Appendix I

An Introduction to R by Stefano Iacus

I.1 Introduction

The R software is, at first glance, user unfriendly in the sense that it is a real programming language rather than a point&click software. A typical statistical analysis in R is an interactive session where the user types a command on the R Console. Indeed, the standard installation of R has a very limited graphical user interface (GUI) which can be enhanced using the package Rcmdr. There are also integrated environments like the free software RStudio (http://www.rstudio.org) which is a spreadsheet-like tool with R as a back end or the commercial product Revolution R (http://www.revolutionanalytics.com). These few pages are intended to be a quick start guide to the R language so that the reader of this book can immediately start to replicate the analyses of previous chapters. In this appendix we will also mention additional tools which may be valuable in more advanced analyses and production environments. The reader is also invited to read the quick guide called An Introduction to R that comes with every installed version of R or the introductory book on the R environment by Dalgaard (2008).

I.2 How to obtain R

R is an open source and free software available in ready-to-install binary form for many platforms. Although the home page of the software is http://www.R-Project.org, the main repository for downloading R is called CRAN “The Comprehensive R Archive Network” and its main address is http://cran.r-project.org/. Three main sections are available on that page “Linux,” “MacOS X,” and “Windows.” In each section, one should check for the “base” installation of R. For example, the base installation of R for Windows is available at http://cran.r-project.org/bin/windows/base but the basic installation for MacOS X is available at http://cran.r-project.org/bin/macosx. For Linux users there are different subdirectories which contain binary files for debian, redhat, suse, ubuntu but these users can always install R from the source code. Expert users can try to install the 64-bit version of R which allows the manipulation of larger databases in memory but the cost is that some external packages may not be available on some platforms.

I.3 Type rather than “point & click”

All the commands are given as inputs to R after the prompt > (R> in our case) and are analyzed by the R parser after the user presses the “return”/“enter” key (or a new line character is encountered in the case of a script file).
2 Modern Industrial Statistics

R> cat("help me!")
help me!

R inputs can be multiline; hence, if the R parser thinks that the user did not complete some command (because of unbalanced parentheses or quotation marks), on the next line a + symbol will appear instead of a prompt.

R> cat("help me!
+

This can be quite frustrating for novice users, so it is better to know how to exit from this impasse. Depending on the implementation of R, usually pressing CTRL+C or ESC on the keyboard helps. Otherwise, for GUI versions of R, pushing the “stop” button of the R console will exit the parser. Of course, another solution is to complete the command (with a “)” in our example).

I.3.1 The workspace

Almost every command in R creates an object and not just text output, and objects live in the workspace. The workspace, or groups of objects, can be saved and loaded into R with the save.image (or save) and load commands, respectively. The user is prompted about saving workspace when exiting from R. This workspace is saved in the current directory as a hidden (on some operating systems) file named .RData and reloaded automatically the next time R is started.

I.3.2 Graphics

Usually R graphics are displayed on a device that corresponds to a window for a GUI version of R (for example, under MS-Windows, X11, or Mac OS X). Otherwise a Postscript file Rplots.ps is generated in the current working directory. Sometimes, in interactive uses of R, it is useful to use par(ask=TRUE) to pause R at each new plot or par(ask=FALSE) to avoid such pauses. We do not discuss the multiple R graphic systems here but the reader can refer to Murrel (2005) and Deepayan (2008).

I.3.3 Getting help

The casual user will find it very hard to get started without prior knowledge of which command is needed to perform a particular task. The help system is not that useful either. But the R system is devised in such a way that every command has its help page, and the documentation always matches the actual implementation of the command. To get information about a particular command one should use the help, such as help(load) or ?help. For some special operators, the user should specify the argument like this: ?"for", ?"+", etc. When the documentation contains the section “Examples” with the R code inside, the code from that page can be executed automatically with example(topic), where topic is the corresponding R command of interest, try e.g. example(plot).

If one wants to execute a fuzzy search on the help system, one can use the command help.search("topic") and R will return several options which partly match the word topic, try e.g. help.search("regression"). It is also possible to extend the search for a term or for more complicated queries to the Web using RSiteSearch, try for example, RSiteSearch("nonlinear regression"). The search will be extended to all documentation pages for packages in the R repository and to all the pertinent mailing lists.1 The website for R mailing lists and related projects is http://stat.ethz.ch/mailman/listinfo.

