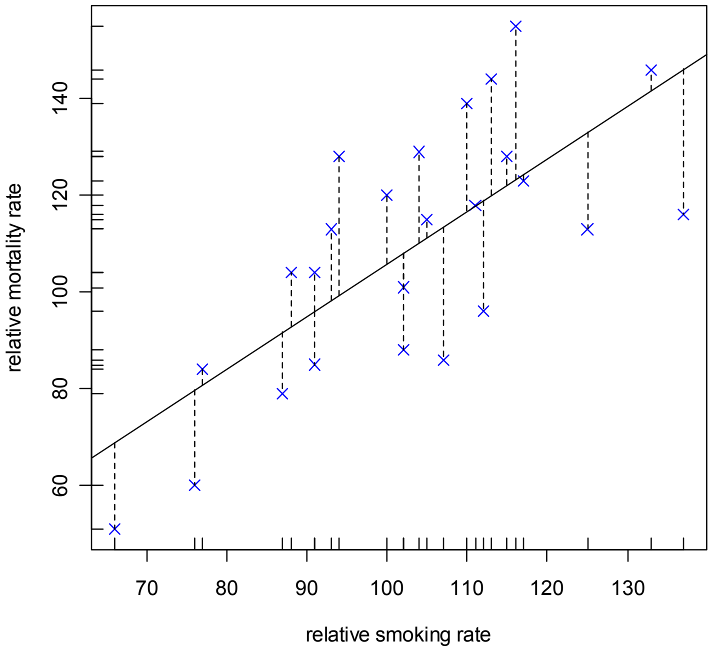
An Introduction to R

18th – 19th January 2007

University of Aberdeen



Alex Douglas & Janine Illian

Contents

1.0 Introduction

The purpose of this two day workshop and accompanying guide is to introduce the uninitiated to the use of R, an interactive environment for statistical computing. R itself is not difficult to learn, but just like any new language the initial learning curve is a little steep and you will need to use it frequently otherwise it's easy to forget. Our suggestion to you, is that whilst you are getting to grips with R, uninstall any other statistics software you have on your computer and only use R. This will hopefully remove the temptation to 'just do it quickly' in a more familiar environment and consequently slow down your learning of R. Believe us, anything you can do in your existing statistics software package you can do in R (often better and more efficiently).

A few notes about the workshop and guide. We have tried to simplify the content of this course as much as possible and have based it on our own personal experience. It is not intended as a complete R manual or an introductory statistics course, although we will be using some basic statistics to highlight R's capabilities. Our aim is to help you climb the initial learning curve and provide you with the basic skills to enable you to further your experience in using R. We have included a number of practical exercises for you to work through during the workshop and encourage you complete these in your own time - you certainly won't learn how to use R by watching other people do it. We would also encourage you to practice what you have learnt during the workshop on your own data and also work through one of the many freely available introductory texts on R once the workshop is over.

Good luck and have fun.

1.1 What is R?

R is a statistical analysis system initially created by Ross Ihaka and Robert Gentleman in 1996. It can be regarded as an implementation of the S language, which was developed at Bell Laboratories by Rick Becker, John Chambers and Allan Wilks. The S language also forms the basis of the S-PLUS systems. To find the difference between R and S, go to

http://cran.r-project.org/doc/FAQ/R-FAQ.html.

R can be used both as a programming language, and as a software package which you can use to manipulate your data, perform calculations, conduct statistical analyses and display graphics. Some advantages of using R include

- R is open source and freely available.
- R has an extensive and coherent set of tools for statistical analysis.

- R has an extensive and highly flexible graphical facility capable of producing publication quality figures.
- R has an expanding set of publicly available 'packages' of routines for special or unusual analyses.
- R has an extensive support network with numerous online and freely available documents.

Some disadvantages include

- R has a limited graphical interface which may make it harder to learn at the outset.
- R has no commercial support (but see above).
- The command language is a programming language so users must be aware of syntax issues

Initially, for those who haven't used R before, R may seem rather daunting and complex. Whilst the initial learning curve is admittedly a little steep, R offers the user a degree of flexibility and control not usually available in other more traditional statistical packages. Most importantly, learning to use R will change the way you think about statistics and provide you with a more holistic and coherent approach to your data analysis. Undoubtedly, time invested in learning R now will be more than repaid at a later date.

1.2 Installing R in Windows

Most people using this guide will be running R on a computer with a Windows operating system. Information on installing and running R on a variety of other platforms can be found at

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

R can be downloaded as a self-extracting file from the Comprehensive R Archive Network (CRAN) website at

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

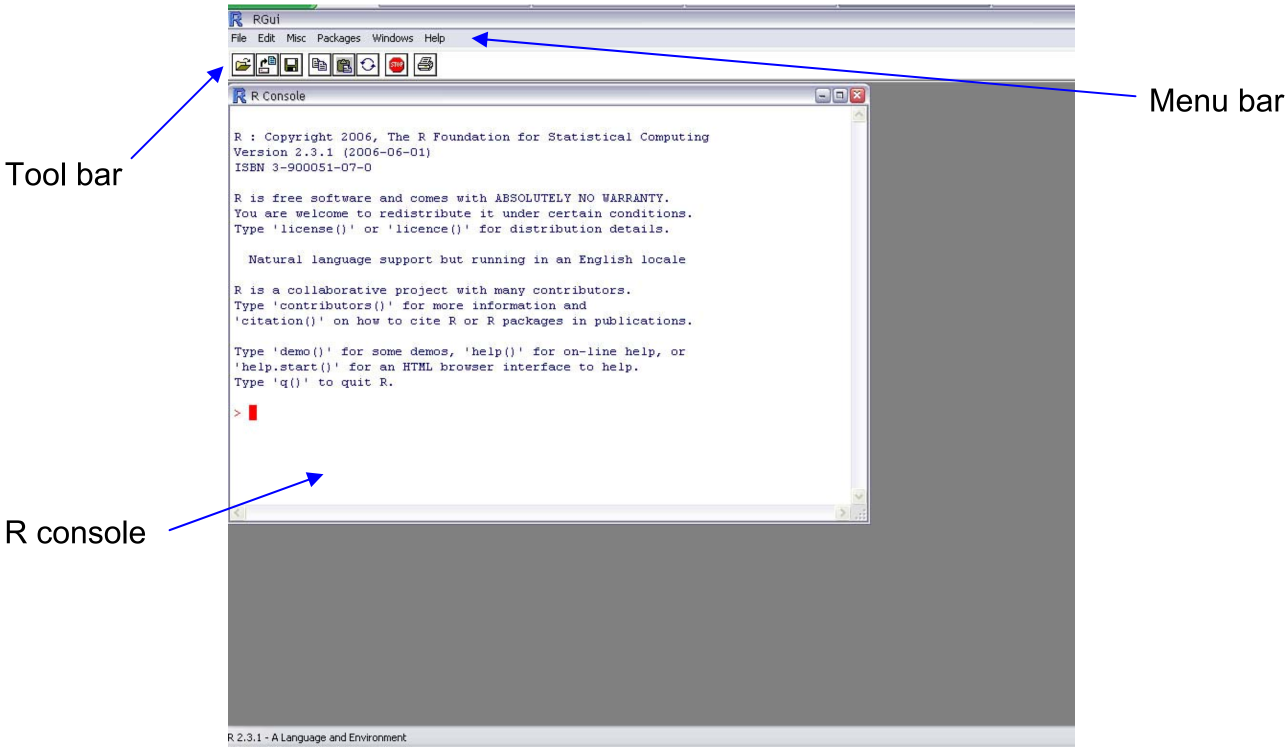
Click on 'Windows 95 and Later', click on 'base' and then 'R-2.4.0-win32.exe' (the version available whilst writing this guide). Double click on the downloaded R executable file and follow the on screen instructions. Full installation instructions can be found at the CRAN website. Although various contributed packages are now included with the standard R distribution, you may need to install other packages to perform particular analyses (see section 1.7 for details of how to do this).

1.3 Starting R on a University of Aberdeen network PC

First, log onto the University network using your usual username and password. Double click on the Life Science and Medicine folder on the Desktop, then the Biological Sciences folder and finally to the Zoology folder. Shortcuts to both R and Tinn-R should be visible. Double click on either to start the program.

1.4 The R console

Once you have started R you will see the standard graphical user interface (GUI). The GUI is rather spartan with a limited number of menu and toolbar commands (Figure 1).


RGui
File Edit Misc Packages Windows Help
R Console
Menu bar
Tool bar
R console
R 2.3.1 - A Language and Environment

```
R : Copyright 2006, The R Foundation for Statistical Computing
Version 2.3.1 (2006-06-01)
ISBN 3-900051-07-0

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

Figure 1: The R console

The GUI contains a menu bar and a tool bar where you can access commonly used commands. It also provides a console window where your commands will be typed at the command line prompt (>). Alternatively, you can use an external script editor to enter your commands, but more on this in section 1.9. In addition, a graphics window will appear automatically when using any plotting function and a R help window will appear when you ask for information about a particular command (see section 1.5 for more information).

1.5 R help and support

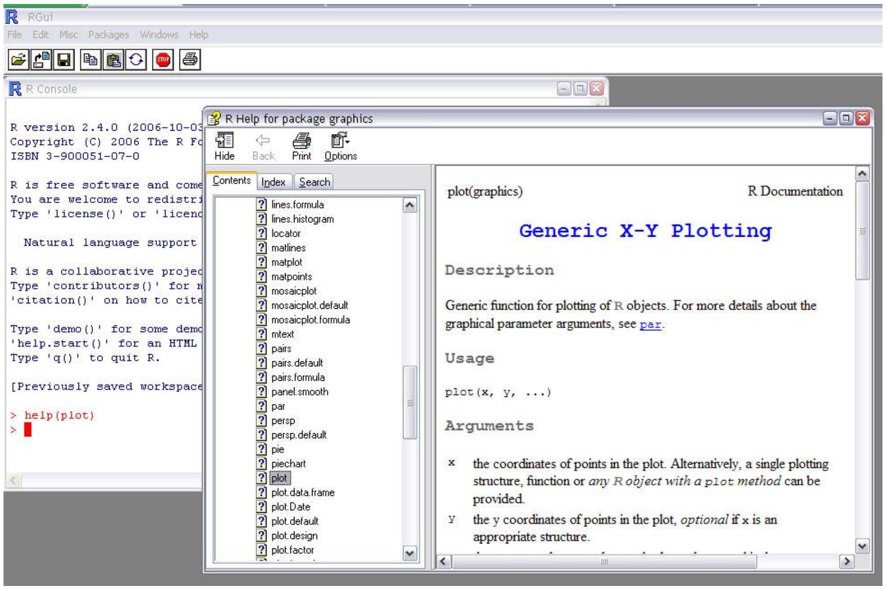
This guide is intended as a brief introduction to R and as such you will soon be using functions and packages that are beyond its scope. Fortunately, one of the strengths of R is its comprehensive and easily accessible help system and a wealth of online resources where you can obtain further information. To access R's built-in help facility to get specific information on any named function simply type in the R console at the command line prompt

```
> help(plot)
```

Or alternatively

```
> ?plot
```

The above example will display help for the function `plot()` in a separate R help window (Figure 2).



```
R version 2.4.0 (2006-10-03
Copyright (C) 2006 The R Fo
ISBN 3-900051-07-0

R is free software and come
You are welcome to redistri
Type 'license()' or 'licenc

  Natural language support

R is a collaborative projec
Type 'contributors()' for m
'citation()' on how to cite

Type 'demo()' for some demo
'help.start()' for an HTML
Type 'q()' to quit R.

