An introduction to R: free software for modeling & computing

Juri Hinz[1]

[1] University of Technology Sydney

February 19, 2019

What is R

R can be considered as a *free* implementation of the Splus

- Splus is a value-added version of S sold by Insightful Corporation (formerly Math- Soft, Inc.
- S is a very high level language and an environment for data analysis and graphics.

S was written by Richard A. Becker, John M. Chambers, and Allan R. Wilks at Lucent Technologies (formerly AT&T Bell Laboratories) Statistics Research Department.

S is a very general tool, so that applications are not restricted to any particular subject area. S has been used for business, finance and experimental science.

S is a language. That is why the authors of S prefer that you not call S a statistics package, despite the fact that S is used in statistics and graphics.

S System was recognized with the 'Association for Computing Machinery (ACM) Software Systems Award'. Software systems recognized by this previous awards include UNIX operating system, World Wide Web, TCP/IP, and Postscript language.

The ACM named John M. Chambers as the recipient of the Award, mentioning that Chambers's work 'forever altered the way people analyze, visualize, and manipulate data... S is an elegant, widely accepted, and enduring software system, with conceptual integrity...

S-Plus is a fully supported and documented (statistical) application based on S. It is available in both UNIX, Windows, and Linux. S-Plus is a very comprehensive environment with over 4,000 built-in functions. There are add-on packages for financial data analysis, see http://www.insightful.com.

The disadvantages to use R instead of Splus are

- no graphical surface (unlike in Splus)
- no support (unlike for Splus users)
- some packages are less user-friendly than in Splus
- some packages have a better documentation in Splus and less bugs

However, the advantages to use R instead of Splus are

- R is free (under GNU)
- there are many contributed packages for R
- broad scientific community can be connected (http://www.r-project.org/)

For these reasons, we focus on R instead on Splus in this lecture. Both systems are closely related. In most applications, it should be not difficult to modify an R-scripts to run on Splus.

Getting started

The latest copy of R can be downloaded from the CRAN (Comprehensive R Archive Network) web site:

http://lib.stat.cmu.edu/R/CRAN/

For some problems additional packages are required. R packages can also be downloaded from the same site. An easier way, however is to install R first, to run it and to install packages via R. Manuals, FAQ, mailing lists, reference material can be found on the R web page.

Most frequently, R is used as a dialog-like environment

- user keys in a command
- R gives a response (text or graphical output)

There is no graphical surface. It turns out that the work becomes much more efficient, if the user communicates with R via

Rstudio

RStudio is a set of integrated tools designed to help you be more productive with R. It includes a console, syntax-highlighting editor that supports direct code execution, as well as tools for plotting, history, debugging and workspace management.

https://www.rstudio.com/products/rstudio/download/

After start, the R language expects input after the prompt

```
>
```

you can finish the program by typing

```
>q()
```

any specific named function, for example solve, the command is

```
> help(solve)
```

You can obtain help from the system.

- to launch a Web browser that allows to show the help pages type

  ```
  >help.start()
  ```

- to obtain help on particular topic, (say, on *solve* routine) type

  ```
  >?solve
  ```

R operates on named data structures. The simplest such structure is the numeric vector, which is a single entity consisting of an ordered collection of numbers. To set up a vector named x, say, consisting of five numbers, namely 10.4, 5.6, 3.1, 6.4 and 21.7, use the R command

```
> x <- c(10.4, 5.6, 3.1, 6.4, 21.7)
```

In this context, the concatenation function *c*() creates the vector, which is assigned to the object *x*. Equivalent commands are

```
> assign("x", c(10.4, 5.6, 3.1, 6.4, 21.7))
> c(10.4, 5.6, 3.1, 6.4, 21.7) -> x
```

If an expression is used as a complete command, the value is printed and lost. So now if we were to use the command

```
> 1/x
```

the reciprocals of the five values would be printed at the terminal (and the value of x, of course, unchanged). If you want to recall and to save the value of the last calculation (say as an object with name *value*), type

```
>value <- .Last.value
```

If you want to look at an object, just type its name, for instance

```
>value
```

gives the result of the last calculation, according to the previous assignment.

The entities that R creates and manipulates are known as objects. These may be variables, arrays of numbers, character strings, functions, or more general structures built from such components.