There is a rich repository of quick guides or electronic books on R and its use in different disciplines which can be found under the section “Documentation/Contributed” on CRAN. The direct link to the page is http://cran.r-project.org/other-docs.html. Finally, we mention the “Task Views,” which are collections of R packages organized by macro areas, for example, “Multivariate.” These are again hosted on CRAN and the direct link is http://cran.r-project.org/web/views.

1 If you have a question, have a look at the mailing list archives first.
I.3.4 Installing packages

R is a layered software constituted by a core of basic functionalities which serves as the glue for additional capabilities grouped as a collection of functions in the so-called packages. The notion of package in R corresponds to the one of tool or module in other softwares. There are more than 3000 additional packages available for R as of May 2011.

This book, like many others, requires several add-on packages which are not distributed with the basic R system. The main repository for R packages is the CRAN. To install a package in the R system, one should use the command `install.packages()` with a package name as argument, that is, `install.packages("pls")` to install the package `pls` for partial least squares regression. R GUIs usually offer some options to access the list of all packages at the repository and install those selected by point and click actions.

Another important source of R packages is the R-Forge repository. This is a repository mainly for developers but where users can also find pre-release of developer versions of packages already on CRAN or even packages not necessarily hosted on CRAN. The home page of R-Forge is http://r-forge.r-project.org. For example, to install the package `permute` (to generate restricted permutations of data) from R-Forge, one can use a command like

```r
R> install.packages("permute",repos="http://R-Forge.R-project.org")
```

where we have specified to `install.packages()` the argument `repos` with a proper web address.

In order to install the complete suite of packages from a particular Task View, one first needs to install the `ctv` package and load it into R

```r
R> install.packages("ctv")
R> library("ctv")
```

and then install the packages from a Task View, say, “Multivariate,” with

```r
R> install.views("Multivariate")
```

I.4 Objects

As mentioned, most functions in R return objects rather than text output. Clearly objects can be created anew with commands. We now describe how to create, inspect and manipulate objects.

I.4.1 Assignments

To create an object it is necessary to use the operator `<-` which has the meaning “assign the right-hand side to the left-hand side,” or use the more common operator `=` as in the following lines in which we create an object named `x` and assign the number 4 to it:

```r
R> x <- 4
R> x = 4
```

Similarly, one can use the operator `->` which assigns the left-hand side to the right-hand side, that is, `x -> 4`. The following command creates a more interesting vector `y` containing the numbers 2, 7, 4, and 1 concatenated into a single object using the function `c()`:

```r
R> y <- c(2,7,4,1)
R> y
[1] 2 7 4 1
```

A matrix can be created using the `matrix` command

```r
R> z <- matrix(1:30, 5, 6)
R> z
```
Where `1:30` produces a sequence from 1 to 30 by unitary step, that is,

```r
R> 1:30
```

```
[1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
```

The command `matrix` requires at least three arguments, where the second and third are the number of rows and columns and the first one is an object which is used recursively to fill the elements of the matrix. Of course, we can create an empty matrix with

```r
R> matrix(,5,6)
```

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

where `NA` is the R symbol for the missing values, or empty numerical vectors with `numeric`

```r
R> numeric(4)
```

```
[1] 0 0 0 0
```

The command `ls()` shows the current content of the workspace

```r
R> ls()
```

```
[1] "x" "y" "z"
```

Notice that all objects which are created but not assigned, are not kept in the workspace. Like `numeric`, there are several commands to allocate objects for the different data types available in R, that is, we have functions such as `integer`, `character`, etc. or use the function `vector` as follows

```r
R> vector(mode="numeric", 4)
```

```
[1] 0 0 0 0
```

is equivalent to `numeric(4)`. Objects can also have length zero

```r
R> w1 <- numeric(0)
R> w1
```

```
numeric(0)
```

or can be initialized as `NULL`

```r
R> w2 <- NULL
R> w2
```

```
NULL
```

which is useful if one wants to enlarge these objects later in subsequent tasks. In the above, `w1` and `w2` are objects of different types and, in particular, `w1` is an object of class `numeric` while `w2` is not. It is also possible to use the command
assign to create objects and this is sometimes useful when the name of the object has to be created dynamically in the 
R code. The following is an example of use in which O1 is created as before and O2 is created via assign

