R introductory course

Daniele Amberti and Longhow Lam

Nov-2011

daniele dot amberti at gmail dot com

Contents

List of Figures

This handout is based on 'An Itnroduction to R' by Longhow Lam, see [1].

1 Introduction

1.1 What is R?

While the commercial implementation of S, S-PLUS, is struggling to keep its existing users, the open source version of S, R, has received a lot of attention in the last five years. Not only because the R system is a free tool, the system has proven to be a very effective tool in data manipulation, data analysis, graphing and developing new functionality. The user community has grown enormously the last years, and it is an active user community writing new R packages that are made available to others.

If you have any questions or comments on this document please do not hesitate to contact me.

The best explanation of R is given on the R web site http://www.r-project.org. The remainder of this section and the following section are taken from the R web site.

R is a language and environment for statistical computing and graphics. It is a GNU project which is similar to the S language and environment which was developed at Bell Laboratories (formerly AT&T, now Lucent Technologies) by John Chambers and colleagues. R can be considered as a different implementation of S. There are some important differences, but much code written for S runs unaltered under R.

R provides a wide variety of statistical (linear and non linear modeling, classical statistical tests, time-series analysis, classification, clustering, ...) and graphical techniques, and is highly extensible. The S language is often the vehicle of choice for research in statistical methodology, and R provides an Open Source route to participation in that activity.

One of R's strengths is the ease with which well-designed publication-quality plots can be produced, including mathematical symbols and formulae where needed. Great care has been taken over the defaults for the minor design choices in graphics, but the user retains full control.

R is available as Free Software under the terms of the Free Software Foundation's GNU General Public License in source code form. It compiles and runs on a wide variety of UNIX platforms and similar systems (including FreeBSD and Linux), Windows and MacOS.

1.2 The R environment

R is an integrated suite of software facilities for data manipulation, calculation and graphical display. It includes

- an effective data handling and storage facility,
- a suite of operators for calculations on arrays, in particular matrices,
- a large, coherent, integrated collection of intermediate tools for data analysis,
- graphical facilities for data analysis and display either on-screen or on hardcopy, and
- a well-developed, simple and effective programming language which includes conditionals, loops, user-defined recursive functions and input and output facilities.

The term 'environment' is intended to characterize it as a fully planned and coherent system, rather than an incremental accretion of very specific and inflexible tools, as is frequently the case with other data analysis software.

R, like S, is designed around a true computer language, and it allows users to add additional functionality by defining new functions. Much of the system is itself written in the R dialect of S, which makes it easy for users to follow the algorithmic choices made. For computationally-intensive tasks, C, C++ and Fortran code can be linked and called at run time. Advanced users can write C code to manipulate R objects directly.

Many users think of R as a statistics system. We prefer to think of it of an environment within which statistical techniques are implemented. R can be extended (easily) via packages. There are about eight packages supplied with the R distribution and many more are available through the CRAN family of Internet sites covering a very wide range of modern statistics.

R has its own LaTeX-like documentation format, which is used to supply comprehensive documentation, both on-line in a number of formats and in hardcopy.

1.3 Obtaining and installing R

R can be downloaded from the 'Comprehensive R Archive Network' (CRAN). You can download the complete source code of R, but more likely as a beginning R user you want to download the precompiled binary distribution of R. Go to the R web site http://www.r-project.org and select a CRAN mirror site (or simply http://cran.r-project.org) and download the base distribution file, under Windows: `R-2.14.0-win.exe`. At the time of writing the latest version is 2.14.0. We will mention user contributed packages in the next section.

The base file has a size of around 45MB, which you can execute to install R. The installation wizard will guide you through the installation process. It may be useful to install all the On-line pdf manuals and, under Windows, to customize Internet Access startup options to make use of Internet Explorer proxy settings. You can select appropriate options in the installation wizard.

1.4 Your first R session

Start the R system, the main window (RGui) with a sub window (R Console) will appear as in figure 1.1.

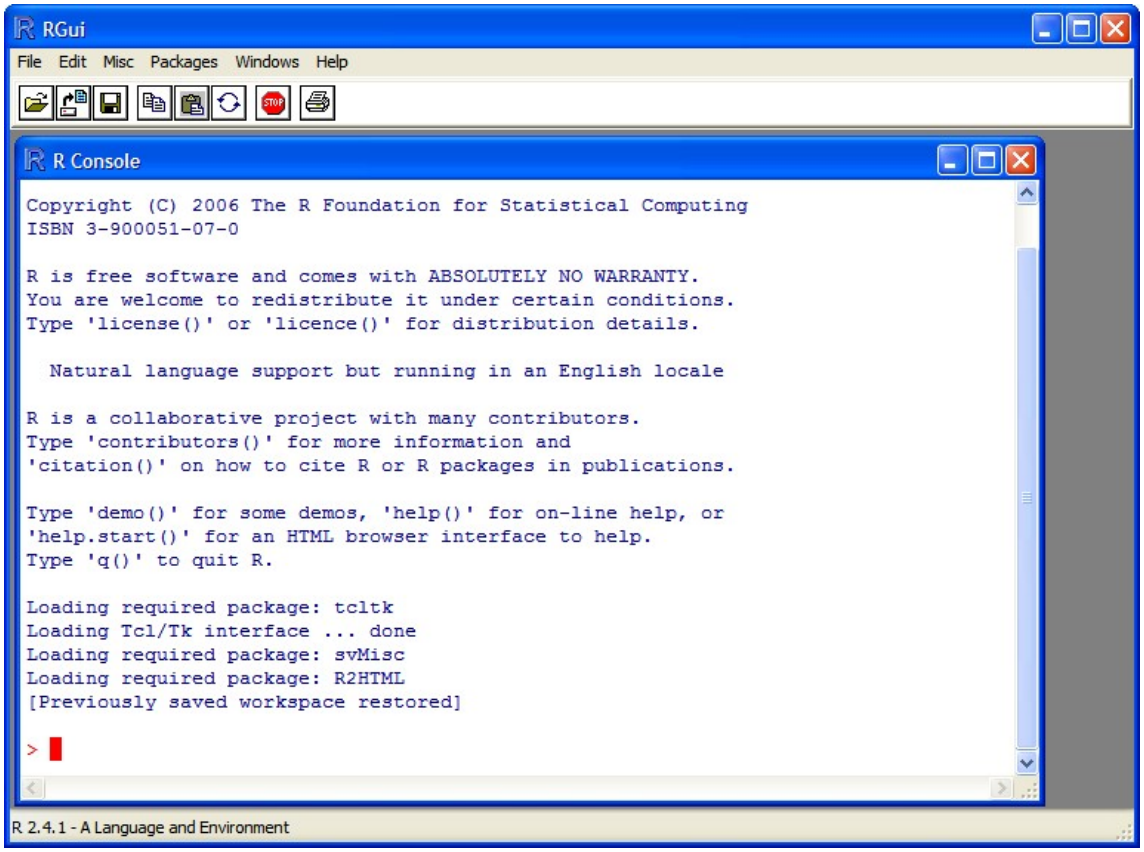

Figure 1.1: The R system on Windows

In the 'Console' window the cursor is waiting for you to type in some R commands. For example, use R as a simple calculator:

```
> print("Hello world!")

[1] "Hello world!"
```

```
> 1+1

[1] 2

> 1 + sin(9)

[1] 1.412118

> 234/87754

[1] 0.002666545

> (1 + 0.05)^8

[1] 1.477455

> 23.76*log(8)/(23 + atan(9))

[1] 2.019920
```

Results of calculations can be stored in objects using the assignment operators:

- An arrow (<-) formed by a smaller than character and a hyphen without a space!
- The equal character (=).

These objects can then be used in other calculations. To print the object just enter the name of the object. There are some restrictions when giving an object a name:

- A syntactically valid name consists of letters, numbers and the dot (.) or underscore (_) characters.
- Object names cannot contain 'strange' symbols like !, +, -, #.
- Object names can contain a number but cannot start with a number.
- Object names can start with a dot but not followed by a number.
- R is case sensitive, `X` and `x` are two different objects, as well as `temp` and `temP`.
- Reserved words in R's parser are not valid object names.

```
> x = sin(9)/75
> y = log(x) + x^2
> x

[1] 0.005494913

> y

[1] -5.203902
```

```
> m <- matrix(c(1,2,4,1), ncol=2)
> m

     [,1] [,2]
[1,]    1    4
[2,]    2    1

> solve(m)

           [,1]       [,2]
[1,] -0.1428571  0.5714286
[2,]  0.2857143 -0.1428571
```

To list the objects that you have in your current R session use the function `ls` or the function `objects`.

```
> ls()

[1] "m" "x" "y"
```

So to run the function `ls` we need to enter the name followed by an opening (and and a closing). Entering only `ls` will just print the object, you will see the underlying R code of the the function `ls`. Most functions in R accept certain arguments. For example, one of the arguments of the function `ls` is `pattern`. To list all objects starting with the letter x:

```
> x2 = 9
> y2 = 10
> ls(pattern="x")

[1] "x"  "x2"
```

If you assign a value to an object that already exists then the contents of the object will be overwritten with the new value (without a warning!). Use the function `rm` to remove one or more objects from your session.

```
> rm(x, x2)
```

To conclude your first session, we create two small vectors with data and a scatterplot.

```
> z2 <- c(1,2,3,4,5,6)
> z3 <- c(6,8,3,5,7,1)
> plot(z2,z3)
> title("My first scatterplot")
```

After this very short R session which barely scratched the surface, we hope you continue using the R system. The following chapters of this document will explain in detail the different data types, data structures, functions, plots and data analysis in R.

1.5 The available help

1.5.1 The on line help

There is extensive on line help in the R system, the best starting point is to run the function `help.start()`. This will launch a local page inside your browser with links to the R manuals, R FAQ, a search engine and other links.

In the R Console the function `help` can be used to see the help file of a specific function.

```
help(mean)
```

Use the function `help.search` to list help files that contain a certain string.

```
help.search("reserved")
Help files with alias or concept or title matching 'reserved' using fuzzy matching:


base::Reserved                  Reserved Words in R
DoE.base::fix                   Function to preserve class design when editing a design
MASS::npr1                      US Naval Petroleum Reserve No. 1 data
quantmod::getSymbols.FRED       Download Federal Reserve Economic Data - FRED(R)


Type '?PKG::FOO' to inspect entries 'PKG::FOO', or 'TYPE?PKG::FOO' for entries like
'PKG::FOO-TYPE'.

> help.search("robust")
Help files with alias or concept or title matching 'robust' using fuzzy
matching:

hubers(MASS)            Huber Proposal 2 Robust Estimator of Location
                        and/or Scale
rlm(MASS)               Robust Fitting of Linear Models
summary.rlm(MASS)       Summary Method for Robust Linear Models
line(stats)             Robust Line Fitting
runmed(stats)           Running Medians -- Robust Scatter Plot
                        Smoothing

Type 'help(FOO, package = PKG)' to inspect entry 'FOO(PKG) TITLE'.
```

The R manuals are also on line available in pdf format. In the RGui window go the help menu and select 'manuals in pdf'.