During an R session, objects are created and stored by name (we discuss this process in the next session). The R command

```
> objects()
```

can be used to display the names of (most of) the objects which are currently stored within R. The collection of objects currently stored is called the workspace.

At the end of each R session you are given the opportunity to save all the currently available objects. If you indicate that you want to do this, the objects are written to a file called '.RData' in the current directory, and the command lines used in the session are saved to a file called '.Rhistory'.
When R is started at later time from the same directory it reloads the workspace from this file. At the same time the associated commands history is reloaded. It is recommended that you should use separate working directories for analysis conducted with R.

Sometimes objects from the previous calculations need to be removed. Here the functions *rm*() or *remove*() are used. To remove the object *value* type

```
> rm(value)
```

to remove a collection of objects, list them inside the command

```
> rm(obj1, obj2, obj3)
```

to remove all objects (to start anew) type

```
rm(list=ls(all=TRUE))
```

If commands (script file) are stored in an external file, say 'commands.R' in the working directory 'work', they may be executed at any time in an R session with the command

```
> source("commands.R")
```

The function *sink*(),

```
> sink("record.lis")
```

will divert all subsequent output from the console to an external file (say 'record.lis'). The command

```
> sink()
```

restores it to the console once again.

There are four atomic data types in R.

- Numeric

```
> value <- 605
> value
[1] 605
```

- Character

```
> string <- "Hello World"
> string
[1] "Hello World"
```

- Logical

```
> 2 <- 4
[1] TRUE
```

- Complex number

```
> cn <- 2 + 3i
> cn
[1] 2+3i
```

The attribute of an object becomes important when manipulating objects. All objects have two attributes, the mode and their length. The R function mode can be used to determine the mode of each object, while the function length will help to determine each object's length.

```
> mode(value)
[1] "numeric"
> length(value)
[1] 1
```

In many practical examples, some of the data elements will not be known and will therefore be assigned a missing value. The code for missing values in R is NA. This indicates that the value or element of the object is unknown. Any operation on an NA results in an NA. The is.na() function can be used to check for missing values in an object.

```
> value <- c(3,6,23,NA)
> is.na(value)
[1] FALSE FALSE FALSE TRUE
> any(is.na(value))
[1] TRUE
> na.omit(value)
[1] 3 6 23
```

Indefinite and Infinite values (Inf, -Inf and NaN) can also be tested using the

is.finite, *is.infinite*, *is.nan*, *is.number*

functions in a similar way as shown above.

```
> value1 <- 5/0
> value2 <- log(0)
> value3 <- 0/0
> cat("value1 = ",value1," value2 = ",value2,
" value3 = ",value3,"\n")
>value1 = Inf value2 = -Inf value3 = NaN
```

The following list of arithmetic and logical operations available

Operator	Description	Example
$+$	Addition	`> 2+5` `[1] 7`
$-$	Subtraction	`> 2-5` `[1] -3`
$\times$	Multiplication	`>2*5` `[1] 10`
$/$	Division	`> 2/5` `[1] 0.4`
$\wedge$	Exponentiation	`> 2^5` `[1] 32`
$\%/\%$	Integer Divide	`> 5%/%2` `[1] 2`
$\%\%$	Modulo	`> 5%%2` `[1] 1`

Operator	Description	Example
==	Equals	> value1 [1] 3 6 23 > value1==23 [1] FALSE FALSE TRUE
!=	Not Equals	> value1 != 23 [1] TRUE TRUE FALSE
<	Less Than	> value1 < 6 [1] TRUE FALSE FALSE
>	Greater Than	> value1 > 6 [1] FALSE FALSE TRUE
<=	Less Than or Equal To	> value1 <= 6 [1] TRUE TRUE FALSE
>=	Greater Than or Equal To	> value1 >= 6 [1] FALSE FALSE TRUE
&	Elementwise And	> value2 [1] 1 2 3 > value1==6 & value2 <= 2 [1] FALSE TRUE FALSE
|	Elementwise Or	> value1==6 | value2 <= 2 [1] TRUE TRUE FALSE
&&	Control And	> value1[1] <- NA > is.na(value1) && value2 == 1 [1] TRUE
||	Control Or	> is.na(value1) || value2 == 4 [1] TRUE
xor	Elementwise Exclusive Or	> xor(is.na(value1), value2 == 2) [1] TRUE TRUE FALSE
!	Logical Negation	> !is.na(value1) [1] FALSE TRUE TRUE

The four most frequently used types of data objects in R are

- **vectors:** vector represents a set of elements of the same mode
- **matrices:** matrix is a set of elements appearing in rows and columns where the elements are of the same mode
- **data frames:** data frame is similar to a matrix object but the columns can be of different modes
- **lists** list is a generalization of a vector and represents a collection of data objects

Vectors.

