Department of Methodology

Workshop in Applied Analysis Software MY591

Introduction to R

Instructor
Mai Hafez
Contact: m.m.hafez@lse.ac.uk

Course Convenor (MY591)
Dr. Aude Bicquelet (LSE, Department of Methodology)
Contact: A.J.Bicquelet@lse.ac.uk

1 Purpose and Outline of the course

The MY591 course consists of a number of introductory training courses on computer packages for conducting qualitative or quantitative analysis. This class is an introduction to R. R is a language and environment for statistical computing and graphics. R provides a wide variety of statistical (linear and nonlinear modelling, classical statistical tests, time-series analysis, classification, clustering, ...) and graphical techniques, and is highly extensible. It provides an Open Source route to participation in research in statistical methodology.

This class aims at getting you introduced to R and familiarising with its environment so you can start using it for your own research. You will have the opportunity to get hands-on experience. Although some statistical analysis will be carried out using R during this session, however it is not within the scope of this class to explain those statistical concepts. Our main concern is how to implement those techniques within R.

In this session we will show you how to get started with R, how to manage your data and produce some descriptive statistics and graphics. In a more advanced session, some more advanced statistical analysis techniques are covered.

2 What is R?

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.

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 software is freely available under the GNU General Public License. The R project homepage is http://www.r-project.org/. You can download the software to install on your own computer from http://www.stats.bris.ac.uk/R/.

3 The R help system

There are a number of different ways of getting help in R.

- If you have a query about a specific function; typing `?` and then the function's name at the prompt will bring up the relevant help page.

```
> ?mean
> ?setwd
> ?t.test
```

- If your problem is of a more general nature, then typing `help.start()` will open up a window which allows you to browse for the information you want. The search engine on this page is very helpful for finding context specific information.

There are many books on R and new ones are coming out all the time. The following are two of those:

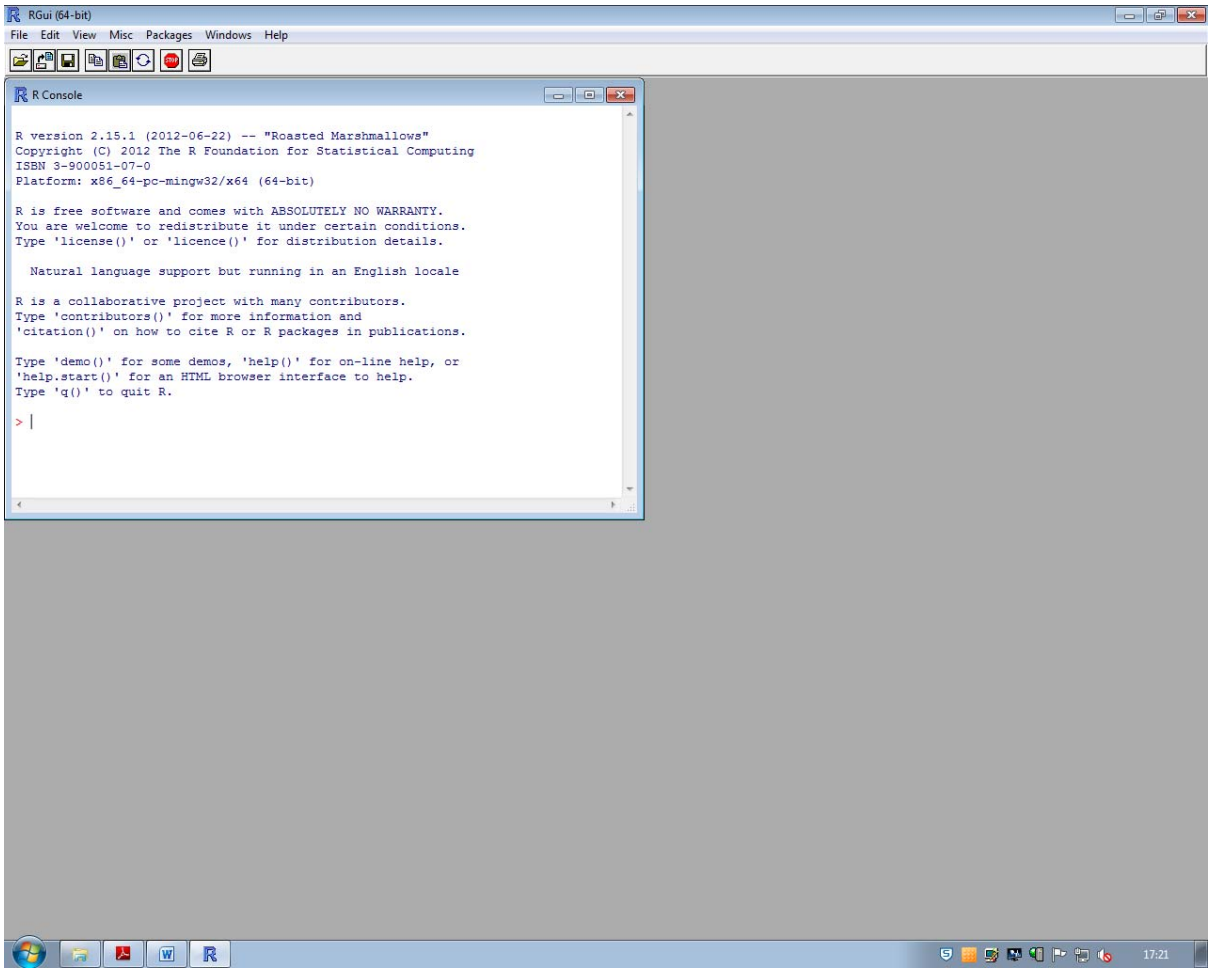
- Venable, W. N. and Ripley, B. D. (2002) Modern Applied Statistics with S, (Fourth Edition), Springer. (QA276.4 V44)
- Venables, W. N., Smith, D.M. and the R Core Development Team (2001) An Introduction to R, freely available from http://cran.r-project.org/doc/manuals/R-intro.pdf.

