R: An Introduction

How to Use this User Guide

This handbook accompanies the taught sessions for the course. Each section contains a brief overview of a topic for your reference and then one or more exercises.

Exercises are arranged as follows:

- A title and brief overview of the tasks to be carried out;
- A numbered set of tasks, together with a brief description of each;
- A numbered set of detailed steps that will achieve each task.

Some exercises, particularly those within the same section, assume that you have completed earlier exercises. Your teacher will direct you to the location of files that are needed for the exercises. If you have any problems with the text or the exercises, please ask the teacher or one of the demonstrators for help.

This book includes plenty of exercise activities – more than can usually be completed during the hands-on sessions of the course. You should select some to try during the course, while the teacher and demonstrator(s) are around to guide you. Later, you may attend follow-up sessions at IT Services called Computer8, where you can continue work on the exercises, with some support from IT teachers. Other exercises are for you to try on your own, as a reminder or an extension of the work done during the course.

Text Conventions

A number of conventions are used to help you to be clear about what you need to do in each step of a task.

- In general, the word **press** indicates you need to press a key on the keyboard. **Click**, **choose** or **select** refer to using the mouse and clicking on items on the screen. If you have more than one mouse button, click usually refers to the left button unless stated otherwise.
- Labels and titles on the screen are shown **like this**.
- Drop-down menu options are indicated by the name of the options separated by a vertical bar, for example **File|Print**. In this example you need to select the option **Print** from the **File** menu or tab. To do this, click when the mouse pointer is on the **File** menu or tab name; move the pointer to **Print**; when **Print** is highlighted, click the mouse button again.
- A button to be clicked will look **like this**.
- The names of software packages are identified *like this*, and the names of files to be used **like this**.
- Boxes with the broken borders have examples contained within.
- Boxes with double borders have cautionary exposition within.

Software Used

R 2.14.0 or later versions

Files Used

text_sample.Rd, age_weight.txt, district.txt, expenditure.txt, expenditure.csv

The other datasets used have been preloaded in the R package 'datasets'. To list all the datasets available, use the command

```
> data()
```

This will bring up a window listing the datasets and providing a short description of each. To obtain a more detailed description, use the command help(). For example, the line below brings up a html file describing the 'Biochemical Oxygen Demand' experiment in greater detail.

```
> help("BOD")
```

Revision Information

Version	Date	Author	Changes made
2.0	Jan 2013	Esther Ng	Created

Copyright

Acknowledgements

David Baker, Denise Cattell and Kathryn Wenczek for their help in organising the course and putting the course notes together

Contents

1 Introduction 7
1.1. What you should already know 7
1.2. What will you learn? 7
2 Getting Started 8
2.1. What is R? 8
2.1.1. Advantages 8
2.1.2. Disadvantages 8
2.2. Download and Installation 8
2.3. R and other tools/languages 8
2.3.1. Matlab 8
2.3.2. SPSS/SAS 8
2.3.3. C++ 8
2.4. R on different platforms 9
2.5. Commands 9
2.6. Getting Help 9
Exercise 1 Getting Started 12
3 Basic Data Structures 13
3.1. Vectors 13
3.1.1. Creating numerical vectors 13
3.1.2. Accessing elements of numerical vectors 14
3.1.3. Arithmetic operations on numerical vectors 15
3.1.4. Useful vector commands 15
3.2. Lists 16
3.3. Matrices 16
3.3.1. Creating matrices 16
3.3.2. Accessing elements of a matrix 18
3.3.3. Useful matrix commands 19
3.4. Data frames 21
3.5. Factors 21
Exercise 2 Vectors and Matrices 22
4 Reading and Writing data 24
4.1. Reading Text Files 24
4.2. Reading built-in data 24

4.3. Editing Data in Spreadsheet style 24
4.4. Writing out data 25
Exercise 3 Reading and Writing Files 25
5 Scripts, Objects and the Workspace 27
5.1. Objects and the Workspace 27
5.2. Directories 28
5.3. Running scripts 28
Exercise 4 Scripts, Objects and Workspaces 29
6 Control Statements and Loops 31
6.1. If/Else and Ifelse 31
6.2. Loops 31
Exercise 5 Loops and If/Else statements 33
7 Working with text in R 34
7.1. Characters and Strings 34
7.2. Useful commands 35
7.2.1. Count number of characters in the string 35
7.2.2. Split the string 35
7.2.3. Search text strings for a word 36
Exercise 6 Text Manipulations in R 36
8 Graphs and Charts 37
8.1. High level plotting commands 37
8.1.1. Generic plot() command 37
8.1.2. Boxplot 38
8.1.3. Histogram 38
8.1.4. Options 39
8.2. Low Level Plotting Functions 42
Exercise 7 Plotting functions in R 43
9 Statistics with R 45
9.1. Measures of Central Tendency and Spread 45
9.2. Tests for continuous vs discrete variables 45
9.2.1. 2 groups 45
9.2.2. 3 or more groups 45
9.3. Tests for discrete vs discrete variables 48
9.3.1. Chi Square Test 48
9.3.2. Fisher's Exact Test 48

9.4. Tests for continuous vs continuous variables 49
9.4.1. Pearson Correlation 49
9.4.2. Spearman Correlation 49
9.5. Regression 51
9.6. Probability Distributions 52
9.7. Other Statistical Features 52
9.8. Packages 53
Exercise 8 Statistics in R 53
10 Writing your own functions 54
11 References 55

1 Introduction

Welcome to Introduction to R!

This booklet accompanies the course delivered by Oxford University IT services, IT Learning Programme. Although the exercises are clearly explained so that you can work through them independently, you will find that it will help if you also attend the taught session where you can get advice from the teacher, demonstrator(s) and even each other!

If at any time you are not clear about any aspect of the course, please make sure you ask your teacher or demonstrator for some help. If you are away from the class, you can get help by email from your teacher or from help@it.ox.ac.uk.

1.1. What you should already know

While basic knowledge of programming would be helpful, it is not essential. Knowledge of statistics would be helpful in understanding the later chapters.

1.2. What will you learn?

This course will teach basic R programming for data analysis and presentation. While it provides a framework of tools, please remember that data is highly individualised from study to study, hence methods which seem applicable to one study are not necessarily generalizable to another study. R has an active network of users who are constantly developing packages for various disciplines. These packages are open-source and can be freely downloaded from

http://cran.r-project.org/

2 Getting Started

2.1. What is R?

R is a collection of software tools for data manipulation, analysis and presentation. It is an implementation of the S language. There are several advantages and disadvantages

2.1.1. Advantages

Extensive collection of statistical functions and operators for matrix calculations

Effective data handling and storage

Simple syntax

Good graphical facilities for data display

2.1.2. Disadvantages

Steeper learning curve compared to SPSS or SAS for users with no background in programming

Slow performance compared to languages like C/C++

Larger room for error as R tends to make assumptions rather than produce error messages when commands are unclear

2.2. Download and Installation

Detailed instructions can be found at http://cran.r-project.org/ for Windows, Unix and Mac. For certain packages, previous versions for R may be needed due to various dependencies. These previously versions can also be downloaded from the website mentioned above.

2.3. R and other tools/languages

2.3.1. Matlab

For those with a background in engineering and a knowledge of Matlab, this document facilitates the translation of Matlab syntax into R syntax as there are functions which are very similar in both environments http://cran.r-project.org/doc/contrib/Hiebeler-matlabR.pdf

2.3.2. SPSS/SAS

For those who have mainly used SPSS/SAS for their statistical needs, R may seem confusing at first due to the command line interface. However, this document http://www.et.bs.ehu.es/~etptupaf/pub/R/RforSAS&SPSSusers.pdf provides more information to smoothen the transition between the different environments

2.3.3. C++

C++ code is often used to speed up calculations in R. The 2 languages can be interfaced more easily using this package. http://cran.r-project.org/web/packages/Rcpp/vignettes/Rcpp-introduction.pdf

2.4. R on different platforms

Most functions in R are common to all platforms. On a unix platform, the graphical device may need to be setup with a command such as

```
$ sys.putenv("DISPLAY"=":0.0")
```

This may not work on all machines, and other settings may need to be adjusted.

When executing a script file from the command line in unix, the following can be used

```
$ R CMD BATCH my_script_file.R
```

Other details on differences between platforms can be found on http://www.r-project.org/user-2006/Slides/IacusEtAl.pdf

2.5. Commands

In R, the command prompt is '>'. If you see '+' instead of '>', it means that the command is incomplete. For example, if there are unpaired brackets or inverted commas, '+' will be seen instead of '>'. Commands can either run by typing them directly into the console or by typing them into the R editor then highlighting, right-clicking and selecting the option 'run selection'.

The output of a command can either be immediately printed to the screen or it can be stored in a variable.

For example, the output of a command 'mean' can either be immediately printed to the screen (in which case it is not stored) or it can be stored in a variable (in this case the variable named 'answer') and later accessed by typing the variable name at the command prompt.

```
> mean(5,6,7)
[1] 5
> mean(5,6,7)->answer
> answer
[1] 5
```

In R, the output of a command can be assigned to a variable with '->' or '='. Direction of assignment does not matter, hence 'a<-5' is equivalent to '5->a', both assign the value of 5 to the variable 'a'.

Of note, all alphanumeric symbols are allowed in variable names, in addition to '.' And '_'. However, variable names cannot start with a number eg. 'a2' is allowed but not '2a'

Commands can be separated by a new line or by a semi-colon(;). They can be commented out with a hashmark(#).

R has a command history function, in which previously issued commands can be recalled with the 2 vertical keys.

2.6. Getting Help

To get help on any specific function, use the command 'help' or '?'. For example, to search for help on the command 'sum', at the command line, type

```
> help(sum) or
> ?sum
```

This will bring up the help page http://127.0.0.1:26807/library/base/html/sum.html

This page contains several sections

sum {base} R Documentation

This refers to the package from which the function comes

Sum of Vector Elements

Description

`sum` returns the sum of all the values present in its arguments.

Usage

This shows how the function is used, together with the default settings. In this case, NAs are not removed by default. If removal is wished, use 'na.rm=TRUE'

```
sum(..., na.rm = FALSE)
```

Arguments

This refers to input arguments of the function

`...` numeric or complex or logical vectors.

`na.rm` logical. Should missing values (including `NaN`) be removed?

Details

This provides further details on the function, including appropriate usage and meaning of default settings

This is a generic function: methods can be defined for it directly or via the `Summary` group generic. For this to work properly, the arguments `...` should be unnamed, and dispatch is on the first argument.