[Previously saved workspace

> help(plot)
>
```

plot(graphics) R Documentation

Generic X-Y Plotting

Description

Generic function for plotting of R objects. For more details about the graphical parameter arguments, see par.

Usage

`plot(x, y, ...)`

Arguments

x the coordinates of points in the plot. Alternatively, a single plotting structure, function or *any R object with a* `plot` *method* can be provided.

y the y coordinates of points in the plot, *optional* if x is an appropriate structure.

Figure 2: The R help window

The first line of the help contains information such as the name of the function and the package where the function can be found. There are also other headings which provide more specific information such as

Description: gives a brief description of the function.
Usage: gives the name of the arguments associated with the function and possible default values (options).
Arguments: provides more detail regarding each argument.
Details: gives further details of the function.
Value: if applicable, gives the type of object returned by the function or the operator.

See Also: provides information on other help pages with similar or related content.

Examples: gives some examples of using the function. You can also access examples at any time by using the `example()` function (i.e. `example(plot)`)

Alternatively, you can use

```
> help("plot")
```

This method has the advantage of allowing you to search for help on non conventional characters (i.e. {, [, *). If in doubt always use quotes.

In order to search for help in R it is necessary to use the `help.search()` function. For example

```
> help.search(“plot”)
```

Gives the following

```
Help files with alias or concept or title matching 'plot' using regular
expression matching:

base-defunct(base)      Defunct Functions in Base Package
glm.diag.plots(boot)    Diagnostics plots for generalized linear models
jack.after.boot(boot)   Jackknife-after-Bootstrap Plots
lines.saddle.distn(boot)
                        Add a Saddlepoint Approximation to a Plot
plot.boot(boot)         Plots of the Output of a Bootstrap Simulation
av.plots(car)           Added-Variable Plots
ceres.plots(car)        Ceres Plots
cr.plots(car)           Component+Residual (Partial Residual) Plots
```

The name of each entry is given on the left with the corresponding package in parentheses. A short description of the function is provided on the right. In the above example, the second entry can be displayed by typing

```
> help(glm.diag.plots, package=”boot”)
```

Use the command `?help.search` for further details and examples.

Help in html format can be called from within the console by typing

```
> help.start()
```

This function launches your web browser and allows you to browse the help pages using hyperlinks (Figure 3). One particularly useful feature of html help is the ability to search the R help pages using keywords and also search individual packages (although you can also do this from the R console).

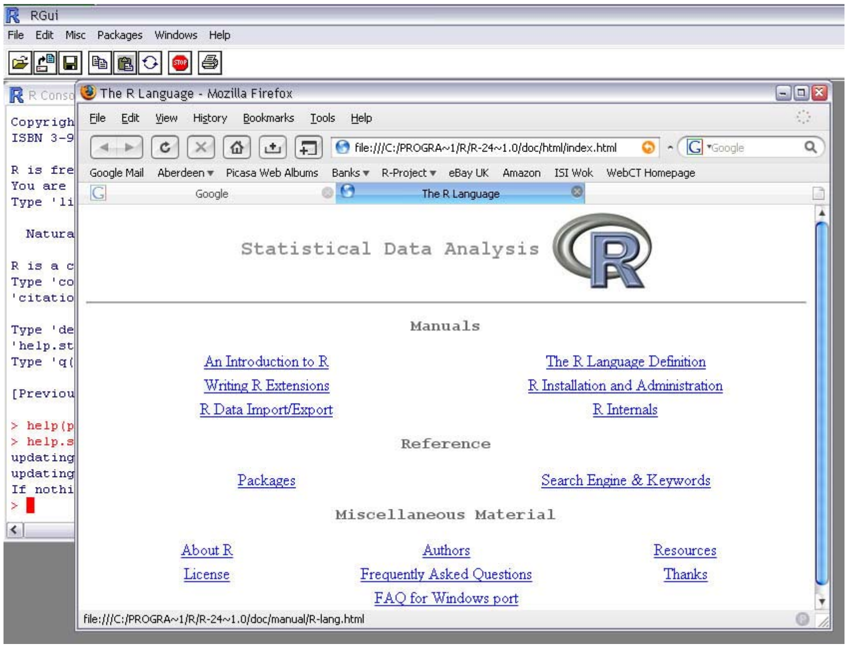

RGui
File Edit Misc Packages Windows Help
R Console
The R Language - Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///C:/PROGRA~1/R/R-24~1.0/doc/html/index.html
Google Mail Aberdeen Picasa Web Albums Banks R-Project eBay UK Amazon ISI Wok WebCT Homepage
Google
The R Language
Statistical Data Analysis
Manuals
An Introduction to R
The R Language Definition
Writing R Extensions
R Installation and Administration
R Data Import/Export
R Internals
Reference
Packages
Search Engine & Keywords
Miscellaneous Material
About R
Authors
Resources
License
Frequently Asked Questions
Thanks
FAQ for Windows port
file:///C:/PROGRA~1/R/R-24~1.0/doc/manual/R-lang.html

Figure 3: Web browser html help

Another useful function is `apropos()`. This function can be used to list all functions containing a specified character string. For example

```
> apropos(help)
[1] ".helpForCall"   "help"           "help.search"    "help.start"
[5] "link.html.help"
```

lists all the functions with "help" in their name.

You can also access help, FAQs and search the R help archives via the pull down menu. Just select the appropriate option and follow the on screen instructions (Figure 4).

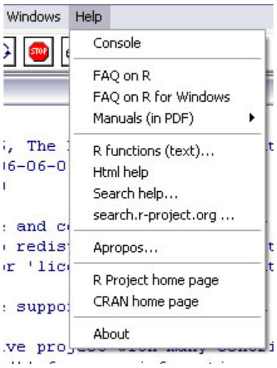


Figure 4: Access help via the menu commands

1.6 Other sources of information

There are a large number of resources available online, many of which can be found on the R-Project homepage (http://www.r-project.org/). These include a searchable RHelp archive, an R Wiki, R mailing lists (can be a bit scary but very useful!) and a variety of user-contributed documents. Some particularly useful pdfs are

"An Introduction to R" is based on the former "Notes on R" and gives an introduction to the language and how to use R for doing statistical analysis and graphics.

"R for Beginners" by Emmanuel Paradis.

"Using R for Data Analysis and Graphics – Introduction, Examples and Commentary" by John Maindonald.

"Simple R" by John Verzani.

"Practical Regression and Anova using R" by Julian Faraway

"R reference card" by Tom Short.

Another useful web page is an R graphics gallery maintained by Romain François which provides examples of R graphics with the code used to produce them. The website can be found at http://addictedtor.free.fr/graphiques/

1.7 R packages

The standard installation of R contains a library of many useful packages. Other packages can be downloaded from the CRAN website which currently hosts over a 100 packages used for various purposes. A list of available packages can be found at the CRAN website or by typing

```
> CRAN.packages()
```

in the R console.

To install a particular package use:

```
> install.packages(“name of package”)
```

or if you want to install more than one package :

```
> install.packages(c(“package1”, “package2”))
```

In order to determine which packages are already installed on your system use:

```
> installed.packages()
```

and to update your packages use:

```
> update.packages()
```

You can also install, update and view packages using the drop down menu option (Figure 5)

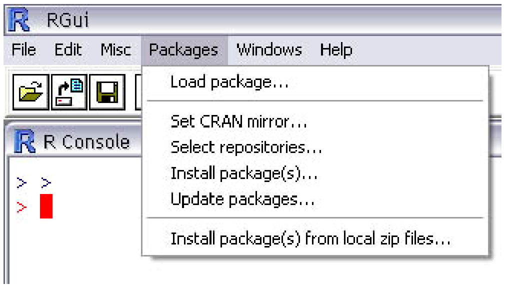
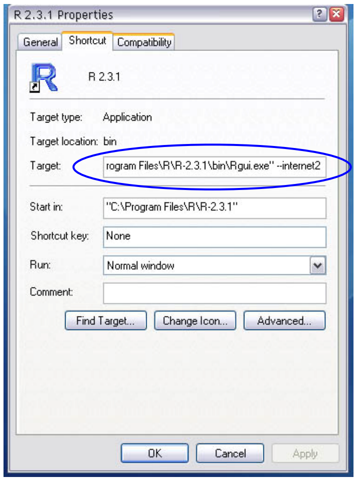


Figure 5: The packages menu options

If you are unable to install packages directly, you can manually download each package as a compressed file (*.zip) and perform a local zip file installation using the menu option (see above).

A note about installing packages on computers connected to the University of Aberdeen network. Due to the presence of the proxy server, which all internet connections have to go through, you will find that you will not be able to perform remote installations or updates of packages. A useful work around is to include the argument --internet2 in the command line of your R program shortcut. To do this, right click on your R icon and select properties. In the 'Target' box enter
--internet2 after the file path (Figure 6). Make sure leave a space between the quotes and the argument.



Target type: Application

Target location: bin

Target: rogram Files\R\R-2.3.1\bin\Rgui.exe" --internet2

Start in: "C:\Program Files\R\R-2.3.1"

Shortcut key: None

Run: Normal window

Comment:

Figure 6: Including the –internet2 argument

R will now use the same settings as the windows internet options on your computer, so make sure you have the proxy settings correctly configured.

Once you have installed a package, it can be loaded into R using the `library()` command. For example, to load the package `nlme` you should enter

```
> library(nlme)
```

Be aware that loaded packages (other than those in the base installation) are not kept in the R workspace between R sessions. If you restart R you will need to reload any packages that you wish to use.

1.8 Working with R

When you begin using R in earnest, you soon find that you will want to save the results of particular analyses for later reference (further details on this are given in section 2.6). When R is installed the default working directory is automatically set to the installation folder. You can check this by typing

```
> getwd()
```

which will display the file path of your current working directory. When you save anything in R it will be saved to the current working directory. For most users, this is not very convenient, so to change the working directory enter

```
setwd("path of new directory")
```

For example, if you wish to change your working directory to 'D:\R\rdata' then type