A vector may created by using concatenation function, *c*. This function binds elements together, whether they are of character form, numeric or logical.

```
> value.num <- c(3,4,2,6,20)
> value.char <- c("koala","kangaroo","echidna")
> value.logical.1 <- c(F,F,T,T)
# or
> value.logical.2 <- c(FALSE,FALSE,TRUE,TRUE)
```

Also, *rep* and *seq* functions can be used. The *rep* function replicates elements of vectors. For example,

```
> value <- rep(5,6)
> value
[1] 5 5 5 5 5 5
```

The seq function creates a regular sequence of values to form a vector.

The following script shows some simple examples of creating vectors using this function.

```
> seq(from=2,to=10,by=2)
[1] 2 4 6 8 10
> seq(from=2,to=10,length=5)
[1] 2 4 6 8 10
> 1:5
[1] 1 2 3 4 5
```

As well as using each of these functions individually to create a vector, the functions can be used in combination. For example,

```
> value <- c(1,3,4,rep(3,4),seq(from=1,to=6,by=2))
> value
[1] 1 3 4 3 3 3 3 1 3 5
```

uses the rep and seq functions inside the concatenation function to create the vector value.

It is important to remember that elements of a vector are expected to be of the same mode. So an expression

```
> c(1:3,"a","b","c")
```

will produce an error message.

The scan function is used to enter in data at the terminal. This is useful for small data sets.

```
> value <- scan()
1: 3 4 2 6 20
6:
> value
[1] 3 4 2 6 20
```

Computation with vectors is achieved using an element-by-element operation. This is useful when writing code because it avoids 'for loops'. However, care must be taken when doing arithmetic with vectors, especially when one vector is shorter than another.

In the latter circumstance, short vectors are recycled. This could lead to problems if 'recycling' was not meant to happen. An example is shown below. Other functions return a single value when applied to a vector *small*

```
> x <- runif(10) # generates random vector
                 # of length 10 iid
> x
[1] 0.3565455 0.8021543 0.6338499 0.9511269
[5] 0.9741948 0.1371202 0.2457823 0.7773790
[9] 0.2524180 0.5636271
> y <- 2*x + 1 # recycling short vectors
> y
[1] 1.713091 2.604309 2.267700 2.902254 2.948390
[6] 1.274240 1.491565 2.554758 1.504836 2.127254
```

Some functions take vectors of values and produce results of the same length.

```
> z <- (x-mean(x))/sd(x) # see also 'scale'
> z
[1] -0.69326707 0.75794573 0.20982940 1.24310440
[5] 1.31822981 -1.40786896 -1.05398941 0.67726018
[9] -1.03237897 -0.01886511
> mean(z)
[1] -1.488393e-16
> sd(z)
```

Function	Description
cos, sin, tan	Cosine, Sine, Tangent
acos, asin, atan	Inverse functions
cosh, sinh, tanh	Hyperbolic functions
acosh, asinh, atanh	Inverse hyperbolic functions
log	Logarithm (any base, default is natural logari
log10	Logarithm (base 10)
exp	Exponential (e raised to a power)
round	Rounding
abs	Absolute value
ceiling, floor, trunc	Truncating to integer values
gamma	Gamma function
lgamma	Log of gamma function

Function	Description
sum	Sum elements of a vector
mean	arithmetic mean
max, min	Maximum and minimum
prod	Product of elements of a vector
sd	standard deviation
var	variance
median	50th percentile

The dim function can be used to convert a vector to a matrix

```
> value <- rnorm(6)
> dim(value) <- c(2,3) #here one defines dimensions
> value
[,1] [,2] [,3]
[1,] 0.7093460 -0.8643547 -0.1093764
[2,] -0.3461981 -1.7348805 1.8176161
```

This fills the matrix column by column.