4 Getting started with R

To get started with R, you have to go through the following steps:

- Create a folder (working directory)
  In **My computer** use `File →New →Folder` to create a new folder in your H: space and give your new folder the name `MY591R`.
- Get data files from Moodle
  In Moodle, go to `My Courses →MY591 →R`. Select the data files using right-click → `Save target as`, then save to the MY591R folder that you have just created in your H: space.
- Start R
  Go to `Start → All Programs →Specialist and Teaching Software → Statistics → R → R2.15` to start the R software package. You will see a window that looks like the one displayed below.
- Change the working directory to the one you have just created

```
> setwd("H:/MY591R")
```



4.1 Objects

R works on *objects*. All objects have the following properties:

- `mode`: tells us what kind of thing the object is - possible modes include `numeric, complex, logical, character` and `list.`
- `length`: is the number of components that make up the object.

At the simplest level, an object is a convenient way to store information. In Statistics, we need to store observations of a variable of interest. This is done using a numeric vector. Note that there are no scalars in R; a number is just a numeric vector of length 1.
If an object stores information, we need to name it so that we can refer to it later (and thus recover the information that it contains). The term used for the name of an object is *identifier*.
An identifier is something that we choose. Identifiers can be chosen fairly freely in R. The points below are a few simple rules to bear in mind.

- In general any combination of letters, digits and the dot character can be used although it is obviously sensible to choose names that are reasonably descriptive.
- You cannot start an identifier with a digit or a dot so `moonbase3.sample` is acceptable but `3moons.samplebase` and `.sample3basemoon` are not.
- Identifiers are CASE SENSITIVE so `moon.sample` is different from `moon.Sample`. It is easy to get caught out by this.
- Some characters are already assigned values. These include `c, q, t, C, D, F, I` and `T.` Avoid using these as identifiers.

Typically we are interested in data sets that consist of several variables. In R, data sets are represented by an object known as a *data frame*. As with all objects, a data frame has the intrinsic attributes mode and length; data frames are of mode `list` and the `length` of a data frame is the number of variables that is contains. In common with many larger objects, a data frame has other attributes in addition to mode and length. These are:

- `names`: these are the names of the variables that make up the data set,
- `row.names`: these are the names of the individuals on whom the observations are made,
- `class`: this attribute can be thought of as a detailed specification of the kind of thing the object is; in this case the class is `"data.frame".`

The class attribute tells certain functions (generic functions) how to deal with the object. For example, objects of class `"data.frame"` are displayed on screen in a particular way.