```
setwd("D:\\R\\rdata")
```

Notice the use of '\\' instead of '\'. You can also use '/' instead of '\'. The working directory can also be changed using the drop down File menu and by selecting 'Change dir…' (Figure 7)

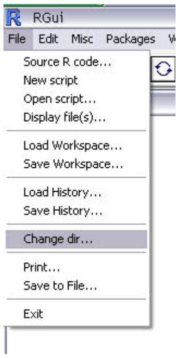
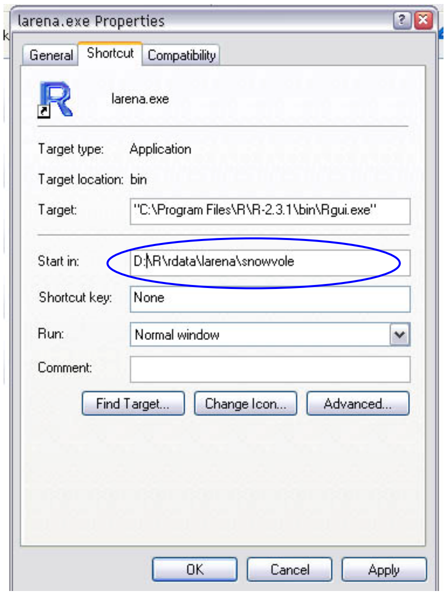


Figure 7: Changing the working directory via the menu command

Another option is to create a separate folder for each of your projects. You can then copy an R shortcut into each folder and rename it to whatever you like. For each shortcut you will need to change the 'Start in' file path properties so that R automatically sets the working directory to the current folder (Figure 8). To do this, right click on the shortcut and select properties. Enter the file path to the current folder in the 'Start in' box and then click OK.



larena.exe

Target type: Application

Target location: bin

Target: "C:\Program Files\R\R-2.3.1\bin\Rgui.exe"

Start in: D:\R\rdata\larena\snowvole

Shortcut key: None

Run: Normal window

Comment:

Figure 8: Changing the 'start in' configuration

Now when you double click on your R icon, R will start and automatically set the working directory to that folder.

1.9 Using an external script editor

The command line is fine for entering short and simple commands, however, when things start to get a little bit more complex you will find using an external script editor much easier. There are a number of excellent and freely available script editors around (just Google WinEdt or Crimson editor) but for this course we will be using Tinn-R. Used in its simplest way, a script editor is just a program in which you can type your commands and then transfer them to R. In Tinn-R you are able to link directly to R and submit your commands at the push of a button. However, we will be using it more simply by copying our commands and pasting them into R manually. The great advantage of using a script editor is that you are able to modify a series commands and submit them all at once rather than having to scroll through numerous commands you typed in previously. Also, by saving your script you will have a complete record of your analysis which you can refer back to. You can obtain more information on using Tinn-R from the website http://www.sciviews.org/Tinn-R/

1.10 Quitting R

To quit R, simply type `q()` in the R console at the command line prompt (>) or use the drop down File menu and select Exit. R will ask you whether you wish to save the workspace image. For now select no (details of how to save your work is given in section 2.6)

1.11 Notation convention used in this guide

A few typographical conventions are used in this guide. These include different fonts and styles for urls, and `R commands`. A series of actions required to access menu commands are identified as File | Change dir… (click on File menu and then Change dir…). Any text that begins with a '#' will be ignored by R and is used to insert comments to help clarify points.

1.12 Preparation for the rest of the workshop

Perhaps you are by now beginning to think to yourself –"why am I bothering with R, everything seems too complicated, I think I'll just use my usual stats package". Don't worry, this is a very natural reaction, and you won't be the only one thinking it. To be fair, R can seem a little complicated at first and very different to many of the packages you might already be using. However, in our experience, a little perseverance at this point will be more than paid back at a later date by an increase in scope and flexibility of your data exploration and analysis and also an increase in productivity and efficiency. There are a few things you can do to ease the 'pain':

1. Keep an accurate and comprehensive record of your work in R. Save script files, keep a note book and annotate these liberally for later reference.

2. Begin using R to explore and analyse your own data as soon as possible. Use R as often as possible.

3. Seek advice and guidance from other people who are also using R. These may be your friends, supervisors, postdocs or contributors to the Rhelp mailing lists. However, be sure to have thoroughly searched the existing help archives for the answer to your question before you submit a query. You will receive very short shrift from the contributors if you don't.

4. Remember, you don't need to know everything there is to know about R. R is just a tool to help you address questions you are interested in, not an end unto itself.

2.0 Some basics

Before we continue, a few comments about the R language:

- Data, functions, results etc. are stored as named variables. The value of any variable can be displayed by typing its name. The value could be quite complex, e.g. a table of all your data for the season, rather than just a simple number. You can perform operations with these variables with operators (arithmetic, logical) and functions, e.g. plot(x).

- R is a case sensitive language. i.e. 'A' is not the same as 'a' and can be used to name different variables.

- Anything that follows a # on the command line is taken as a comment and ignored by R. Comments can be included almost anywhere.

- Commands are generally separated by a new line, but can also be separated by a semicolon ;

- A series of commands can be grouped together using braces, `{` and `}`

- A continuation prompt (+) will appear when you hit return but the command is still not complete i.e. you forget to close a bracket when using `plot(x` . Just finish the command on the new line.

- You can recall and re-execute previous commands using the ↑and ↓ keys on your keyboard.

- In general, R is fairly tolerant of extra spaces inserted into commands, however, spaces should not be inserted into operators i.e. `<-` should not read `<  -`

2.1 R as a calculator

One of the simplest tasks you can ask R to perform is to enter arithmetic expressions and receive a result. For example

```
> 2+2
[1] 4
```

The answer is of course 4. The `[1]` in front of the result is R's method of listing numbers and is more useful when you are listing more numbers. The other obvious arithmetic operators are `-`, `*`, `/` for subtraction, multiplication and division respectively.

There are a huge range of mathematical functions in R, some of the most useful include

```
log()       # logarithm to base e
log10()     # logarithm to base 10
exp()       # natural antilog
sqrt()      # square root
4^2         # 4 to the power of 2
3^-1        # 3^-1
pi          # the number π = 3.1415926
```

2.2 Assigning values to variables

To display the value of a variable you simply type its name. For example, if variable b has a value 20

```
> b
[1] 20
```

To assign a value to a variable use the ‘gets’ `<-` operator. Specifically ‘gets’ is a composite operator comprised of a ‘less than’ symbol < and a minus sign `-` . To assign one value to a variable enter

```
> b <- 20          # literally b ‘gets’ 20
> b                # displays the content of b
[1] 20
```

You can also assign the value of an arithmetic expression to a variable

```
> c <- 20+20
> c
[1] 40
```

A variable may also contain many values (a vector). These can be assigned in a number of different ways. The simplest method is to use, c, which means concatenate (literally to link or join together)

```
> w <-c(2,3,1,6,4,3,3,7) #creates a vector with these numbers
> w
[1] 2 3 1 6 4 3 3 7
```

If you have a large number of values to assign, c can be unwieldy. Alternatively you can use the `scan()` function. `scan()` allows you to enter each value in turn using the keyboard by pressing return after each entry. Finish the data entry by pressing return twice. For example

```
> a <- scan()
1: 2                     # data entry starts
2: 4
3: 5
4: 3
5: 6
6: 7
7: 3
8:                       # data entry ends by hitting return twice
Read 7 items
> a                      # displays contents of vector a
[1] 2 4 5 3 6 7 3
>
```

Sometimes it is useful to create a vector that contains a regular sequence of values. To do this enter

```
> d <- 1:10         # creates a vector of whole numbers from 1 to 10
> d
 [1] 1  2  3  4  5  6  7  8  9 10
```

A sequence of values with non-integer steps can be created using the `seq()`function.

```
> e <-seq(1,5,0.5)  # creates a sequence from 1 to 5 in 0.5 steps
> e
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

To generate repeated values in your vector use the `rep()` function

```
> e <-rep(2,times=10)    # repeats 2, 10 times
> e
 [1] 2 2 2 2 2 2 2 2 2 2
```

You can also repeat non-numeric values

```
> f <- rep("abc",times=3)     # repeats abc 3 times
> f
[1] "abc" "abc" "abc"
```

or repeat a series

```
> g <-rep(1:5,times=3)        # repeats the series 1 to 5, 3 times
> g
 [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
```

or elements of a series

```
> h <-rep(1:5,each=3)       # repeats each element of the series 3 times
> h
 [1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
```

To automatically generate levels of factors you can use the `gl()` function

```
> gl(4,3)              # generates a factor with 4 levels with 3 repeats
 [1] 1 1 1 2 2 2 3 3 3 4 4 4
Levels: 1 2 3 4        # automatically identifies the level as a factor
```

2.3 Vector arithmetic and functions in R

Vectors can be manipulated using the same functions described above. However, you must be careful when adding or subtracting vectors of different lengths. Some examples of vector arithmetic are given below

```
> x <- c(1,2,3,4)
> y <- c(5,6,7,8)
> x*y
[1] 5 12 21 32
> y/x
[1] 5.000000 3.000000 2.333333 2.000000
> y-x
[1] 4 4 4 4
> x^y
[1] 1 64 2187 65536
```

Some typical functions used with vectors include `mean(),var(), sd(),range(),length(),max(),min(), summary()`. Some examples of these functions are

```
> y <- c(4,2,5,6,4,3,5,6,7,4,3)
> z <- 1:11
> mean(y)              # calculates the mean of y
[1] 4.454545
> var(y)               # calculates the variance of y
[1] 2.272727
> sd(y)                # calculates the standard deviation of y
[1] 1.507557
> range(z)             # give the range of values in z
[1]  1 11
> length(z)            # gives the number of values in z
[1] 11
```

```
> summary(y)        # produces a table of summary statistics of y
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  2.000   3.500   4.000   4.455   5.500   7.000
```

2.4 Sorting, ordering and manipulating data

You may find you want to extract particular elements from a vector. In order to extract a single element use square brackets, [], containing the position, or index, of the element . For example

```
> y <- c(4,2,5,6,4,3,5,6,7,4,3)    # creates a vector y
> y[3]                   # extracts the 3rd element in the variable y
[1] 5                    # the value of the 3rd element
```

To extract more than one element, not necessarily in sequence

```
> y[c(2,4,6,8,10)]       # extracts the values for elements 2,4,6,8,10
[1] 2 6 3 6 4
```

and to extract a range of elements in sequence

```
> y[3:9]                 # extracts the values of elements 3 to 9
[1] 5 6 4 3 5 6 7
```

It is often useful to be able to extract elements from a vector using a logical condition. For example, to extract all elements greater than 4 in the variable y enter

```
> y[y>4]                 # extracts all elements with values greater
[1] 5 6 5 6 7            # than 4
```

Other examples include

```
> y[y>=2]                     # extracts all elements with values
 [1] 4 2 5 6 4 3 5 6 7 4 3    # greater or equal to 2

> y[y!=6]                     # extracts all elements with values
[1] 4 2 5 4 3 5 7 4 3         # different from 6

> y <- c(4,2,5,6,4)
> y[y>=4] <- 10               # replaces all elements with a value
> y                           # greater or equal to 4 with the value 10
 [1] 10  2 10 10 10
```

Vectors can be sorted and ordered using the functions `sort()`and `rev()`. Some examples are given below

```
> y <- c(4,2,5,6,4,3,5,6,7,4,3)
> sort(y)
 [1] 2 3 3 4 4 4 5 5 6 6 7      # places all elements in ascending order


> rev(sort(y))                  # places all elements in descending
[1] 7 6 6 5 5 4 4 4 3 3 2       # order
```

Note, however, that if you sort the original variable (`y<-sort(y)`) that no 'unsort' function exists, so be sure this is what you want to do (you can of course assign the sorted values to a new variable `y.sorted<-sort(y)`).

Sorting a single vector is generally not that useful. More often we would like to sort a vector according to the values of another vector. To do this use `order()`

```
> height <- c(180,155,160,167,181)
> order(height)
 [1] 2 3 4 1 5
```

To interpret this, lets start with the `order(height)`output. The first value, `2`, (remember ignore `[1]`) should be read as 'the smallest value of `height` is the second element of `height`'. If we check this by looking at `height`, you can see that element 2 has a value of 155, which is the smallest value. The second smallest value in `height` is the 3rd element of `height`, which when we check is 160. The largest value of `height` is element 5 which is 181.

Now suppose the variable `height` is the height (in cm) of five different women. We know the names of these women and can store their names in a variable called `names`.

```
> names <-c ("Joanna","Charlotte","Helen","Karen","Amy")
```

Now we can order the names of the women according to their height

```
> height.ord <- order(height) # creates a variable of ordered height
> names[height.ord]       # orders names using the order of height
 [1] "Charlotte" "Helen"     " Karen"    "Joanna"    " Amy"
```

You are probably thinking 'what's the use of this?' Well, imagine you have a dataset which contains two columns of data and you want to sort each column. If you just use `sort()` to sort each column separately, the values of each column will become uncoupled from each other. By ordering one column and then ordering the other column based on the value of the first column you will keep the correct association of values. More on this in section 3.3.