To convert back to a vector we simply use the dim function again.

```
> dim(value) <- NULL
```

Alternatively we can use the *matrix* function to convert a vector to a matrix

```
> matrix(value,2,3)
[,1] [,2] [,3]
[1,] 0.7093460 -0.8643547 -0.1093764
[2,] -0.3461981 -1.7348805 1.8176161
```

If we want to fill by rows instead then we can use the following script

```
> matrix(value,2,3,byrow=T)
[,1] [,2] [,3]
[1,] 0.709346 -0.3461981 -0.8643547
[2,] -1.734881 -0.1093764 1.8176161
```

To bind a row onto an already existing matrix, the *rbind* function can be used

```
> value <- matrix(rnorm(6),2,3,byrow=T)
> value2 <- rbind(value,c(1,1,2))
> value2
[,1] [,2] [,3]
[1,] 0.5037181 0.2142138 0.3245778
[2,] -0.3206511 -0.4632307 0.2654400
[3,] 1.0000000 1.0000000 2.0000000
```

To bind a column onto an already existing matrix, the *cbind* function can be used

```
> value3 <- cbind(value2,c(1,1,2))
[,1] [,2] [,3] [,4]
[1,] 0.5037181 0.2142138 0.3245778 1
[2,] -0.3206511 -0.4632307 0.2654400 1
[3,] 1.0000000 1.0000000 2.0000000 2
```

The function *data.frame* converts a matrix or collection of vectors into a data frame

```
> value3 <- data.frame(value3)
> value3
X1 X2 X3 X4
1 0.5037181 0.2142138 0.3245778 1
2 -0.3206511 -0.4632307 0.2654400 1
3 1.0000000 1.0000000 2.0000000 2
```

Observe that this function automatically assigns names to rows and columns.

Another example joins two columns of data together.

```
> value4 <- data.frame(rnorm(3),runif(3))
> value4
rnorm.3. runif.3.
1 -0.6786953 0.8105632
2 -1.4916136 0.6675202
3 0.4686428 0.6593426
```

Row and column names are already assigned to a data frame but they may be changed using the names and row.names functions.

To view the row and column names of a data frame:

```
> names(value3)
[1] "X1" "X2" "X3" "X4"
> row.names(value3)
[1] "1" "2" "3"
```

Alternative labels can be assigned by doing the following

```
> names(value3) <- c("C1","C2","C3","C4")
> row.names(value3) <- c("R1","R2","R3")
```

Names can also be specified within the data.frame function itself.

```
> data.frame(C1=rnorm(3),C2=runif(3),
row.names=c("R1","R2","R3")
C1 C2
R1 -0.2177390 0.8652764
R2 0.4142899 0.2224165
R3 1.8229383 0.5382999
```

Accessing elements (of a vector or matrix) is achieved through a process called indexing. Indexing may be done by

- a vector of positive integers: to indicate inclusion
- a vector of negative integers: to indicate exclusion
- a vector of logical values: to indicate which are in and which are out
- a vector of names: if the object has a names attribute

The first example involves producing a random sample of values between one and five, twenty times and determining which elements are equal to 1.

```
> x <- sample(1:5, 20, rep=T)
> x
[1] 3 4 1 1 2 1 4 2 1 1 5 3 1 1 1 2 4 5 5 3
> x == 1
[1] FALSE FALSE TRUE TRUE FALSE TRUE FALSE FALSE TR
[10] TRUE FALSE FALSE TRUE TRUE TRUE FALSE FALSE FA
[19] FALSE FALSE
> ones <- (x == 1) # parentheses unnecessary
```

We now want to replace the ones appearing in the sample with zeros and store the values greater than 1 into an object called y.

```
> x[ones] <- 0
> x
[1] 3 4 0 0 2 0 4 2 0 0 5 3 0 0 0 2 4 5 5 3
> others <- (x > 1) # parentheses unnecessary
> y <- x[others]
> y
[1] 3 4 2 4 2 5 3 2 4 5 5 3
```

The following command queries the x vector and reports the position of each element that is greater than 1.

```
> which(x > 1)
[1] 1 2 5 7 8 11 12 16 17 18 19 20
```

To exclude values, negative index vectors are used. Thus

```
> y <- x[-(1:5)]
```

gives y all but the first five elements of x.