R> O1 <- 1:4
R> O1
[1] 1 2 3 4
R> ls()
[1] "O1" "w1" "w2" "x" "y" "z"
R> assign("O2", 5:8)
R> ls()
[1] "O1" "O2" "w1" "w2" "x" "y" "z"
R> O2
[1] 5 6 7 8

I.4.2 Basic object types

Object classes can be created from scratch in the R language, and this is usually the case for many R packages, but the 
basic classes are integer, numeric, complex, character, etc. which can be aggregated into vectors, matrices, 
arrays or lists. While vectors, arrays and lists contain elements all of the same type, the lists are more general and can 
contain objects of different sizes and types but in addition can also be nested. For example, the following code loads a 
data set and estimates a linear model via lm and assigns the result to an object mod. The statistical analysis per se is 
not relevant here, we just notice that an estimated regression model in R is not just an output of coefficients with their 
significance, but an object

R> data(cars)
R> mod <- lm(dist~speed, data=cars)
R> mod
Call: lm(formula = dist ~ speed, data = cars)
Coefficients:
(Intercept) speed
 -17.579 3.932

We now look at the structure of the object created by the linear regression using the command str which inspects the 
structure of the object.

R> str(mod)
List of 12
$ coefficients : Named num [1:2] -17.58 3.93
 ..- attr(*, "names")= chr [1:2] "(Intercept)" "speed"
$ residuals : Named num [1:50] 3.85 11.85 -5.95 12.05 2.12 ...
 ..- attr(*, "names")= chr [1:50] "1" "2" "3" "4" ...
$ effects : Named num [1:50] -303.914 145.552 -8.115 9.885 0.194 ...
 ..- attr(*, "names")= chr [1:50] "(Intercept)" "speed" "" ""
$ rank : int 2
$ fitted.values: Named num [1:50] -1.85 -1.85 9.95 9.95 13.88 ...
 ..- attr(*, "names")= chr [1:50] "1" "2" "3" "4" ...
$ assign : int [1:2] 0 1
$ qr : List of 5
 ..$ qr : num [1:50, 1:2] -7.071 0.141 0.141 0.141 0.141 ...
For the above we see that `mod` is essentially a list object of 12 elements and it is of class ‘lm’ (for linear models). For example, the first one is called `coefficients` and can be accessed using the symbol $ as follows:

R> mod$coefficients

(Intercept)  speed
     -17.58       3.93

R> str(mod$coefficients)
Named num [1:2] -17.58 3.93
 - attr(*, "names") = chr [1:2] "(Intercept)" "speed"
The vector coefficients is a “named vector.” One can obtain or change the names of the elements of a vector with

\[R> \text{names}(\text{mod$coefficients})\]

\[\text{[1]} \ "\text{(Intercept)}" \ "\text{speed}\]

or change them with

\[R> \text{names}(\text{mod$coefficients}) <- c("alpha", "beta")\]

\[R> \text{mod$coefficients}\]

```
  alpha  beta
-17.579095 3.932409
```

Similarly, one can assign or get the names of the rows or the columns of an R matrix

\[R> z\]

```
 [1,]  1  6 11 16 21 26
 [2,]  2  7 12 17 22 27
 [3,]  3  8 13 18 23 28
 [4,]  4  9 14 19 24 29
 [5,]  5 10 15 20 25 30
```

\[R> \text{rownames}(z) <- c("a","b","c","d","e")\]

\[R> \text{colnames}(z) <- c("A","B","C","D","E","F")\]

\[R> z\]

```
   A B C D E F
  a 1 6 11 16 21 26
  b 2 7 12 17 22 27
  c 3 8 13 18 23 28
  d 4 9 14 19 24 29
  e 5 10 15 20 25 30
```