2.5 The R workspace

All variables created in R are stored in what is known as the *workspace*. To see what variables are in the workspace, you can use the function `ls()` to list them (this function doesn't need any argument between the parentheses).

```
> x <- c(1,4,7,3,2,9,7,6,7)     # create some variable
> y <- 1:9
> z <- seq(1:5,0.5)
> ls()                          # lists variables in the workspace
[1] "last.warning" "x"            "y"            "z"
```

Currently we have 4 variables in the workspace: a system variable "last.warning" and the 3 variables we created "x", "y" and "z".

To remove variables from the workspace (you'll want to do this occasionally when your workspace gets too cluttered), use the `rm()` function. To remove the variable "x" from the workspace enter

```
> rm(x)
> ls()                     # check whether "x" has been removed
[1] "last.warning" "y"            "z"
```

You can remove all variables from the workspace using

```
> rm(list=ls())            # removes all variables from the workspace
> ls()
character(0)               # no variables in the workspace
```

You can also list and remove all variables from the workspace using the Misc drop down menu: Misc | List objects or Misc | Remove all objects.

2.6 Saving your work

You can save your workspace image at any time using

```
> save.image()
```

This will save your workspace to a file called .R Workspace (in windows) in your working directory (you will have already set this to something sensible). You can also specify an alternative file name

```
> save.image("test1.RData")          # saves the workspace as test1
```

You can also save your workspace using File | Save workspace… You will be prompted to save your workspace every time you exit R.

A word of caution. Saving the workspace image saves all the data stored in R created during a session but will not save any of the output displayed in the console window. If you want to save your output then we suggest that you copy and paste it into a suitable text editor (suggestion: use `Courier font` so that the formatting is identical to the R Console). In addition, you can save everything in the R Console by using the 'Save to File…' command found in the File drop down menu.

The history of commands entered during a session can also be saved using the `savehistory()` function. To reload a saved history, use `loadhistory().`Of course, if you have used an external script editor (like Tinn-R) to write your commands and then copied them into R, you will automatically have a record of the commands used (just save your script file). This is extremely useful if you wish to re-run or make changes to the original analyses or show your analyses to colleagues for advice. This is something that is impossible with the more traditional 'point and click' statistics packages and is one of the major strengths of R (or any other command line based software for that matter).

Every time you start R, any previously saved workspace image will be automatically loaded. You can manually load a workspace using the File | load workspace… menu option.

3.0 Dataframes

Until now, you have entered data into R as a single vector. However, most (if not all!) of you will have much more complicated datasets from your various experiments and surveys. Learning how to import your data into R, and how manipulate it, is one of the most important skills you will need to master.

3.1 What are dataframes?

R handles data in variables called dataframes. A dataframe contains rows and columns with the rows referring to different observations or measurements and the columns containing different variables. This setup will be familiar to those of you who use Microsoft excel to manage and store your data. The values in the dataframe don’t have to be just numbers, they can also be, text, logical, dates etc.

For example, the dataframe below contains the results of an experiment to determine the effect of removing the tip of petunia plants grown at 3 levels of nitrogen and in 2 blocks on growth. The dataframe has 8 variables (columns) and each row represents an individual plant. The variables ‘tip treatment’ and ‘nitrogen level’ are categorical and ‘shoot height’, ‘shoot weight’, ‘leaf area’, ‘side shoot area’ and ‘flower number’ are continuous. Although the variable ‘block’ has numerical values, these do not have an order (the plants were either grown in block 1 or block 2 which have no order) and could also be treated as categorical (i.e. they could also have been called A and B). You will see why this is important later (see section 3.3).

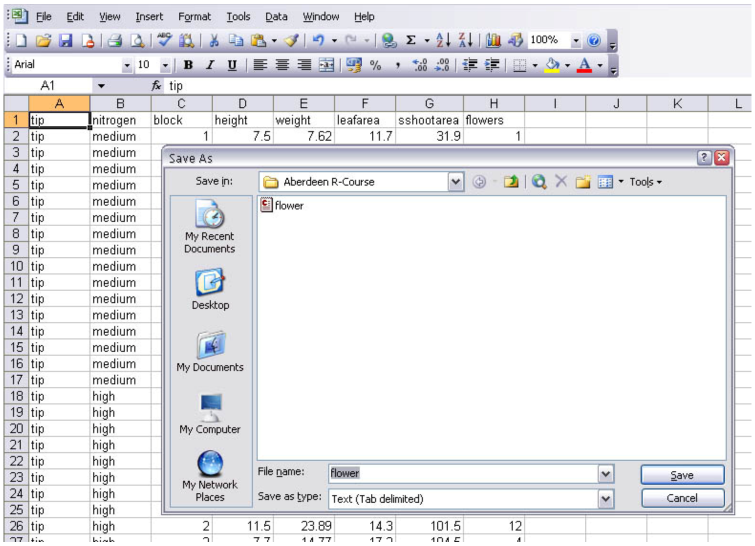
Tip treatment	Nitrogen level	Block	Shoot height	Shoot weight	Leaf area	Side shoot area	Flower number
tip	medium	1	7.5	7.62	11.7	31.9	1
tip	medium	1	10.7	12.14	14.1	46	10
tip	medium	2	11	11.56	12.6	31.3	6
tip	medium	2	7.1	8.16	29.6	9.7	2
tip	high	1	12.6	18.66	18.6	54	9
tip	high	1	10	18.07	16.9	90.5	3
tip	high	2	10.1	15.49	12.6	77.2	12
tip	high	2	8.5	17.82	20.5	54.4	3
tip	low	1	8	6.88	9.3	16.1	4
tip	low	1	8	10.23	11.9	88.1	4
tip	low	2	7.4	10.89	13.3	9.5	5
tip	low	2	3.1	8.74	16.1	39.1	3
notip	medium	1	5.6	11.03	18.6	49.9	8
notip	medium	1	5.3	9.29	11.5	82.3	6
notip	medium	2	3.5	12.93	16.6	109.3	3
notip	medium	2	8.5	10.04	12.3	113.6	4

notip	high	1	8.5	22.53	20.8	166.9	16
notip	high	1	8.5	17.33	19.8	184.4	12
notip	high	2	1.2	18.24	16.6	148.1	7
notip	high	2	2.6	16.57	17.1	141.1	3
notip	low	1	3.9	7.17	13.5	52.8	6
notip	low	1	2.3	7.28	13.8	32.8	6
notip	low	2	5.2	5.79	11	67.4	5
notip	low	2	2.2	9.97	9.6	63.1	2

3.2 Importing data into R

Once you have your dataframe correctly formatted you will need to save it in a file format that R recognises. Fortunately, R is able to recognise a wide variety of file formats, although in reality you will probably only regularly use one or two. The easiest method of importing your data is to save your dataframe in Excel (or an equivalent program i.e. OpenOffice Calc) as a tab delimited file.

In excel, select File | Save as.. from the menu and navigate to the folder where you wish the file to be saved (Figure 9). Enter the file name (keep it fairly short) in the ‘File name:’ dialogue box. In the ‘Save as Type:’ dialogue box click on the down arrow to open the drop down menu and select ‘Text (Tab delimited)’ as your file type. Select Ok to save the file. Your file will now be saved as *filename*.txt



Save in: Aberdeen R-Course

File name: flower

Save as type: Text (Tab delimited)

	A	B	C	D	E	F	G	H
1	tip	nitrogen	block	height	weight	leafarea	sshootarea	flowers
2	tip	medium	1	7.5	7.62	11.7	31.9	1
3	tip	medium						
4	tip	medium						
5	tip	medium						
6	tip	medium						
7	tip	medium						
8	tip	medium						
9	tip	medium						
10	tip	medium						
11	tip	medium						
12	tip	medium						
13	tip	medium						
14	tip	medium						
15	tip	medium						
16	tip	medium						
17	tip	medium						
18	tip	high						
19	tip	high						
20	tip	high						
21	tip	high						
22	tip	high						
23	tip	high						
24	tip	high						
25	tip	high						
26	tip	high	2	11.5	23.89	14.3	101.5	12

Figure 9: Saving text files in Excel

This file can now be read directly into R using the `read.table()` function. So to read the file ‘flower.txt’ into R, enter

```
petunia <-read.table(“D:\\RData\\Rcourse\\flower.txt”, header =T)
```

The above example has read the file *‘flower.txt’* into R, converted it to a dataframe and assigned (using the ‘gets’ operator `<-`) it to an variable called *petunia*. There are a few things to note about the above command. Firstly, the whole file path and the file name needs to be enclosed in double quotes (i.e. “D:\\flower.txt”). If your working directory is set to the directory which contains the file, you don’t need to include the entire file path just the file name. The `header=T` (which can also be written as `header=TRUE`) option specifies that the first row of your dataframe contains the variable names (i.e. ‘nitrogen’, ‘block’ etc). If this is not the case you can specify `header=F`. If the first column of your dataframe contains unique row names you can also include the `row.names=1` argument. Also notice the use of double backslashes (\\) instead of the more familiar single backslash (\) in the file path. Two final points to be aware of. Firstly, `read.table()` will fail if there are any spaces in the variable names in row 1 of the dataframe. Either keep your column headings as single words or replace the space with a dot (i.e. replace shoot height with shoot.height). Secondly, if you have missing data in your dataframe (i.e. empty cells) you must use NA to represent these missing values (if you have used something else you need to use the `na.strings=””` argument, see `?read.table` for further information).

There are additional optional arguments that can be used with `read.table()` which may be useful. Use `?read.table` to explore these further.

A number of variants of the `read.table()` function exist. The most useful of these are `read.csv`, `read.csv2`. The former assumes that the fields are separated by a comma and the latter assumes they are separated by semicolons and that a comma is used instead of a decimal point (as in many European countries). Further variants include `read.delim` for reading in delimited files and `read.fwf` for fixed width formats. You can also install the package ‘foreign’ into R which will allow you to import data files from many other statistical software packages, including SAS, SPSS and Minitab.

To see the contents of the dataframe simply type the variable name

```
> petunia
```

```
    treat nitrogen block height weight leafarea shootarea flowers
1    tip   medium     1    7.5   7.62     11.7      31.9       1
2    tip   medium     1   10.7  12.14     14.1      46.0      10
3    tip   medium     2   10.4  10.48     10.5      57.8       5
4    tip   medium     2   12.3  13.48     16.1      36.9       8
5    tip     high     1   12.6  18.66     18.6      54.0       9
6    tip     high     1   10.0  18.07     16.9      90.5       3
7    tip     high     2   11.5  23.89     14.3     101.5      12
8    tip     high     2    7.7  14.77     17.2     104.5       4
9    tip      low     1    8.0   6.88      9.3      16.1       4
10   tip      low     1    8.0  10.23     11.9      88.1       4
```

```
11   tip      low    2   7.4  10.89     13.3       9.5       5
12   tip      low    2   3.1   8.74     16.1      39.1       3
13 notip   medium    1   5.6  11.03     18.6      49.9       8
14 notip   medium    1   5.3   9.29     11.5      82.3       6
15 notip   medium    2   5.4  11.36     17.8     104.6      12
16 notip   medium    2   3.9   9.07      9.6      90.4       7
17 notip   medium    2   3.9  12.97     17.0      97.5       5
18 notip     high    1   8.5  22.53     20.8     166.9      16
19 notip     high    1   8.5  17.33     19.8     184.4      12
20 notip     high    2   4.7  13.42     19.8     124.7       5
21 notip     high    2   5.0  16.82     17.3     182.5      15
22 notip      low    1   3.9   7.17     13.5      52.8       6
23 notip      low    1   2.3   7.28     13.8      32.8       6
24 notip      low    2   2.4   9.10     14.5      78.7       8
25 notip      low    2   5.7   9.05      9.6      63.2       6
```