Here one more example where an object has a names attribute to identify its components. In this case a sub-vector of the names vector may be used

```
> fruit <- c(5, 10, 1, 20)
> names(fruit) <- c("orange", "banana", "apple", "p
> lunch <- fruit[c("apple","orange")]
```

Lists

Lists can be created using the list function. Like data frames, they can incorporate a mixture of modes into the one list and each component can be of a different length or size. For example, the following is an example of how we might create a list from scratch.

```
> L1 <- list(x = sample(1:5, 20, rep=T),
y = rep(letters[1:5], 4), z = rpois(20, 1))
> L1
$x
[1] 2 1 1 4 5 3 4 5 5 3 3 3 4 3 2 3 3 2 3 1
$y
[1] "a" "b" "c" "d" "e" "a" "b" "c" "d" "e" "a" "b"
[13] "c" "d" "e" "a" "b" "c" "d" "e"
$z
[1] 1 3 0 0 3 1 3 1 0 1 2 2 0 3 1 1 0 1 2 0
```

There are a number of ways of accessing the first component of a list. We can either access it through the name of that component (if names are assigned) or by using a number corresponding to the position the component corresponds to. The former approach can be performed using subsetting ([[]]) or alternatively, by the extraction operator ($). Here are a few examples:

```
> L1[["x"]]
[1] 2 1 1 4 5 3 4 5 5 3 3 3 4 3 2 3 3 2 3 1
> L1$x
[1] 2 1 1 4 5 3 4 5 5 3 3 3 4 3 2 3 3 2 3 1
> L1[[1]]
[1] 2 1 1 4 5 3 4 5 5 3 3 3 4 3 2 3 3 2 3 1
```

To extract a sublist, we use single brackets. The following example extracts the first component only.

```
> L1[1]
$x
[1] 2 1 1 4 5 3 4 5 5 3 3 3 4 3 2 3 3 2 3 1
```

Working with Lists The length of a list is equal to the number of components in that list. So in the previous example, the number of components in L1 equals 3.

We confirm this result using the following line of code:

```
> length(L1)
[1] 3
```

To determine the names assigned to a list, the names function can be used. Names of lists can also be altered in a similar way to that shown for data frames.

```
> names(L1) <- c("Item1","Item2","Item3")
```

Indexing components of the list (if they are vectors) can be achieved in a similar way to how data frames are indexed:

```
> L1$Item1[L1$Item1>2]
[1] 4 3 4 5 3 3 3 5 3 3 5
```

Joining two lists can be achieved either using the concatenation function or the append function. The following two scripts show how to join two lists together using both functions.

Concatenation function:

```
> L2 <- list(x=c(1,5,6,7),
y=c("apple","orange","melon","grapes"))
> c(L1,L2)
$Item1
[1] 2 4 3 4 1 5 3 1 1 2 3 3 5 2 1 3 2 3 5 1
$Item2
[1] "a" "b" "c" "d" "e" "a" "b" "c" "d" "e" "a" "b"
[13]"c" "d" "e" "a" "b" "c" "d" "e"
$Item3
[1] 0 0 2 1 1 0 2 0 0 1 1 1 0 0 1 1 1 3 0 2
$x
[1] 1 5 6 7
$y
[1] "apple" "orange" "melon" "grapes"
```

Append Function: (actually inseart)

```
> append(L1,L2,after=2)
$Item1
[1] 2 4 3 4 1 5 3 1 1 2 3 3 5 2 1 3 2 3 5 1
$Item2
[1] "a" "b" "c" "d" "e" "a" "b" "c" "d" "e" "a"
[12]"b" "c" "d" "e" "a" "b" "c" "d" "e"
$x
[1] 1 5 6 7
$y
[1] "apple" "orange" "melon" "grapes"
$Item3
[1] 0 0 2 1 1 0 2 0 0 1 1 1 0 0 1 1 1 3 0 2
```

Adding elements to a list can be achieved by

- adding a new component name:

```
> L1$Item4 <- c("apple","orange","melon","grapes
# alternative way
> L1[["Item4"]] <- c("apple","orange","melon","g
```

- adding a new component element, whose index is greater than the length of the list

```
L1[[4]] <- c("apple","orange","melon","grapes")
> names(L1)[4] <- c("Item4")
```

It is important to understand lists, since there are also many functions within R that produce a list as output

Data sorting

Ordering is usually best done indirectly: Find an index vector that achieves the sort operation and use it for all vectors that need to remain together. The function order allows sorting with tie-breaking: Find an index vector that arranges the first of its arguments in increasing order.