1.5.2 The R mailing lists and the R Journal

There are several mailing lists on R, see the R website. The main mailing list is R-help, web interfaces are available where you can browse trough the postings or search for a specific key word. If you have a connection to the internet, then the function `RSiteSearch` in R can be used to search for a string in the archives of all the R mailing lists.

```
> RSiteSearch("robust")

A search query has been submitted to http://search.r-project.org
The results page should open in your browser shortly
```

Another very useful webpage on the internet is www.Rseek.org, a sort of R search enigine. Also take a look at the R Journal, at http://journal.r-project.org.

1.6 The R workspace, managing objects

Objects that you create during an R session are hold in memory, the collection of objects that you currently have is called the workspace. This workspace is not saved on disk unless you tell R to do so. This means that your objects are lost when you close R and not save the objects, or worse when R or your system crashes on you during a session.

When you close the RGui or the R console window, the system will ask if you want to save the workspace image. If you select to save the workspace image then all the objects in your current R session are saved in a file `.RData`. This is a binary file located in the working directory of R, which is by default the installation directory of R.

During your R session you can also explicitly save the workspace image. Go to the 'File' menu and then select 'Save Workspace...', or use the `save.image` function.

```
> ## save to the current working directory
> save.image()
> ## just checking what the current working directory is
> getwd()

[1] "C:/"

> ## save to a specific file and location
> save.image("C:\\RIntroductoryCourse.RData")
```

If you have saved a workspace image and you start R the next time, it will restore the workspace. So all your previously saved objects are available again. You can also explicitly load a saved workspace file, that could be the workspace image of someone else. Go the 'File' menu and select 'Load workspace...'.

1.7 R Packages

One of the strengths of R is that the system can easily be extended. The system allows you to write new functions and package those functions in a so called 'R package' (or 'R library'). The R package may also contain other R objects, for example data sets or documentation. There is a lively R user community and many R packages have been written and made available on CRAN for other users. Just a few examples, there are packages for portfolio optimization, drawing maps, exporting objects to html, time series analysis, spatial statistics and the list goes on and on.

When you download R, a number of packages are downloaded as well. To use a function in an R package, that package has to be attached to the system. When you start R not all of the downloaded packages are attached, only seven packages are attached to the system by default. You can use the function `search` to see a list of packages that are currently attached to the system, this list is also called the *search path*.

```
> search()

[1] ".GlobalEnv"        "package:stats"     "package:graphics"
[4] "package:grDevices" "package:utils"     "package:datasets"
[7] "package:methods"   "Autoloads"         "package:base"
```

The first element of the output of `search` is `".GlobalEnv"`, which is the current workspace of the user. To attach another package to the system you can use the menu or the `library` function. Via the menu: Select the 'Packages' menu and select 'Load package...', a list of available packages on your system will be displayed. Select one and click 'OK', the package is now attached to your current R session. Via the `library` function:

```
> exists("shoes")

[1] FALSE

> library(MASS)
> exists("shoes")

[1] TRUE

> shoes

$A
 [1] 13.2  8.2 10.9 14.3 10.7  6.6  9.5 10.8  8.8 13.3

$B
 [1] 14.0  8.8 11.2 14.2 11.8  6.4  9.8 11.3  9.3 13.6
```

The function `library` can also be used to list all the available libraries on your system with a short description. Run the function without any arguments

```
> library()
```

If you have a connection to the internet then a package on CRAN can be installed very easily. To install a new package go to the 'Packages' menu and select 'Install package(s)...'. Then select a CRAN mirror near you, a (long) list with all the packages will appear where you can select one or more packages. Click 'OK' to install the selected packages. Note that the packages are only installed on your machine and not loaded (attached) to your current R session. As an alternative to the function `search` use `sessionInfo` to see system packages and user attached packages.

```
> sessionInfo()

R version 2.12.2 (2011-02-25)
Platform: x86_64-pc-mingw32/x64 (64-bit)

locale:
[1] LC_COLLATE=Italian_Italy.1252  LC_CTYPE=Italian_Italy.1252
[3] LC_MONETARY=Italian_Italy.1252 LC_NUMERIC=C
[5] LC_TIME=Italian_Italy.1252

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] MASS_7.3-11
```

1.8 Conflicting objects

It is not recommended to do, but R allows the user to give an object a name that already exists. If you are not sure if a name already exists, just enter the name in the R console and see if R can find it. R will look for the object in all the libraries (packages) that are currently attached to the R system. R will not warn you when you use an existing name.

```
> sum

function (..., na.rm = FALSE)  .Primitive("sum")

> sum = 10
> sum

[1] 10
```

The object `sum` already exists in the base package, but is now *masked* by your object `sum`. To get a list of all masked objects use the function `conflicts`.

```
> conflicts()

[1] "body<-" "sum"
```

You can safely remove the object `sum` with the function `rm` without risking deletion of the `sum` function. Calling `rm` removes only objects in your working environment by default.

1.9 Editors for R scripts, notebook graphical user interfaces

1.9.1 The editor in RGui

The console window (command line interface) in R is only useful when you want to enter one or two statements. It is not useful when you want to edit or write larger blocks of R code. In the RGui window you can open a new script, go to the 'File' menu and select 'New Script'. An empty R editor will appear where you can enter R code. This code can be saved, it will be a normal text file, normally with a .R extension. Existing text files with R code can be opened in the RGui window.

To run code in an R editor, select the code and use <Ctrl>-R to run the selected code. You can see that the code is parsed in the console window, any results will be displayed there.

1.9.2 Other editors

The built-in R editor is not the most fancy editor you can think of. It does not have much functionality. Since writing R code is just creating text files, you can do that with any text editor you like. If you have R code in a text file, you can use the `source` function to run the code in R. The function reads and executes all the statements in a text file.

```
# In the console window
source("C:\\Temp\\MyRfile.R")
```

There are free text editors that can send the R code inside the text editor to an R session. Some free editors that are worth mentioning are RStudio (http://www.rstudio.org), Eclipse (www.eclipse.org), Tinn-R (http://www.sciviews.org/Tinn-R) and JGR (speak 'Jaguar' http://jgr.markushelbig.org).

RStudio

RStudio is an integrated development environment (IDE) for R. RStudio combines an intuitive user interface with powerful coding tools to help you get the most out of R. It is available on all major platforms. Built in functionality includes:

- Support for R and Sweave files with one-click Pdf.
- Code completion.

- Searchable History.
- Code Transformations.
- Pane layout with source, workspace, history, plots, help and more.
- Project management and Version control.

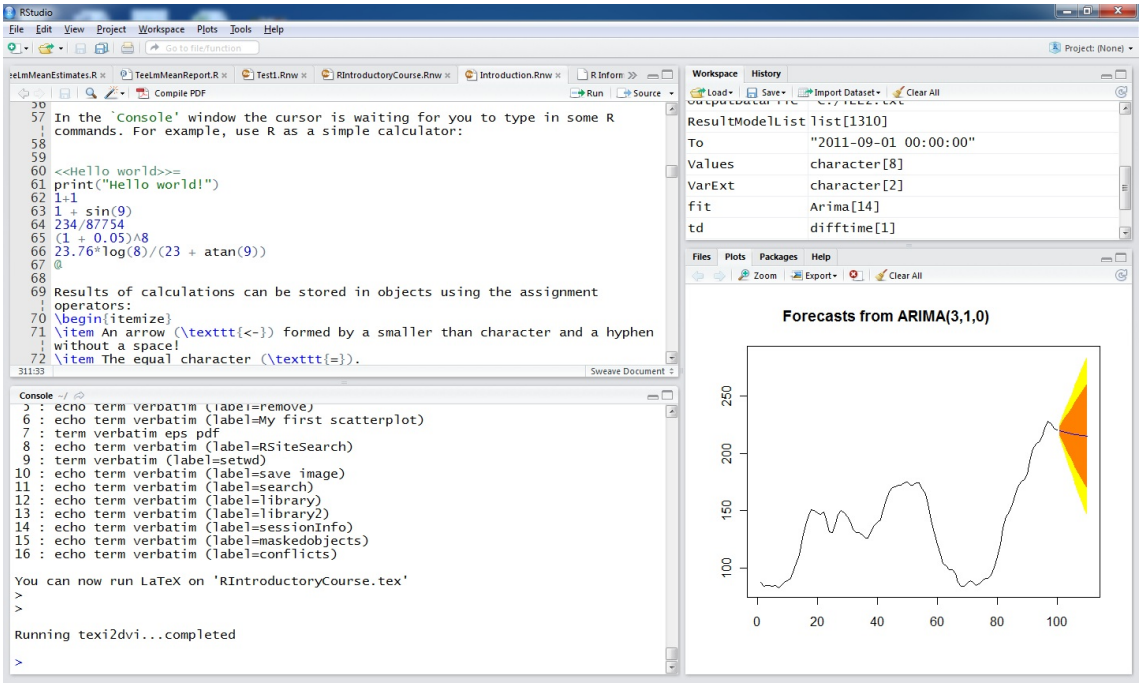

Figure 1.2: R integrated in the RStudio development environment

Eclipse

Eclipse is more than a text editor it is an environment to create, test manage and maintain (large) pieces of code. Built in functionality includes:

- Managing different text files in a project.
- Version control, recall previously saved versions of your text file.
- Search in multiple files.

The eclipse environment allows user to develop so called perspectives (or plug-ins). Such a plug-in customizes the Eclipse environment for a certain programming language. Stephan Wahlbrink has written an Eclipse plug-in for R, called 'StatEt'. See www.walware.de/goto/statet and see [2]. This plug-in adds extra 'R specific' functionality:

- Start an R console or terminal within Eclipse.

- Color coding of key words.
- Run R code in Eclipse by sending it to the R console.
- Insert predefined blocks of R code (templates).
- Supports writing R documentation files (*.Rd files).