To list the names of your variables in the dataframe use the `names()` function

```
> names(petunia)
[1] “treat”      “nitrogen”   “block”      “height”     “weight”
[6] “leafarea”   “shootarea” “flowers”
```

Alternatively, you can gain more information about your dataframe using the `attributes()` or `str()` commands

```
> attributes(petunia)
$names
[1] “treat”      “nitrogen”   “block”      “height”     “weight”
[6] “leafarea”   “shootarea” “flowers”

$class
[1] “data.frame”

$row.names
 [1] “1”  “2”  “3”  “4”  “5”  “6”  “7”  “8”  “9”  “10” “11” “12”
“13” “14” “15”
[16] “16” “17” “18” “19” “20” “21” “22” “23” “24” “25”
```

3.3 Selecting entries in the dataframe

To access single variables in a data frame, use the dollar symbol ($) between the data frame and column names

```
> petunia$height         # extracts all values from the variable height

 [1]  7.5 10.7 11.2  6.0 10.4  9.8  6.9  9.4 10.4 12.3 10.4 11.0  7.1
6.0  9.0
[16]  4.5 12.6 10.0 10.0  8.5 14.1 10.1  8.5  6.5 11.5  7.7  6.4  8.8
9.2  6.2
```

```
[31]  6.3 17.2  8.0  8.0  6.4  7.6  9.7 12.3  9.1  8.9  7.4  3.1  7.9
8.8  8.5
```

It can get rather tedious when repeatedly accessing variables in this manner. Fortunately you can get R to access variables without the $ notation by first attaching the dataframe to R's search path using the `attach()` function. For example

```
> attach(petunia)
```

You will now be able to access a variable by just typing the variable name

```
> height
```

```
[1]   7.5 10.7 11.2  6.0 10.4  9.8  6.9  9.4 10.4 12.3 10.4 11.0  7.1
6.0  9.0
[16]  4.5 12.6 10.0 10.0  8.5 14.1 10.1  8.5  6.5 11.5  7.7  6.4  8.8
9.2  6.2
[31]  6.3 17.2  8.0  8.0  6.4  7.6  9.7 12.3  9.1  8.9  7.4  3.1  7.9
8.8  8.5
```

To detach a dataframe from R's search path, use the `detach()` function (i.e. `detach(petunia)`). It is good practice to detach all dataframes before exiting R.

It is often useful to be able to extract parts of a dataframe. You can extract specific elements, whole columns or rows, or parts of the dataframe based on some logical test.

For example

```
> petunia[2,4]
[1] 10.7
```

extracts the element from the second row, fourth column (which corresponds to the height of a plant grown with a tip, in medium nitrogen in block 1 – check it in the dataframe on page 25). This is also the same as `petunia$height[2]`

If you want to extract values from more than one column and/or row then

```
> petunia[1:10,1:4]
```

```
   treat nitrogen block height
1  tip   medium     1    7.5
2  tip   medium     1   10.7
3  tip   medium     1   11.2
4  tip   medium     1    6.0
5  tip   medium     1   10.4
```

```
6  tip   medium     1    9.8
7  tip   medium     1    6.9
8  tip   medium     1    9.4
9  tip   medium     2   10.4
10 tip   medium     2   12.3
```

extracts rows 1 to 10, columns 1 to 4.

If you do not specify a row or column, then R will extract all elements in all rows or columns. For example, to select all the columns of the first 10 rows

```
> petunia[1:10,]

   treat nitrogen block height weight leafarea shootarea flowers
1  tip   medium     1    7.5   7.62      11.7       31.9       1
2  tip   medium     1   10.7  12.14      14.1       46.0      10
3  tip   medium     1   11.2  12.76       7.1       66.7      10
4  tip   medium     1    6.0   8.78      11.9       20.3       1
5  tip   medium     1   10.4  13.58      14.5       26.9       4
6  tip   medium     1    9.8  10.08      12.2       72.7       9
7  tip   medium     1    6.9  10.11      13.2       43.1       7
8  tip   medium     1    9.4  10.28      14.0       28.5       6
9  tip   medium     2   10.4  10.48      10.5       57.8       5
10 tip   medium     2   12.3  13.48      16.1       36.9       8
```

Notice in the example above that there is no value after the comma so all columns have been included. If you want to include all rows of the first 4 columns then use `petunia[,1:4]`

You can also select parts of the dataframe based on a logical test. In order to select rows with the treatment ‘tip’, nitrogen level ‘medium’ and a height greater than 6 cm use

```
> petunia[height>6 & treat==”tip” & nitrogen==”medium”,]

   treat nitrogen block height weight leafarea shootarea flowers
1  tip   medium     1    7.5   7.62      11.7       31.9       1
2  tip   medium     1   10.7  12.14      14.1       46.0      10
3  tip   medium     1   11.2  12.76       7.1       66.7      10
5  tip   medium     1   10.4  13.58      14.5       26.9       4
6  tip   medium     1    9.8  10.08      12.2       72.7       9
7  tip   medium     1    6.9  10.11      13.2       43.1       7
8  tip   medium     1    9.4  10.28      14.0       28.5       6
9  tip   medium     2   10.4  10.48      10.5       57.8       5
10 tip   medium     2   12.3  13.48      16.1       36.9       8
11 tip   medium     2   10.4  13.18      11.1       56.8      12
12 tip   medium     2   11.0  11.56      12.6       31.3       6
13 tip   medium     2    7.1   8.16      29.6        9.7       2
15 tip   medium     2    9.0  10.20      10.8       90.1       6
```

Notice the use of ‘==’ to specify ‘equals to’ and again no value after the comma.

Remember when we used the function `order()` to sort one vector based on the order of another vector (Page 20 to jog your memory). This comes in very handy if you want to sort columns in your dataframe but keep each value associated with the correct row. For example, if we want all of the rows in the dataframe sorted by height we can use

```
> petunia[order(petunia[,4]),1:8]

    treat nitrogen block height weight leafarea shootarea flowers
68 notip     high     1    1.2  18.24     16.6     148.1       7
62 notip   medium     2    1.8  10.47     11.8     120.8       9
86 notip      low     1    1.8   6.01     17.6      46.2       4
72 notip     high     1    2.1  19.15     15.6     176.7       6
63 notip   medium     2    2.2  10.70     15.3      97.1       7
84 notip      low     1    2.2   9.97      9.6      63.1       2
82 notip      low     1    2.3   7.28     13.8      32.8       6
89 notip      low     2    2.4   9.10     14.5      78.7       8
.
.
.
17   tip     high     1   12.6  18.66     18.6      54.0       9
21   tip     high     1   14.1  19.12     13.1     113.2      13
32   tip     high     2   17.2  19.20     10.9      89.9      14
```

The above command translates to: sort all columns (1:8) of the dataframe petunia in ascending order of height ([,4] height is the fourth column).

An alternative method of selecting parts of the dataframe is to use the `subset()` function

```
> tipplants <- subset(petunia, treat==”tip” & nitrogen==”medium”
& block==”2”)
> tipplants

   treat nitrogen block height weight leafarea shootarea flowers
9    tip   medium     2   10.4  10.48     10.5      57.8       5
10   tip   medium     2   12.3  13.48     16.1      36.9       8
11   tip   medium     2   10.4  13.18     11.1      56.8      12
12   tip   medium     2   11.0  11.56     12.6      31.3       6
13   tip   medium     2    7.1   8.16     29.6       9.7       2
14   tip   medium     2    6.0  11.22     13.0      16.4       3
15   tip   medium     2    9.0  10.20     10.8      90.1       6
16   tip   medium     2    4.5  12.55     13.4      14.4       6
```

In this example a new variable, `tipplants`, has been created and contains values of plants with tips intact, grown at medium levels of nitrogen in block 2.

In order to get a summary of your dataframe you can type

```
> summary(petunia)

   treat      nitrogen      block         height          weight
 notip:48    high  :32   Min.   :1.0   Min.   :1.200   Min.   : 5.790
 tip  :48    low   :32   1st Qu.:1.0   1st Qu.:4.475   1st Qu.: 9.027
             medium:32   Median :1.5   Median : 6.400  Median :11.395
                         Mean   :1.5   Mean   : 6.794  Mean   :12.155
                         3rd Qu.:2.0   3rd Qu.: 8.925  3rd Qu.:14.537
                         Max.   :2.0   Max.   :17.200  Max.   :23.890

   leafarea       shootarea         flowers
 Min.   : 5.80   Min.   :  5.80   Min.   : 1.000
 1st Qu.:11.07   1st Qu.: 39.05   1st Qu.: 4.000
 Median :13.45   Median : 70.05   Median : 6.000
 Mean   :14.05   Mean   : 79.78   Mean   : 7.063
 3rd Qu.:16.45   3rd Qu.:113.28   3rd Qu.: 9.000
 Max.   :49.20   Max.   :189.60   Max.   :17.000
```

Continuous variables (i.e. height, weight etc) are summarised as the mean, minimum, maximum, median, first quartile and third quartile. The number of values in each level of categorical variable is also given. Notice that R has assumed that the variable block is a continuous variable as the level labels are numerical (1 and 2). To declare block as a factor use the `factor()` function

```
> petunia$block <- factor(petunia$block)
```

To check whether this has worked

```
> is.factor(petunia$block)        # ask R whether block is a factor
[1] TRUE                          # R answers yes
```

and the summary now reads as a categorical variable

```
> summary(petunia$block)
 1  2
48 48
```

If you want to create a table of summary data of a variable as a function of different categories you can use the `tapply()` function

```
> tapply(height, treat, mean) # note, the dataframe has already been
                              # attached
 notip    tip
4.7375   8.8500
```

The above command provides the mean of the height of plants in the ‘tip’ and ‘notip’ treatments.

Note: if your dataframe contains missing values coded as NA’s, R will return an NA for which ever summary you have requested.

```
> tapply(height, treat, mean)
 notip    tip              # one of the height values in the tip treatment
4.7375     NA              # contained a missing value
```

To avoid this, include the argument `na.rm=T` in your command

```
> tapply(height, treat, mean, na.rm=T)
   notip      tip
4.737500 8.865217
```

You can also get a full summary of a specified group

```
> tapply(height, treat, summary)
$notip
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.200   3.150   4.500   4.737   5.725  10.900

$tip
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   3.10    7.05    8.80    8.85   10.40   17.20
```

or can summarise the variable by more than one category using `list()`

```
> tapply(height,list(treat,nitrogen), mean)
         high    low   medium
notip 5.70625 3.66875 4.8375
tip   9.60000 8.03750 8.9125
```

Other useful functions for extracting summaries of data are `lapply()` and `sapply()`. The former returns a list whereas the latter tries to simplify the result to a vector.