If `na.rm` is `FALSE` an `NA` or `NaN` value in any of the arguments will cause a value of `NA` or `NaN` to be returned, otherwise `NA` and `NaN` values are ignored.

Logical true values are regarded as one, false values as zero. For historical reasons, `NULL` is accepted and treated as if it were `integer(0)`.

Value

This refers to output of the function

The sum. If all of `...` are of type integer or logical, then the sum is integer, and in that case the result will be `NA` (with a warning) if integer overflow occurs. Otherwise it is a length-one numeric or complex vector.

NB: the sum of an empty set is zero, by definition.

This refers to the class of the function. This information is not usually needed by the general user.

S4 methods

This is part of the S4 `Summary` group generic. Methods for it must use the signature `x, ..., na.rm`.

'plotmath' for the use of `sum` in plot annotation.

References

This section references, and is especially useful for recently developed statistical tools

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

This section contains related functions and is often useful to explore as there may be combined functions that save you writing code

`colSums` for row and column sums.

[Package *base* version 2.14.2 Index]

Exercise 1 Getting Started

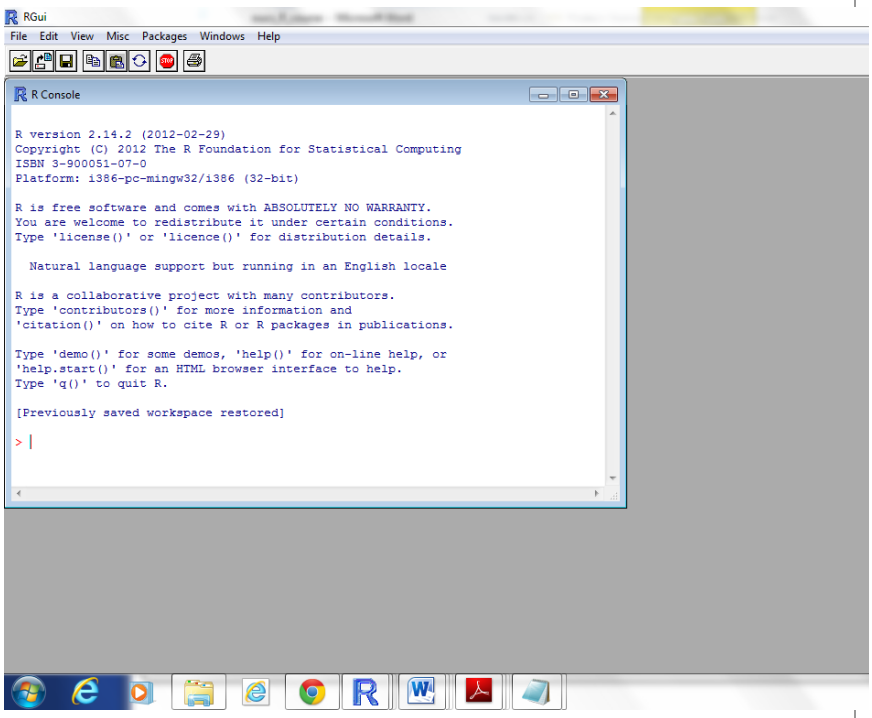
In this exercise, we will explore the graphical user interface in R and the help functions

Task 1

Open the R console; Learn how to use the help facility

Step 1

Open R. Click on the Start button on the Task Bar then click on R. This will bring up the R console.



Step 2

At the console, type

> help.start()

This will bring up a html page from the official CRAN website

http://127.0.0.1:31317/doc/html/index.html

Navigate to the link 'An Introduction to R'. This link contains an immense amount of helpful information, inaddition to a Sample Session in Appendix A which should be worked through when you have the time.

Step 3

We will be using the save() command later. In the R console, type

>?save()

to bring up the html page on save(). What are the differences between the save.image() and save() commands?

3 Basic Data Structures

3.1. Vectors

3.1.1. Creating numerical vectors

The simplest data structure in R is the numerical vector. This can be created with the c() command or with the assign command. The statements below are equivalent.

> x <- c(5,4,3,2,1)

> assign("x", c(5,4,3,2,1))

Sequences or repeats of numbers can be generated in R. To generate a sequence of consecutive numbers, either the colon operator or the function seq() can be used. If the sequence has steps of greater than 1, the seq() command can be used with a third argument.

```
> a<-c(1:10) ## this produces a sequence of 1 to 10 in steps of 1
> a
 [1]  1  2  3  4  5  6  7  8  9 10
> a<-seq(1,10) ## this achieves the same effect as the command above
> a
 [1]  1  2  3  4  5  6  7  8  9 10
> b<-seq(1,10,2) ## this produces a sequence of 1 to 10 in steps of 2
> b
[1] 1 3 5 7 9
> d<-rep(1,10) ## this produces a vector of 10 'ones'
> d
 [1] 1 1 1 1 1 1 1 1 1 1
```

Vectors or parts of vectors can be concatenated together, with the c() command

```
> c(y,y,5)->y2
> y2
 [1] 5 4 3 2 1 5 4 3 2 1 5
```

An empty vector can also be created and populated with numbers. This is useful when writing loops in which the result of repeated calculations are used to populate a vector.

```
> my_vector <- vector() ## creates empty vector
> my_vector[1] <- 42 ## inserts the value of 42 into the first slot of the vector
> my_vector[2] <- 43 ## inserts the value of 43 into the second slot of the vector
> length(my_vector) ## checks the length of the vector
[1] 2
```

3.1.2. Accessing elements of numerical vectors

To access each element of the vector, square brackets are used. To access a few elements of the vector, a colon can be used.

```
> y <- x[5] ## this assigns the value of 1 to y since the fifth element of
x is 1
> z <- x[1:3] ## this creates a new vector z with 3 numbers – 5,4,3
```

Index vectors can be created to select specific elements within a vector. There are different types of index vectors.

a) Logical index vectors

A logical index vector is comprised of a series of 'TRUE' and 'FALSE' elements.

```
> a<-seq(1,10) ## produces sequence of numbers from 1 to 10
> a<7 -> b ## creates a logical vector based on the conditions provided
> b
 [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
> a[b]
> [1] 1 2 3 4 5 6
> a[a<7]  ## combines the above steps
 [1] 1 2 3 4 5 6
```

b) Vector of positive integers

In this case, the values in the index vector must be smaller than the length of the vector being indexed. The elements that correspond to the elements of the index vector are concatenated into a new vector

```
> primary.vector <- seq(1,10,2)
> index.vector <- c(1,3,5)
> primary.vector
[1] 1 3 5 7 9
> primary.vector[index.vector]
[1] 1 5 9
```

c) Vector of negative integers

This specifies the values to be excluded.

```
> index.vector <- c(-1,-3,-5)
> primary.vector[index.vector]
[1] 3 7
```

3.1.3. Arithmetic operations on numerical vectors

Arithmetic operations can be performed on vectors. These are performed on each number within the vector. In the example below, 1 is subtracted from each number within the vector y2.

```
> y2-1
 [1] 4 3 2 1 0 4 3 2 1 0 4
```

Caution: When shorter vectors are used in the expression, they are recycled. In the example below, the numbers in y are added individually to the numbers in y2, with the first number of y added to the first number of y2, the second number of y added to the second number of y2 etc, until the fifth number of y is reached. After that, the first number of y is added to the sixth number of y2. While other environments may generate a warning message when vectors of different lengths are added together, R will not generate such warnings. As such, users are more prone to errors since it is more likely that one has wrongly tried to added 2 vectors of different lengths rather than wanting one vector to be fractionally recycled to be added to another vector.

The use of the term 'vector' and later 'matrix' in this context does not pertain to linear algebra. The vector operations performed with the commands detailed above are strictly for element-wise operations. Linear algebra commands are different. For example, the symbol for matrix multiplication is %*%. While linear algebra is beyond the scope of this course, an excellent guide to Linear Algebra in R can be found in the link below.

http://bendixcarstensen.com/APC/linalg-notes-BxC.pdf

3.1.4. Useful vector commands

The length() command will find the length of the vector

```
> length(y2)
> [1] 11
```

The max() command will find the largest element of the vector

```
> max(y2)
> [1] 5
```

The min() command will find the smallest element of the vector

```
> min(y2)
> [1] 1
```

The sum() command will find the sum of all elements in the vector

```
> sum(y2)
> [1] 35
```

The mean() command will find the average of all elements in the vector

```
> mean(y2)
[1] 3
```

The sd() command will find the standard deviation of all elements in the vector

```
> sd(y2)
```

```
[1] 1.490712
```

The sort() command returns vector sorted in numerical order

```
> sort(y2)
 [1] 1 1 2 2 3 3 4 4 5 5 5
```

3.2. Lists

A list is an ordered collection of objects known as components. Lists are like vectors except for a few differences…

a) Different types of objects can be concatenated into lists eg. a list can consist of a numeric vector, a matrix and a character string. The components of a vector all have to be of the same type eg. all numbers, all characters etc.

b) The components of a list are accessed by double square brackets ([[ ]]) whereas the components of a vector are accessed with single square brackets ([ ])

c) Linear algebra can be performed on vectors of numbers but not lists of numbers

d) Lists are recursive, meaning that one can create a list of lists of lists. This is not possible with vectors.

e) Components of lists can be accessed by name through the $ operator. This is not possible with a vector

```
> list() -> my_list ## creates an empty list
> 42 -> my_list[[1]] ## populates the list with numbers
> 43 -> my_list[[2]]
> names(my_list) <- c("first_entry","second_entry") ## assigns names to the list components
> my_list$first_entry ## accesses components of the list using the name
[1] 42
```

Caution: When to use lists and when to use vectors?

When performing mathematical or statistical calculations, it is easier to use vectors because one will be sure that all components of the vector will be of the numerical type and one can perform linear algebra with them. Lists are useful when manipulating data that includes different types of information eg. databases of names, addresses, ages etc.

3.3. Matrices

An array is a subscripted collection of data elements. A matrix is a 2 dimensional array. Since matrices are the most commonly used form of array, we will describe matrices in detail.