Ties are broken by the second argument and any remaining ties are broken by a third argument.

```
> x <- sample(1:5, 20, rep=T)
> y <- sample(1:5, 20, rep=T)
> z <- sample(1:5, 20, rep=T)
> xyz <- rbind(x, y, z)
> dimnames(xyz)[[2]] <- letters[1:20]
> xyz
a b c d e f g h i j k l m n o p q r s t
x 4 4 2 4 3 4 4 1 2 2 5 3 1 5 5 3 4 5 3 4
y 5 5 2 5 2 3 5 4 4 2 4 2 1 4 3 4 4 2 2 2
z 4 5 3 2 4 2 4 5 5 2 4 2 4 5 3 4 3 4 4 3
> o <- order(x, y, z)
> xyz[, o]
m h j c i l e s p t f q d a g b r o k n
x 1 1 2 2 2 3 3 3 3 4 4 4 4 4 4 4 5 5 5 5
y 1 4 2 2 4 2 2 2 4 2 3 4 5 5 5 5 2 3 4 4
z 4 5 2 3 5 2 4 4 4 3 2 3 2 4 4 5 4 3 4 5
```

The operator %*% is used for matrix multiplication. An $n \times 1$ or $1 \times n$ matrix may of course be used as an n-vector if in the context such is appropriate.
Conversely, vectors which occur in matrix multiplication expressions are automatically promoted either to row or column vectors, whichever is multiplicatively coherent, if possible, (although this is not always unambiguously possible, as we see later).

If, for example, *A* and *B* are square matrices of the same size, then

```
> A * B
```

is the matrix of element by element products and

```
> A %*% B
```

is the matrix product. If *x* is a vector, then

```
> x %*% A %*% x
```

is a quadratic form.

The function $t(X)$ calculates the transpose of X.

Example

```
> (mat.1 <- matrix(c(1,0,1,1), nrow=2))
[,1] [,2]
[1,] 1 1
[2,] 0 1
> (mat.2 <- matrix(c(1,1,0,1), nrow=2))
[,1] [,2]
[1,] 1 0
[2,] 1 1
> solve(mat.1) # This inverts the matrix
[,1] [,2]
[1,] 1 -1
[2,] 0 1
```

```
> mat.1 %*% mat.2 # Matrix multiplication
[,1] [,2]
[1,] 2 1
[2,] 1 1
> mat.1 + mat.2 # Matrix addition
[,1] [,2]
[1,] 2 1
[2,] 1 2
> t(mat.1) # Matrix transposition
[,1] [,2]
[1,] 1 0
[2,] 1 1
> det(mat.1) # Matrix determinant
[1] 1
```

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
- $diag(M)$, where M is a matrix, gives the vector of main diagonal entries of M.
- somewhat confusingly, if k is a single integer value then $diag(k)$ is the $k \times k$ identity matrix!

In R,

```
> solve(A,b)
```

solves the linear system $Ax = b$. The inverse of the matrix A can be computed by

```
solve(A)
```

The function *eigen*(*Sm*) calculates the eigenvalues and eigenvectors of a symmetric matrix *Sm*. The result of this function is a list of two components named values and vectors. The assignment

```
> ev <- eigen(Sm)
```

will assign this list to *ev*. Then *ev*$*val* is the vector of eigenvalues of *Sm* and *ev*$*vec* is the matrix of corresponding eigenvectors.

By default the routine *eigen()* checks the input matrix for symmetry, but it is probably better to specify whether the matrix is symmetric by construction or not using the parameter *symmetric*.

```
> J <- cbind(c(20,3),c(3,18))
> j <- eigen(J,symmetric=T)
> j$vec%*%diag(j$val)%*%t(j$vec)
     [,1] [,2]
[1,]   20    3
[2,]    3   18
```

If the more general singular value decomposition is desired, we use instead *svd()*. For the QR factorization, we use *qr()*.

R can numerically minimize an arbitrary function using either *nlm()* or *optim()*. The latter because it lets the user choose which optimization method.

The *nlm()* function takes as an argument a function and a starting vector at which to evaluate the function.