4.2 Workspace and working directories

During an R session, a number of objects will be generated; for example we may generate vectors, data frames and functions. For the duration of the session, these objects are stored in an area of memory referred to as the *workspace*. If we want to save the objects for future use, we instruct R to write them to a file in our current *working directory* (directory is just another name for a folder). Note the distinction: things in memory are temporary (they will be lost when we log out); files are more permanent (they are stored on disk and the information they contain can be loaded into memory during our next session). Managing objects and files is an important part of using R effectively.

5 Using R as a calculator

The simplest thing that R can do is to evaluate arithmetic expressions:

```
> 1
[1] 1
> 1+4.23
[1] 5.23
> 1+1/2*9-3.14
[1] 2.36
# Note the order in which operations are performed
# in the final calculation
```

Comments in R

R ignores anything after a # sign in a command. We will follow this convention. Anything after a # in a set of R commands is a comment.

6 Vectors and assignment

We can create vectors at the command prompt using the concatenation function `c(...)`.

```
c(object1,object2,...)
```

This function takes arguments of the same mode and returns a vector containing these values.

```
> c(1,2,3)
[1] 1 2 3
> c("Ali","Bet","Cat")
[1] "Ali" "Bet" "Cat"
```

In order to make use of vectors, we need identifiers for them (we do not want to have to write vectors from scratch every time we use them). This is done using the assignment operator `<-` .

```
name <- expression
```

name now refers to an object whose value is the result of evaluating expression.

```
> numbers <- c(1,2,3)
> people <- c("Ali","Bet","Cat")
```

```
> numbers
[1] 1 2 3
> people
[1] "Ali" "Bet" "Cat"
# Typing an object's identifier causes R
# to print the contents of the object
```

Simple arithmetic operations can be performed with vectors:

```
> c(1,2,3)+c(4,5,6)
[1] 5 7 9
> numbers + numbers
[1] 2 4 6
> numbers - c(8,7.5,-2)
[1] -7.0 -5.5 5.0
> c(1,2,4)*c(1,3,3)
[1] 1 6 12
> c(12,12,12)/numbers
[1] 12 6 4
```

Note in the above example that multiplication and division are done element by element.

Reusing commands

If you want to bring back a command which you have used earlier in the session, press the up arrow key ". This allows you to go back through the commands until you find the one you want. The commands reappear at the command line and can be edited and then run by pressing return.

The outcome of an arithmetic calculation can be given an identifier for later use:

```
> calc1 <- numbers + c(8,7.5,-2)
> calc2 <- calc1 * calc1
> calc1
[1] 9.0 9.5 1.0
> calc2
[1] 81.00 90.25 1.00
> calc1 <- calc1 + calc2
> calc1
[1] 90.00 99.75 2.00
> calc2
[1] 81.00 90.25 1.00
# Note: in the final step we have updated the value of calc1
```

```
# by adding calc2 to the old value; calc1 changes but calc2
is unchanged
```

If we try to add together vectors of different lengths, R uses a recycling rule; the smaller vector is repeated until the dimensions match.

```
> small <- c(1,2)
> large <- c(0,0,0,0,0,0)
> large + small
[1] 1 2 1 2 1 2
```

If the dimension of the larger vector is not a multiple of the dimension of the smaller vector, a warning message will be given. The concatenation function can be used to concatenate vectors.

```
> c(small,large,small)
[1] 1 2 0 0 0 0 0 0 1 2
```

We have now created a number of objects. To ensure clarity in the following examples we need to remove all of the objects we have created.

```
> rm(list=objects())
```

We want to work with data sets. In general we have multiple observations for each variable. Vectors provide a convenient way to store observations.