3.3.1. Creating matrices

Arrays and matrices can be created with the array and matrix commands respectively. Note that the if byrow=TRUE for the matrix() function, the data will be entered by row, if byrow=FALSE, it will be entered by column, like the array() function.

```
> my_matrix <- matrix(c(1,2,3, 7,8,9), nrow = 2, ncol=3, byrow=TRUE)
> my_matrix
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    7    8    9
> my_array <- array(c(1,7,2,8,3,9), c(2,3))
>my_array
       [,1] [,2] [,3]
[1,]    1    2    3
[2,]    7    8    9
```

An empty matrix can be created and populated with data

```
> my_matrix <- matrix(nrow=3,ncol=3)
> my_matrix[2,2] <- 42
> my_matrix
     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA   42   NA
[3,]   NA   NA   NA
```

R inserts NA for any unpopulated elements in a matrix or vector

A matrix can also be created by binding vectors or matrices together using the cbind() or rbind() functions which binds vectors in terms of columns and rows respectively.

```
> rbind(my_matrix,my_matrix) ->combined_matrix
> combined_matrix
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    7    8    9
[3,]    1    2    3
[4,]    7    8    9
```

Names can be assigned to the rows and columns of a matrix with the commands rownames() and colnames().

```
> rbind(my_matrix,my_matrix) ->combined_matrix
> rownames(combined_matrix) <-
c("basket_1","basket_2","basket_3","basket_4")
> colnames(combined_matrix) <-
c("number_of_apples","number_of_oranges","number_of_pears")
```

```
> combined_matrix
         number_of_apples number_of_oranges number_of_pears
basket_1                1                 2               3
basket_2                7                 8               9
basket_3                1                 2               3
basket_4                7                 8               9
```

3.3.2. Accessing elements of a matrix

The elements of a matrix can be accessed with 2 numbers separated in a square bracket separated by a comma, with the first number being the row and the second number being the column.

> combined_matrix[2,1]

[1] 7

Similar, several entries can be accessed at once with a colon, in the same way as a vector

> combined_matrix[2,1:3]

[1] 7 8 9

Elements can be accessed conditionally too.

For example, if one wanted to replace all numbers smaller than 3 with 0, this can be done

```
> combined_matrix[combined_matrix<3] <-0
> combined_matrix
     [,1] [,2] [,3]
[1,]    0    0    3
[2,]    7    8    9
[3,]    0    0    3
[4,]    7    8    9
```

R does this by creating an index matrix of TRUE and FALSE depending on the condition imposed.

```
> combined_matrix<3
      [,1]  [,2]  [,3]
[1,]  TRUE  TRUE FALSE
[2,] FALSE FALSE FALSE
[3,]  TRUE  TRUE FALSE
[4,] FALSE FALSE FALSE
```

This can also be performed on selected rows or columns. The example below uses the original object ‘combined_matrix’ and replaces numbers less than 3 with 0 only in the first row.

```
> combined_matrix[1,which(combined_matrix[1,]<3)] <-0
> combined_matrix
     [,1] [,2] [,3]
[1,]    0    0    3
[2,]    7    8    9
[3,]    1    2    3
[4,]    7    8    9
```

In this way, matrix elements can be accessed or modified according to conditions in other matrices. For example, if one had a matrix of gene expression values as well as a separate matrix of 'YES' and 'NO' to indicate whether a particular measurement is valid or not, one could insert a number representing an invalid measurement eg. '-999' into the matrix of gene expression values depending on whether the corresponding word was 'YES' or 'NO' in the validity matrix.

An example is given below

```
> validity_matrix
     [,1]  [,2]  [,3]
[1,] "YES" "YES" "NO"
[2,] "NO"  "YES" "YES"
> measurement_matrix
     [,1] [,2] [,3]
[1,]    4    4    3
[2,]    6    6    7
```

> measurement_matrix[validity_matrix=="NO"] <- -999 ## this inserts the value '-999' into the measurement matrix to represent entries that correspond to 'NO' in the validity matrix. Note that the conditional operator for equals to is '==', like in most other environments.

```
> measurement_matrix
     [,1] [,2] [,3]
[1,]    4    4 -999
[2,] -999    6    7
```

3.3.3. Useful matrix commands

The dim() command outputs the dimensions of the matrix, just as the 'length' command outputs the length of the matrix.

```
> dim(measurement_matrix)
[1] 2 3
```

The order() command returns an index of the numerical order. This is especially useful as the matrix can be sorted in ascending or descending or of a particular column (or row). For example, if one took example of fruits in a basket in section 3.3.1, one could order the baskets in terms of the number of oranges in them

```
> combined_matrix
         number_of_apples number_of_oranges number_of_pears
basket_1                1                 2               3
basket_2                7                 8               9
basket_3                1                 2               3
basket_4                7                 8               9
> combined_matrix[order(combined_matrix[,2]),]->
combined_matrix_orange_ordered
> combined_matrix_orange_ordered
         number_of_apples number_of_oranges number_of_pears
basket_1                1                 2               3
basket_3                1                 2               3
basket_2                7                 8               9
basket_4                7                 8               9
```

The functions rowMeans, colMeans, colSums and rowSums are usefully in finding the average of rows/columns and the sums of rows/columns respectively

```
> colMeans(combined_matrix_orange_ordered)
 number_of_apples number_of_oranges   number_of_pears
              4                 5                 6
> rowMeans(combined_matrix_orange_ordered)
basket_1 basket_3 basket_2 basket_4
               2        2        8        8
> colSums(combined_matrix_orange_ordered)
 number_of_apples number_of_oranges   number_of_pears
                       16                20                24
> rowSums(combined_matrix_orange_ordered)
basket_1 basket_3 basket_2 basket_4
       6        6       24       24
```

The function t() is useful in transposing the matrix so that the rows become columns and the columns become rows.

```
> t(combined_matrix_orange_ordered) ->
combined_matrix_orange_ordered_transposed
> combined_matrix_orange_ordered_transposed
                 basket_1 basket_3 basket_2 basket_4
number_of_apples         1        1        7        7
number_of_oranges        2        2        8        8
number_of_pears          3        3        9        9
```

Caution: All components of a matrix have to be of the same type. If a character is inserted into a matrix of numbers, all components of the matrix will be coerced into a character type.

The type of object can be checked with the following commands – is.numeric, is.character, is.matrix, is.list, is.vector.

```
> is.numeric(combined_matrix_orange_ordered[1,3]) ## this checks the type
of element in the matrix at index [1,3]

[1] TRUE

> combined_matrix_orange_ordered[1,1] <- "unknown" ## this inserts the
character string into the matrix at index[1,1], hence coercing all other
elements of the index into character type

> combined_matrix_orange_ordered

         number_of_apples number_of_oranges number_of_pears

basket_1 "unknown"        "2"               "3"

basket_3 "1"              "2"               "3"

basket_2 "7"              "8"               "9"

basket_4 "7"              "8"               "9"

> is.numeric(combined_matrix_orange_ordered[1,3]) ## this demonstrates that
the elements in the matrix are now no longer numeric type, but are
character type

[1] FALSE

> is.character(combined_matrix_orange_ordered[1,3])

[1] TRUE
```

The command typeof() can be used to find the type an object is. However, it may not give full information eg. in the case of a character matrix, it will just report 'character'.

3.4. Data frames

A data frame is like a matrix that may contain elements of different types.

Data frames can be constructed with the function data.frame(). Alternatively, a list can be coerced into a data frame using the function as.data.frame()

Data can be read into R to form a data frame using the read.table() function (discussed later).

When dealing with data frames (unlike matrices), the attach() and detach() functions allow for a database to be loaded into R as a copy and modified temporarily without changing the original database.

3.5. Factors

A factor is a vector object used to specify grouping of the components of other vectors. An example is given below

```
> c("chocolate","vanilla","chocolate","strawberry","vanilla","vanilla","cho
colate") -> flavours

> factor(flavours) -> fflavours

> fflavours
```

```
[1] chocolate  vanilla    chocolate  strawberry vanilla    vanilla
chocolate
Levels: chocolate strawberry vanilla
```

Factors can also be constructed from a vector of numbers eg.

```
> y<-c(3,3,5,6,6) ## creates a vector of numbers
> y
[1] 3 3 5 6 6
> as.factor(y) -> y.f ## converts these numbers to factors
> y.f
[1] 3 3 5 6 6
Levels: 3 5 6
```

Caution: as.numeric will convert factors in a different way compared to characters. This is something that one has to beware of because data is sometimes read in terms of factors and other times in terms of characters (depending on the settings). Applying 'as.numeric' to a string of factors believed to be characters will provide an unexpected result. The example shown below is based on the vector 'y.f' created above.

> as.numeric(y.f) ## converts these factors back to numbers (a different set of numbers will result)

[1] 1 1 2 3 3

> as.numeric(as.character(y.f)) ## converts these factors back to numbers (original set of numbers will result)

[1] 3 3 5 6 6

Exercise 2 Vectors and Matrices

In this exercise, we will explore vector and matrix manipulations in R

Task 1 Explore the properties of a vector	**Step 1** The 'WWWusage' dataset provides a time series of the number of users connected to the internet through a server each minute. Find out how many entries there are in this vector with the command _____
	Step 2 Find the average of all the entries with the command _____
	Step 3 Extract the 40th to 60th entry in the vector and find that median of those 21 entries with the command _____

Task 2 Explore the properties of a matrix	**Step 1** The WorldPhones dataset describes the number of telephones in each region of the world (in the thousands). Find the dimensions of the dataset with the command _____
	Step 2 Find the difference between the number of phones in the two regions with the highest and lowest number of phones respectively in 1957 _____
	Step 3 In which year did Africa have 1411 thousand phones? _____
	Step 4 Which area had 45939 thousand phones in 1951? _____
	Step 5 Find the total number of phones in all areas in 1959 _____

4 Reading and Writing data

4.1. Reading Text Files

Text files can be read into R with the read.table() command. There are several options that can be supplied with this command

a) skip – this refers to the number of lines in the file that should be skipped before the actual data is input (default is zero)

b) header – this refers to whether the first line should be read in as column names (R will count the number of entries in the first and second row, setting this to true if there are 1 fewer entries in the first row than the subsequent rows)

c) fill – this refers to whether rows which have fewer entries than others are filled with blank fields (if this is not explicitly stated and there are rows with fewer fields than others, R will produce a warning message)

d) na.strings – this refers to the character string that symbolises 'not applicable'. The default setting is "NA".

e) sep – this is the field separator. The default setting is whitespace ie. " "

The file is read into R and a dataframe is created. This can be converted to a matrix with the as.matrix() command. This is useful when mathematical operations are performed on the data.