3.4 Datasets included with R

During the workshop you will be required to access a number of example dataframes that are include with the base installation of R. To obtain a list of datasets type

```
> data()
```

A window will open and the available datasets are listed.

```
Data sets in package ‘datasets’:

AirPassengers           Monthly Airline Passenger Numbers 1949
Bjsales                 Sales Data with Leading Indicator
Bjsales.lead (Bjsales)  Sales Data with Leading Indicator
BOD                     Biochemical Oxygen Demand
CO2                     Carbon Dioxide uptake in grass plants
ChickWeight             Weight versus age of chicks on different
Dnase                   Elisa assay of Dnase
EuStockMarkets          Daily Closing Prices of Major European
                        Indices, 1991-1998
Formaldehyde            Determination of Formaldehyde
```

To access the dataset called **CO2**, simply type

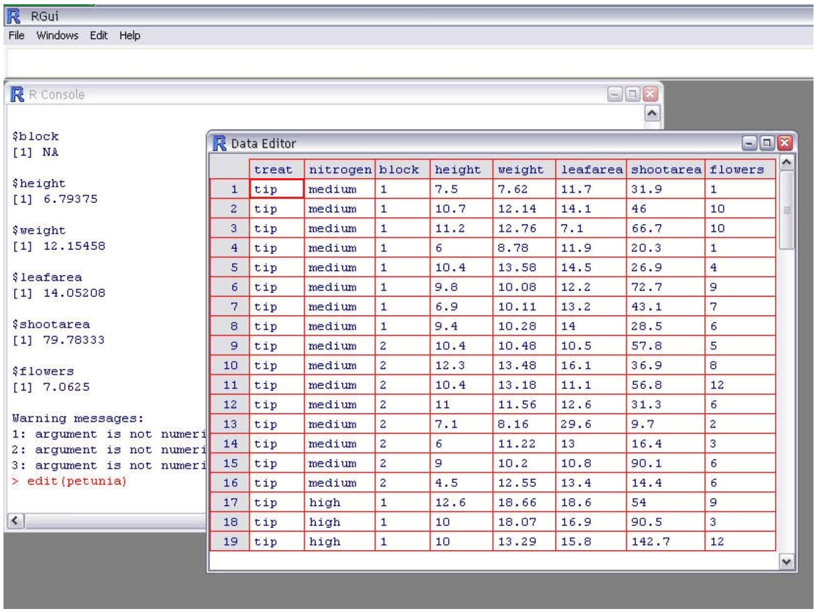
```
> data(CO2)
```

3.5 Using R’s data editor

It is sometimes useful to use R’s built in data editor to change the odd value or variable in a dataframe. To access the editor

```
> edit(petunia)
```

which brings up a separate window containing your data (Figure 10).



```
$block
[1] NA

$height
[1] 6.79375

$weight
[1] 12.15458

$leafarea
[1] 14.05208

$shootarea
[1] 79.78333

$flowers
[1] 7.0625

Warning messages:
1: argument is not numeri
2: argument is not numeri
3: argument is not numeri
> edit(petunia)
```

	treat	nitrogen	block	height	weight	leafarea	shootarea	flowers
1	tip	medium	1	7.5	7.62	11.7	31.9	1
2	tip	medium	1	10.7	12.14	14.1	46	10
3	tip	medium	1	11.2	12.76	7.1	66.7	10
4	tip	medium	1	6	8.78	11.9	20.3	1
5	tip	medium	1	10.4	13.58	14.5	26.9	4
6	tip	medium	1	9.8	10.08	12.2	72.7	9
7	tip	medium	1	6.9	10.11	13.2	43.1	7
8	tip	medium	1	9.4	10.28	14	28.5	6
9	tip	medium	2	10.4	10.48	10.5	57.8	5
10	tip	medium	2	12.3	13.48	16.1	36.9	8
11	tip	medium	2	10.4	13.18	11.1	56.8	12
12	tip	medium	2	11	11.56	12.6	31.3	6
13	tip	medium	2	7.1	8.16	29.6	9.7	2
14	tip	medium	2	6	11.22	13	16.4	3
15	tip	medium	2	9	10.2	10.8	90.1	6
16	tip	medium	2	4.5	12.55	13.4	14.4	6
17	tip	high	1	12.6	18.66	18.6	54	9
18	tip	high	1	10	18.07	16.9	90.5	3
19	tip	high	1	10	13.29	15.8	142.7	12

Figure 10: R’s data editor

You can make changes to the dataframe but they will not be saved when you close the editor. To save changes either assign the edited dataframe to a new variable

```
> petunia1 <- edit(petunia)
```

Or use `fix()`

```
> fix(petunia)
```

If you want to create a new dataframe with R's data editor, use the `data.frame()` function

```
> newpetunia <-edit(data.frame())
```

This command creates a blank dataframe which opens in the data editor. You can enter data and change variable names and once you close the window these changes are saved in the dataframe called *newpetunia*. Remember that this will not change the original file that you loaded or created in Excel.

4.0 Graphics in R

Summarising your data, either numerically or graphically, is an important (if often overlooked) component of any data analyses. Fortunately, R has excellent graphics capabilities and can be used whether you want to produce plots for initial data exploration, model validation or highly complex publication quality graphs. To see some examples of graphics produced in R type

```
> demo(graphics)
```

and hit return to scroll through the examples.

When graphics are created in R they are (unless otherwise told) displayed in the active graphical device or window. If no such window is open when a graphical function is executed, R will open one. However, each time a new plot is produced in the graphics window it replaces the old one. You can save a history of your graphs by activating the 'Recording' option in the History | Recording menu. You can access the old graphs by using the 'Page Up' and 'Page Down' keys to scroll through the graphs. Alternatively, you can simply open a new active graphics window by using the function `x11()`.

You can print your graph directly from the graphics window (using either the File | Print… menu or right click over the graph) or copy the graph to the clipboard (using either File | Copy to the clipboard or right click over the graph) and paste it into a word processor. You can copy the graph as either a metafile or a bitmap image, however we would suggest using the metafile option. A graphic can be saved in many formats including bitmap, metafile, postscript, PDF or jpeg (use File | Save as.. and choose the required format). For publication quality graphs we would recommend using PDF or postscript as these formats can be scaled with no loss of quality.

4.1 Basic plots

There are many functions in R used to produce graphs, ranging from the very basic to the highly complex. It is impossible to cover every aspect of producing graphics in R in this introductory guide. However, we will cover most of the more common methods of graphing data and briefly describe how to customise the standard format.

The most common function used to produce graphs in R is the `plot()` function. For example, to plot a scatterplot of the height of petunia plants (Figure 11).

```
> plot(height)        # assumes petunia has been attached
```

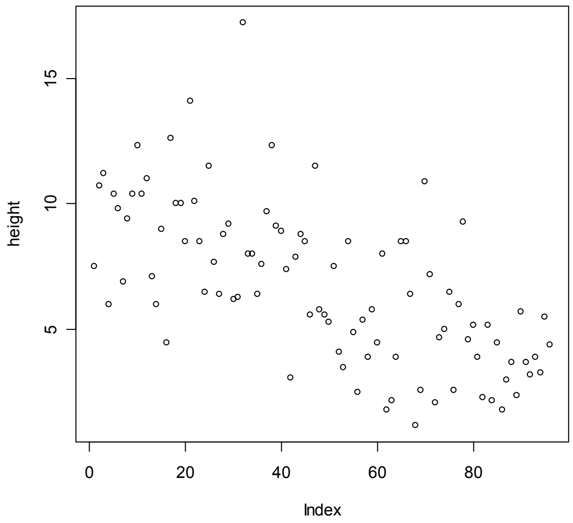


Figure 11: Scatterplot of a single continuous variable

R has plotted the values of height (on the y axis) against an index of the values since there is only one variable to plot. The index is just the order of the values as they appear in the dataframe. If you want to sort the values you can use `plot(sort(height))` The variable labels have been automatically included as axes labels and the axes scale has been automatically set.

Note: if you have not attached `petunia` to R's search path (using `attach(petunia)`) the `plot` function will fail as R will be unable to find the variable `height`

```
> plot(height)
Error in plot(height) : object "height" not found
```

As many of the plotting functions do not allow you to specify the dataframe directly (using `data =` for example), a useful function to use is `with()`. For example

```
> with(petunia,plot(height))    # tells R to plot height using petunia
```

To plot a continuous (dependent) variable Y against a continuous (independent) variable X use either

```
> plot(X,Y)
```

or

```
> plot(Y~X)                    # ~ = 'tilda'
```

Notice that with the first method, R plots the first variable (X) along the horizontal axis and the second variable (Y) along the vertical axis. The second command is an example of using the formula method (should be read as “Y described by X” more on this in section 5.3) which will plot the first variable (Y) on the vertical and the second variable (X) on the horizontal axis. Sometimes the formula method offers more flexibility.

For example, to plot the shoot area of petunia plants against weight (Figure 12)

```
> plot(weight,shootarea)
```

or

```
> plot(shootarea ~ weight)
```

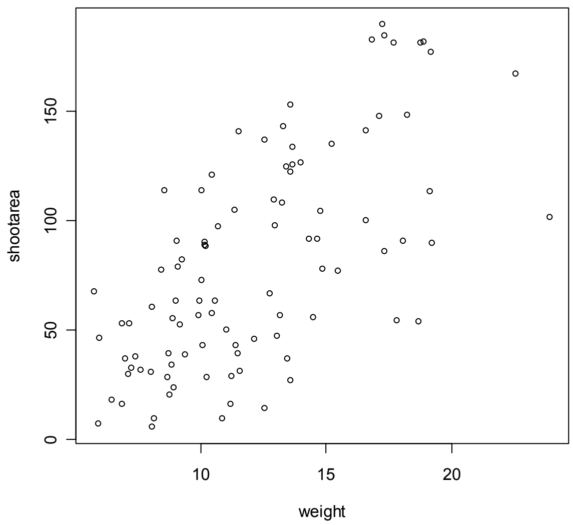


Figure 12: Scatterplot of two continuous variables

The graphs so far have been pretty basic. However, the `plot()` function has numerous options which you can change from the default settings to allow almost complete control over the look of the graph. More details on how to do this is given in section 5.3.

You can also specify the type of graph you wish to plot using the option `type=””`. For example, you can plot just the points (`type=”p”`), lines (`type=”l”`), both points and lines connected (`type=”b”`) and both points and lines with the lines running through the points (`type=”o”`). To plot both points and lines

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

An example of all four types of graph is shown in Figure 13. A special case is `type="n"` which plots the axes but not the data. This can be useful if you want to customise a graph as you will see in section 4.2.

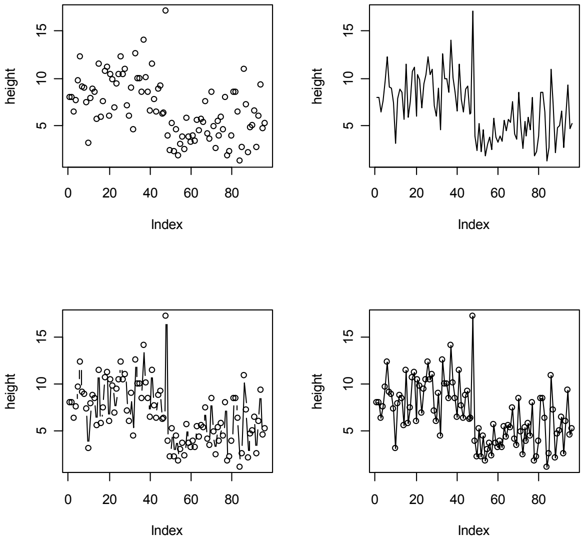


Figure 13: Examples of different plotting styles

The `hist()` function allows you to draw a histogram of a variable in order to gain an impression of its frequency distribution. To plot a histogram of `height` (Figure14)

```
> hist(height)          # assumes petunia is attached
```