7 Simple Statistical Functions

Example - sheep weight

We have taken a random sample of the weight of 5 sheep in the UK. The weights (kg) are

```
84.5 72.6 75.7 94.8 71.3
```

We are going to put these values in a vector and illustrate some standard procedures:

```
> weight <- c(84.5, 72.6, 75.7, 94.8, 71.3)
> weight
[1] 84.5 72.6 75.7 94.8 71.3
> total <- sum(weight)
> numobs <- length(weight)
> meanweight <- total/numobs
> meanweight
[1] 79.78
```

```
# We have worked out the mean the hard way. There is a quick
way ...

> mean(weight)
[1] 79.78
```

You can try other simple statistical functions. Most functions to generate descriptive statistics are reasonably obvious:

```
> median(weight)

> range(weight)

> sd(weight) standard deviation

> mad(weight) mean absolute deviation

> IQR(weight) inter-quartile range

> min(weight) minimum

> max(weight) maximum
```

8 Data frames

A data frame is an R object that can be thought of as representing a data set. A data frame consists of variables (columns vectors) of the same length with each row corresponding to an experimental unit. The general syntax for setting up a data frame is

```
name <- data.frame(variable1, variable2, ...)
```

Individual variables in a data frame are accessed using the $ notation:

```
name $variable
```

Once a data frame has been created we can view and edit it in a spreadsheet format using the command `fix(...)`. New variables can be added to an existing data frame by assignment.

Example – sheep again

Suppose that, for each of the sheep weighed in the example above, we also measure the height at the shoulder. The heights (cm) are

```
86.5 71.8 77.2 84.9 75.4
```

We will set up another variable for height. We would also like to have a single structure in which the association between weight and height (that is, that they are two measurements of

the same sheep) is made explicit. This is done by adding each variable to a dataframe. We will call the data frame `sheep` and view it using `fix(sheep).`

```
> height <- c(86.5, 71.8, 77.2, 84.9, 75.4)
> sheep <- data.frame(weight, height)
> mean(sheep$height)
[1] 79.16
> fix(sheep)
# the spreadsheet window must be closed before we can continue
```

Suppose that a third variable consisting of measurements of the length of the sheep's backs becomes available. The values (in cm) are

```
130.4 100.2 109.4 140.6 101.4
```

We can include a new variable in the data frame using assignment. Suppose we choose the identifier `backlength` for this new variable:

```
> sheep$backlength <- c(130.4, 100.2, 109.4, 140.6, 101.4)
```

Look at the data in spreadsheet format to check what has happened.

9 Descriptive analysis

A set of descriptive statistics is produced by the function `summary(...).` The argument can be an individual variable or a data frame. The output is a table.

```
> summary(sheep$weight)
Min. 1st Qu. Median Mean 3rd Qu. Max.
71.30 72.60 75.70 79.78 84.50 94.80

> summary(sheep)
Weight                height                backlength
Min.    : 71.30   Min.      : 71.80   Min.      : 100.2
1st Qu. : 72.60   1st Qu.   : 75.40   1st Qu.   : 101.4
Median  : 75.70   Median    : 77.20   Median    : 109.4
Mean    : 79.78   Mean      : 79.16   Mean      : 116.4
3rd Qu. : 84.50   3rd Qu.   : 84.90   3rd Qu.   : 130.4
Max.    : 94.80   Max.      : 86.50   Max.      : 140.6
> IQR(sheep$height)
[1] 9.5
> sd(sheep$backlength)
[1] 18.15269
```

10 Session management and visibility

All of the objects created during an R session are stored in a *workspace* in memory. We can see the objects that are currently in the workspace by using the command `objects()`. Notice the `()`, these are vital for the command to work.

```
> objects()
[1] "height" "meanweight" "numobs" "sheep" "total"
[6] "weight"
```

The information in the variables height and weight is now *encapsulated* in the data frame sheep. We can tidy up our workspace by removing the height and weight variables (and various others that we are no longer interested in) using the `rm(...)` function. Do this and then check what is left.

```
> rm(height,weight,meanweight,numobs,total)
> objects()
[1] "sheep"
```

The `height` and `weight` variables are now only accessible via the sheep data frame.

```
> weight
Error: Object "weight" not found
> sheep$weight
[1] 84.5 72.6 75.7 94.8 71.3
```