When reading tab delimited text files rather than white-space delimited text files, the command read.delim() can be used. In read.delim(), the default setting for sep is "\t". When reading a file with comma-separated values, the command read.csv() can be used.

Caution: It is always a good idea to check whether your file has been read in correctly using command such as these

```
head(x,n=yL) – prints first y lines of x
```

```
head(x,n=yL) – prints last y lines of x
```

`summary(x)` – this provides further information about the objects depending on its class. In the case of a numerical matrix, it supplies information on the measures of central tendency and spread

4.2. Reading built-in data

There are several datasets automatically supplied with R, for testing purposes. These datasets can be accessed with the command 'data().' The specific dataset can be loaded into R as follows…

```
> data(AirPassengers)
```

This creates an object called AirPassengers containing the built-in data.

Some R packages also contain built in data. This can be viewed using the data(package="package_name") command. The data can then be loaded in using the data(dataset_name, package="package_name") command.

4.3. Editing Data in Spreadsheet style

Data can be edited in a spreadsheet-like environment with the command

```
>edit(data_old) -> data_new
```

In this way, the final object is assigned to data_new. If the objective is to alter the original dataset, the command fix(data_old), which is equivalent to

```
>edit(data_old) -> data_old
```

4.4. Writing out data

Data can be written out into a text file using the write.table() command. Like the read.table() command, there are several options that can be used.

Append – if true, the output is appended to the file of the same name. If false, the existing file is destroyed and replaced by the new file (default is false)

Quote – This determines whether characters are surrounded by double quotes (default is true)

Sep – This determines what the field separator string is (default is whitespace “”)

Eol – This determines the character to be printed at the end of each line (default is “\n”)

Row.names – this determines whether the row names of x are to be written out

Col.names = this determine s whether the column names are to be written out

Write.csv() can be used to create a comma-separated-value file that is readable by Microsoft Excel.

Exercise 3 Reading and Writing Files In this exercise, we will explore reading and writing data into R	
Task 1 Reading data	**Step 1** The text file ‘expenditure.txt’ contains the personal expenditure of US citizens in the 1940s to 1960s. Open the file in Microsoft notepad to check the format, in particular whether it has a header row.
	Step 2 In R, read the file into an object called ‘x’ with the command _____
	Step 3 In R, find the standard deviation of Health and Medical Expenditure through all the years represented in the data with the command _____
	Step 4 Open the csv file ‘expenditure.csv’ in Microsoft Excel. Read it into R with the read.csv() command. Check that the resulting object is similar to the object read in from the text file.
Task 2 Writing data	**Step 1** The `Puromycin` data frame has 23 rows and 3 columns of the reaction velocity versus substrate concentration in an enzymatic reaction involving untreated cells or cells treated with Puromycin. There are 3 columns – substrate concentration, rate and state (treated vs untreated). Explore the structure of this dataset with the head() and summary() commands

	Step 2 Create a matrix with the rows of the puromycin dataframe that have 'treated' in the third column with the following command. _____
	Step 3 Write out this matrix into a text file with the following command _____ Open this text file in Microsoft Notepad and check that it contains what you expected.
	Step 4 Create a matrix with the rows of the puromycin dataframe that have a rate of greater than 100 counts/min/min with the following command _____
	Step 5 Write out this matrix into a csv file with the following command _____ Open this csv file in Microsoft Excel and check that it contains what you expected.

5 Scripts, Objects and the Workspace

5.1. Objects and the Workspace

The basic unit that is manipulated in R is an object. An object can be a variable, a function, or general structure built from different components. In an R session, objects are created and stored. They can be accessed with the ls() command.

To remove an object, the command rm() can be used. To clear the workspace of all objects, the command rm(list=ls()) can be used.

The objects used in a session (and the command history) can be stored in a file with the command save.image(). The objects can be reloaded to the workspace with the command load(). If one wishes to save specific objects, this can be performed with the save() command and a list of objects as the argument.

An example is given below.

```
> a<-1
> b<-2
> c<-3
> d<-4
> ls()
[1] "a" "b" "c" "d" ## this lists the objects in the current workspace
> save.image("my_workspace.RData") ## this saves all the objects in the
current workspace
> rm(list=ls()) ## this clears the workspace of all objects
> ls() ## this lists the objects in the workspace (none at present)
character(0)
> load("my_workspace.RData") ## this loads up the stored workspace file
> ls() ## the objects stored within the workspace file are loaded into the
current workspace
[1] "a" "b" "c" "d"
> rm(d) ## this removes the object 'd'
> ls()
[1] "a" "b" "c"
> save("a","b",file="my_objects.Rd") ## this selectively stores 2 objects -
a and b
> rm(list=ls())
> load("my_objects.Rd") ## this loads up the 2 objects stored in the Rd
file
> ls()
[1] "a" "b"
```

The R workspace has memory limits (dependent on the machine it is run on), which means that extremely large datasets may have to be analysed in chunks. The command memory.size() provides the total amount of memory used by R while the command memory.limit() provides the memory limit for the workspace. The memory limit can be altered by providing the command memory.limit () with an argument eg.

```
> memory.size() ## this reports the memory currently used
[1] 14.81
> memory.limit() ## this reports the memory limit of the workspace
```

```
[1] 3583
> memory.limit(3600) ## this changes the limit to the value of the argument supplied
[1] 3600
> memory.limit() ## this reports the new memory limit
[1] 3600
```

To end the session, the command q() is used. At this point, you will be prompted with a question 'Save Workspace Image?' If you click on yes, all objects in the workspace are written to a file called '.RData' and the command lines are written to a file called '.Rhistory'. When R is started from the same directory, both files are automatically loaded.

Caution: It is a good idea to store your workspaces separately according to project rather than '.Rhistory' or '.Rdata' since variables with common names eg. 'x' or 'foo' may take on different values for different projects, leading to mistaken identities. When saving a script file, the '.R' extension has to be added explicitly as R will not add it for you, unlike the addition of '.doc' by Microsoft word or '.xls' by Microsoft Excel.

5.2. Directories

When loading workspace files or script files, R has to be started within the directory that the files are stored. To check which directory R is in, one can use the command getwd(). To change directory within R, one can use the command setwd().

If one does not wish to change directory, an alternative is to use the full path to the file when loading it into R.

An example is given below.

```
> getwd() ## this prints the current directory that R is in
[1] "C:/Me/u0302066/Documents"
> setwd("C:/You/u0302066/Documents") ## this changes the directory
> getwd()
[1] "C:/You/u0302066/Documents"
> load("C:/Me/u0302066/Documents/my_workspace.RData") ## this loads a workspace located in a different directory by stating the full path to the workspace
> ls()
[1] "a" "b" "c" "d"
```

5.3. Running scripts

So far, we have been running commands by typing them into the R editor and clicking on 'run selection' or by typing them directly in the console. Commands typed into the R editor can also be stored in a file, with the extension '.R' and run at the console with the command 'source'. For example

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

will run all commands in the file 'my_commands.R'

The function 'sink' can be used to divert all output from the console into a file. For example,

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

will divert all subsequent output to an external file 'my_record.lis' and the command sink() will restore it to the console again.

Exercise 4 Scripts, Objects and Workspaces In this exercise, we will explore scripts, objects and workspaces in R	
Task 1 Writing and executing scripts	**Step 1** Click on File|New script. This opens up a box titled 'Untitled – R editor'.
	Step 2 Type into the box rm(list=ls()) a<-1 b<-2 ls()
	Step 3 Execute the command by highlighting it then right click | run line or selection. What output do you get? _____
	Step 4 In the R editor, type in a command that creates a new variable 'sum_of_a_and_b' by adding a and b together _____
	Step 5 Remove all objects in the workspace with the command _____
	Step 6 Save the script file as 'my_script_file.R' by doing File|Save as. Make sure that your cursor is in the R editor and not in the R console when you do this.
	Step 7 Execute the script file by typing source("my_script_file.R") in the R console
Task 2 Saving and loading objects	**Step 1** Create a 2 new variables – 'difference_of_a_and_b', 'product_of_a_and_b'.
	Step 2 Save these 2 objects in a file called 'my_objects.Rd' with the following command _____
	Step 3 Quit R without saving the workspace.

	Step 4 Open a new session of R and load these 2 objects into the workspace with the following command _____

6 Control Statements and Loops

6.1. If/Else and Ifelse

The if/else syntax is as follows

> if (condition) expr_1 else expr_2

An example is given below.

```
> y<-1
> if(y>2) x<-3 else x <-4
> x
[1] 4
```

The conditional expression often contains the 'and' or 'or' operators, which are represented by '&&' and '||' (or '&' and '|' described later). Here is an example of how to use them.

```
> z<-1
> y<-1
> if(y<2&&z<2) x<-3 else x <-4
> x
[1] 3
```

The above statement can also be written in the ifelse syntax as follows

```
> ifelse(y<2&&z<2,3,4)->x
> x
[1] 3
```

Caution: The 'and' and 'or' operators can take 2 forms. 'And' can be represented by '&' or '&&'. 'Or' can be represented by '|' or '||'. The 'short circuit' form of the operators are '&&' and '||' respectively. These only evaluate the second argument if necessary, unlike '&' and '|', which evaluate both arguments. Additionally, if the arguments are vectors, '&' and '|' returns a vector of 'TRUE' and 'FALSE' while '&&' and '||' return a single output based on the first element of the vector (in addition to a warning). With that in mind, how should we decide which to use? For most conditional testing, the arguments are single elements rather than vectors, hence && can be used safely. The warning would come in useful in case you have unintentionally included a vector as an argument.

6.2. Loops

This can be achieved with 'for' or 'while'.

The syntax is as follows...

> for (i in n:m){expr} where n is the first value and m is the last value of i for which the expression within curly brackets should be evaluated

An example is given below

```
> my_results <- vector() ## this creates an empty vector to store results
> my_matrix <- matrix(c(1,2,3, 7,8,9), nrow = 2, ncol=3, byrow=TRUE)
> my_matrix ## this generates a matrix for testing purposes
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    7    8    9
> for(i in 1:3){ ##this loops through the numbers 1 to 3
+ mean(my_matrix[,i])->my_results[i] ## this finds the mean of each column
in the matrix
+ print(i)} ## this prints the counter so that we know which column we are
up to
[1] 1
[1] 2
[1] 3
> my_results
[1] 4 5 6
```

Printing the counter is useful especially when running very long loops, so that we are able to monitor and make sure that the loop is still running and has not hanged itself.