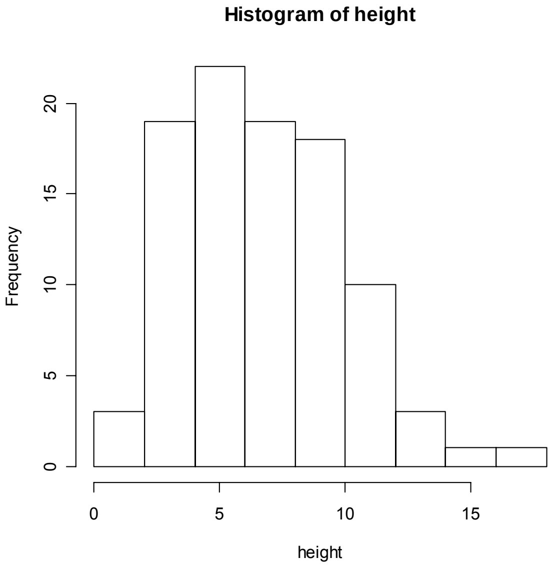


Figure 14: A histogram with automatic breaks

R automatically creates the breaks (or bins) in the histogram unless you specify otherwise by using the `break=` argument (Figure 15)

```
> brk <- seq(0,18,2)            # creates a vector to specify the breaks
> hist(height, breaks=brk)
```

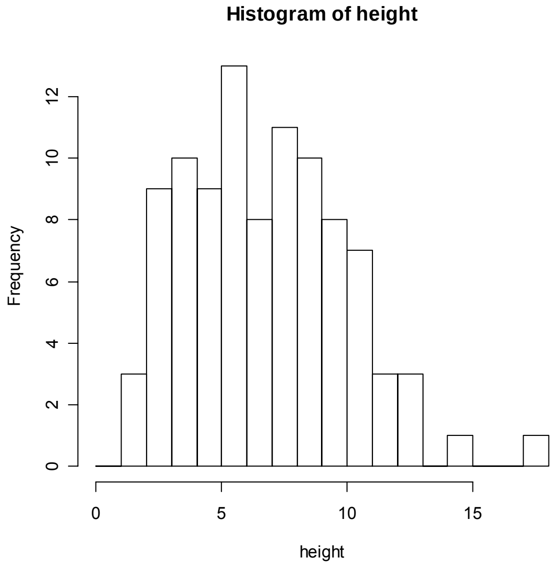


Figure 15: A histogram with user defined breaks

You can also display your data as a proportion rather than a frequency by specifying `freq=F` (literally frequency = FALSE) (Figure 16)

```
> hist(height, freq=F, breaks=brk)
```

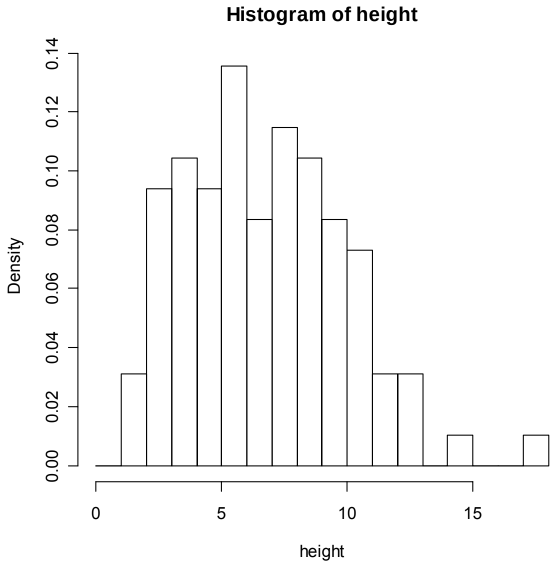


Figure 16: A histogram with proportions displayed

An alternative to plotting a histogram is to plot a density curve. You can superimpose a density curve onto the histogram using the `density()`and `lines()` functions (Figure 17)

```
> dens <- density(height)        # defines the density curve to dens
> hist(height, breaks=brk, freq=F) # plots the histogram
> lines(dens)                       # plots the curve on the graph
```

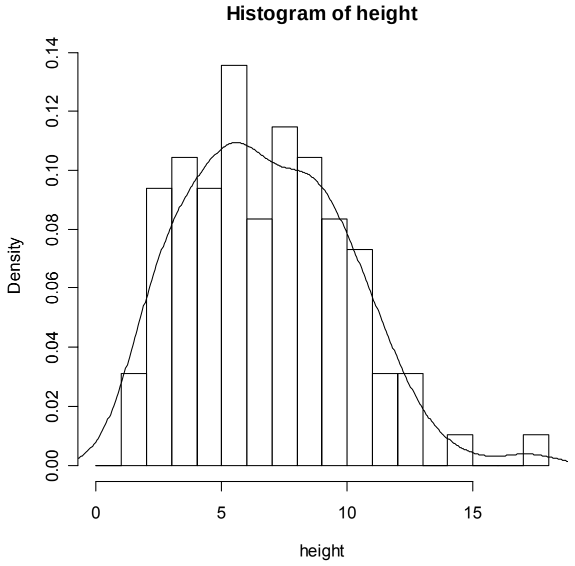


Figure 17: A histogram with a kernel density estimate (automatic bandwidth selected)

Another method of plotting graphical summaries of distributions is to use a box and whiskers plot. To call this in R us the `boxplot()` function (Figure 18)

```
> boxplot(height)
```

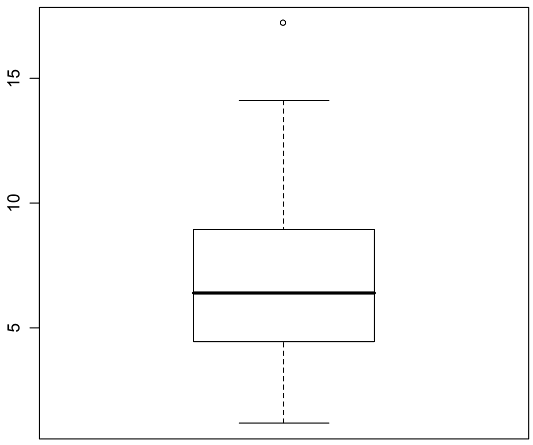


Figure 18: A simple boxplot of one continuous variable

To summarise distributions grouped by a categorical variable, use the formula method to specify what to plot (Figure 19)

```
> boxplot(height~treat, notch=T)    # produces boxplots of height
                                    # for tip and notip treatments
```

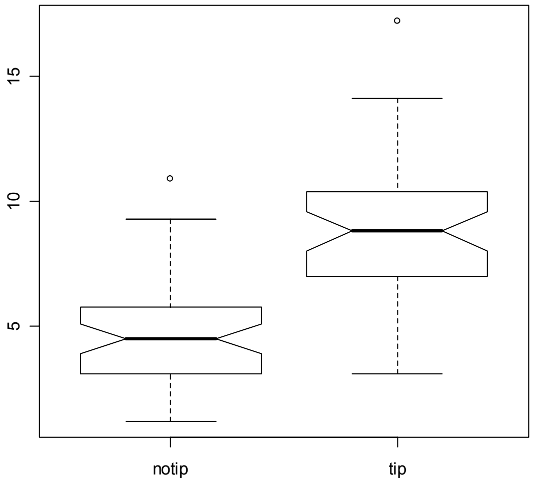


Figure 19: A boxplot grouped by a categorical variable with two levels

By specifying notch=T, a notch is drawn in each side of the boxes. If the notches of two plots do not overlap there is 'evidence' to suggest that the two medians differ.

If you want to place tick marks on the axes which correspond to the position of the data points (Figure 20) use the rug() function.

The argument side= tells R on which axis to plot the tick marks (1=bottom, 2=left, 3=top, 4=right)

```
> boxplot(height~treat, notch=T)
> rug(height[treat=="notip"],side=2)# tick marks of 'notip' on left axis
> rug(height[treat=="tip"],side=4)# tick marks of 'tip' on right axis
```

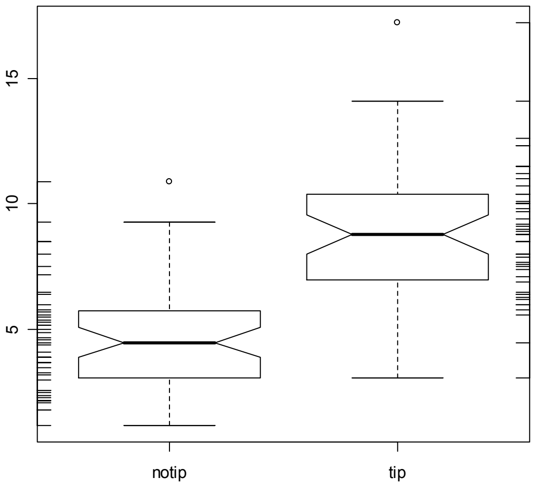


Figure 20: A histogram with rug marks

If there are only few observations in each group, or if the range of values differ markedly in each group, it is more reasonable to use a dotplot (Figure 21). A dotplot (or strip chart as R calls them) plots each value individually

```
> stripchart(height~treat)
```

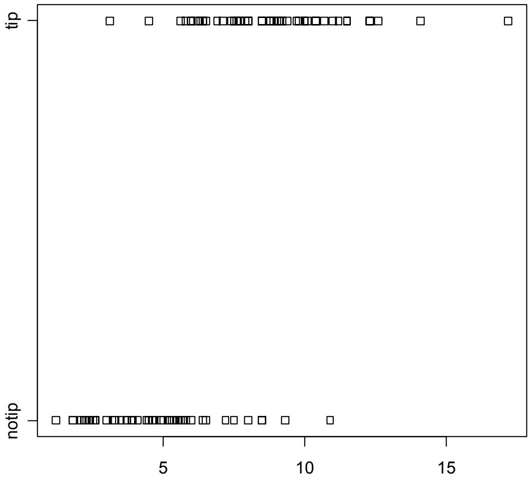


Figure 21: An example of a dotplot

If you have a reasonably large number of data points, a common problem is that some data points will overlay others. To avoid this you can use the `jitter` argument (Figure 22)

```
> stripchart(height~treat, method="jitter", jitter=0.05)
```

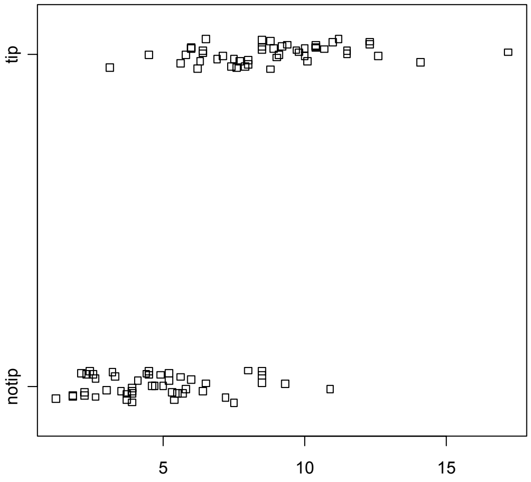


Figure 22: A dotplot with jitter

The value specified by the `jitter` argument determines how far the points are spread apart.

With datasets that contain many continuous variables, it is often important to determine whether any of the variables are inter-related. Plotting multivariate data can sometimes be a real challenge, but R makes it easy. To plot all continuous variables (you can also plot categorical variables) in the dataframe `petunia` simply use the `pairs()` function.

```
> pairs(petunia[4:8])
```

The `pairs()` function plots a matrix of all variables on all possible axes. In the example above, columns 4 to 8 contain the continuous variables (height, weight, leafarea, shootarea and flowers) so in this case we just want to plot these (Figure 23).

