The London School of Economics and Political Science

Department of Methodology

Workshop in Applied Analysis Software

MY591

Advanced R Workshop

**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

Academic year

**2012-2013**

# 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 session 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 session is designed for people who have been introduced to R before. It builds on an introductory workshop on R that is also offered within MY591. However, you do not need to be an expert. The first part of this session will be dedicated for refreshing your memory about the basics you need almost for every R session. Once you have familiarised yourself with the R environment, the rest of the class aims at getting you introduced to more advanced features and topics in R so you can start using it for your own research. You will have the opportunity to get hands-on experience. Although some relatively advanced 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 provide a quick review on R essentials then show you how to use R in a number of advanced statistical analysis techniques including writing functions, simulation, and linear models.

©J Penzer 2006: this hand-out is based on notes originally created by J Penzer for a course on Computational Statistics at LSE.

# 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/.

# 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 Edi-

tion), 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>.

# 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

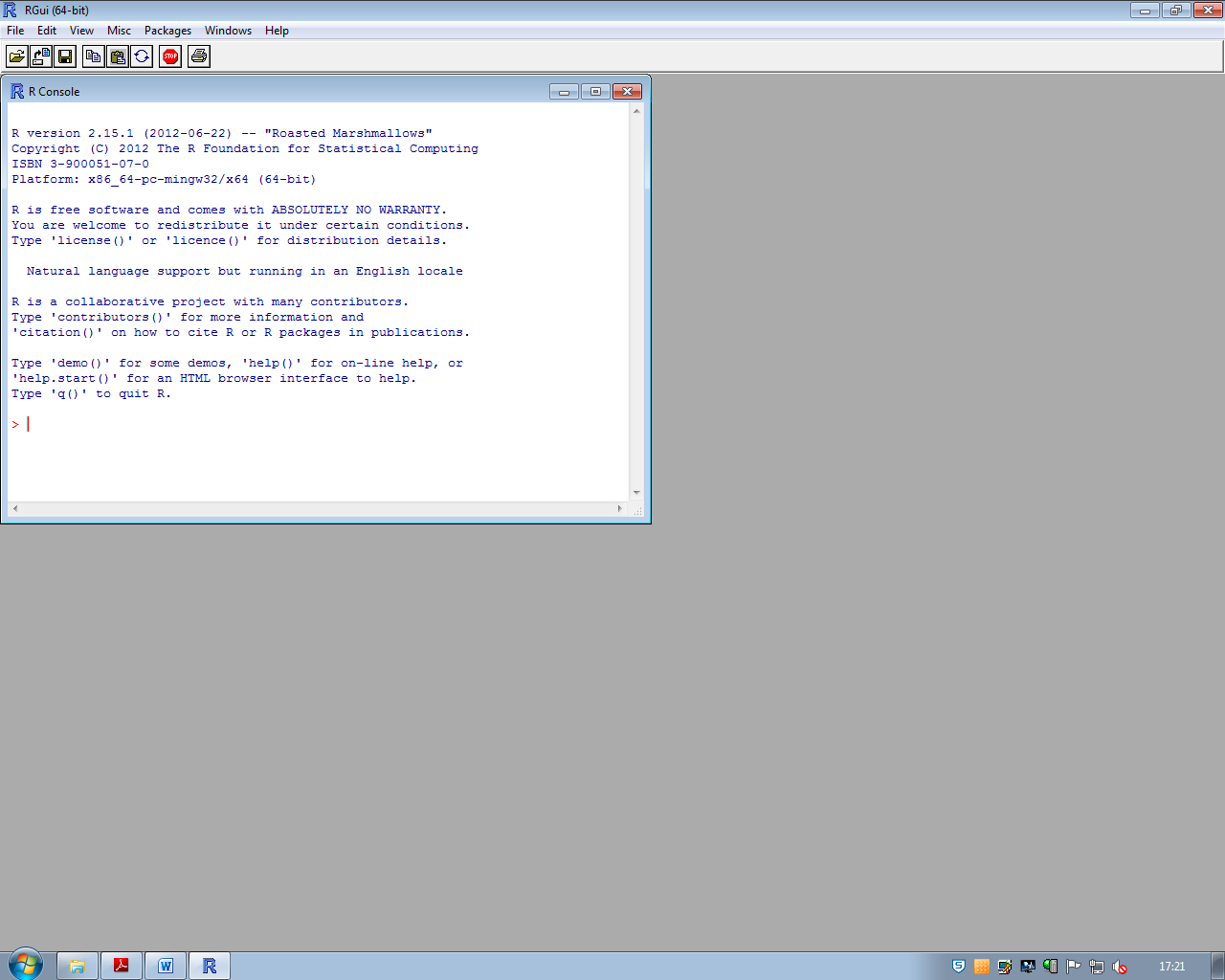
In Moodle, go to My Courses 🡪MY591 🡪R. Select the data files using right-click 🡪 Save target as, then save to the MY591R folder that you have just created in your H: space.