The while syntax for the above statement is as follows

```
> i<-1
> while(i<4){
+ mean(my_matrix[,i])->my_results[i]
+ print(i)
+ i<-i+1}
[1] 1
[1] 2
[1] 3
> my_results
[1] 4 5 6
```

As you might have realised by now, both loops basically perform the same function as the colMeans() command. In R, there are many built-in functions that perform complicated operations. It is always good to do a search using the help function and look at related commands on the help page to see if you can avoid writing your own functions or loops.

An alternative to writing loops is to use the 'apply' command as it runs faster than loops.

The syntax for the apply() command is

apply(X,margin,fun) where the function 'fun' is applied to the matrix 'X' either row-wise (in which case margin=1) or column-wise (in which case margin=2)

The syntax for the above loops would be

apply(my_matrix,2,mean) -> my_result

Exercise 5 Loops and If/Else statements In this exercise, we will explore loops and if/else statements in R	
Task 1 Loops	**Step 1** The dataset 'airquality' contains daily airquality measurements in New York in 1973. It is a dataframe with observation of 6 variables – mean ozone in parts perbillion, solar radiation, wind in miles perhour, temperature in degrees fahrenhait, month and day of month. View the first few lines of the dataset with the command head().
	Step 2 For each day, we would like to create a hypothetical 'wind-temperature' score calculated by Wind *2 + temperature of that day – temperature of the next day Write a loop to create this score for each day and store it in a vector. It is not necessary to calculate this value for the last day in the dataset. You should end up with a vector the length of the number of rows in the data frame minus one. Hint – Let the loop variable i be the row-number
Task 2 If else statements	**Step 1** For each day, we would like to create a conditional score based on temperature and solar radiation. If the solar radiation is higher than 150 units and the temperature is higher than 60 degrees fahrenhait, the score should be 1. If not, it should be 0. Write an ifelse statement to calculate this score for the all days in the dataset. You should end up with a vector of zeros and ones. The vector should be the same length as the number of rows in the dataset.

7 Working with text in R

7.1. Characters and Strings

So far the functions that we have described mainly discuss numerical types. In this chapter, we will discuss several common and useful functions for working with text in R. This will be useful for working with charts and graphs as described in the next chapter.

A character is the single unit of text. Characters can be joined together to form text strings eg. “This is a text string”

Text string can be joined together to form vectors of text strings. Of course, characters can also be joined together to form vectors of characters. However, joining characters to form a vector is not equivalent to joining characters together to form a text string. To do the latter, one needs the ‘paste’ command (described later).

```
> "I am a text string" -> text_string_1
> "I am another text string" -> text_string_2 ## creates 2 text strings
> c(text_string_1,text_string_2)->text_string_vector ## joins 2 text
strings into a vector
> text_string_vector ## prints out the vector
[1] "I am a text string"       "I am another text string"
> text_string_vector[2] ## accesses the second element of the vector
[1] "I am another text string"
> c(text_string_vector,"I am yet another text string") ->
text_string_vector ## adds a third text string element to the text_string
vector
> text_string_vector
[1] "I am a text string"           "I am another text string"
[3] "I am yet another text string"
```

As can be seen above, adding another text string to the vector of text string does not join text strings together. To do that, we need the paste command. An example is given below.

```
> paste(text_string_1,text_string_2,sep=" ") -> text_string_3
> text_string_3
[1] "I am a text string I am another text string"
```

The separator can be changed with the ‘sep’ option.

```
> paste(text_string_1,text_string_2,sep=" and ") -> text_string_3
> text_string_3
[1] "I am a text string and I am another text string"
```

Like text strings, characters can either be pasted together or joined together to form a vector

```
> paste("t","e","x","t","s","t","r","i","n","g",sep="") -> my_text_string
> c("t","e","x","t","s","t","r","i","n","g") -> my_text_vector
> my_text_string
[1] "textstring"
> my_text_vector
 [1] "t" "e" "x" "t" "s" "t" "r" "i" "n" "g"
```

When files are read into R, numbers are sometimes read in as characters. These can be converted into numbers with the as.numeric() command. If there are any characters in the vector, they will be converted to "NA".

7.2. Useful commands

7.2.1. Count number of characters in the string

nchar() reports how many characters there are in the string

```
> my_text_string
[1] "textstring"

> nchar(my_text_string)
[1] 10
```

7.2.2. Split the string

The substr() command splits the text string with the following syntax substr(text,start,stop)

```
> substr(my_text_string,1,4)
[1] "text"
```

The strsplit() command splits the text string at a delimiter with the following syntax strsplit(text,delimiter).

```
> strsplit("why is are there so many ones at the end of this line","
")[[1]][1]
[1] "why"
```

This begs the question – why is there are need for the [[1]]?

This is because strsplit() operates on vectors and splits each element of the vector along the delimiter, then creates a list with each element of the vector being each element of the list. Hence, if there were a vector of text strings, it would be split into a list of text strings as follows

```
> text_vector<-(c("I bet R","is having a laugh"))
> strsplit(text_vector," ")
[[1]]
[1] "I"   "bet" "R"

[[2]]
[1] "is"     "having" "a"      "laugh"
```

To access the word "laugh", we would do

```
> strsplit(text_vector," ")[[2]][4]
[1] "laugh"
```

7.2.3. Search text strings for a word

Vectors of text strings can be searched for a word. The syntax is as follows grep("search_term",text).

```
> grep("laugh",text_vector)
[1] 2
```

This reports that the word laugh is in the second text string of the vector of text strings

Exercise 6 Text Manipulations in R In this exercise, we will practise manipulating text in R	
Task 1 Working with text vectors and text strings	**Step 1** The file 'text_sample.Rd' contains 2 objects, 'txt1' and 'txt2'. Load the file into R with the command load().
	Step 2 Create a new vector by concatenating elements of each vector together. The new vector should read "my" "dog" "loves" "my" "cat"
	Step 3 Paste the elements of this new vector together to create a text string that reads "my dog loves my cat"
Task 2 Splitting text	**Step 1** Split the new string "my dog loves my cat" with the whitespace delimiter " "
	Step 2 Extract the second word "dog" from the output of step 1
	Step 3 Extract the word 'do' from the word 'dog' with the substr() command

8 Graphs and Charts

There are 2 different types of plotting commands – high level and low level.
High level commands create a new plot on the graphics device, low level commands add information to an existing plot.

8.1. High level plotting commands

These always start a new plot, erasing anything already on the graphics device. Axes, labels and titles are created with the automatic default settings.

8.1.1. Generic plot() command

The plot() function is the most commonly used graphical function in R. The type of plot that results depends on the arguments supplied.

If plot(x,y) is typed in, a scatterplot of y against x is produced if both are vectors.

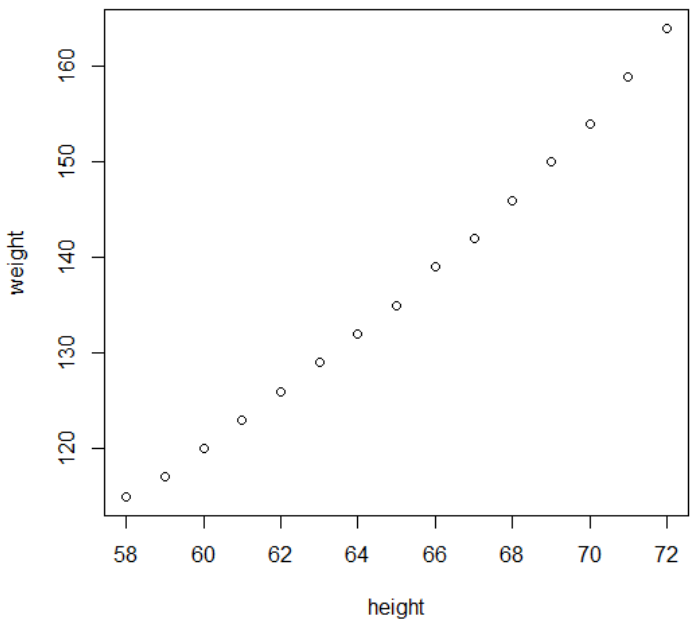
If plot (x) is typed in and x is a vector, the values of x will be plotted against their index. If x is a matrix with 2 columns, the first column will be plotted against the second column.

Other formats include plot(x~y), plot(f,y) where f is a factor object and plot(~expr) etc. These can detailed in the help pages on plot.

The women dataset gives the average heights and weights for American women aged 30–39. It consists of a matrix with the first column being height and the second column being weight.

To produce a scatterplot representing the relationship between height and weight, we can use the following command

```
> plot(women)
```



8.1.2. Boxplot

These can be performed with the boxplot() command

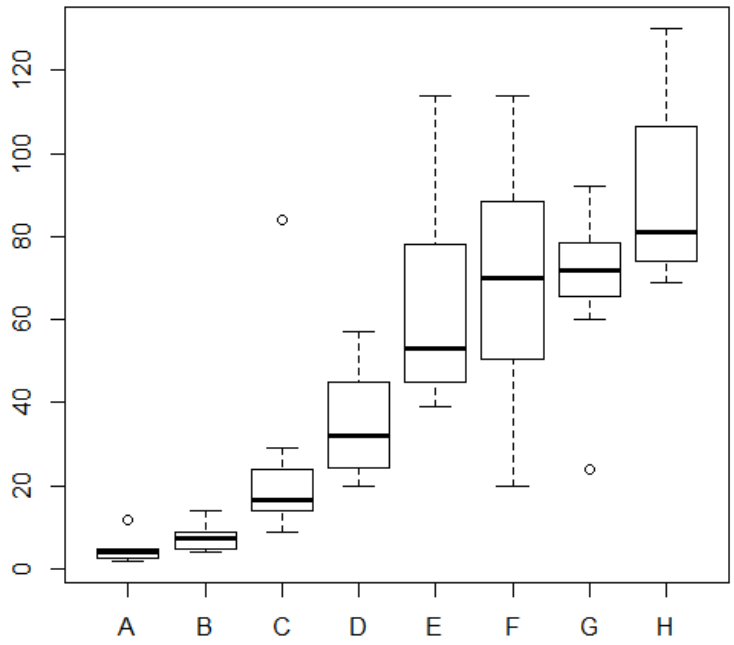
The OrchardSprays dataset in R contains information on a study on the effect of different concentrations of lime sulphur in repelling honey bee. There are 64 observations on 4 variables – treatment groups (these different groups are represented by alphabets, with A being the highest concentration of lime sulphur, G being the lowest concentration of lime sulphur, H being no lime sulphur at all), response (as measured by decrease in concentration of sugar solution that the lime sulphur is dissolved in), row position and column position (latin square design)

Boxplots express the relationship between 2 variables, one continuous and one discrete. They represent a five point summary - the smallest observation (sample minimum), lower quartile (Q1), median (Q2), upper quartile (Q3), and largest observation (sample maximum).