The fist argument of the user-defined function should be the parameter(s) over which R will minimize the function, additional arguments to the function (constants) should be specified by name in the *nlm()* call.

```
> g <- function(x,A,B){
+ out <- sin(x[1])-sin(x[2]-A)+x[3]^2+B
+ out
+ }
> results <- nlm(g,c(1,2,3),A=4,B=2)
> results$min
[1] 6.497025e-13
> results$est
[1] -1.570797e+00 -7.123895e-01 -4.990333e-07
```

If function maximization is wanted one should multiply the function by -1 and minimize.
If *optim()* is used, one can instead pass the parameter *control=list(fnscale=-1)*, which indicates a multiplier for the objective function and gradient. It can also be used to scale up functions that are nearly flat so as to avoid numerical inaccuracies.

Other optimization functions which may be of interest are

optimize() for one-dimensional minimization,

uniroot() for root finding, and

deriv() for calculating numerical derivatives. Also,

constrOptim() provides constrained optimization functionality.

We can use the function *integrate()* from the *stats* package to do one-dimensional integration of a known function. For example,

```
> g <- function(x){
+  exp(-1/2*((.12-x)^2+(.07-x)^2+(.08-x)^2))
+ }
> const <- integrate(g,-Inf,Inf)$value
```

We notice that *integrate()* returns additional information, such as error bounds, so we extract the value using *$value*. Also, in addition to a function name, *integrate()* takes limits of integration, which—as in this case—may be infinite.

A function can be treated as any other object in R. It is created with the assignment operator and *function()*, which is passed an argument list (use the equal sign to denote default arguments; all other arguments will be required at runtime). The code that will operate on the arguments follows, surrounded by curly brackets if it comprises more than one line.

This means the following functions are equivalent

```
> g <- function(x,Alpha=1,B=0) sin(x[1])-sin(x[2]-A
> f <- function(x,Alpha=1,B=0){
  out <- sin(x[1])-sin(x[2]-Alpha)+x[3]^2+B
  return(out)
  }
```

Because R does not distinguish what kind of data object a variable in the parameter list is, we should be careful how we write our functions.

Function parameters are passed by value, so changing them inside the function does not change them outside of the function. Also variables defined within functions are unavailable outside of the function. If a variable is referenced inside of a function, first the function scope is checked for that variable, then the scope above it, etc.

In other words, variables outside of the function are available to code inside for reading, but changes made to a variable defined outside a function are lost when the function terminates. For example, define

```
a<-c(1,2)
 k<-function(){
 cat("Before: ",a,"\n")
 a<-c(a,3)
 cat("During: ",a,"\n")
 }
```

then run

```
 k()
Before:  1 2
During:  1 2 3
>  cat("After: ",a,"\n")
After:  1 2
```

If a function wishes to write to a variable defined in the scope above it, it can use the “super-assignment” operator $<<-$.

Looping is performed using the *for* command. It's syntax is as in the example

```
> for (i in 1:20){
+ cat(i)
> }
```

Where *cat()* may be replaced with the block of code we wish to repeat. Instead of *1:20*, a vector or matrix of values can be used. The index variable will take on each value in the vector or matrix and run the code contained in curly brackets.

If we simply want a loop to run until something happens to stop it, we could use the *repeat* loop combined with the command *break*

```
> repeat {
+ g <- rnorm(1)
+ if (g > 2.0) break
+ cat(g);cat("\n") # new line
> }
```

The semicolon acts to let R know where the end of our command is, when we put several commands on a line.

For example, the above is equivalent to

```
> repeat {g <- rnorm(1);if (g>2.0) break;cat(g);cat
```

To help the programmer avoid the sluggishness associated with writing and executing loops, R has a command to call a function with each of the rows or columns of an array. We specify one of the dimensions in the array, and for each element in that dimension, the resulting cross section is passed to the function.

For example, if X is a 50x10 array representing 10 quantities associated with 50 individuals and we want to find the mean of each row (or column), we could write

```
> apply(X,1,mean) # for a 50-vector of
                  # individual (row) means
> apply(X,2,mean) # for a 10-vector of observation
                  # (column) means
```

Further, multi-dimensional arrays can be treated in similar way, where several dimensions (beyond rows/columns are fixed). This technique is very useful, since R is slow when it comes to loops.

For reading and writing in files, R uses the working directory. To find this directory, the command *getwd()* (get working directory) can be used, and the working directory can be changed with *setwd("C:/data")*. It is necessary to give the path to a file if it is not in the working directory.