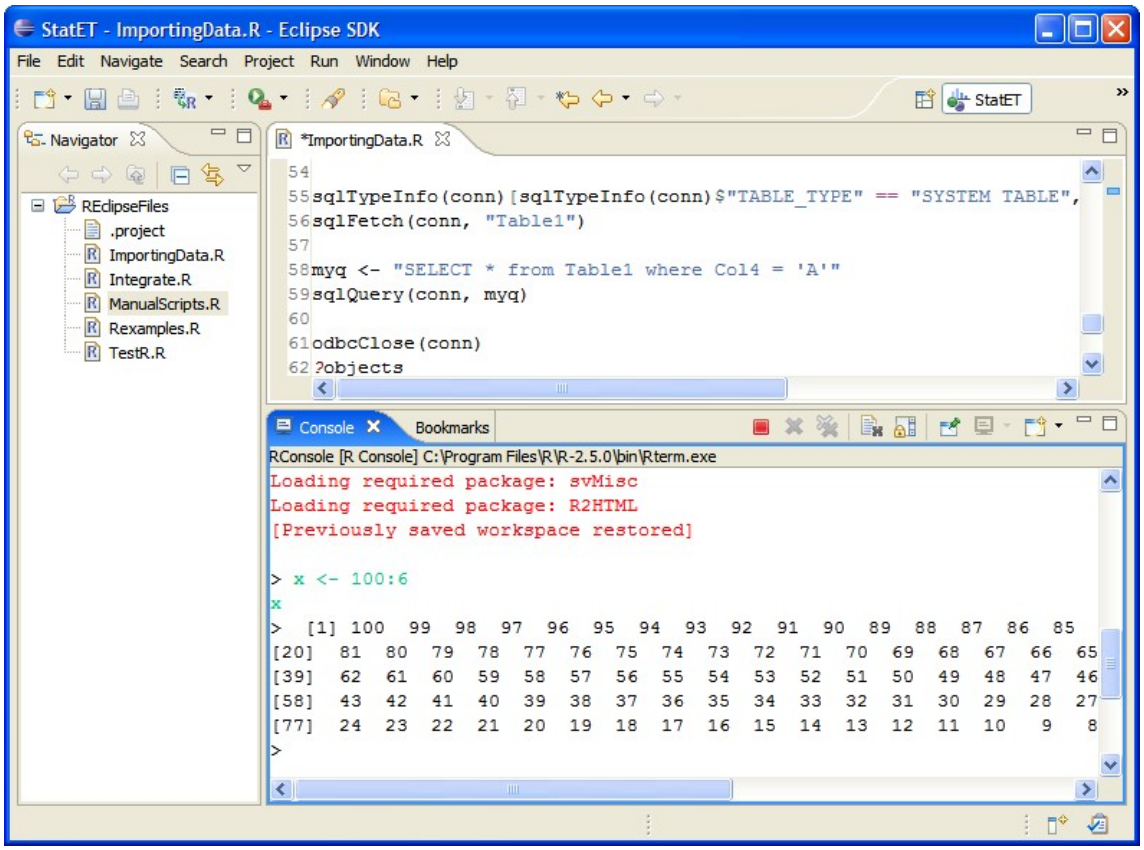

Figure 1.3: R integrated in the Eclipse development environment

Tinn-R

Tinn stands for Tinn is not Notepad, it is a text editor that was originally developed to replace the boring Notepad. With each new version of Tinn more features were added, and it has become a very nice environment to edit and maintain code. Tinn-R is the special R version of Tinn. It allows color highlighting of the R language and sending R statements to an R Console window.

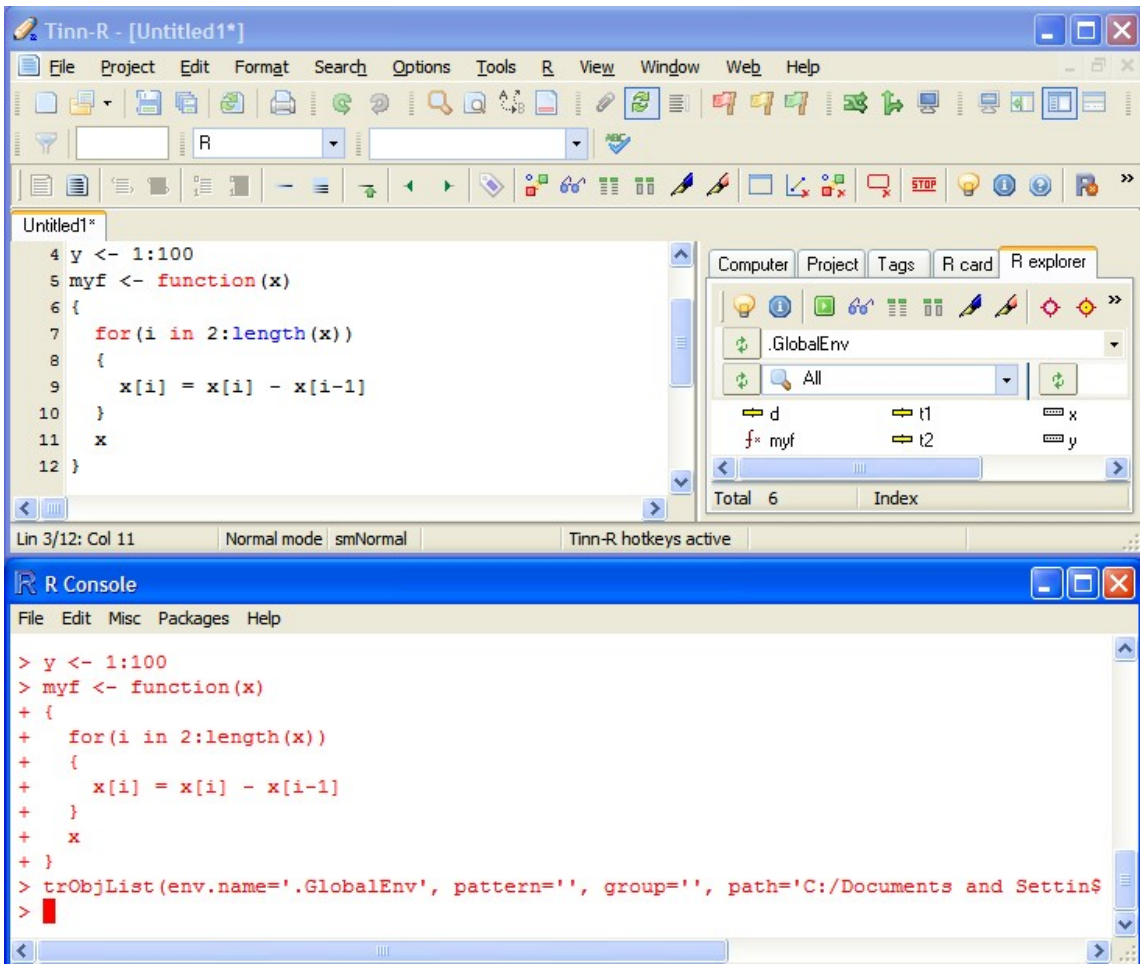

Figure 1.4: The Tinn-R and an the R Console environment

JGR

JGR (Java GUI for R) is a universal and unified Graphical User Interface for R. It includes among others: an integrated editor, help system 'Type-on' spreadsheet and an object browser.

1.10 Menus and dialog boxes graphical user interfaces

This type of GUI is commonly found in current statistical packages like SPSS, JMP, Minitab, Systat, Statistica, Splus.

Rcmdr

The R-Commander GUI consists of a window containing several menus, buttons, and information fields. In addition, the Commander window contains script and output text windows useful to learn R language from the GUI. Rcmdr is available as a package, it is possible to get a copy of the latest released version of the Rcmdr package through CRAN as explained in section 1.7 on page 13.

Deducer

Deducer is designed to be a free easy to use alternative to proprietary data analysis software such as SPSS, JMP, and Minitab. It has a menu system to do common data manipulation and analysis tasks, and an excel-like spreadsheet in which to view and edit data frames. Deducer is designed to be used with the Java based R console JGR, though it supports a number of other R environments (e.g. Windows RGUI and RTerm). Deducer is available as a package, it is possible to get a copy of the latest released version of the Deducer package through CRAN as explained in section 1.7.