* 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")



## 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.

## 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.

# 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.

# 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.

# 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

# 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.

# 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

# 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

# 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)

# R Essentials

## 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

## 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[Maths>50 & Statistics>50 & English>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()

# 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()).

# 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*.

# 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 : µ = 80;*

*H1 : µ ≠ 80.*

We set significance level of 5%, that is α= 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)

# A linear model

## Simple Linear Regression

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?

## Multiple Linear Regression

A model with several variables is constructed using the operator + in the model formula. In this context + denotes inclusion not addition. We can use the operator - is used for exclusion in model formulae. When models are updated, we use a . to denote contents of the original model. The use of these operators is best understood in the context of an example.

To illustrate consider the Cars93 data from the MASS package. These are the values of variables recorded on 93 cars in the USA in 1993. Consider the MPG.highway (fuel consumption in highway driving) variable. It is reasonable to suppose that fuel consumption may, in part, be determined by the size of the vehicle and by characteristics of the engine. We start by using pairs(...) to generate a scatter plot matrix for four of the variables in the data set. A linear model is fitted using the lm(...) function (notice we use the named argument data to specify the data set as an alternative to attaching the data).

> library(MASS)

> ?Cars93

> names(Cars93)

> pairs(Cars93[c("MPG.highway","Horsepower","RPM","Weight")], col=2)

> lmMPG1 <- lm(MPG.highway ~ Horsepower + RPM + Weight, data=Cars93)

> summary(lmMPG1)

The anova(...) function gives us the (sequential) analysis of variance table using the order in which the variables are specified.

> anova(lmMPG1)

In the analysis of variance table for this example, adding each variable in the order specified gives a significant reduction in the error sum of squares. However, the explanatory variables are highly correlated; changing the order in which they are included alters the analysis of variance table.

> lmMPG2 <- lm(MPG.highway ~ Weight + Horsepower + RPM, data=Cars93)

> anova(lmMPG2)

Notice that we could have achieved the same outcome by updating our lmMPG1 model object.

Below we use (.-Weight) to denote the existing right-hand-side of the model formula with the variable Weight removed.

> lmMPG2 <- update(lmMPG1, ~ Weight + (.-Weight), data=Cars93)

> anova(lmMPG2)

When explanatory variables are correlated, inclusion or exclusion of one variable will affect the significance (as measured by the t-statistic) of the other variables. For example, if we exclude the Weight variable, both Horsepower and RPM become highly significant. Variables cannot be chosen simply on the basis of their significance in a larger model. This is the subject of the next section.

## Model selection

The MASS package provides a number of functions that are useful in the process of variable selection. Consider the situation where we are interested in constructing the best (according to some criteria that we will specify later) linear model of MPG.highway in terms of ten of the other variables in model.