The advantage of this encapsulation of information is that we could now have another data frame, say `dog`, with `height` and `weight` variables without any ambiguity. However, the `$` notation can be a bit cumbersome. If we are going to be using the variables in the `sheep` data frame a lot, we can make them visible from the command line by using the `attach(...)` command. When we have finished using the data frame, it is good practice to use the `detach()` command (notice the empty `()` again) so the encapsulated variables are no longer visible.

```
> weight
Error: Object "weight" not found
> attach(sheep)
> weight
[1] 84.5 72.6 75.7 94.8 71.3
> detach()
> weight
Error: Object "weight" not found
```

11 Importing data

In practice, five observations would not provide a very good basis for inference about the entire UK sheep population. Usually we will want to import information from large (potentially very large) data sets that are in an electronic form. We will use data that are in a plain text format.
The variables are in columns with the first row of the data giving the variable names. The file sheep.dat contains weight and height measurements from 100 randomly selected UK sheep.
We are going to copy this file to our working directory `MY591R`. The information is read into R using the `read.table(...)` function. This function returns a data frame.

- Make sure you have the data file in your working directory `H:MY591R`
- Read data into R
  `> sheep2 <- read.table("sheep.dat", header=TRUE)`

Using `header = TRUE` gives us a data frame in which the variable names in the first row of the data file are used as identifiers for the columns of our data set. If we exclude the `header=TRUE`, the first line will be treated as a line of data. In order to view or amend your data use

```
> fix(sheep2)
```

12 R Essentials

12.1 Regular sequences

A regular sequence is a sequence of numbers or characters that follow a fixed pattern. These are useful for selecting portions of a vector and in generating values for categorical variables. We can use a number of different methods for generating sequences. First we investigate the `:` sequence generator:

```
> 1:10
[1] 1 2 3 4 5 6 7 8 9 10
> 10:1
[1] 10 9 8 7 6 5 4 3 2 1
> 2*1:10
[1] 2 4 6 8 10 12 14 16 18 20
> 1:10 + 1:20
[1] 2 4 6 8 10 12 14 16 18 20 12 14 16 18 20 22 24 26 28 30
> 1:10-1
[1] 0 1 2 3 4 5 6 7 8 9
> 1:(10-1)
[1] 1 2 3 4 5 6 7 8 9

# Notice that : takes precedence over arithmetic
operations.
```

The seq function allows a greater degree of sophistication in generating sequences. The function definition is

```
seq(from, to, by, length, along)
```

The arguments `from` and `to` are self-explanatory. `by` gives the increment for the sequence and `length` the number of entries that we want to appear. Notice that if we include all of these arguments there will be some redundancy and an error message will be given. `along` allows a vector to be specified whose length is the desired length of our sequence. Notice how the named arguments are used below. By playing around with the command, see whether you can work out what the default values for the arguments are.

```
> seq(1,10)
[1] 1 2 3 4 5 6 7 8 9 10
> seq(to=10, from=1)
[1] 1 2 3 4 5 6 7 8 9 10
> seq(1,10,by=0.5)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0 7.5
8.0
[16] 8.5 9.0 9.5 10.0
> seq(1,10,length=19)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0 7.5
8.0
[16] 8.5 9.0 9.5 10.0
> seq(1,10,length=19,by=0.25)
Error in seq.default(1, 10, length = 19, by = 0.25) :
Too many arguments
> seq(1,by=2,length=6)
[1] 1 3 5 7 9 11
> seq(to=30,length=13)
[1] 18 19 20 21 22 23 24 25 26 27 28 29 30
> seq(to=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] 26 27 28 29 30
```

Finally you can use the function `rep`. Find out about rep using the R help system then experiment with it.

```
> ?rep
> rep(1, times = 3)
[1] 1 1 1
> rep((1:3), each =5)
[1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
```

12.2 Indexing vectors and subset selection

We often only want to use a portion of the data or a subset whose members satisfy a particular criteria. The notation for referring to a part of a vector is the square brackets `[]`.

This is known as the *subscripting operator*. The content of the brackets is a vector, referring to as an *indexing vector*. If the indexing vector is integer valued (often a regular sequence), the elements with indices corresponding to the values of the indexing vector are selected. If there is a minus sign in front then all elements except those indexed by the sequence will be selected.