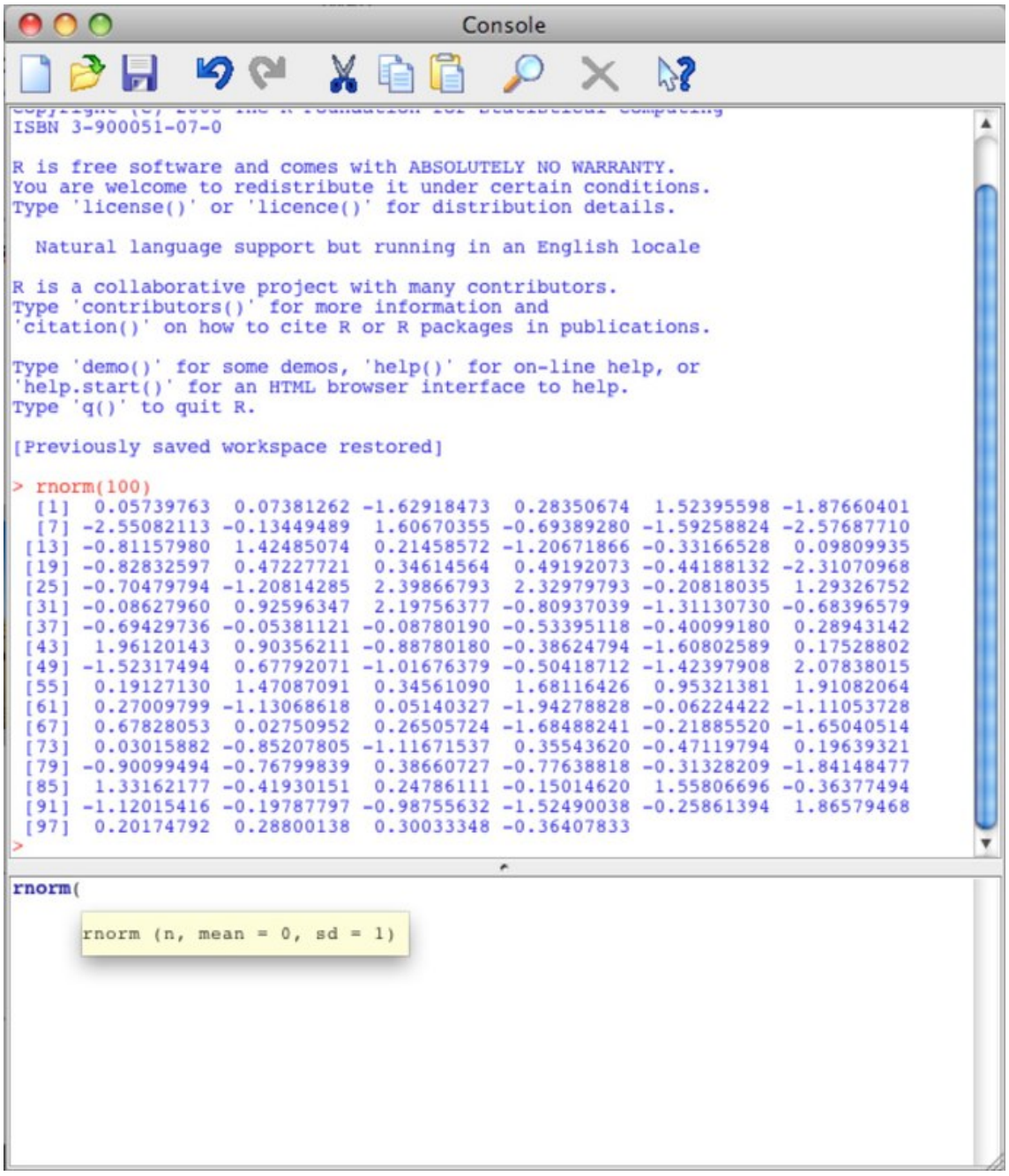

Figure 1.5: The JGR interface

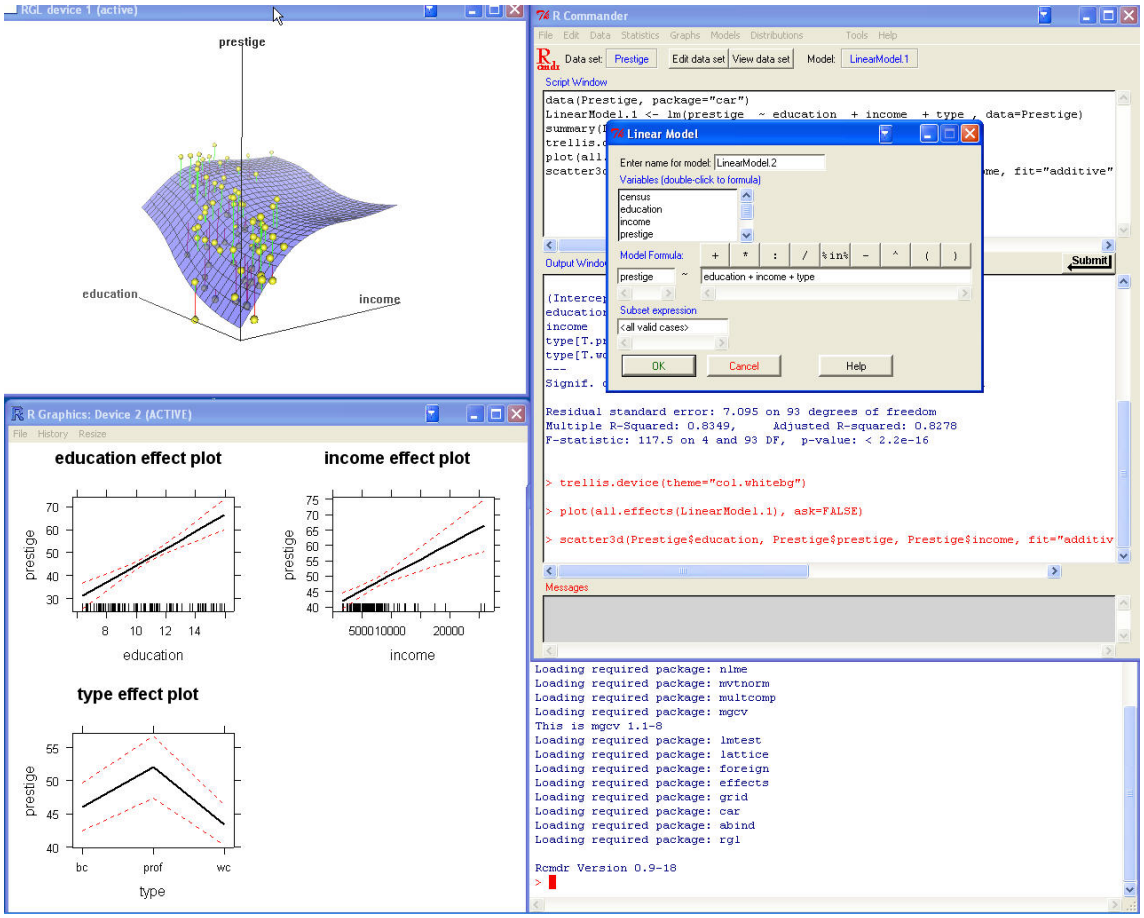

Figure 1.6: The Rcmdr interface

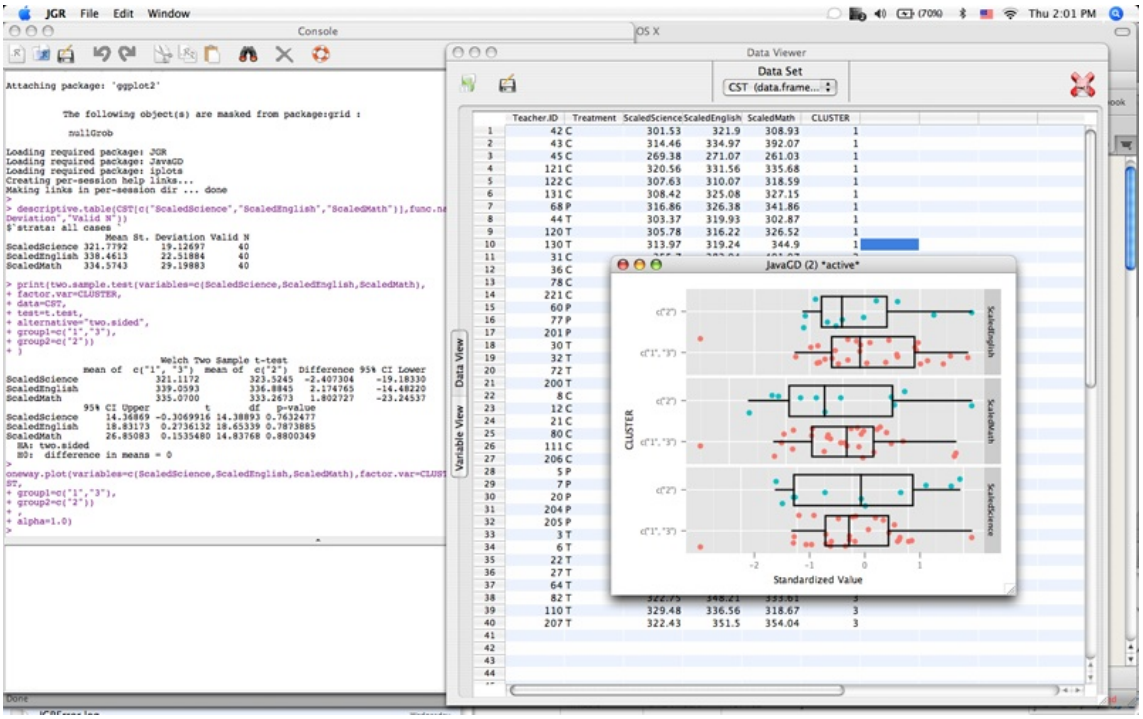

Figure 1.7: The Deducer interface

2 Data Objects

In this section we will discuss the different aspects of data types and structures in R. Operators such as `c` and `:` will be used in this section as an illustration and will be discussed in the next section. If you are confronted with an unknown function, you can ask for help by typing in the command `help`:

```
> help(c)
```

A help text will appear and describe the purpose of the function and how to use it.

2.1 Data types

Technically data types are those returned by `typeof` and it represents the (R internal) type or storage mode of any object. Current values are the vector types 'logical', 'integer', 'double', 'complex', 'character', 'raw' and 'list', 'NULL, 'closure' (function), 'special' and 'builtin' (basic functions and operators), 'environment', 'S4' (some S4 objects) and others that are unlikely to be seen at user level.

In this section we will adopt a user oriented approach describing basic data types adding factors, dates, times, missing data, infinite values and data structures (e.g. matrices)

2.1.1 Double

If you do calculations on numbers, you can use the data type double to represent the numbers. Doubles are numbers like `3.1415`, `8.0` and `8.1`. Doubles are used to represent continuous variables like the weight or length of a person.

```
> x <- 8.14
> y <- 8.0
> z <- 87.0 + 12.9
```

Use the function `is.double` to check if an object is of type double. Alternatively, use the function `typeof` to ask R the type of the object `x`. Type numeric is identical to double (and real).

```
> typeof(x)
```

```
[1] "double"

> is.double(8.9)

[1] TRUE

> test <- 1223.456
> is.double(test)

[1] TRUE

> is.numeric(test)

[1] TRUE

> is.real(test)

[1] TRUE
```

Keep in mind that doubles are just approximations to real numbers. Mathematically there are infinity many numbers, the computer can ofcourse only represent a finite number of numbers. Not only can numbers like π or $\sqrt{2}$ not be represented exactly, less exotic numbers like 0.1 for example can also not be represented exactly.

One of the conseqeunces of this is that when you compare two doubles with each other you should take some care. Consider the following (surprising) result.

```
> 0.3 == 0.1 + 0.1 + 0.1

[1] FALSE
```

2.1.2 Character

A character object is represented by a collection of characters between double quotes ("). For example: `"x"`, `"test character"` and `"iuiu8ygy-iuhu"`. One way to create character objects is as follows.

```
> x <- c("a","b","c")
> x

[1] "a" "b" "c"

> mychar1 <- "This is a test"
> mychar1

[1] "This is a test"
```

```
> mychar2 <- "This is another test"
> mychar2

[1] "This is another test"

> charvector <- c("a", "b", "c", "test")
> charvector

[1] "a"    "b"    "c"    "test"
```

The double quotes indicate that we are dealing with an object of type 'character'.

2.1.3 Logical

An object of data type logical can have the value `TRUE` or `FALSE` and is used to indicate if a condition is true or false. Such objects are usually the result of logical expressions.

```
> x <- 9
> y <- x > 10
> y

[1] FALSE
```

The result of the function `is.double` is an object of type logical (TRUE or FALSE).

```
> is.double(9.876)

[1] TRUE
```

Logical expressions are often built from logical operators:

`<`	smaller than
`<=`	smaller than or equal to
`>`	larger than
`>=`	larger than or equal to
`==`	is equal to
`!=`	is unequal to

The logical operators `and`, `or` and `not` are given by `&`, | and `!`, respectively.

```
> x <- c(9,166)
> y <- (3 < x) & (x <= 10)
```

Calculations can also be carried out on logical objects, in which case the `FALSE` is replaced by a zero and a one replaces the `TRUE`. For example, the sum function can be used to count the number of `TRUE`'s in a vector or array.

```
> x <- 1:15
> ## number of elements in x larger than 9
> sum(x>9)
```

```
[1] 6
```

2.1.4 Integer

Integers are natural numbers. They can be used to represent counting variables, for example the number of children in a household. In R integers are mostly used for internal representation of a factor, see 2.1.5, because of this it is necessary to explicitly declare a numer as integer to get the appropriate type but fortunately mixing objects of type 'double' and 'integer' is not a problem.

```
> nchild <- as.integer(3)
> is.integer(nchild)
```

```
[1] TRUE
```

Note that 3.0 is not an integer, nor is 3 by default an integer!

```
> nchild  <- 3.0
> is.integer(nchild)
```

```
[1] FALSE
```

```
> nchild <- 3
> is.integer(nchild)
```

```
[1] FALSE
```

So a 3 of type 'integer' in R is something different than a 3.0 of type 'double'. However, you can mix objects of type 'double' and 'integer' in one calculation without any problems.

```
> x <- as.integer(7)
> y <- 2.0
> z <- x/y
> z
```

```
[1] 3.5
```

In contrast to some other programming languages, the answer is of type double and is 3.5. The maximum integer in R is $2^{31} - 1$.

```
> as.integer(2^31 - 1)
```

```
[1] 2147483647
```

```
> as.integer(2^31)
```

```
[1] NA
```

2.1.5 Factor

The factor data type is used to represent categorical data (i.e. data of which the value range is a collection of codes). For example:

- variable 'sex' with values male and female.
- variable 'blood type' with values: A, AB and O.