In this example, we can represent the relationship between response and treatment group by

```
> boxplot(decrease~treatment,data=OrchardSprays)
```

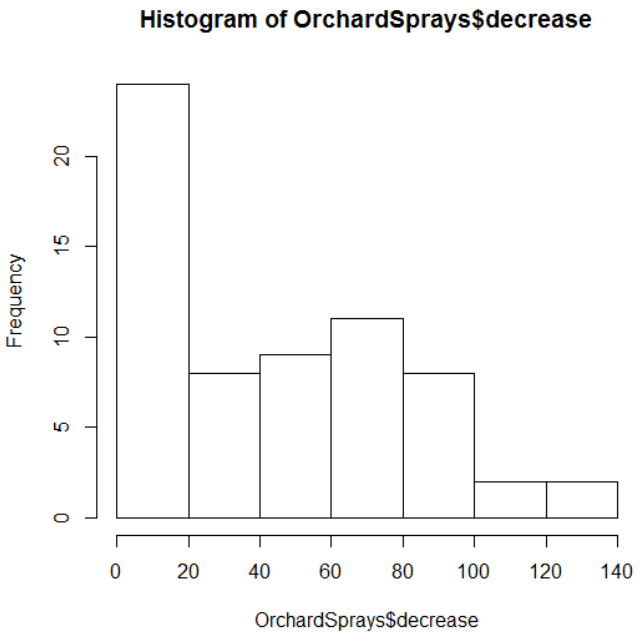
The resultant boxplot shows a clear relationship between response to treatment and concentration of lime sulphur in the sugar solution, which differs between the groups.



8.1.3. Histogram

A histogram groups continuous variables into categories and plots them in terms of frequency. The bins of a histogram can be adjusted according to the distribution of data with the 'breaks' option. In this example, we can plot a histogram of the treatment response

```
> hist(OrchardSprays$decrease)
```



There are many other plotting functions eg.

qqplot() – does a quantile-quantile plot

persp(x,y,z) – draws a 3D contour surface representing the relationship between 3 variables

These different plot functions are detailed in the document http://cran.r-project.org/doc/manuals/R-intro.pdf

8.1.4. Options

Various additional details can be added to the plot

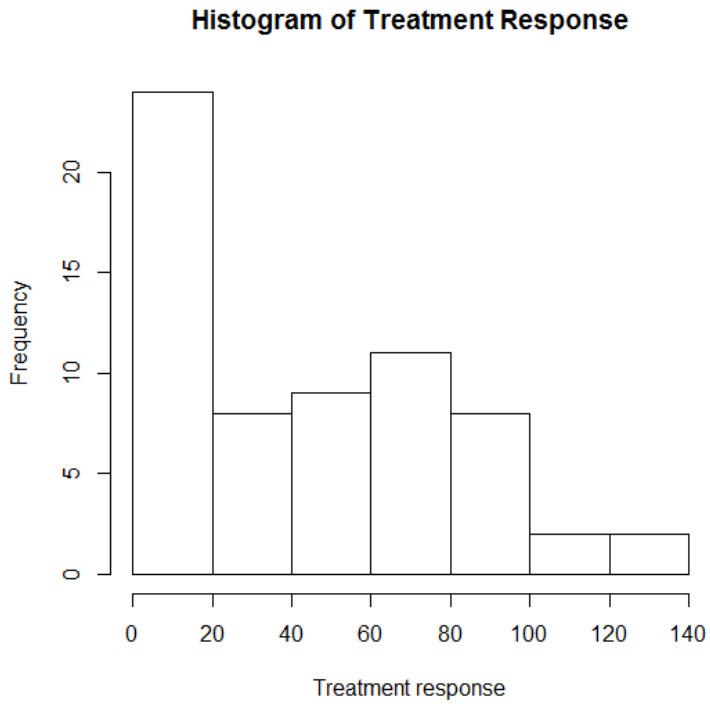
To add a title, use the 'main' option

To change the axis labels, use the 'xlab' and 'ylab' options

To change the axis margins, use the 'xlim' and 'ylim' options

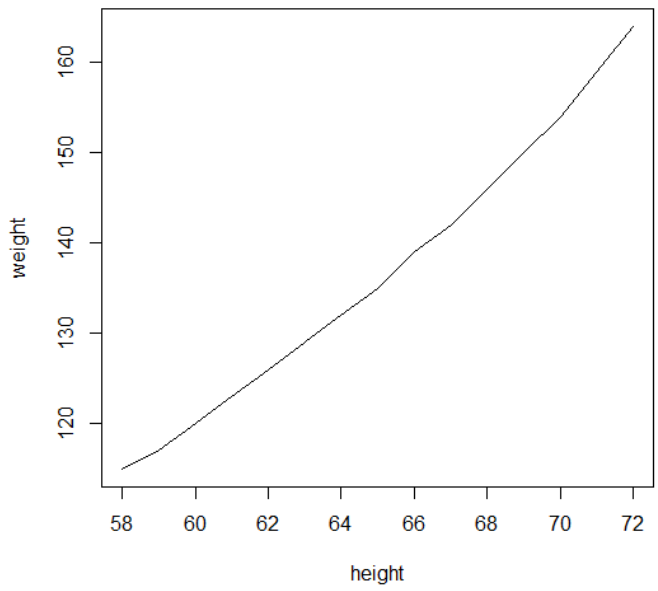
For example, the plot above can be modified with the following command

```
> hist(OrchardSprays$decrease, main="Histogram of Treatment Response",
xlab="Treatment response")
```

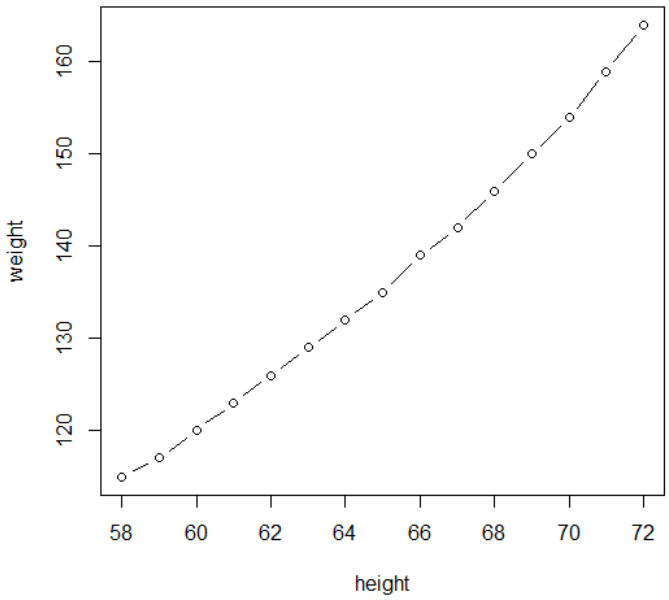


The option 'type' can be used to modify the type of graph produced. Type="p" is the default and results in individual points. Type="l" plots lines and type="b" plots points connected by lines. For example, in the case of women's height and weight, the resultant plots are as follows

```
> plot(women,type="l")
```



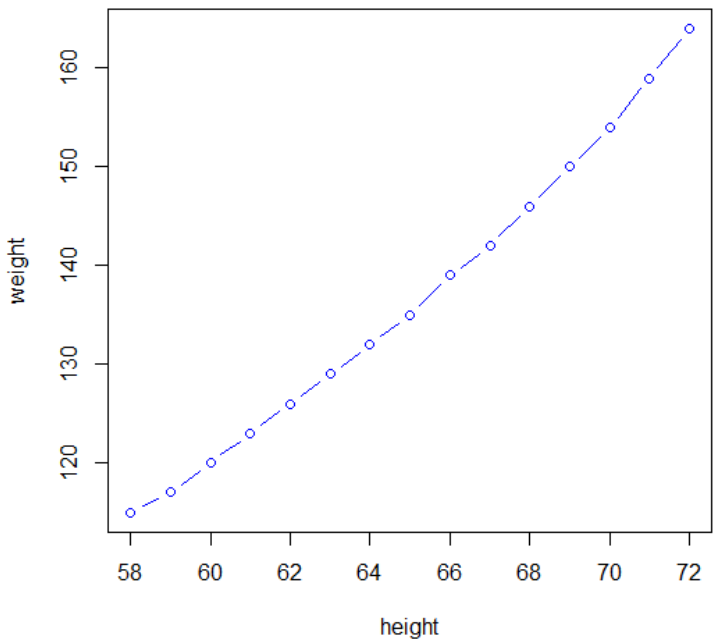
```
> plot(women,type="b")
```



The option ‘col’ can be used to change the colour of the graph.

