A)
> A%*%X
     [,1] [,2]
[1,]    1    0
[2,]    0    1
> X%*%A
     [,1] [,2]
[1,]    1    0
[2,]    0    1
> ev<-eigen(A)
> ev
$values
[1]  5.3722813 -0.3722813

$vectors
           [,1]       [,2]
[1,] -0.5657675 -0.9093767
[2,] -0.8245648  0.4159736

> det(A)
[1] -2
> sd<-svd(A); sd
$d
[1] 5.4649857 0.3659662

$u
           [,1]       [,2]
[1,] -0.5760484 -0.8174156
[2,] -0.8174156  0.5760484

$v
           [,1]       [,2]
[1,] -0.4045536  0.9145143
[2,] -0.9145143 -0.4045536
> sd$u%*%diag(sd$d)%*%t(sd$v)
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Bibliography

References

[1] W. N. Venables, D. M. Smith and the R Development Core Team, *An Introduction to R*, 2015

[2] Peter Dalgaard, *Introductory Statistics with R*, 2nd ed., Springer Verlag, 2008.

[3] Norman Matloff, *THE ART OF R PROGRAMMING. A Tour of Statistical Software Design*, No Starch Press, San Francisco, 2011