A factor variable can be useful in various statistical analysis including regression, see section ??.

An individual code of the value range is also called a *level* of the factor variable. So the variable 'sex' is a factor variable with two levels, male and female.

Sometimes people confuse factor type with character type. Characters are often used for labels in graphs, column names or row names. Factors must be used when you want to represent a discrete variable in a data frame and want to analyze it.

Factor objects can be created from character objects or from numeric objects, using the function factor. For example, to create a vector of length five of type factor do the following:

```
> sex <- c("male","male","female","male","female")
```

The object `sex` is a character object. You need to transform it to factor.

```
> sex <- factor(sex)
> sex
```

```
[1] male   male   female male   female
Levels: female male
```

Use the function `levels` to see the different levels a factor variable has.

```
> levels(sex)
```

```
[1] "female" "male"
```

Note that the result of the `levels` function is of type character. Another way to generate the sex variable is as follows:

```
> sex <- c(2,2,1,2,1)
```

The object 'sex' is an integer variable, you need to transform it to a factor.

```
> sex <- factor(sex)
> sex
```

```
[1] 2 2 1 2 1
Levels: 1 2
```

The object 'sex' looks like, but is not an integer variable. The 1 represents level "1" here. So arithmetic operations on the sex variable are not possible:

```
> sex + 7
```

```
[1] NA NA NA NA NA
```

It is better to rename the levels, so level "1" becomes female and level "2" becomes male:

```
> levels(sex) <- c("female", "male")
> sex
```

```
[1] male   male   female male   female
Levels: female male
```

You can transform factor variables to double or integer variables using the `as.double` or `as.integer` function.

```
> sex.numeric <- as.double(sex)
> sex.numeric
```

```
[1] 2 2 1 2 1
```

The 1 is assigned to the female level, only because alphabetically female comes first. If the order of the levels is of importance, you will need to use *ordered factors*. Use the function `ordered` and specify the order with the levels argument. For example:

```
> Income <- c("High","Low","Average","Low","Average","High","Low")
> Income <- ordered(Income, levels=c("Low","Average","High"))
> Income
```

```
[1] High    Low     Average Low     Average High    Low
Levels: Low < Average < High
```

The last line indicates the ordering of the levels within the factor variable. When you transform an ordered factor variable, the order is used to assign numbers to the levels.

```
> Income.numeric  <- as.double(Income)
> Income.numeric
```

```
[1] 3 1 2 1 2 3 1
```

The order of the levels is also used in linear models. If one or more of the regression variables are factor variables, the order of the levels is important for the interpretation of the parameter estimates see section **??**.

2.1.6 Dates and Times

To represent a calendar date in R use the function `as.Date` to create an object of class Date.

```
> temp <- c("01-01-2000", "31-03-2000")
> z <- as.Date(temp, "%d-%m-%Y")
> z

[1] "2000-01-01" "2000-03-31"

> data.class(z)

[1] "Date"

> format(z, "%Y-%m-%d")

[1] "2000-01-01" "2000-03-31"
```

You can add a number to a date object, the number is interpreted as the number of day to add to the date.

```
> z + 19

[1] "2000-01-20" "2000-04-19"
```

You can subtract one date from another, the result is an object of class 'difftime'

```
> dz = z[2] -z[1]
> dz

Time difference of 90 days

> data.class(dz)

[1] "difftime"
```

In R the classes `POSIXct` and `POSIXlt` can be used to represent calendar dates and times. You can create `POSIXct` objects with the function `as.POSIXct`. The function accepts characters as input, and it can be used to not only to specify a date but also a time within a date.

```
> t1 <- as.POSIXct("2000-01-01")
> t2 <- as.POSIXct("2000-03-31 15:34")
> t1

[1] "2000-01-01 CET"

> t2

[1] "2000-03-31 15:34:00 CEST"
```

A handy function is `strptime`, it is used to convert a certain character representation of a date (and time) into another character representation. You need to provide a conversion specification that starts with a % followed by a single letter.

```
> # first creating four characters
> x <- c("01/01/2000", "01/02/2000", "03/31/2000", "07/30/2000")
> z <- strptime(x, "%m/%d/%Y")
> zt <- as.POSIXct(z)
> zt

[1] "2000-01-01 CET"  "2000-01-02 CET"  "2000-03-31 CEST" "2000-07-30 CEST"

> # pasting 4 character dates and 4 character times together
> dates <- c("01/01/2000", "01/02/2000", "03/31/2000", "07/30/2000")
> times <- c("23:03:20", "22:29:56", "01:03:30", "18:21:03")
> x <- paste(dates, times)
> z <- strptime(x, "%m/%d/%Y %H:%M:%S")
> zt <- as.POSIXct(z, tz = "GMT")
> zt

[1] "2000-01-01 23:03:20 GMT" "2000-01-02 22:29:56 GMT"
[3] "2000-03-31 01:03:30 GMT" "2000-07-30 18:21:03 GMT"
```

An object of type `POSIXct` can be used in certain calculations, a number can be added to a `POSIXct` object. This number will be the interpreted as the number of seconds to add to the POSIXct object.

```
> zt + 13

[1] "2000-01-01 23:03:33 GMT" "2000-01-02 22:30:09 GMT"
[3] "2000-03-31 01:03:43 GMT" "2000-07-30 18:21:16 GMT"
```

You can subtract two POSIXct objects, the result is a so called 'difftime' object.

```
> t2 <- as.POSIXct("2000-01-01 23:03:20")
> t1 <- as.POSIXct("2000-03-31 01:03:30")
> d <- t2-t1
> d
```

```
Time difference of -89.04178 days
```

A 'difftime' object can also be created using the function `as.difftime`, and you can add a difftime object to a POSIXct object. Due to a bug in R this can only safely be done with the function `"+.POSIXt"`.

```
> "+.POSIXt"(zt, d)
```

```
[1] "1999-10-04 22:03:10 GMT" "1999-10-05 21:29:46 GMT"
[3] "2000-01-02 00:03:20 GMT" "2000-05-02 17:20:53 GMT"
```

To extract the weekday, month or quarter from a POSIXct object use the handy R functions `weekdays`, `months` and `quarters`. Another handy function is `Sys.time`, which returns the current date and time.

```
> weekdays(zt)
```

```
[1] "sabato"   "domenica" "venerdì"  "domenica"
```

There are some R packages that can handle dates and time objects. For example, the packages zoo, chron, tseries, its, xts and Rmetrics. Especially Rmetrics has a set of powerful functions to maintain and manipulate dates and times. See [3].

2.1.7 Complex

Objects of type 'complex' are used to represent complex numbers. In statistical data analysis you will not need them often. Use the function `as.complex` or `complex` to create objects of type complex.

```
> test1 <- as.complex(-1)
> test1
```

```
[1] -1+0i
```

```
> sqrt(test1)
```

```
[1] 0+1i
```

```
> typeof(test1)
```

```
[1] "complex"
```

Note that by default calculations are done on real numbers, so `sqrt(-1)` results in `NA`.

2.1.8 Missing data and Infinite values

We have already seen the symbol `NA`. In R it is used to represent 'missing' data (*Not Available*). It is not really a separate data type, it could be a missing double or a missing integer. To check if data is missing, use the function `is.na` or use a direct comparison with the symbol NA. There is also the symbol `NaN` (*Not a Number*), which can be detected with the function `is.nan`.

```
> x <- as.double( c("1", "2", "qaz"))
> is.na(x)

[1] FALSE FALSE  TRUE

> z <- sqrt(c(1,-1))
> is.nan(z)

[1] FALSE  TRUE
```

Infinite values are represented by `Inf` or `-Inf`. You can check if a value is infinite with the function `is.infinite`. Use `is.finite` to check if a value is finite.

```
> x <- c(1,3,4)
> y <- c(1,0,4)
> x/y

[1]   1 Inf   1

> z <- log(c(4,0,8))
> is.infinite(z)

[1] FALSE  TRUE FALSE
```

In R `NULL` represents the null object. `NULL` is used mainly to represent the lists with zero length, and is often returned by expressions and functions whose value is undefined.

2.2 Data structures

Before you can perform statistical analysis in R, your data has to be structured in some coherent way. To store your data R has the following structures:

- vector
- matrix
- array
- data frame

- time-series
- list

2.2.1 Vectors

The simplest structure in R is the vector. A vector is an object that consists of a number of elements of the same type, all doubles or all logical. A vector with the name 'x' consisting of four elements of type 'double' (10, 5, 3, 6) can be constructed using the function `c`.

```
> x <- c(10, 5, 3, 6)
> x

[1] 10  5  3  6
```

The function `c` merges an arbitrary number of vectors to one vector. A single number is regarded as a vector of length one.

```
> y <- c(x,0.55, x, x)
> y

 [1] 10.00  5.00  3.00  6.00  0.55 10.00  5.00  3.00  6.00 10.00  5.00  3.00
[13]  6.00
```

Typing the name of an object in the commands window results in printing the object. The numbers between square brackets indicate the position of the following element in the vector.