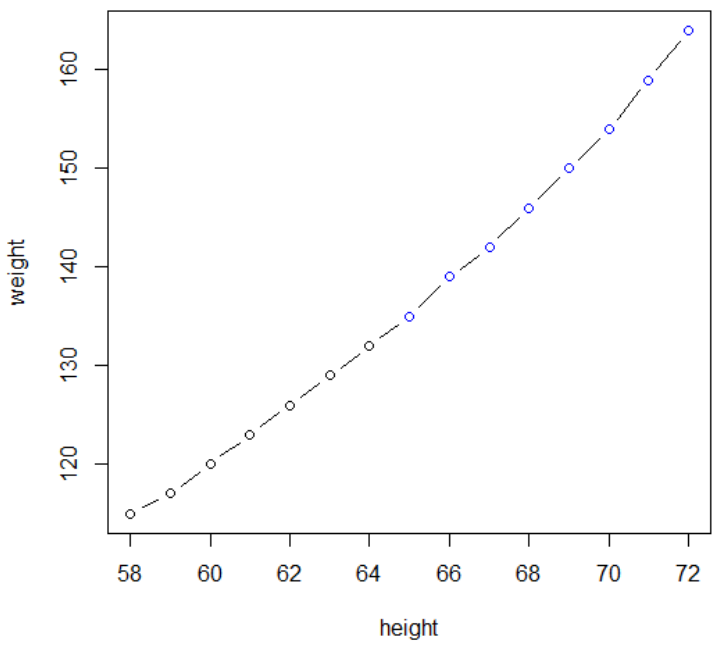
For example, this command colours all the points blue.

```
> plot(women,type="b",col="blue")
```



The points can also be selectively coloured by making the colour argument a vector as follows

```
> plot(women,type="b",col=c(rep("black",7),rep("blue",8)))
```



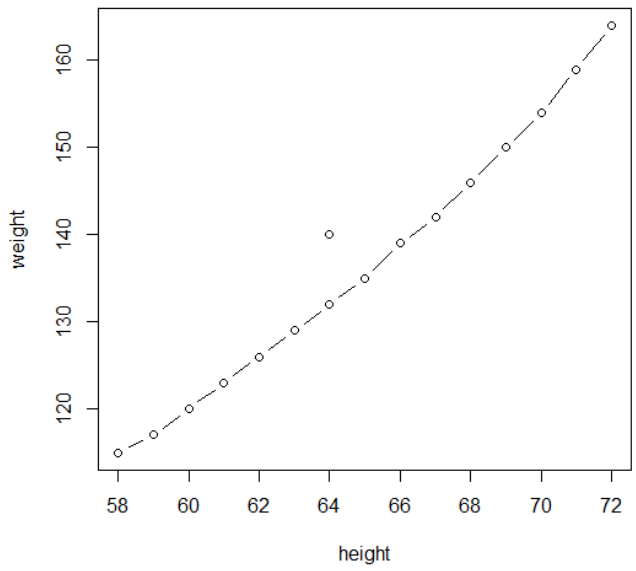
8.2. Low Level Plotting Functions

To add additional features to plots, these commands can be used. They do not erase the current plot.

Points(x,y) adds points to the plot. lines(x,y) adds lines to the plot

For example, the command below adds the point with the corresponding coordinates to the plot

```
> points(64,140)
```



Some other options for modifying the graph include

Text() – adds text to the graph

Legend() – adds a legend to the graph

Abline() – adds a line in the format y=ax+b

Polygon() – adds a polygon

A full list can be found in the 'Introduction to R' manual on the R website.

Exercise 7 Plotting functions in R In this exercise, we will explore plotting functions in R	
Task 1 Drawing line graphs	**Step 1** The dataset 'longley' contains economic variables that observed yearly from 1947 to 1962. Plot the GNP on the y axis and the year on the x axis. Use type="b" to create a plot with both the points and a line joining them. Label the axes accordingly. Give your plot an appropriate title.
	Step 2 Using the command points(), add the figures for 'Unemployed' to the plot. Use type="b" to create a plot with both points and lines joining them. Colour the points blue with the option 'col'.
	Step 3 What problem do you notice? Modify the graph limits with the 'ylim' optionand replot both lines.
Task 2 Drawing histograms	**Step 1** Draw a histogram of the variable 'Employed'. Give it an appropriate title and axes labels. Does it follow a Gaussian distribution?

9 Statistics with R

9.1. Measures of Central Tendency and Spread

These can be obtained with the following commands, which calculate the required measure from the vector argument supplied

```
> test_vector <- seq(10)
> test_vector
 [1]  1  2  3  4  5  6  7  8  9 10
> mean(test_vector)
[1] 5.5
> median(test_vector)
[1] 5.5
> sd(test_vector)
[1] 3.02765
> max(test_vector)
[1] 10
> min(test_vector)
[1] 1
> summary(test_vector)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1.00    3.25    5.50    5.50    7.75   10.00
```

9.2. Tests for continuous vs discrete variables

9.2.1. 2 groups

With normally distributed measurements in 2 groups, one can measure the relationship between continuous and discrete variables with a parametric paired or unpaired t test with the command t.test().

This test has a few options. The default setting is an unpaired test but this can be changed with the option (paired=FALSE). The default setting is a 2 tailed test but this can be altered with the option (alternative=LESS or alternative=GREATER).

When the data is not normally distributed, one would use the non-parametric Wilcoxon test with the commandwilcox.test, with the options paired=TRUE(Wilcoxon signed rank test), or paired=FALSE(Wilcoxon rank sum test, otherwise known as Mann Whitney U test).

9.2.2. 3 or more groups

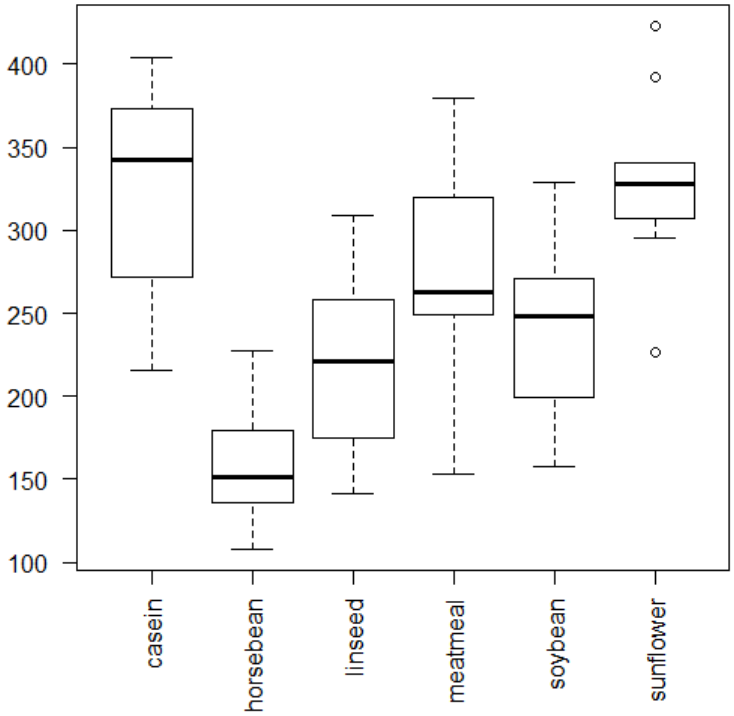
When the data is distributed normally, one can use the parametric Analysis of Variance test with the command aov(). When the data is not normally distributed, one can use the non-parametric Kruskal-Wallis test with the command kruskal.test().

The following example is from the dataset called 'chickwts'. This describes an experiment to compare the effectiveness of different types of feed supplements on the growth rate of chickens. This dataset consists of 71 observations on 2 variables. The first variable is chick weight after 6 weeks and the second variable is the grouping based on the type of feed. There are 6 types of feeds, which can be accessed with the following command.

```
> summary(chickwts)
     weight             feed
 Min.   :108.0   casein   :12
 1st Qu.:204.5   horsebean:10
 Median :258.0   linseed  :12
 Mean   :261.3   meatmeal :11
 3rd Qu.:323.5   soybean  :14
 Max.   :423.0   sunflower:12
```

Let's say we wish to find out how feed affects chick weight, one quick way to do this is to visualise the data with a boxplot.

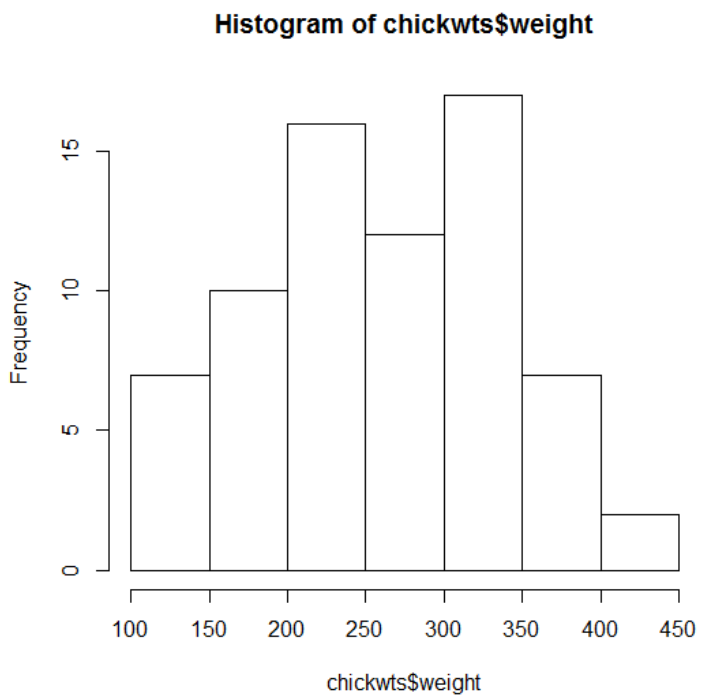
```
> boxplot(weight~feed,data=chickwts,las=2) ## las=2 turns the x labs
horizontally
```



Let's say we wish to find out whether chicks fed with casein have a significantly higher weight than chicks fed with horsebean.

The first step is to check whether the data is normally distributed. This can be done by plotting a histogram of the weights.

```
> hist(chickwts$weight)
```



Since the data is normally distributed, we can use the unpaired t test as follows

```
>t.test(chickwts$weight[which(chickwts$feed=="casein")],chickwts$weight[whi
ch(chickwts$feed=="horsebean")])


        Welch Two Sample t-test


data:  chickwts$weight[which(chickwts$feed == "casein")] and
chickwts$weight[which(chickwts$feed == "horsebean")]
t = 7.3423, df = 18.36, p-value = 7.21e-07
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 116.6982 210.0685
sample estimates:
mean of x mean of y
 323.5833  160.2000
```

The output provides more information about the results of the t test. The t statistic is shown, together with the degrees of freedom, 95% confidence interval and p value. The means of each of group are shown. When var.equal = FALSE (by default), the welch 2 sample test is

used, which is an adaptation of the Student's t test adapted for unequal variances. If var.equal=TRUE, the variances are pooled.

If we wish to conduct an analysis of variance amongst all the feed groups, we can use the following

```
> summary(aov(chickwts$weight~chickwts$feed))
              Df Sum Sq Mean Sq F value   Pr(>F)
chickwts$feed  5 231129   46226   15.37 5.94e-10 ***
Residuals     65 195556    3009
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

This tells us that there are significant differences between at least 2 groups. However, we do not know which pairs of groups have differences and which do not. To do this, we can perform posthoc tests.