**Forward search**

In a forward search, from some starting model, we include variables one by one. We have established that MPG.highway is reasonably well explained by Weight. A model with just the Weight variable is a reasonable starting point for our forward search.

> lmMPG3 <- lm(MPG.highway ~ Weight, data=Cars93)

> summary(lmMPG3)

We may want to consider what the effect of adding another term to this model is likely to be.

We can do this using the addterm(...) function.

> addterm(lmMPG3, ~.+EngineSize + Horsepower + RPM + Rev.per.mile + Fuel.tank.capacity + Length + Wheelbase + +Width+ Turn.circle, test="F")

This function adds a single term from those listed and displays the corresponding F statistic.

The second argument is a model formula; the term ~ . denotes our existing model. If we select variables according to their significance (most significant first) the variable Length would be the next to be included.

> lmMPG4 <- update(lmMPG3, ~ .+Length)

> summary(lmMPG4)

This process of variable inclusion can be repeated until there are no further significant variables to include. Notice something strange here; the coefficient for Length is positive. Does this seem counter intuitive? What happens when we remove Weight from the model?

> summary(update(lmMPG4, \_ .-Weight))

**Backwards elimination**

An alternative approach is to start with a large model and remove terms one by one. The function dropterm(...) allows us to see the impact of removing variables from a model. We start by including all ten candidate explanatory variables.

> lmMPG5 <- lm(MPG.highway ~ Weight + EngineSize + Horsepower + + RPM + Rev.per.mile + Fuel.tank.capacity + Length + +Wheelbase + Width + Turn.circle, data=Cars93)

> dropterm(lmMPG5, test="F")

Rather surprisingly, it is clear from this output that, in the presence of the other variables, Horsepower is not significant. Horsepower can be removed using the update(...) function and repeat the process.

> lmMPG6 <- update(lmMPG5, ~ .-Horsepower)

> dropterm(lmMPG6, test="F")

Which variable does the output of this command suggest should be dropped next?

**Step-wise selection**

The process of model selection can be automated using the step(...) function. This uses the Akaike information criteria (AIC) to select models. AIC is a measure of goodness of fit that penalises models with too many parameters; low AIC is desirable. The function can be used to perform a forward search.

> lmMPG7 <- lm(MPG.highway ~ 1, data=Cars93)

> step(lmMPG7, scope=list(upper=lmMPG5), direction="forward")

Here the starting model (lmMPG7) just contains a constant term. The scope=list(upper=lmMPG5)argument tells R the largest model that we are willing to consider. The process stops when the model with the smallest AIC is that resulting from adding no further variables to the model. Using a similar strategy we can automate backwards selection.

> step(lmMPG5, scope=list(lower=lmMPG7), direction="backward")

True step-wise regression allows us to go in both directions; variables may be both included and removed (if these actions result in a reduction of AIC).

> step(lmMPG7, scope=list(upper=lmMPG5))

At each stage the output shows the AIC resulting from removing variables that are in the model, including variables that are outside the model or doing nothing <none>. The result is a model with four of original ten variables included.

Model selection procedures are based on arbitrary criteria. There is no guarantee that the resulting model will be a good model (in the sense of giving good predictions) or that is will be sensible (in terms of what we know about the economics/physics/chemistry/. . . of the process under consideration). The procedures discussed in this section may also produce undesirable results in the presence of outliers or influential points.

## Polynomial regression

We have considered models that are linear in both the parameters and the explanatory variables. Higher order terms in the explanatory variables are readily included in a regression model. A regression model with a quadratic term is

In polynomial regression the lower order terms are referred to as being marginal. For example, is marginal to . If the marginal term is absent, a constraint is imposed on the fitted function; if is excluded from quadratic regression, the regression curve is constrained to be symmetric about the origin (which is not usually what we want). In a variable selection problem, removal of marginal terms is not usually considered.

A quadratic regression model is fitted in R by including a quadratic term in the model formula.