Now create a data frame and give it the name `marks.msc` by reading in the data from the file `marks.dat` into R. This is a toy data set with marks for six MSc students on three courses.

```
> marks.msc<-read.table("marks.dat", header=TRUE)
> names(marks.msc)
[1] "courseA" "courseB" "courseC"
> names(marks.msc)<-c("Maths","Statistics","English")
# changing the names of the courses
> marks.msc$Maths
[1] 52 71 44 90 23 66
> attach(marks.msc)
> Maths[1]
[1] 52
> Maths[c(1,2,6)]
[1] 52 71 66
> Maths[1:4]
[1] 52 71 44 90
> Maths[-(1:4)]
[1] 23 66
> Maths[seq(6,by=-2)]
[1] 66 90 71
> Maths[rep((1:3),each=2)]
[1] 52 52 71 71 44 44
> row.names(marks.msc)
[1] "1" "2" "3" "4" "5" "6"
> row.names(marks.msc)<-c("Ali", "Bet", "Cat", "Dan",
"Eli", "Foo")
> row.names(marks.msc)[1:3]
[1] "Ali" "Bet" "Cal"
```

If the contents of the `[]` brackets is a logical vector of the same length as our original, the result is just those elements for which the value of the logical vector is true. We can now work things like:

- the Statistics marks which were higher than `70`,
- the Maths marks for people who got less than `65` in Statistics,
- the English marks for people whose total marks are less than `200`.

These are implemented below. Try to work out an English statement for what the fourth command is doing (note 50 is the pass mark for these exams).

```
> Statistics[Statistics>70]
[1] 82 78
> Maths[Statistics<65]
[1] 44 23 66
> English[(Maths+Statistics+English)<200]
[1] 71 55 52 61
> English[ST402>50 & ST419>50 & ST422>50]
[1] 71 84 68 61
```

In practice we may be more interested in the names of the students with Statistics marks over 70 rather than the marks themselves. Try to work out what the last two of the commands below are doing:

```
> row.names(marks.msc)[Statistics>70]
[1] "Bet" "Dan"
> row.names(marks.msc)[Maths<50 | Statistics<50 |
English<50]
[1] "Cal" "Eli"
>      names(marks.msc)[c(sum(Maths),      sum(Statistics),
sum(English)) > 350]
[1] "Statistics" "English"
> detach()
```

13 Quitting and returning to a saved workspace

Before ending a session it is advisable to remove any unwanted objects and then save the workspace in the current working directory. The command to save a workspace is `save.image(...)`. If we do not specify a file name the workspace will be saved in a fie called `.Rdata`. It is usually a good idea to use something a little more informative than this default

- Save current workspace

```
> save.image("introduction.Rdata")
> quit()
```

On quitting you will be given the option of saving the current workspace. If you say Yes the current workspace will be saved to the default file .Rdata (we have already saved the workspace, there is no need to save it again).

In order to return to the workspace we have just saved. Restart R, set the appropriate working directory and then use the `load(...)` command to bring back the saved objects.

```
> setwd("H:/MY591R")
> load("introduction.Rdata")
> objects()
```

```
# you should have all the objects from the previous
session
```

- List file in current working directory
  `> dir()`

The `dir()` command gives a listing of the files in the current working directory. You will see that your MY591 directory now contains a file `introduction.Rdata` along with your data files. This file contains the objects from our current session, that is, the sheep data frame. It is important to be clear about the distinction between objects (which are contained in a workspace, listed by `objects()`) and the workspace (which can be stored in a file, contained in a directory, listed by `dir()`).

14 Graphics in R

Graphics form an important part of any descriptive analysis of a data set; a histogram provides a visual impression of the distribution of the data and comparison with specific probability distributions, such as the normal, are possible using a quantile-quantile (qq) plot. The distribution of several variables can be compared using parallel boxplots and relationships investigated using scatter plots. Some plots are specific to a type of data; for example, in time series analysis, time series plots and correlograms (plots that are indicative of serial correlation) are commonly used. Graphical methods also play a role in model building and the analysis of output from a fitting process. In particular, diagnostic plots are used to determine whether a model is an adequate representation of the data.