```
>pairwise.t.test(chickwts$weight,chickwts$feed,p.adjust.method="none",pool.
sd=FALSE,var.equal=FALSE)

        Pairwise comparisons using t tests with non-pooled SD

data:  chickwts$weight and chickwts$feed

          casein  horsebean linseed meatmeal soybean
horsebean 7.2e-07 -         -       -        -
linseed   0.00026 0.00687   -       -        -
meatmeal  0.09866 0.00011   0.02933 -        -
soybean   0.00352 0.00016   0.19799 0.22523  -
sunflower 0.82151 1.7e-08   2.4e-05 0.04441  0.00043
```

9.3. Tests for discrete vs discrete variables

9.3.1. Chi Square Test

This is used when comparing 2 discrete variables to measure whether the observed proportions are significantly different from the null hypothesis. There should be more than 5 observations for each cell. The R command is chisq.test().

9.3.2. Fisher's Exact Test

This is used when there are fewer than 5 observations for each cell. The command is fisher.test()

In this example, we create a 2x2 matrix of the number of male and female students in a 2 classes – physics and biology. We would like to find out whether there is a statistical difference in the proportion of males compared to females in each subject class. Since there are more than 5 subjects in each category, we should use the chi square test.

```
> gender_subject <- matrix(c(23,45, 25,57), nrow = 2, ncol=2, byrow=TRUE)
> rownames(gender_subject) <- c("males","females")
> colnames(gender_subject) <- c("physics","biology")
> gender_subject
        physics biology
males        23      45
females      25      57
> chisq.test(gender_subject)

        Pearson's Chi-squared test with Yates' continuity correction
data:  gender_subject
X-squared = 0.0677, df = 1, p-value = 0.7947
```

9.4. Tests for continuous vs continuous variables

9.4.1. Pearson Correlation

This is used when both variables are normally distributed. The R command is cor.test(x,y,method=”pearson”)

9.4.2. Spearman Correlation

This is used when one or both variables are not normally distributed. The R command is cor.test(x,y,method=”spearman”)

The example below comes from the ‘women’ dataset, which documents the heights and weights of a sample of American women.

The first step is to check whether both height and weight variables are normally distributed. This can be performed as follows either with a histogram (previous example) or with the Shapiro wilk test.

```
> shapiro.test(women[,1])

        Shapiro-Wilk normality test

data:  women[, 1]
W = 0.9636, p-value = 0.7545

> shapiro.test(women[,2])
```

```
        Shapiro-Wilk normality test

data:  women[, 2]
W = 0.9604, p-value = 0.6986
```

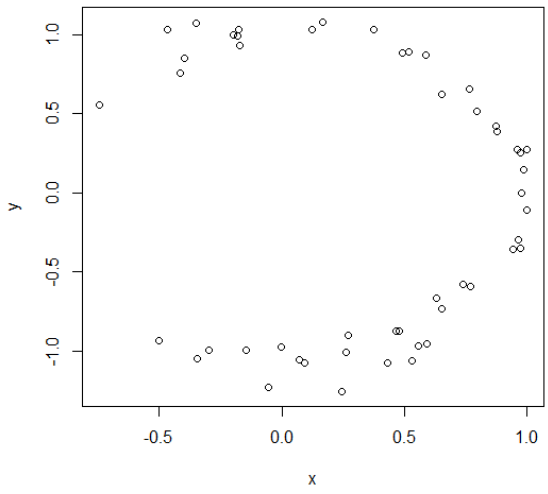
Since both height and weight do not show significant deviation from normality, Pearson correlation can be used.

```
> cor.test(women[,1],women[,2],method="pearson")

        Pearson's product-moment correlation

data:  women[, 1] and women[, 2]
t = 37.8553, df = 13, p-value = 1.088e-14
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.9860970 0.9985447
sample estimates:
      cor
0.9954948
```

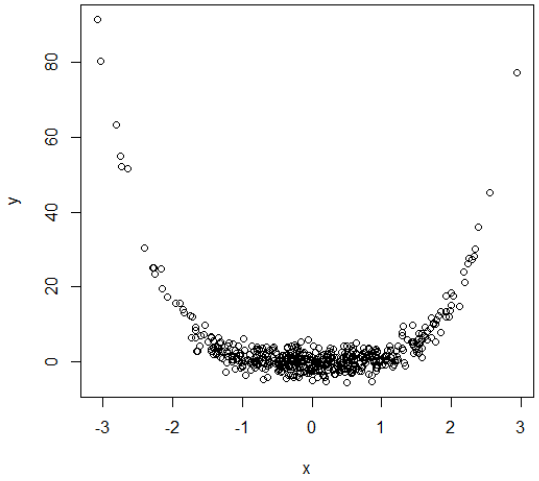
The relevant statistics can be extracted as follows

```
> cor.test(women[,1],women[,2],method="pearson")$p.value
[1] 1.088019e-14
> cor.test(women[,1],women[,2],method="pearson")$estimate
      cor
0.9954948
```

Caution: Correlation coefficient of zero does not necessarily mean that there is no relationship between the 2 variables. The scatterplots below demonstrate non-linear relationships between 2 variables



Pearson correlation = -0.09



Pearson correlation = -0.08

9.5. Regression

Fitting linear models can be performed with the command lm(). The example below examines regression of weight on height for the R dataset 'women' described above.

```
> summary(lm(women[,1]~women[,2]))

Call:
lm(formula = women[, 1] ~ women[, 2])

Residuals:
     Min       1Q   Median       3Q      Max
-0.83233 -0.26249  0.08314  0.34353  0.49790

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 25.723456   1.043746   24.64 2.68e-12 ***
women[, 2]   0.287249   0.007588   37.85 1.09e-14 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.44 on 13 degrees of freedom
Multiple R-squared: 0.991,      Adjusted R-squared: 0.9903
F-statistic:  1433 on 1 and 13 DF,  p-value: 1.091e-14
```

The relevant values can be extracted as follows

P value : summary(lm(women[,1]~women[,2]))$coeff[2,4]

Regression coefficient: summary(lm(women[,1]~women[,2]))$coeff[2,1]

Residuals: summary(lm(women[,1]~women[,2]))$residuals

Other regression models can be fitted with the glm function using the 'link' option. The various options are given below. They can also be found on the glm function help page.

```
binomial(link = "logit")
gaussian(link = "identity")
Gamma(link = "inverse")
inverse.gaussian(link = "1/mu^2")
poisson(link = "log")
quasi(link = "identity", variance = "constant")
quasibinomial(link = "logit")
quasipoisson(link = "log")
```

9.6. Probability Distributions

R has a set of built-in functions related to probability distributions.

These functions can evaluate the

- probability density function (prefix the name with 'd')

- cumulative distribution function (prefix the name with 'p')

- quantile function (prefix the name with 'q')

- simulate from the distribution (prefix the name with 'r')

where the name refers to a set of R names eg. 'binom' (binomial), 'chisq' (chi square), 'hyper' (hypergeometric) etc. The full list can be found on the official R manual accessible here

http://cran.r-project.org/doc/manuals/R-intro.pdf

9.7. Other Statistical Features

R has a large compendium of statistical features, which are constantly being added by users in the form of new packages (later chapter).

Some of the commonly used statistical features include

- Maximum likelihood models – commands depend on the model being fitted. More information can be found here http://www.stat.umn.edu/geyer/5931/mle/mle.pdf
- Mixed models – the nlme package is recommended, with the functions lme() and nlme()
- Local approximating regression – the loess() function performs a non-parametric regression
- Principal Components Analysis/Singular Value Decomposition – The former can be performed with the prcomp() or princomp() function. The latter can be performed with the svd() function. Be careful of whether rows/columns are standardised or centred
- Robust regression – Many functions are contained in the MASS package
- Clustering – various types of clustering functions exist eg. hclust() does hierarchical clustering
- Tree-based models – these can be found in packages rpart and tree

- Machine learning – various unsupervised learning algorithms can be easily implemented in R eg. support vector machines with the function svm(), neural networks with the package nnet(), random forests with the package randomForest

9.8. Packages

Many of these statistical features are found in packages developed by users. These packages can be installed within the R environment with the command

> install.packages("package_name")

Packages have been developed for statistical analysis in different disciplines:

1. Bioinformatics

Many of the packages for bioinformatics are embedded with the 'bioconductor' environment. Bioconductor can be downloaded from http://www.bioconductor.org/

2. Social Sciences

Most of the functions needed for social sciences can be found in the base packages. Further details are here http://cran.r-project.org/web/views/SocialSciences.html

3. Financial Engineering

Many of the functions can be found in this suite of packages 'R/Rmetrics'

These are but a few examples. The CRAN repository contains a full list of packages that can be downloaded. Packages that not within that repository have to be downloaded and installed manually with by clicking on **Packages|Install packages from local zip file**

Exercise 8 Statistics in R In this exercise, we will explore statistics in R	
Task 1 Comparing 2 groups	**Step 1** The text file 'district.txt' contains the ages of residents in 2 different districts. Plot the ages as a histogram to asses normality.
	Step 2 Use the appropriate test to check whether there is a statistically significant difference between the ages of residents in the 2 districts.
Task 2 Correlation	**Step 1** The file 'ageweight.txt' contains information about the age and weights of participants in a clinical study. There are 2 groups of participants – healthy controls and patients with carpal tunnel syndrome. Plot the age variable vs the weight variable on a graph to assess if the relationship is linear.
	Step 2 Create histograms of ages and weights to assess whether these variables have a normal distribution. What correlation test is appropriate for this situation?
	Step 3 Perform the correlation test on the whole dataset
	Step 4 Perform the correlation test separately on patients and on controls

10 Writing your own functions

When you need to perform a repeated function in R, a function can be written to perform this. The syntax to write a function is

```
my_function <- functionx(x){
commands(x) -> y
return(y)}
```

Taking the example in the 'loops' exercise

The dataset in question is 'airquality'

```
> head(airquality)
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
```

For each day, we would like to create a conditional score based on temperature and solar radiation. If the solar radiation is higher than 150 units and the temperature is higher than 60 degrees fahrenhait, the score should be 1. If not, it should be 0. Write an ifelse statement to calculate this score for the all days in the dataset. You should end up with a vector of zeros and ones. The vector should be the same length as the number of rows in the dataset.

Instead of writing a loop, we can write a function and apply it to the matrix. The argument to the function would be row of the matrix

The function would be written as follows

```
calc_score <- function(x){
ifelse(x[2]>150|x[4]>60,1,0) -> y
return(y)}
```

The function can then be applied to the matrix as follows

```
apply(airquality, 1, calc_score) -> result
```

11 References

The manual titled “An Introduction to R” located on the official R website at http://cran.r-project.org/doc/manuals/R-intro.pdf is the main reference used in the creation of this document. Other references, with hyperlinks, are documented in the relevant text passages.