R can read data stored in text (ASCII) files with *read.table* (which has several variants). The function *read.table* has for effect to create a data frame, and so is the main way to read data in tabular form. For instance, if one has a file named data.dat, the command:

```
> mydata <- read.table("data.dat")
```

will create a data frame named *mydata*, and each variable (column) will be named, by default, by

$$V1, \quad V2, \quad \ldots$$

and can be accessed individually by *mydata*\$*V*1, *mydata*\$*V*2, ... or by *mydata*["*V*1"], *mydata*["*V*2"], ... or, by *mydata*[, 1], *mydata*[, 2], ...

There are several options for read.table

```
read.table(file, header = FALSE, sep = "",
quote = "\"' ", dec = "."
row.names, col.names, as.is = FALSE,
na.strings = "NA",
colClasses = NA, nrows = -1,
skip = 0, check.names = TRUE,
fill = !blank.lines.skip,
strip.white = FALSE, blank.lines.skip = TRUE,
comment.char = "#")
```

`file`	the name of the file (within `""` or a variable of mode character), possibly with its path (the symbol \ is not allowed and must be replaced by /, even under Windows), or a remote access to a file of type URL (http://...)
`header`	a logical (`FALSE` or `TRUE`) indicating if the file contains the names of the variables on its first line
`sep`	the field separator used in the file, for instance `sep="\t"` if it is a tabulation
`quote`	the characters used to cite the variables of mode character
`dec`	the character used for the decimal point
`row.names`	a vector with the names of the lines which can be either a vector of mode character, or the number (or the name) of a variable of the file (by default: `1`, `2`, `3`, ...)
`col.names`	a vector with the names of the variables (by default: `V1`, `V2`, `V3`, ...)
`as.is`	controls the conversion of character variables as factors (if `FALSE`) or keeps them as characters (`TRUE`); `as.is` can be a logical, numeric or character vector specifying the variables to be kept as character
`na.strings`	the value given to missing data (converted as `NA`)
`colClasses`	a vector of mode character giving the classes to attribute to the columns
`nrows`	the maximum number of lines to read (negative values are ignored)
`skip`	the number of lines to be skipped before reading the data
`check.names`	if `TRUE`, checks that the variable names are valid for R
`fill`	if `TRUE` and all lines do not have the same number of variables, "blanks" are added
`strip.white`	(conditional to `sep`) if `TRUE`, deletes extra spaces before and after the character variables
`blank.lines.skip`	if `TRUE`, ignores "blank" lines
`comment.char`	a character defining comments in the data file, the rest of the line after this character is ignored (to disable this argument, use `comment.char = ""`)

The function *write.table* writes in a file an object, typically a data frame but this could well be another kind of object (vector, matrix, . . .). The arguments and options are:

```
write.table(x, file = "", append = FALSE, quote = T
eol = "\n", na = "NA", dec = ".", row.names = TRUE,
col.names = TRUE, qmethod = c("escape", "double"))
```

`x`	the name of the object to be written
`file`	the name of the file (by default the object is displayed on the screen)
`append`	if `TRUE` adds the data without erasing those possibly existing in the file
`quote`	a logical or a numeric vector: if `TRUE` the variables of mode character and the factors are written within `""`, otherwise the numeric vector indicates the numbers of the variables to write within `""` (in both cases the names of the variables are written within `""` but not if `quote = FALSE`)
`sep`	the field separator used in the file
`eol`	the character to be used at the end of each line (`"\n"` is a carriage-return)
`na`	the character to be used for missing data
`dec`	the character used for the decimal point
`row.names`	a logical indicating whether the names of the lines are written in the file
`col.names`	id. for the names of the columns
`qmethod`	specifies, if `quote=TRUE`, how double quotes `"` included in variables of mode character are treated: if `"escape"` (or `"e"`, the default) each `"` is replaced by `\"`, if `"d"` each `"` is replaced by `""`

To record a group of objects of any type, we can use the command

```
save(x,y, z, file= "xyz.RData")
```

To ease the transfer of data between different machines, the option *ascii = TRUE* can be used.

The data (which are now called a workspace in R's jargon) can be loaded later in memory with *load("xyz.RData")*. The function *save.image()* is a short-cut for

```
save(list=ls(all=TRUE), file=".RData")}
```