As anticipated, lists can be nested. For example, the object model inside mod is itself a list

\[R> \text{str(mod$model)}\]

```
'data.frame': 50 obs. of 2 variables:
$ dist : num 2 10 4 22 16 10 18 26 34 17 ...
$ speed: num 4 4 7 7 8 9 10 10 10 11 ...
- attr(*, "terms")=Classes 'terms', 'formula' length 3 dist ~ speed
  ..- attr(*, "variables")= language list(dist, speed)
  ..- attr(*, "factors")= int [1:2, 1] 0 1
  ..- attr(*, "dimnames")=List of 2
  .. . . . .$ : chr [1:2] "dist" "speed"
  .. . . . .$ : chr "speed"
  ..- attr(*, "term.labels")= chr "speed"
  ..- attr(*, "order")= int 1
  ..- attr(*, "intercept")= int 1
  ..- attr(*, "response")= int 1
  ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
  ..- attr(*, "predvars")= language list(dist, speed)
  ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
  ..- attr(*, "names")= chr [1:2] "dist" "speed"
```

or, more precisely a data.frame which is essentially a list with the property that all the elements have the same length. The data.frame object is used to store data sets, like the cars data set
and it is assumed that the elements of a data.frame correspond to variables, while the length of each object is the same as the sample size.

### I.4.3 Accessing objects and subsetting

We have seen that $\mathrm{\$}$ can be used to access the elements of a list and hence of a data.frame, but R also offer operators for enhanced subsetting. The first one is $\mathrm{[\]}$ which returns an object of the same type as the original object

R> y

[1] 2 7 4 1
R> y[2:3]

[1] 7 4

R> str(y)
	num [1:4] 2 7 4 1
R> str(y[2:3])
	num [1:2] 7 4

or, for matrix-like objects

R> z

A B C D E F
a 1 6 11 16 21 26
b 2 7 12 17 22 27
c 3 8 13 18 23 28
d 4 9 14 19 24 29
e 5 10 15 20 25 30
R> z[1:2, 5:6]

EF
a 21 26
b 22 27

and subsetting can occur also on non-consecutive indexes

R> z[1:2, c(1,3,6)]

A C F
a 1 11 26
b 2 12 27

or in a different order

R> z[1:2, c(6,5,4)]

F E D
a 26 21 16
b 27 22 17
One can subset objects also using names, e.g.

```r
R> z[c("a","c"), "D"]
  a  c
16 18
```

We can also use a syntax such as

```r
R> z["c",]
A B C D E F
 3 8 13 18 23 28
```

leaving one argument out to mean “run all the elements” for that index. Further, R allows negative indexes which are used to exclude indexes

```r
R> z[c(-1,-3),]
A B C D E F
 b 2 7 12 17 22 27
d 4 9 14 19 24 29
e 5 10 15 20 25 30
```

but positive and negative indexes cannot be mixed.

The subsetting operator `[` also works for lists

```r
R> a <- mod[1:2]
R> str(a)

List of 2
$ coefficients: Named num [1:2] -17.58 3.93
  ..- attr(*, "names")= chr [1:2] "alpha" "beta"
$ residuals : Named num [1:50] 3.85 11.85 -5.95 12.05 2.12 ...
  ..- attr(*, "names")= chr [1:50] "1" "2" "3" "4" ...
```

where we have extracted the first two elements of the list `mod` using `mod[1:2]`. We can use names as well and the commands below return the same objects

```r
R> str(mod["coefficients"])

List of 1
$ coefficients: Named num [1:2] -17.58 3.93
  ..- attr(*, "names")= chr [1:2] "alpha" "beta"
R> str(mod[1])

List of 1
$ coefficients: Named num [1:2] -17.58 3.93
  ..- attr(*, "names")= chr [1:2] "alpha" "beta"
```

Notice that `mod[1]` returns a list with one element but not just the element inside the list. For this purpose one should use the subsetting operator `[[`. The next group of commands returns the element inside the list

```r
R> str(mod[[1]])

Named num [1:2] -17.58 3.93
  - attr(*, "names")= chr [1:2] "alpha" "beta"
R> str(mod[["coefficients"]])

Named num [1:2] -17.58 3.93
  - attr(*, "names")= chr [1:2] "alpha" "beta"
```
We have mentioned that a data.frame looks like a particular list, but with more structure and is used to store data sets. The latter are always thought of as matrices and indeed it is possible to access the elements of a data.frame using the subsetting rules for matrixes, that is,

\begin{verbatim}
R> cars[,1]
[1]  4  4  7  7  8  9 10 10 10 11 11 12 12 12 12 13 13 13 13 14 14 14
[23] 14 15 15 15 16 16 17 17 17 18 18 18 18 19 19 19 20 20 20 20 20 22
[45] 23 24 24 24 24 25
\end{verbatim}