Use the function `round` to round the numbers in a vector.

```
> round (y,3)  # round to 3 decimals

 [1] 10.00  5.00  3.00  6.00  0.55 10.00  5.00  3.00  6.00 10.00  5.00  3.00
[13]  6.00
```

Mathematical calculations on vectors

Calculations on (numerical) vectors are usually performed on each element. For example, `x*x` results in a vector which contains the squared elements of `x`.

```
> x

[1] 10  5  3  6
```

```
> z <- x*x
> z

[1] 100  25   9  36
```

The symbols for elementary arithmetic operations are `+`, `-`, `*`, `/`. Use the `^` symbol to raise power. Most of the standard mathematical functions are available in R. These functions also work on each element of a vector. For example the logarithm of x:

```
> log(x)

[1] 2.302585 1.609438 1.098612 1.791759
```

Function name	Operation
`abs`	absolute value
`asin acos atan`	inverse geometric functions
`asinh acosh atanh`	inverse hyperbolic functions
`exp log`	exponent and natural logarithm
`floor ceiling trunc`	creates integers from floating point numbers
`gamma lgamma`	gamma and log gamma function
`log10`	logarithm with basis 10
`round`	rounding
`sin cos tan`	geometric functions
`sinh cosh tanh`	hyperbolic functions
`sqrt`	square root

Table 2.1: Some mathematical functions that can be applied on vectors

The recycling rule

It is not necessary to have vectors of the same length in an expression. If two vectors in an expression are not of the same length then the shorter one will be repeated until it has the same length as the longer one. A simple example is a vector and a number (which is a vector of length one).

```
> sqrt(x) + 2

[1] 5.162278 4.236068 3.732051 4.449490
```

In the above example the 2 is repeated 4 times until it has the same length as `x` and then the addition of the two vectors is carried out. In the next example, `x` has to be repeated 1.5 times in order to have the same length as `y`. This means the first two elements of `x` are added to `x` and then `x*y` is calculated.

```
> x <- c(1,2,3,4)
> y <- c(1,2,3,4,5,6)
> z <- x*y
> z

[1]  1  4  9 16  5 12
```

Generating vectors

Regular sequences of numbers can be very handy for all sorts of reasons. Such sequences can be generated in different ways. The easiest way is to use the column operator (:).

```
> index <- 1:20
> index

 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

A descending sequence is obtained by `20:1`. The function seq together with its arguments from, to, by or length is used to generate more general sequences. Specify the beginning and end of the sequence and either specify the length of the sequence or the increment.

```
> u <- seq(from=-3,to=3,by =0.5)
> u

 [1] -3.0 -2.5 -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0  1.5  2.0  2.5  3.0
```

The following commands have the same result:

```
> u <- seq(-3,3,length=13)
> u

 [1] -3.0 -2.5 -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0  1.5  2.0  2.5  3.0

> u <- (-6):6/2
> u

 [1] -3.0 -2.5 -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0  1.5  2.0  2.5  3.0
```

The function `seq` can also be used to generate vectors with POSIXct elements (a sequence of dates). The following examples speak for them selves.

```
> seq(as.POSIXct("2000-01-01"), by = "month", length = 6)

[1] "2000-01-01 CET"  "2000-02-01 CET"  "2000-03-01 CET"  "2000-04-01 CEST"
[5] "2000-05-01 CEST" "2000-06-01 CEST"

> seq(ISOdate(2000,1,1), ISOdate(2005,6,1), "years")
```

```
[1] "2000-01-01 12:00:00 GMT" "2001-01-01 12:00:00 GMT"
[3] "2002-01-01 12:00:00 GMT" "2003-01-01 12:00:00 GMT"
[5] "2004-01-01 12:00:00 GMT" "2005-01-01 12:00:00 GMT"
```

The function `rep` repeats a given vector. The first argument is the vector and the second argument can be a number that indicates how often the vector needs to be repeated.

```
> rep(1:4, 4)

 [1] 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4

> rep(1:4, each = 4)

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

The second argument can also be a vector of the same length as the vector used for the first argument. In this case each element in the second vector indicates how often the corresponding element in the first vector is repeated.

```
> rep(1:4, c(2,2,2,2))

[1] 1 1 2 2 3 3 4 4

> rep(1:4, 1:4)

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

For information about other options of the function `rep` type `help(rep)`. To generate vectors with random elements you can use the functions `rnorm` or `runif`. There are more of these functions.

```
> x <- rnorm(10)                          # 10 random standard normal numbers
> y <- runif(10,4,7)        # 10 random numbers between 4 and 7
```

2.2.2 Matrices

Generating matrices

A matrix can be regarded as a generalization of a vector. As with vectors, all the elements of a matrix must be of the same data type. A matrix can be generated in several ways. For example:

- Use the function `matrix`:

```
> x <- matrix(1:8,2,4,byrow=F)
> x
```

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

By default the matrix is filled by column. To fill the matrix by row specify `byrow = T` as argument in the matrix function.

- Use the function `dim`:

```
> x <- 1:8
> x

[1] 1 2 3 4 5 6 7 8

> dim(x) <- c(2,4)
> x

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

1. Use the function `cbind` to create a matrix by binding two or more vectors as column vectors. The function `rbind` is used to create a matrix by binding two or more vectors as row vectors.

```
> cbind(c(1,2,3),c(4,5,6))

     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

> rbind(c(1,2,3),c(4,5,6))

     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Calculations on matrices

A matrix can be regarded as a number of equal length vectors pasted together. All the mathematical functions that apply to vectors also apply to matrices and are applied on each matrix element.

```
> x*x^2   # All operations are applied on each matrix element
```

```
     [,1] [,2] [,3] [,4]
[1,]    1   27  125  343
[2,]    8   64  216  512

> max(x)  # returns the maximum of all matrix elements in x

[1] 8
```

You can multiply a matrix with a vector. The outcome may be surprising:

```
> x <- matrix(1:16,ncol=4)
> y <- 7:10
> x*y

     [,1] [,2] [,3] [,4]
[1,]    7   35   63   91
[2,]   16   48   80  112
[3,]   27   63   99  135
[4,]   40   80  120  160

> x <- matrix(1:28,ncol=4)
> y <- 7:10
> x*y

     [,1] [,2] [,3] [,4]
[1,]    7   80  135  176
[2,]   16   63  160  207
[3,]   27   80  119  240
[4,]   40   99  144  175
[5,]   35  120  171  208
[6,]   48   91  200  243
[7,]   63  112  147  280
```

As an exercise: try to find out what R did.

To perform a matrix multiplication in the mathematical sense, use the operator: `%*%`. The dimensions of the two matrices must agree. In the following example the dimensions are wrong:

```
> x <- matrix(1:8,ncol=2)
> print(try(x %*% x))

[1] "Error in x %*% x : gli argomenti non sono compatibili\n"
attr(,"class")
[1] "try-error"
```

A matrix multiplied with its transposed `t(x)` always works.

```
> x %*% t(x)

     [,1] [,2] [,3] [,4]
[1,]   26   32   38   44
[2,]   32   40   48   56
[3,]   38   48   58   68
[4,]   44   56   68   80
```

R has a number of matrix specific operations, for example:

Function name	Operation
`chol(x)`	Choleski decomposition
`col(x)`	matrix with column numbers of the elements
`diag(x)`	create a diagonal matrix from a vector
`ncol(x)`	returns the number of columns of a matrix
`nrow(x)`	returns the number of rows of a matrix
`qr(x)`	QR matrix decomposition
`row(x)`	matrix with row numbers of the elements
`solve(A,b)`	solve the system Ax=b
`solve(x)`	calculate the inverse
`svd(x)`	singular value decomposition
`var(x)`	covariance matrix of the columns

Table 2.2: Some functions that can be applied on matrices

A detailed description of these functions can be found in the corresponding help files, which can be accessed by typing for example `?diag` in the R Console.

2.2.3 Arrays

Arrays are generalizations of vectors and matrices. A vector is a one-dimensional array and a matrix is a two dimensional array. As with vectors and matrices, all the elements of an array must be of the same data type. An example of an array is the three-dimensional array 'iris3', which is a built-in data object in R. A three dimensional array can be regarded as a block of numbers.

```
> dim(iris3)   # dimensions of iris

[1] 50  4  3
```

All basic arithmetic operations which apply to matrices are also applicable to arrays and are performed on each element.

```
> test <- iris + 2*iris
```

The function `array` is used to create an array object

```
> newarray <- array(c(1:8, 21:28, 331:338), dim = c(2,4,3))
> newarray

, , 1

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

, , 2

     [,1] [,2] [,3] [,4]
[1,]   21   23   25   27
[2,]   22   24   26   28

, , 3

     [,1] [,2] [,3] [,4]
[1,]  331  333  335  337
[2,]  332  334  336  338
```

2.2.4 Data frames

Data frames can also be regarded as an extension to matrices. Data frames can have columns of different data types and are the most convenient data structure for data analysis in R. In fact, most statistical modeling routines in R require a data frame as input.

One of the built-in data frames in R is 'mtcars'.

```
> head(mtcars) # only the first rows of a data.frame

                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

The data frame contains information on different cars. Usually each row corresponds with a case and each column represents a variable. In this example the 'carb' column is of data type 'double' and represents the number of carburetors. See the help file for more information on this data frame; `?mtcars`.

Data frame attributes

A data frame can have the attributes names and row.names. The attribute names contains the column names of the data frame and the attribute row.names contains the row names of the data frame. The attributes of a data frame can be retrieved separately from the data frame with the functions `names` and `row.names`. The result is a character vector containing the names.

```
> rownames(mtcars)[1:5] # only the first five row names

[1] "Mazda RX4"          "Mazda RX4 Wag"      "Datsun 710"
[4] "Hornet 4 Drive"     "Hornet Sportabout"

> names(mtcars)

 [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
[11] "carb"

> colnames(mtcars)

 [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
[11] "carb"
```

Creating data frames

You can create data frames in several ways, by importing a data file as in Chapter 3, for example, or by using the function `data.frame`. This function can be used to create new data frames or convert other objects into data frames.

A few examples of the `data.frame` function:

```
> my.logical <- sample(c(T,F),size=5,replace = T)
> my.numeric <- rnorm(5)
> my.df <- data.frame(my.logical,my.numeric)
> my.df
```

```
  my.logical  my.numeric
1       TRUE -1.08569914
2       TRUE -0.08542326
3       TRUE  1.07061054
4       TRUE -0.14539355
5       TRUE -1.16554485
```

```
> my.df <- data.frame(Logical = my.logical, Numeric = my.numeric)
> my.df
```

```
  Logical     Numeric
1    TRUE -1.08569914
2    TRUE -0.08542326
3    TRUE  1.07061054
4    TRUE -0.14539355
5    TRUE -1.16554485
```

```
> test <- matrix(runif(15, 0, 100),5,3) # create a matrix with random elements
> test <- data.frame(test)
> test
```

```
        X1        X2       X3
1 20.65314 66.511519 44.85163
2 12.75317  9.484066 81.00644
3 75.33079 38.396964 81.23895
4 89.50454 27.438364 79.43423
5 37.44628 81.464004 43.98317
```

```
> names(test)
```

```
[1] "X1" "X2" "X3"
```

R automatically creates column names: 'X1', 'X2' and 'X3'. You can use the `names` function to change these column names.

```
> names(test) <- c("Price", "Length", "Income")
> row.names(test) <- c("Paul","Ian","Richard","David","Rob")
> test
```

```
           Price    Length   Income
Paul    20.65314 66.511519 44.85163
Ian     12.75317  9.484066 81.00644
Richard 75.33079 38.396964 81.23895
David   89.50454 27.438364 79.43423
Rob     37.44628 81.464004 43.98317
```

2.2.5 Time-series objects

In R a time-series object (an object of class 'ts') is created with the function `ts`. It combines two components:

- The data, a vector or matrix of numeric values. In case of a matrix, each column is a separate time-series.
- The dates of the data, the dates are equispaced points in time.

```
> # starting from jan-2000, 12 monthly intervals
> myts1 <- ts(data = rnorm(12), start=c(2000), freq = 12)
> # two time-series starting from mar-2000, 3 monthly intervals
> myts2 <- ts(data = matrix(rnorm(6),ncol=2), start=c(2000,3), freq=12)
> myts2

            Series 1    Series 2
Mar 2000 -0.46665535  0.25331851
Apr 2000  0.77996512 -0.02854676
May 2000 -0.08336907 -0.04287046
```

The function `tsp` returns the start and end time, and also the frequency without printing the complete data of the time-series.

```
> tsp(myts2)

[1] 2000.167 2000.333   12.000
```

2.2.6 Lists

A list is like a vector. However, an element of a list can be an object of any type and structure. Consequently, a list can contain another list and therefore it can be used to construct arbitrary data structures. Lists are often used for output of statistical routines in R. The output object is often a collection of parameter estimates, residuals, predicted values etc.