> lmMPG11 <- lm(MPG.highway \_ Weight+I(Weight^2), data=Cars93)

> summary(lmMPG11)

The I(...) is necessary to prevent ^ from being interpreted as part of the model formula. This function forces R to interpret an object in its simplest form - it is also useful for preventing character vectors from being interpreted as factors.

We can fit higher order terms in multiple regression. These often take the form of interactions.

For example, in the model

The parameter measures the strength of the interaction between and . This model can be fitted using

> summary(lm(MPG.highway ~ Weight +Wheelbase+ +I(Weight\*Wheelbase), data=Cars93))

or equivalently

> summary(lm(MPG.highway ~ Weight\*Wheelbase, data=Cars93))

Note that in the second of these commands, R fits the marginal terms automatically.

# Flow control

## Loops: for

A for loop often provides the most obvious implementation.

for (loopvariable in sequence ) expr1

Here sequence is actually any vector expression but usually takes the form of a regular sequence such as 1:5. The statements of expr1 are executed for each value of the loop variable in the sequence. An couple of examples follow.

> for (i in 1:5) print(i)

> attach(mk2nd)

> for (i in 1:length(exam1))

+ { ans <- exam1[i] + exam2[i] + exam3[i]

+ cat(row.names(mk2nd)[i], " total: ", ans, "\n")

+ }

## Conditional statements: if

The if statement in R follows the following standard syntax:

if (condition ) ifbranch

if (condition ) ifbranch else elsebranch

Here the condition is an expression that yields a logical value (TRUE or FALSE) when evaluated. This is typically a simple expression like x > y or dog == cat. An example follows to illustrate the if - else statement.

Suppose that in order to pass student must achieve a total mark of 60 or higher. We can easily write R code to tell us which students have passed.

> for (i in 1:length(exam1))

+ { ans <- exam1[i] + exam2[i] + exam3[i]

+ cat(row.names(mk2nd)[i], ": ")

+ if (ans >= 60) cat("PASS \n")

+ else cat("FAIL \n")

+ }

## Vectorization and avoiding loops

Loops are not efficiently implemented in R. One way of avoiding the use of loops is to use commands that operate on whole objects. For example, ifelse(...) is a conditional statement that works on whole vectors (rather than requiring a loop to go through the elements).

ifelse(condition,vec1,vec2 )

If condition, vec1 and vec2 are vectors of the same length, the return value is a vector whose

ith elements if vec1[i] if condition[i] is true and vec2[i] otherwise. If condition, vec1 and vec2 are of different lengths, the recycling rule is used. Repeating the previous example using vectorization:

> pf <- ifelse(ans>=60, "PASS", "FAIL")

> cat(paste(row.names(mk2nd), ":", pf), fill=12)

# 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. Consider the logistic map function

+1 = (1 - )

We can write an R code to implement this function as follows:

> logistic <- function(r,x) r\*x\*(1-x)

Now that we have defined the function, we can use it to evaluate the logistic function for different values of r and x (including vector values).

> logistic(3,0)

[1] 0

> logistic(3,0.4)

[1] 0.72

> logistic(2,0.4)

[1] 0.48

> logistic(3.5, seq(0,1,length=6))

[1] 0.00 0.56 0.84 0.84 0.56 0.00

The expression whose statements are executed by a call to logistic(...) is just the single line r\*x\*(1-x). This is also the return value of the function. The expression in a function may run to several lines. In this case the expression is enclosed in curly braces { } and the final line of the expression determines the return value.

# Simulation

## Generating (pseudo-)random samples

Using R we can generate random instances from any commonly used distribution. The function

rdistributionname (n,...)

will return a random sample of size n from the named distribution. At the heart of this function, R uses some of the most recent innovations in random number generation. The following are the names used by R for some of the most commonly used distributions:

**Distribution R name additional arguments**

binomial binom size, prob

chi-squared chisq df, ncp

exponential exp rate

gamma gamma shape, scale

normal norm mean, sd

Poisson pois lambda

Student's t t df, ncp

The additional arguments are mostly self-explanatory. The ncp stands for non-centrality parameter and allows us to deal with non-central chi-square and non-central t distributions.