which is equivalent to the following

\begin{verbatim}
R> cars$speed
[1]  4  4  7  7  8  9 10 10 10 11 11 12 12 12 12 13 13 13 13 14 14 14
[23] 14 15 15 15 16 16 17 17 17 18 18 18 18 19 19 19 20 20 20 20 20 22
[45] 23 24 24 24 24 25
\end{verbatim}

\begin{verbatim}
R> cars[1]
[1]  4  4  7  7  8  9 10 10 10 11 11 12 12 12 12 13 13 13 13 14 14 14
[23] 14 15 15 15 16 16 17 17 17 18 18 18 18 19 19 19 20 20 20 20 20 22
[45] 23 24 24 24 24 25
\end{verbatim}

Notice that only the output is not a data.frame while

\begin{verbatim}
R> str(cars[1])
'data.frame': 50 obs. of 1 variable:
$ speed: num 4 4 7 7 8 9 10 10 10 11 ...
\end{verbatim}

\begin{verbatim}
R> head(cars[1])
speed
1   4
2   4
3   7
4   7
5   8
6   9
\end{verbatim}

is a proper (sub) data.frame although the matrix-like subsetting operator as a different behavior if used on columns or rows: cars[,1] returns the element but, for example,

\begin{verbatim}
R> cars[1:3,]
     speed dist
1   4     2
2   4    10
3   7     4
\end{verbatim}

returns a data.frame with the selected number of rows and all columns.

\footnote{The commands \texttt{head} and \texttt{tail} show the first and last rows of a data.frame respectively.}
I.4.4 Coercion between data types

Functions like names, colnames, but also levels, attributes, etc. are used to retrieve and set properties of objects and are called accessor functions. Objects can be transformed from one type to another using functions with names as.*. For example, as.integer transforms an object into an integer whenever possible or eventually returns a missing value

R> pi
[1] 3.141593
R> as.integer(pi)
[1] 3
R> as.integer("3.14")
[1] 3
R> as.integer("a")
[1] NA

Other examples are as.data.frame to transform matrix objects into true data.frame objects and vice versa. For more complex classes one can also try the generic function as.

I.5 S4 objects

Several times we have used the term ”class” for R objects. This is because each object in R belongs to some class and for each class there exist generic functions called methods which perform some task on that object. For example, the function summary provides summary statistics which are appropriate for some object

R> summary(cars)

             speed       dist
    Min. : 4.0  Min. : 2.00
  1st Qu.:12.0  1st Qu.: 26.00
  Median :15.0  Median : 36.00
  Mean :15.4  Mean : 42.98
  3rd Qu.:19.0  3rd Qu.: 56.00
  Max. :25.0  Max. :120.00

R> summary(mod)

Call:
  lm(formula = dist ~ speed, data = cars)

Residuals:
     Min       1Q   Median       3Q      Max
-29.069   -9.525  -2.272   9.215  43.201

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
alpha -17.5791   6.7584  -2.601  0.0123 *
beta  3.9324    0.4155   9.464  1.49e-12 ***

---

Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
Residual standard error: 15.38 on 48 degrees of freedom
Multiple R-squared:  0.6511, Adjusted R-squared:  0.6438
F-statistic: 89.57 on 1 and 48 DF,  p-value: 1.49e-12

The standard set of classes and methods in R is called S3. In this framework, a method for an object of some class is simply an R function named `method.class`, e.g. `summary.lm` is the function which is called by R when the function `summary` is called with an argument which is an object of class `lm`. In R methods like `summary` are very generic and the function `methods` provides a list of specific, methods (which apply to specific types of objects) for some particular method. For example

```r
R> methods(summary)
[1] summary.aov       summary.aovlist
[3] summary.aspell*   summary.connection
[5] summary.data.frame summary.Date
[7] summary.default    summary.ecdf*
[9] summary.factor     summary.glm
[13] summary.loess*    summary.manova
[15] summary.matrix    summary.mlm
[17] summary.nls*     summary.packageStatus*
[19] summary.PDF_Dictionary* summary.PDF_Stream*
[21] summary.POSIXct    summary.POSIXlt
[23] summary.ppr*      summary.prcomp*
[25] summary.princomp* summary.proc_time
[27] summary.srcfile    summary.srcref
[29] summary.stepfun    summary.stl*
[31] summary.table     summary.tukeys*  
```