For example, consider the output of the function `lsfit`. In its most simple form the function fits a least square regression.

```
> x <- 1:5
> y <- x + rnorm(5,0,0.25)
> z <- lsfit(x,y)
> z
```

```
$coefficients
Intercept         X
0.3015801 0.9277260

$residuals
[1]  0.1128445 -0.2134749  0.2943595 -0.3996724  0.2059432

$intercept
[1] TRUE

$qr
$qt
[1] -6.89772890  2.93372729  0.21618290 -0.54791914 -0.01237369

$qr
      Intercept          X
[1,] -2.2360680 -6.7082039
[2,]  0.4472136  3.1622777
[3,]  0.4472136 -0.1954395
[4,]  0.4472136 -0.5116673
[5,]  0.4472136 -0.8278950

$qraux
[1] 1.447214 1.120788

$rank
[1] 2

$pivot
[1] 1 2

$tol
[1] 1e-07

attr(,"class")
[1] "qr"
```

In this example the output value of `lsfit(x,y)` is assigned to object 'z'. This is a list whose first component is a vector with the intercept and the slope. The second component is a vector with the model residuals and the third component is a logical vector of length one indicating whether or not an intercept is used. The three components have the names 'coef', 'residuals' and 'intercept'.

The components of a list can be extracted in several ways:

- component number: `z[[1]]` means the first component of `z` (use double square brackets!).
- component name: `z$name` indicates the component of `z` with name `name`.

To identify the component name the first few characters will do, for example, you can use `z$r` instead of `z$residuals`.

```
> test <- z$r
> test

[1]  0.1128445 -0.2134749  0.2943595 -0.3996724  0.2059432

> z$r[4]  # fourth element of the residuals

[1] -0.3996724
```

Creating lists

A list can also be constructed by using the function `list`. The names of the list components and the contents of list components can be specified as arguments of the `list` function by using the = character.

```
> x1 <- 1:5
> x2 <- c(T,T,F,F,T)
> y <- list(numbers=x1, wrong=x2)
> y

$numbers
[1] 1 2 3 4 5

$wrong
[1]  TRUE  TRUE FALSE FALSE  TRUE
```

So the left-hand side of the = operator in the `list` function is the name of the component and the right-hand side is an R object. The order of the arguments in the list function determines the order in the list that is created. In the above example the logical object 'wrong' is the second component of `y`.

```
> y[[2]]

[1]  TRUE  TRUE FALSE FALSE  TRUE
```

The function `names` can be used to extract the names of the list components. It is also used to change the names of list components.

```
> names(y)
```

```
[1] "numbers" "wrong"
```

```
> names(y) <- c("lots", "valid")
> names(y)
```

```
[1] "lots"  "valid"
```

To add extra components to a list proceed as follows:

```
> y[[3]] <- 1:10
> y$test <- "hello"
> y
```

```
$lots
[1] 1 2 3 4 5

$valid
[1]  TRUE  TRUE FALSE FALSE  TRUE

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

$test
[1] "hello"
```

Note the difference in single square brackets and double square brackets.

```
> y[1]
```

```
$lots
[1] 1 2 3 4 5
```

```
> y[[1]]
```

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

When single square brackets are used, the component is returned as list, whereas double square brackets return the component itself.

Transforming objects to a list

Many objects can be transformed to a list with the function `as.list`. For example, vectors, matrices and data frames.

```
> as.list(1:3)

[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 3
```

2.2.7 The `str` function

A handy function is the `str` function, it displays the internal structure of an R object. The function can be used to see a short summary of an object.

```
> x1 <- rnorm(100)
> x2 <- matrix(rnorm(800),ncol=80)
> myl1 <- list(x1,x2,my.df)
> str(x1)

 num [1:100] 0.124 0.216 0.38 -0.502 -0.333 ...

> str(x2)

 num [1:10, 1:80] 0.977 -0.375 1.053 -1.049 -1.26 ...

> str(my.df)

'data.frame':        5 obs. of  2 variables:
 $ Logical: logi  TRUE TRUE TRUE TRUE TRUE
 $ Numeric: num  -1.0857 -0.0854 1.0706 -0.1454 -1.1655

> str(myl1)

List of 3
 $ : num [1:100] 0.124 0.216 0.38 -0.502 -0.333 ...
 $ : num [1:10, 1:80] 0.977 -0.375 1.053 -1.049 -1.26 ...
 $ :'data.frame':        5 obs. of  2 variables:
  ..$ Logical: logi [1:5] TRUE TRUE TRUE TRUE TRUE
  ..$ Numeric: num [1:5] -1.0857 -0.0854 1.0706 -0.1454 -1.1655
```

3 Importing data

One of the first things you want to do in a statistical data analysis system is to import data. R provides a few methods to import data, we will discuss them in this chapter.

3.1 Text files

In R you can import text files with the function `read.table`. This function is a has many arguments. Arguments to specify the header, the column separator, the number of lines to skip, the data types of the columns, etc. The functions `read.csv` and `read.delim` are functions to read 'comma separated values' files and tab delimited files. These functions call `read.table` with specific arguments.

Suppose we have a text file data.txt, that contains the following text:

```
Author: John Davis
Date: 18-05-2007
Some comments..
Col1, Col2, Col3, Col4
23, 45, A, John
34, 41, B, Jimmy
12, 99, B, Patrick
```

The data without the first few lines of text can be imported to an R data frame using the following R syntax:

```
> myfile <- "C:\\PROJECTS\\RIntroductoryCourse\\Data.txt"
> mydf <- read.table(myfile, skip=3, sep=",", header=TRUE)
> mydf
```

```
  Col1 Col2 Col3     Col4
1   23   45    A     John
2   34   41    B    Jimmy
3   12   99    B  Patrick
```

By default R converts character data in text files to factor type. In the above example the third and fourth columns are of type factor. To leave character data as character data type in R, use the `stringsAsFactors` argument.

```
> mydf <- read.table(
+           myfile,
+           skip=3,
+           sep=",",
+           header=TRUE,
+           stringsAsFactors=FALSE)
```

To specify that certain columns are character and other columns are not you must use the `colClasses` argument and provide the type for each column.

```
> mydf <- read.table(
+           myfile, skip=3, sep=",",
+           header=TRUE, stringsAsFactors=FALSE,
+           colClasses = c("numeric", "numeric", "factor", "character"))
```

There is an advantage in using `colClasses`, especially when the data set is large. If you don't use `colClasses` then during a data import, R will store the data as character vectors before deciding what to do with them.

Character strings in a text files may be quoted and may contain the the separator symbol. To import such text files use the `quote` argument. Suppose we have the following comma separated text file that we want to read.

```
Col1, Col2, Col3
12, 45, 'Davis, Joe'
23, 78, 'White, Jimmy'
```

Use the `read.csv` function as follows to import the above text.

```
> myfile <- "C:\\PROJECTS\\RIntroductoryCourse\\Data.csv"
> read.csv(myfile, quote="'")
```

```
  Col1 Col2          Col3
1   12   45    Davis, Joe
2   23   78  White, Jimmy
```

3.1.1 The `scan` function

The `read.table` function uses the more low level function `scan`. This function may also be called directly by the user, and can sometimes be handy when `read.table` cannot do the job. It reads the data into a vector or list, the user can then manipulate this vector or list. For example, if we use `scan` to read the text file above we get:

```
> scan(myfile, what="character", sep=",", strip.white =TRUE)
```

```
[1] "Col1"         "Col2"         "Col3"         "12"           "45"
[6] "Davis, Joe"   "23"           "78"           "White, Jimmy"
```

3.2 Excel files

To get data from Excel just select cells on a Worksheet, copy (to the clipboard) and then in R use the function `read.table` with arguments `file` and `sep` as follow:

```
read.table("clipboard", sep = "\t")
```

To read and write Excel files you can use the package `gdata`. This package provides the function `read.xls`. If the data is in the first sheet and starts at row 1, where the first row represent the column headers, then the call to `read.xls` is simple.

3.3 The Foreign package

When you download R, a number of packages are downloaded as well, among these foreign package reads data stored by Minitab, S, SAS, SPSS, Stata, Systat, dBaseTo and more. Browse foreign help for details.

4 Data Manipulation

The programming language in R provides many different functions and mechanisms to manipulate and extract data. Let's look at some of those for the different data structures.