We illustrate by sampling from Poisson and normal distributions.

> poissamp <- rpois(400, lambda=2)

> hist(poissamp, breaks=0:10, probability=TRUE)

> normsamp <- rnorm(250, mean=10, sd=5)

> hist(normsamp, breaks=seq(-10,30,length=15), probability=TRUE)

> x <- seq(-10,30,length=200)

> lines(x, dnorm(x, mean=10, sd=5), col=2)

The random number generator works as an iterative process. Thus, consecutive identical commands will not give the same output.

> rnorm(5)

[1] 0.4874291 0.7383247 0.5757814 -0.3053884 1.5117812

> rnorm(5)

[1] 0.38984324 -0.62124058 -2.21469989 1.12493092 -0.04493361

The command set.seed(...) allows us to determine the starting point of the iterative process and thus ensure identical output from the random number generator. This is useful when developing the code for a simulation experiment.

The functions sample(...) can be used to generate random permutations and random samples from a data vector. The arguments to the function are the vector that we would like to sample from and the size of the vector (if the size is excluded a permutation of the vector is generated). Sampling with replacement is also possible using this command.

> nvec <- 10:19

> sample(nvec)

> sample(nvec, 5)

> sample(nvec, replace=TRUE)

> sample(nvec, 20, replace=TRUE)

> cvec <- c("Y","o","u","r","n","a","m","e")

> sample(cvec)

> cat(sample(cvec), "\n")

## Simulation experiments - an example

To demonstrate how a simulation experiment works, we are going to look at the small sample

properties of the sample mean for a Poisson population, a statistic whose asymptotic properties are well known. Consider a population that has a Poisson distribution with mean λ, so Y ~ Pois(λ). We take a sample of size n. We would like to know whether the normal distribution will provide a reasonable approximation to the distribution of the sample mean. we first consider the questions that we would like our simulation to answer:

1. How large does n need to be for the normal distribution to be a reasonable approximation?

2. What is the effect of the value of λ?

Writing down these questions makes clear the factors that we will need to make comparisons

across:

* n, the sample size: the central limit theorem tells us that for large values of n the normal will be a reasonable approximation,
* λ, the parameter of the population distribution: we might expect the shape of the under-lying Poisson distribution to have an effect on the distribution of the sample mean.

We will use the computer to generate simulated samples),..., where is the number of simulated replications. For each of these simulates

samples, we evaluate the sample mean to give a sequence that can be viewed

as instances of the statistic of interest . The first step is to write a function to generate

simulated samples and, from these, simulated values of the statistic.

> poisSampMean1 <- function(n, lambda, r)

+ { meanvec <- c()

+ for (j in 1:r)

+ { sampvals <- rpois(n, lambda)

+ meanvec <- c(meanvec, mean(sampvals))

+ }

+ meanvec

+ }

> set.seed(1)

> poisSampMean1(10, 3, 6)

[1] 3.3 3.4 2.6 3.0 3.3 2.6

To get a visual impression of the simulated sample means we write a function to draw a histogram and plot a normal distribution with the same mean and standard deviation.

> histNorm <- function(data, nbins=21)

+ { hist(data, breaks=seq(min(data), max(data), length=nbins),

+ probability=TRUE, col=5)

+ x <- seq(min(data), max(data), length=200)

+ lines(x, dnorm(x, mean=mean(data), sd=sd(data)), col=2)

+ }

Try experimenting with this function with various values of n and λ with r = 1000. If this runs very slowly, try reducing r. If you get a histogram with strange gaps in, try changing the value of the nbins argument.

> histNorm(poisSampMean2(8,1,1000))

> histNorm(poisSampMean2(100,10,1000))