Non-visible functions are asterisked

The dot “.” naming convention is no good because one can artificially create functions which are not proper methods, for example, the `t.test` function is not the method `t` for objects of class `test` but it is just an R function which performs ordinary two-samples t test. The new system of classes and methods which is now fully implemented in R is called S4. Objects of class S4 apparently behave like all other objects in R but they possess properties called “slots,” which can be accessed differently from other R objects. The next code estimates the maximum likelihood estimator for the mean of a Gaussian law. It uses the function `mle` from the package `stats4` which is an S4 package as the name suggests. Again, we are interested in the statistical part of this example

```r
R> require(stats4)
R> set.seed(123)
R> y <- rnorm(100, mean=1.5)
R> f <- function(theta=0) -sum(dnorm(x=y, mean=theta, log=TRUE))
R> fit <- mle(f)
R> fit

Call:
mle(minuslogl = f)

Coefficients:
  theta
1.590406

We now have a look at the object `fit` returned by the `mle` function

```r
R> str(fit)
Formal class ‘mle’ [package "stats4"] with 9 slots
  ..@ call : language mle(minuslogl = f)
```
Wenowseethatthisisan\textit{S4} object with slots that, as the structure suggests, can be accessed using the symbol \texttt{@} instead of \$. For example,

\begin{verbatim}
R> fit@coef

theta
1.590406
\end{verbatim}

To get the list of methods for \textit{S4} objects, one should use the function \texttt{showMethods}

\begin{verbatim}
R> showMethods(summary)

Function: summary (package base)
object="ANY"
object="mle"
\end{verbatim}

\section*{1.6 Functions}

In the previous section we have created a new function called \texttt{f} to define the log-likelihood of the data. In R, functions are created with the command \texttt{function} followed by a list of arguments and the body of the function (if longer than one line) has to be contained within \texttt{"\{\}"} as in the next example in which we define the payoff function of a call option

\begin{verbatim}
R> g <- function(x, K=110){
+   max(x-K, 0)
+ } 
\end{verbatim}

The function returns the last calculation unless the command \texttt{return} is used. By default, in the function \texttt{g} we have set the strike price \( K=100 \) and \( x \) is the argument which represents the price of the underlying asset.
I.7 Vectorization

Most of the R functions are vectorized, which means that if a vector is passed to a function, the function is applied to each element of the function and a vector of results is returned as in the next example:

```r
R> set.seed(123)
R> x <- runif(5, 90, 150)
R> x
[1] 107.2547 137.2983 114.5386 142.9810 146.4280
R> sin(x)
[1] 0.4263927 -0.8026760 0.9916244 -0.9992559 0.9414204
```

But functions should be prepared to be vectorized. For example, our function `g` is not vectorized:

```r
R> g(x)
[1] 36.42804
```

Indeed, in the body of `g` the function `max` is used and it operates as follows: first `x-K` is calculated:

```r
R> x - 100
[1] 7.254651 37.298308 14.538615 42.981044 46.428037
```

and then the `max` calculates the maximum of the vector `c(x-100, 0)`. To vectorize it we can use the function `sapply` as follows:

```r
R> g1 <- function(x, K=110){
+   sapply(x, function(x) max(x-K, 0))
+ }
R> g1(x)
[1] 0.000000 27.298308 4.538615 32.981044 36.428037
```
and we get five different payoffs. The functions of class *apply are designed to work iteratively on different objects. The function *sapply iterates the vector in the first argument and applies the functions in the second argument. The function *apply works on arrays (e.g. matrices), *lapply iterates over lists, etc.

The usual *for and while constructs exist in R as well, but their use should be limited to real iterative tasks which cannot be parallelized as in our example. A *for version of the function *g can be the following:

```r
R> g2<- function(x, K=110){
+ n <- length(x)
+ val <- numeric(n)
+ for(i in 1:n)
+   val[i] <- max(x[i]-K,0)
+ val
+ }
```

or, in a more R-like fashion, as follows:

```r
R> g3<- function(x, K=110){
+ val <- NULL
+ for(u in x)
+   val <- c(val, max(u-K,0))
+ val
+ }
```

The vectorized versions are usually faster than the ones iterated using *for loops:

```r
R> y <- runif(10000, 90, 150)
R> system.time(g1(y))
    user  system elapsed
   0.018   0.001   0.021
R> system.time(g2(y))
    user  system elapsed
   0.024   0.001   0.024
R> system.time(g3(y))
    user  system elapsed
   0.157   0.047   0.204
```

Notice that the function *g3 is particularly inefficient because instead of allocating and assigning the results, it grows the vector *val dynamically.

### 1.8 Importing data from different sources

R offers some facilities to import data from the most common file formats via the package *foreign which is included in the standard distribution of R. The library *foreign can manage the following file formats

- Stata: R can read and write *.dta data format from Stata version 5 to 11. The two commands are *read.dta and *write.dta.
- **EpiInfo**: Epi Info is a public domain database and statistics package produced by the US Centers for Disease Control and EpiData is a freely available data entry and validation system. R can read the files .REC using the function `read.epiinfo`. Version 2000 of the Epi Info files uses the Microsoft Access file format to store data. This may be readable with the RODBC or RCOM packages.
- **Minitab Portable Worksheet**: R is able to read files in this particular format but, while in general the output of these input functions is a data.frame, in this case the returned object is a list. The input function is `read.mtp`.
- **SAS Transport (XPORT)**: this is the format used by the SAS system to export data. The function `read.xport` reads a file as a SAS XPORT format library and returns a list of data.frames. With the exception of the MacOS X platform, if SAS is installed in the system, it is possible to use the function `read.ssd`, a wrapper function, which executes a SAS script which transforms the .ssd (or .ssd?bdat) SAS files into the XPORT format and the calls for the function `read.xport`.
- **SPSS**: the function `read.spss` can read files stored using the commands “save” and “export” of SPSS.
- **S**: it is also possible to read the old files formats of the Unix and Windows versions of S-PLUS (3.x, 4.x o 2000 32bit). The function is called `read.S`. It is also possible to use the function `data.restore` to read S-PLIS files created with the S-PLUS command `data.dump` using option oldStyle=T.
- **DBF**: the function `read.dbf` reads a DBF file into a data frame, converting character fields to factors, and trying to respect NULL fields.
- **Systat**: the function `read.systat` reads a rectangular data file stored by the Systat SAVE command as (legacy) *.sys or more recently *.syd files.
- **WEKA**: Weka Attribute-Relation File Format (ARFF) files can be read and saved using the commands `read.arff` and `write.arff`.
- **Octave**: the `read.octave` imports a file in the Octave text data format into a list.

Other packages may exist to read files from other commercial or free softwares.

### I.9 Interacting with databases

With R it is possible to interact with database management systems (DBMSs) and RDBMSs (relational). DBMSs and, in particular, RDBMSs are designed to do all of these things well. Their strengths are

- they provide fast access to selected parts of large databases;
- powerful ways to summarize and cross-tabulate columns in databases;
- they store data in more organized ways than the rectangular grid model of spreadsheets and R data frames;
- concurrent access from multiple clients running on multiple hosts while enforcing security constraints on access to the data;
- ability to act as a server to a wide range of clients.

The sort of statistical applications for which DBMS might be used are to extract a 10% sample of the data, to cross-tabulate data to produce a multi-dimensional contingency table, and to extract data group by group from a database for separate analysis.

Most R packages are designed to work with a specific database server. For example, ROrcacle, RPostgreSQL and RSQlite. The RODBC package provides access to databases in a broader sense (including Microsoft Access and Microsoft SQL Server) through an ODBC interface. Typical use of the RODBC package is as follows

```r
R> library(RODBC)
R> myconn <- odbcConnect("mydsn", uid="Othello", pwd="2beornot2be")
R> crimedat <- sqlFetch(myconn, Crime)
R> pundat <- sqlQuery(myconn, "select * from Punishment")
R> close(myconn)
```
The package DBI acts as a generic front-end to some of the specific packages mentioned in above. The package RJDBC uses Java and can connect to any DBMS that has a JDBC driver while RpgSQL is a specialist interface to PostgreSQL built on top of RJBDC.

I.10 Simple graphics manipulation

R offers complete control over its graphics at the cost of learning many small bits of information. On the contrary, by default, the standard R graphics tend to be simple but effective. In some cases, labels on the axis may be dropped to avoid overlaps, or the bins in a histogram may be less than the one specified by the user (R considers this number as a suggestion only), and other similar little things which drive the initial user crazy. We briefly recall here some very basic graphics parameters which can be applied to almost all plots in R. We go through the different options by examples. In Figure I.1 we plot a scatterplot and two regression lines with the command abline with two different line widths: lwd=1 (the default value) and lwd=4.