4.1 Vector subscripts

A part of a vector `x` can be selected by a general subscripting mechanism.

```
x[subscript]
```

The simplest example is to select one particular element of a vector, for example the first one or the last one.

```
> x <- c(6,7,2,4)
> x[1]

[1] 6

> x[length(x)]

[1] 4
```

Moreover, the subscript can have one of the following forms:

A vector of positive natural numbers The elements of the resulting vector are determined by the numbers in the subscript. To extract the first three numbers:

```
> x

[1] 6 7 2 4

> x[1:3]

[1] 6 7 2
```

To get a vector with the fourth, first and again fourth element of `x`:

```
> x[c(4,1,4)]
```

```
[1] 4 6 4
```

One or more elements of a vector can be changed by the subscripting mechanism. To change the third element of a vector proceed as follows:

```
> x[3] <- 4
```

A logical vector The result is a vector with only those elements of `x` of which the logical vector has an element `TRUE`.

```
> x <- c(10,4,6,7,8)
> y <- x >9
> y
```

```
[1]  TRUE FALSE FALSE FALSE FALSE
```

```
> x[y]
```

```
[1] 10
```

or directly

```
> x[x>9]
```

```
[1] 10
```

4.2 Matrix subscripts

As with vectors, parts of matrices can be selected by the subscript mechanism. The general scheme for a matrix `x` is given by:

```
x[subscript]
```

Where subscript has one of the following four forms:

1. A pair `(rows, cols)` where `rows` is a vector representing the row numbers and `cols` is a vector representing column numbers. Rows and/or cols can be empty or negative. The following examples will illustrate the different possibilities.

```
> x <- matrix(1:36, ncol=6)
> ## the element in row 2 and column 6 of x
> x[2,6]
```

```
[1] 32
```

```
> ## the third row of x
> x[3, ]
```

```
[1]  3  9 15 21 27 33
```

```
> ## the element in row 3 and column 1 and
> ## the element in row 3 and column 5
> x[3,c(1,5)]
```

```
[1]  3 27
```

```
> ## show x, except for the first column
> x[,-1]
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    7   13   19   25   31
[2,]    8   14   20   26   32
[3,]    9   15   21   27   33
[4,]   10   16   22   28   34
[5,]   11   17   23   29   35
[6,]   12   18   24   30   36
```

A negative pair results in a so-called minor matrix where a column and row is omitted.

```
> x[-3,-4]
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    7   13   25   31
[2,]    2    8   14   26   32
[3,]    4   10   16   28   34
[4,]    5   11   17   29   35
[5,]    6   12   18   30   36
```

The matrix `x` remains the same, unless you assign the result back to `x`.

```
> x <- x[-3,4]
```

As with vectors, matrix elements or parts of matrices can be changed by using the matrix subscript mechanism and the assignment operator together. To change one element:

```
> x <- matrix(1:36, ncol=6)
> x[4,5] <- 5
```

To change a complete column:

```
> x <- matrix(rnorm(100),ncol=10)
> x[ ,1] <- 1:10
```

4.3 Manipulating Data frames

4.3.1 Extracting data from data frames

A data frame can be considered as a generalized matrix, consequently all subscripting methods that work on matrices also work on data frames. However, data frames offer a few extra possibilities.

```
> names(mtcars)
```

```
 [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
[11] "carb"
```

To select a specific column from a data frame use the $ symbol or double square brackets and quotes:

```
> mpg <- cars$mpg
> mpg <- cars[["mpg"]]
```

The object `mpg` is a vector. If you want the result to be a data frame use single square brackets:

```
> mpg2 <- mtcars["mpg"]
```

When using single brackets it is possible to select more than one column from a data frame. The result is again a data frame:

```
> test <- mtcars[c("mpg","cyl")]
```

To select a specific row by name of the data frame 'mtcars' use the following R code:

```
> mtcars["Volvo 142E", ]
```

```
            mpg cyl disp  hp drat   wt qsec vs am gear carb
Volvo 142E 21.4   4  121 109 4.11 2.78 18.6  1  1    4    2
```

The result is a data frame with one row. To select more rows use a vector of names:

```
> mtcars[c("Volvo 142E", "Fiat X1-9"), ]
```

```
            mpg cyl disp  hp drat    wt qsec vs am gear carb
Volvo 142E 21.4   4  121 109 4.11 2.780 18.6  1  1    4    2
Fiat X1-9  27.3   4   79  66 4.08 1.935 18.9  1  1    4    1
```

If the given row name does not exist, R will return a row with `NA`'s.

```
> mtcars["Lada",]
```

```
   mpg cyl disp hp drat wt qsec vs am gear carb
NA  NA  NA   NA NA   NA NA   NA NA NA   NA   NA
```

Rows from a data frame can also be selected using row numbers. Select cases 10 trough 14 from the cars data frame.

```
> mtcars[10:14,]
```

```
             mpg cyl  disp  hp drat   wt qsec vs am gear carb
Merc 280    19.2   6 167.6 123 3.92 3.44 18.3  1  0    4    4
Merc 280C   17.8   6 167.6 123 3.92 3.44 18.9  1  0    4    4
Merc 450SE  16.4   8 275.8 180 3.07 4.07 17.4  0  0    3    3
Merc 450SL  17.3   8 275.8 180 3.07 3.73 17.6  0  0    3    3
Merc 450SLC 15.2   8 275.8 180 3.07 3.78 18.0  0  0    3    3
```

The first few rows or the last few rows can be extracted by using the functions `head` or `tail`.

```
> head(mtcars,3)
```

```
               mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4     21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710    22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
```

```
> tail(mtcars,2)
```

```
               mpg cyl disp  hp drat   wt qsec vs am gear carb
Maserati Bora 15.0   8  301 335 3.54 3.57 14.6  0  1    5    8
Volvo 142E    21.4   4  121 109 4.11 2.78 18.6  1  1    4    2
```

To subset specific cases from a data frame you can also use a logical vector. When you provide a logical vector in a data frame subscript, only the cases which correspond with a `TRUE` are selected. Suppose you want to get all cars from the cars data frame that have a weight of over 3500. First create a logical vector `tmp`:

```
> tmp <- mtcars$cyl > 6
```

Use this vector to subset:

```
> mtcars[tmp, ]
```

```
                     mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
```

```
Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
```

A handy alternative is the function `subset`. It returns a the subset as a data frame. The first argument is the data frame. The second argument is a logical expression. In this expression you use the variable names without proceeding them with the name of the data frame, as in the above example.

```
> subset(mtcars, cyl > 6 & hp > 250)
```

```
                mpg cyl disp  hp drat   wt qsec vs am gear carb
Ford Pantera L 15.8   8  351 264 4.22 3.17 14.5  0  1    5    4
Maserati Bora  15.0   8  301 335 3.54 3.57 14.6  0  1    5    8
```

5 Statistics

The base installation of R contains many functions for calculating statistical summaries, data analysis and statistical modeling. Even more functions are available in all the R packages on CRAN. In this section we will discuss only some of these functions. For a more comprehensive overview of the statistical possibilities see for example [4] and [5].

5.1 Basic statistical functions

5.1.1 Statistical summaries and tests

A number of functions return statistical summaries and tests. The following table contains a list of only some of the statistical functions in R. The names of the functions usually speak for themselves.

Function	**purpose**
`acf(x, plot=F)`	auto or partial correlation coefficients
`chisq.test(x)`	chi squared goodness of fit test
`cor(x,y)`	correlation coefficient
`ks.test(z)`	Kolmogorov-Smirnov goodness of fit test
`mad(x)`	median absolute deviation
`mean(x)`	mean
`mean(x, trim=a)`	trimmed mean
`median(x)`	median
`quantile(x, probs)`	sample quantile at given probabilities
`range(x)`	the range, i.e. the vector c(min(x), max(x))
`stem(x)`	stem-and-leaf-plot
`t.test(x,...)`	One or two sample Student's t-test
`var(x)`	variance of x or covariance matrix of x
`var(x,y)`	covariance
`var.test(x,y)`	test on variance equality of x and y

Table 5.1: Some functions that calculate statistical summaries.

The remainder of this sub section will give some examples of the above functions.

quantiles

The `quantile` function needs two vectors as input. The first one contains the observations, the second one contains the probabilities corresponding to the quantiles. The function returns the empirical quantiles of the first data vector. To calculate the 5 and 10 percent quantile of a sample from a N(0,1) distribution, proceed as follows:

```
> x <- rnorm(100)
> xq <- quantile(x,c(0.05, 0.5, 1))
> xq
```

```
        5%        50%       100%
-1.4434225 -0.0535039  3.1840445
```

The function returns a vector with the quantiles as named elements.

summary

The function `summary` is convenient for calculating basic statistics of columns of a data frame.

```
> summary(mtcars)
```

```
      mpg             cyl             disp             hp
 Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0
 1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5
 Median :19.20   Median :6.000   Median :196.3   Median :123.0
 Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7
 3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0
 Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0
      drat             wt             qsec             vs
 Min.   :2.760   Min.   :1.513   Min.   :14.50   Min.   :0.0000
 1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89   1st Qu.:0.0000
 Median :3.695   Median :3.325   Median :17.71   Median :0.0000
 Mean   :3.597   Mean   :3.217   Mean   :17.85   Mean   :0.4375
 3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90   3rd Qu.:1.0000
 Max.   :4.930   Max.   :5.424   Max.   :22.90   Max.   :1.0000
       am              gear            carb
 Min.   :0.0000   Min.   :3.000   Min.   :1.000
 1st Qu.:0.0000   1st Qu.:3.000   1st Qu.:2.000
 Median :0.0000   Median :4.000   Median :2.000
 Mean   :0.4062   Mean   :3.688   Mean   :2.812
 3rd Qu.:1.0000   3rd Qu.:4.000   3rd Qu.:4.000
 Max.   :1.0000   Max.   :5.000   Max.   :8.000
```

5.2 Regression models

5.2.1 Linear regression models

R can fit linear regression models of the form

$$y = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p + \epsilon$$

where $\beta = (\beta_0, \cdots, \beta_p)$ are the intercept and p regression coefficients and $x_1, \cdots, x_p$ the p regression variables. The error term ϵ has mean zero and is often modeled as a normal distribution with some variance.

For two regression variables you can use the function `lm` with the following formula

```
y ~ x1 + x2
```

By default R includes the intercept of the linear regression model. To omit the intercept use the formula:

```
y ~ -1 + x1 + x2
```

Be aware of the special meaning of the operators `*`, `-`, `^`, `\` and `:` in linear model formulae. They are not used for the normal multiplication, subtraction, power and division.

The `:` operator is used to model interaction terms in linear models.

```
> fit <- lm(100/mpg ~ hp + wt, data=mtcars)
> fit

Call:
lm(formula = 100/mpg ~ hp + wt, data = mtcars)

Coefficients:
(Intercept)           hp           wt
    0.63051      0.00748      1.14853
```

The function `summary` is convenient for calculating basic statistics on lm fit.

```
> summary(fit)
```

```
Call:
lm(formula = 100/mpg ~ hp + wt, data = mtcars)

Residuals:
     Min       1Q   Median       3Q      Max
-1.68690 -0.49601  0.07663  0.40027  1.42186

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 0.630513   0.409363   1.540  0.13435
hp          0.007479   0.002312   3.235  0.00303 **
wt          1.148525   0.162009   7.089 8.45e-08 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.664 on 29 degrees of freedom
Multiple R-squared: 0.8471,        Adjusted R-squared: 0.8365
F-statistic: 80.33 on 2 and 29 DF,  p-value: 1.494e-12
```

The function `plot` plot basic diagnostic statistics on an lm fit.

```
> plot(fit,  which = 2)
```

Bibliography

[1] L. Longhow, *An Introduction to RâĂİ*. 2010.

[2] Longhow Lam, *A guide to Eclipse and the R plug-in StatET*. www.splusbook.com, 2007.

[3] Diethelm Würtz, "S4 'timedate' and 'timeseries' classes for R," *Journal of Statistical Software*.

[4] W. N. Venables and B. D. Ripley, *Modern Applied Statistics with S*. Springer, September 2003.

[5] J. Maindonald and J. Braun, *Data Analysis and Graphics Using R: An Example-based Approach*. Cambridge University Press, 2007.

Index