R has powerful and flexible graphics capabilities. This section provides a flavour of the sorts of things that are possible rather than a comprehensive treatment.

One of the simplest ways to get a feel for the distribution of data is to generate a histogram. This is done using the `hist(...)` command. By setting the value of arguments of `hist(...)` we can alter the appearance of the histogram; setting `probability = TRUE` will give relative frequencies, `nclass` allows us to suggest the number of classes to use and breaks allows the precise break points in the histogram to be specified.

Now create another data frame and give it the name mk2nd by reading in the data from the file `marks2.dat` into R. This is a data set with marks (out of 40) for three difficult exams for a second year undergraduate group.

```
> mk2nd<-read.table("marks2.dat", header=TRUE)
> fix(mk2nd)
> attach(mk2nd)
> hist(exam1)
> hist(exam1, probability = TRUE)
> hist(exam1, nclass=10)
```

```
> hist(exam1, breaks=c(0,20,25,30,40))
```

`plot(exam1,exam2)` will plot exam2 against exam1 (that is, exam1 on the x-axis and exam2 on the y-axis) while `plot(exam2,exam1)` will plot exam1 against exam2. The position of the argument in the call tells R what to do with it.

```
> plot(exam1, exam2)
> plot(exam2, exam1)
```

An alternative, that is available in R and many other languages, is to use named arguments; for example, the arguments of the plot command are x and y. If we name arguments, this takes precedence over the ordering so `plot(y = exam2, x = exam1)` has exactly the same effect as `plot(x = exam1, y = exam2)` (which is also the same as `plot(exam1,exam2)`).

```
> plot(y=exam2, x=exam1)
> plot(x=exam1, y=exam2)
```

It is often useful to compare a data set to the normal distribution. The `qqnorm(...)` command plots the sample quantiles against the quantiles from a normal distribution. A `qqline(...)` command after `qqnorm(...)` will draw a straight line through the coordinates corresponding to the first and third quartiles. We would expect a sample from a normal to yield points on the qq-plot that are close to this line.

```
> qqnorm(exam2)
> qqline(exam2)
```

Boxplots provide another mechanism for getting a feel for the distribution of data. Parallel boxplots are useful for comparison. The full name is a box-and-whisker plot. The box is made up by connecting three horizontal lines: the lower quartile, median and upper quartile. In the default set up, the whiskers extend to any data points that are within 1.5 times the inter quartile range of the edge of the box.

```
> boxplot(exam1,exam2,exam3)
# The labels here are not very informative

> boxplot(mk2nd)
# Using the data frame as an argument gives a better result

> boxplot(mk2nd, main="Boxplot of exam scores",
ylab="Scores")
# A version with a title and proper y-axis label
```

R has a number of interactive graphics capabilities. One of the most useful is the `identify(...)` command. This allows us to label interesting points on the plot. After an `identify(...)` command, R will wait while the users selects points on the plot using the mouse. The process is stopped using the right mouse button.

```
> plot(exam1,exam2)
> identify(exam1,exam2)
> identify(exam1,exam2,row.names(mk2nd))
# Note the default marks are the position (row number) of
the point in the data frame. Using row names may be more
informative.
```

R allows you to put more than one plot on the page by setting the `mfrow` parameter. The value that `mfrow` is set to is an integer vector of length 2 giving the number of rows and the number of columns.

```
> par(mfrow=c(3,2))
> hist(exam1)
> qqnorm(exam1)
> hist(exam2)
> qqnorm(exam2)
> hist(exam3)
> qqnorm(exam3)
> par(mfrow=c(1,1))
```

R also allows you to change the tick marks and labels, the borders around plots and the space allocated for titles - more can be found in Venables's *et. al*.