```
R> data(cars)
R> par(mfrow=c(1,3))
R> plot(cars)
R> plot(cars, main="lwd=1")
R> abline(lm(dist~speed, data=cars))
R> plot(cars, main="lwd=4")
R> abline(lm(dist~speed, data=cars), lwd=4)
```

In the above we have split the graphical area into one row and three columns and specified a different name at the top of the graphics via the argument main in the command plot.

![Figure I.1](image_url)

Line types can either be specified as an integer (0=blank, 1=solid (default), 2=dashed, 3=dotted, 4=dotdash, 5=longdash, 6=twodash) or as one of the character strings “blank,” “solid,” “dashed,” “dotted,” “dotdash,” “longdash,” or “twodash,” where “blank” uses “invisible lines” (i.e., does not draw them).
Figure I.2

Figure I.3

\[ \hat{\beta} = (X'X)^{-1}X'y \]

the regression line

Figure I.4
An Introduction to \textbf{R} by Stefano Iacus

\textbf{Figure I.5}

\begin{verbatim}
R> plot(1,type="n")
R> abline(h=0.6, lty=2)
R> abline(h=0.8, lty=3)
R> abline(h=0.9, lty=4)
R> abline(h=1.2, lty=5)
R> abline(h=1.4, lty=6)

We can also specify colors using the argument \texttt{col="red"}, for example, if we want to draw plot in red. Colors can be chosen by name or from the palettes

\begin{verbatim}
R> str(colors())

   chr [1:657] "white" "aliceblue" "antiquewhite" ...
R> palette()

[1] "black" "red" "green3" "blue" "cyan" "magenta"
[7] "yellow" "gray"
\end{verbatim}

It is possible to modify the title, subtitle, axes' labels and tick values. Next is an example of this

\begin{verbatim}
R> par(mfrow=c(1,2))
R> x <- c(1,2,5,9,10,11)
R> y <- c(1,7,5,4,3,1)
R> plot(x,y, main="Title", sub="subtitle", ylab="values of y")
R> plot(x,y, main="Title\n under the title", sub="subtitle", axes=FALSE,
     xlab="numbers", type="b")
R> axis(2)
R> axis(1,x,c("one","two","five","nine", "ten","eleven"))
\end{verbatim}

In the above we have also used the argument \texttt{AXES=FALSE} because we want to draw the axes ourselves. With this aim we use \texttt{axes(2)} to draw the vertical axis and \texttt{axis(1,\ldots )} to draw the values of \texttt{X} at given positions and with given labels.

It is also possible to add text and formulas to the graph, making use of the \texttt{expression} command. The text is added using \texttt{text} command and the corresponding horizontal and vertical lines.

\begin{verbatim}
R> plot(cars)
R> abline(lm(dist~speed, data=cars),lty=3,lwd=2)
\end{verbatim}

The text can further be aligned in different ways as the next code shows

```r
R> plot(1, main="Text alignment")
R> text(1,0.8,"hello",adj=0)  # sinistra
R> text(1,0.9,"hello",adj=0.5)  # centro
R> text(1,1.1,"hello",adj=1)  # destra
R> abline(v=1,lty=3)
```

### I.11 Bibliographic notes

There are many basic books apart from the one mentioned earlier (Dalgaard, 2008), such as Crawley (2007), which cover the basic functionalities of the R language. A simple search with the keyword R in on-line book stores will return hundreds of titles. For advanced programming techniques on the standard S language we should mention Chambers (2008) and Venables and Ripley (2000). For S4 programming some recent references are Chambers (2008) and Gentleman (2008). For advanced graphics one should not miss the books of Murrel (2005) and Deepayan (2008).

### Bibliography