15 A hypothesis test

Back to the sheep example

Common wisdom states that the population mean sheep weight is 80kg. The data from 100 randomly selected sheep may be used to test this. This data can be found in the data frame "`sheep2`" that we have already created previously (see section 11). We formulate a hypothesis test in which the null hypothesis is population mean, μ, of UK sheep is 80kg and the alternative is that the population mean takes a different value:

$$H0 : \mu = 80;$$

$$H1 : \mu \neq 80.$$

We set significance level of 5%, that is $\alpha = 0.05$. Assuming that sheep weight is normally distributed with unknown variance, the appropriate test is a t-test (two-tailed). We can use the function `t.test(...)` to perform this test.

```
> attach(sheep2) # to make variables accessible
> t.test(weight, mu=80)
One Sample t-test
data: weight
t = 2.1486, df = 99, p-value = 0.03411
alternative hypothesis: true mean is not equal to 80
```

```
95 percent confidence interval:
80.21048 85.29312
sample estimates:
mean of x
82.7518
```

Notice the first argument of the t-test function is the variable that we want to test. The other arguments are optional. The argument mu is used to set the value of the mean that we would like to test (the default is zero). The output includes the sample value of our test statistic t = 2.1486 and the associated p-value 0.03411. For this example, $p < 0.05$ so we reject H0 and conclude that there is evidence to suggest the mean weight of UK sheep is not 80kg. What conclusion would we have come to if the significance level had been 1%?

We can use the alternative argument to do one-tailed tests. For each of the following, write down the hypotheses that are being tested and the conclusion of the test:

```
> t.test(weight, mu=80, alternative="greater")
> t.test(height, mu=66, alternative="less")
```

**You can use the exam marks data set mk2nd to test whether the population mean for exam1 is equal to, less than or greater than 30. Use `?t.test` to find out about paired arguments and test the hypothesis that the population mean marks for `exam1` and `exam2` are identical.

```
> attach(mk2nd)
> t.test(x=exam1, y=exam2, paired=TRUE)
```

16 A linear model

The weight of sheep is of interest to farmers. However, weighing the sheep is time consuming and emotionally draining (the sheep do not like getting on the scales). Measuring sheep height is much easier. It would be enormously advantageous for the farmer to have a simple mechanism to approximate a sheep's weight from a measurement of its height. One obvious way to do this is to fit a simple linear regression model with height as the explanatory variable and weight as the response.
The plausibility of a linear model can be investigated with a simple scatter plot. The R command `plot(...)` is very versatile; here we use it in one of its simplest forms.

```
> plot(height,weight)
```

Notice that the x-axis variable is given first in this type of plot command. To fit linear models we use the R function `lm(...)`. Once again this is very flexible but is used here in a simple form to fit a simple linear regression of weight on height.

```
> reg.simple <- lm(weight~ height)
```

You will notice two things:

- The strange argument `weight~ height`: this is a *model formula*. The ~ means "described by". So the command here is asking for a linear model in which weight is described by height.

- Nothing happens: no output from our model is printed to the screen. This is because R works by putting all of the information into the object returned by the `lm(...)` function. This is known as a *model object*. In this instance we are storing the information in a model object called `reg.simple` which we can then interrogate using *extractor functions*.

The simplest way to extract information is just to type the identifier of a model object (in this case we have chosen the identifier `reg.simple`). We can also use the `summary(...)` function to provide more detailed information or `abline(...)` to generate a fitted line plot. For each of the commands below make a note of the output.

```
> reg.simple
> summary(reg.simple)
> abline(reg.simple)
```

From the output of these commands write down the slope and intercept estimates. Does height influence weight? What weight would you predict for a sheep with height 56cm?

17 Writing your own functions

We have seen in the previous sections that there are a large number of useful function built into R; these include `mean(...)`, `plot(...)` and `lm(...)`. Explicitly telling the computer to add up all of the values in a vector and then divide by the length every time we wanted to calculate the mean would be extremely tiresome. Fortunately, R provides the `mean(...)` function so we do not have to do long winded calculations. One of the most powerful features of R is that the user can write their own functions. This allows complicated procedures to be built with relative ease.
The general syntax for defining a function is

```
name <- function(arg1, arg2, ...) expr1
```

The function is called by using

```
name(...)
```

When the function is called the statements that make up *`expr1`* are executed. The final line of *`expr1`* gives the return value. Writing functions is not discussed in detail here.

More advanced features of R include more on linear models, distributions, simulations, among many other areas in Statistics.