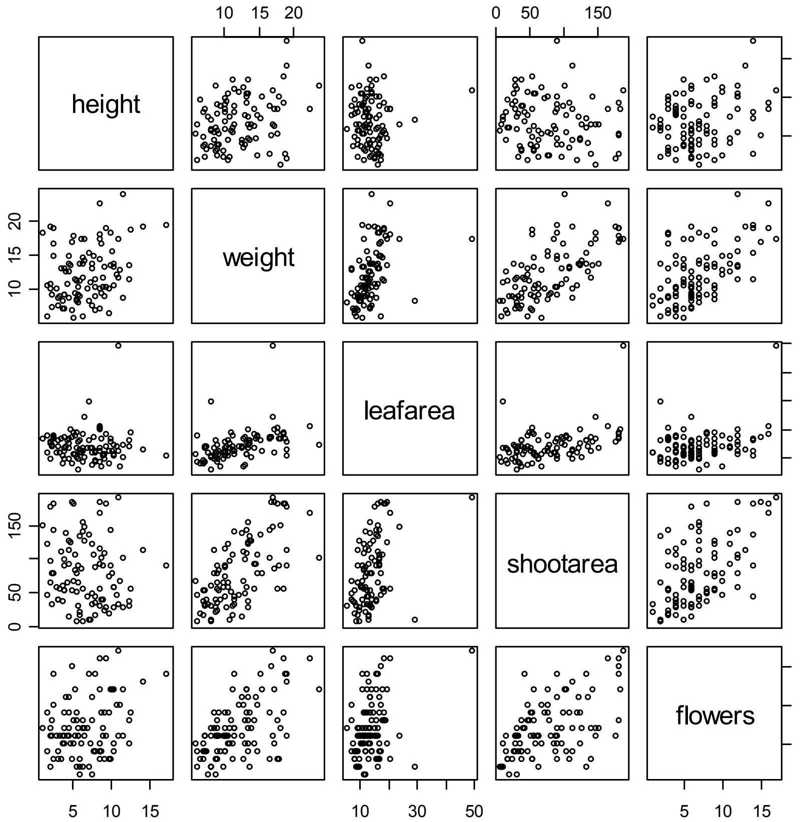


Figure 23: Pair plot of five continuous variables

Interpretation of the matrix of plots takes a bit of getting used to. The rows of the matrix contain the names of the variables on the y axis and the columns contain the names of the variables on the x axis. For example the previous plot of shoot area and weight on page 36 is represented here in the plot 2nd row from the top and 4th from left. A plot of height on the y axis and weight on the x axis is given in the top row, 2nd from left. The corresponding plot of height on the x axis and weight on the y axis is plotted 2nd row from top, 1st from the left.

Additional functions can be called within `pairs()` to aid interpretation of the matrix. For example, `panel.smooth` adds a Lowess smoother to each plot (Figure 24)

```
> pairs(petunia[4:8], panel=panel.smooth)
```

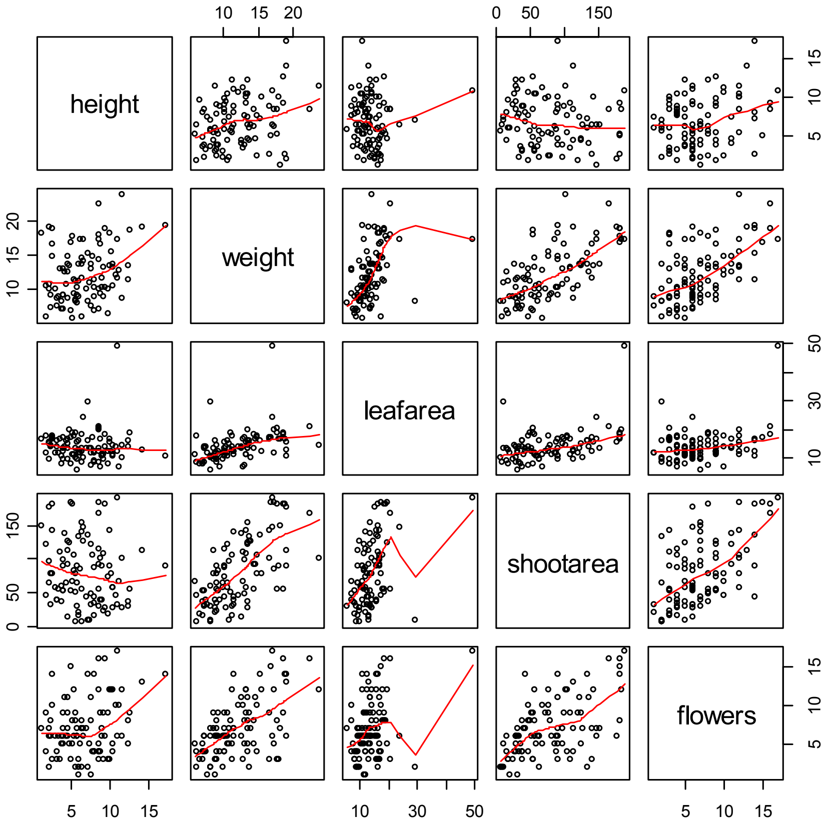


Figure 24: Pair plot with a LOWESS smoother

A really useful little function is given in the help file for the `pairs()` function (remember `?pairs`). The `panel.cor` function puts the absolute correlations of the variables in the upper panels with the size of the text corresponding to the strength of the correlation (Figure 25). The function is

```
## put (absolute) correlations on the upper panels,
## with size proportional to the correlations.
Panel.cor <- function(x, y, digits=2, prefix=””, cex.cor)
{
    usr <- par(“usr”); on.exit(par(usr))
    par(usr = c(0, 1, 0, 1))
    r <- abs(cor(x, y))
    txt <- format(c(r, 0.123456789), digits=digits)[1]
    txt <- paste(prefix, txt, sep=””)
    if(missing(cex.cor)) cex <- 0.8/strwidth(txt)
    text(0.5, 0.5, txt, cex = cex * r)
}
```

Don’t worry about understanding this function (it may be fun to try!) all you need to know is how to use it. Simply copy the text from the help file and paste it into the command line or Tinn-R. This has now defined the function `panel.cor` so you can now use it in the `pairs()` command

```
> pairs(petunia[4:8],lower.panel=panel.smooth, upper.panel
=panel.cor)
```

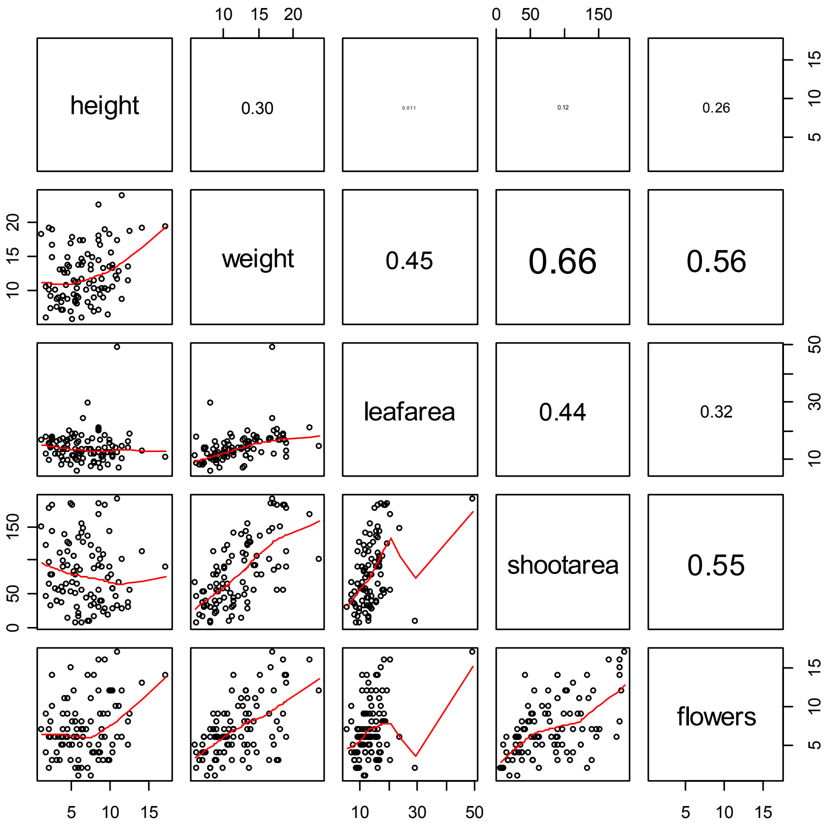


Figure 25: Pair plot with LOWESS smoothers and absolute correlations

The `lower.panel=panel.smooth` argument has plotted each variable and fitted a Lowess smoother in the lower half of the matrix. The `upper.panel =panel.cor` has put the values of the correlations in the upper half of the matrix with the size of the text corresponding to the strength of the correlation. So the relationship between shoot area and weight has an absolute correlation of *r* = 0.66.

When plotting two variables, it is often useful to determine whether a third variable is obscuring or altering any relationship. A really handy plot to use in these situations is a conditioning plot which is called using the `coplot()` function. The `coplot()` function plots two variables but conditioned (|) by a third variable (Figure 26). This variable can be either continuous or categorical. To look at the relationship between the number of flowers and weight of petunia plants conditioned by shoot area

```
> coplot(flowers~weight|shootarea) # plots flowers against weight
                                   # conditioned by shoot area
```

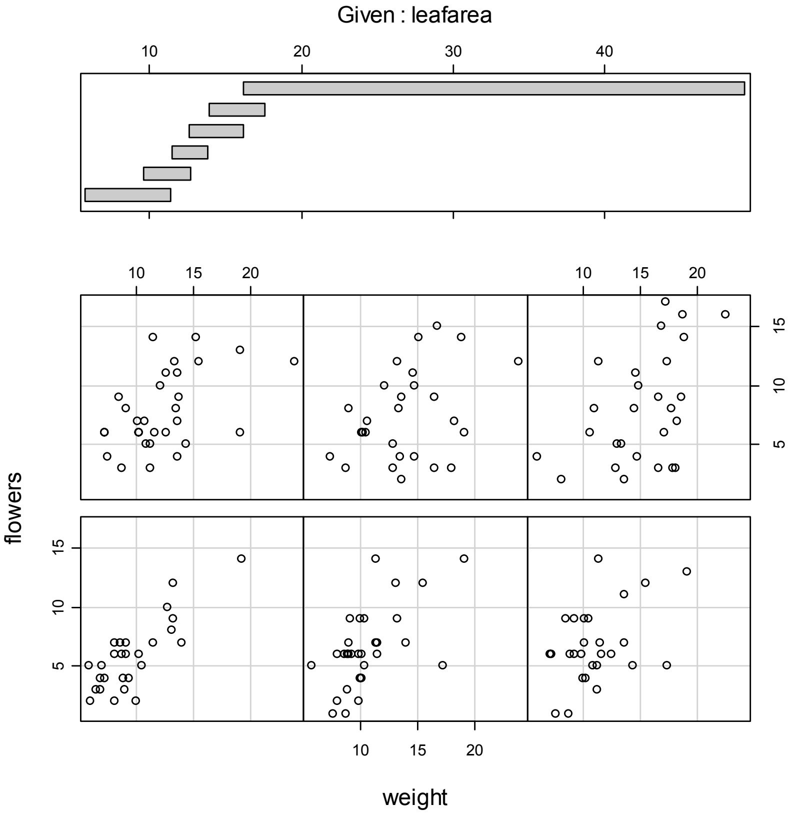


Figure 26: A Coplot of two continuous variables conditioned by another continuous variable

Again, it takes a little practice to interpret coplots. The number of flowers is plotted on the y axis and the weight of plants on the x axis, with 6 separate plots conditioned on the value of leaf area. The panels are read from bottom left to top right along each row. The bottom left panel has the lowest values of leaf area whereas the top right panel has the highest. The panel at the top gives the range of values of leaf area for each of the panels. Notice that the range of values differs between panels and that the ranges overlap from panel to panel.

An example of a coplot with two categorical conditioning variables is to plot flowers as a function of weight for each combination of treatment and nitrogen as shown in Figure 27.

```
> coplot(flowers~weight|treat*nitrogen)
```

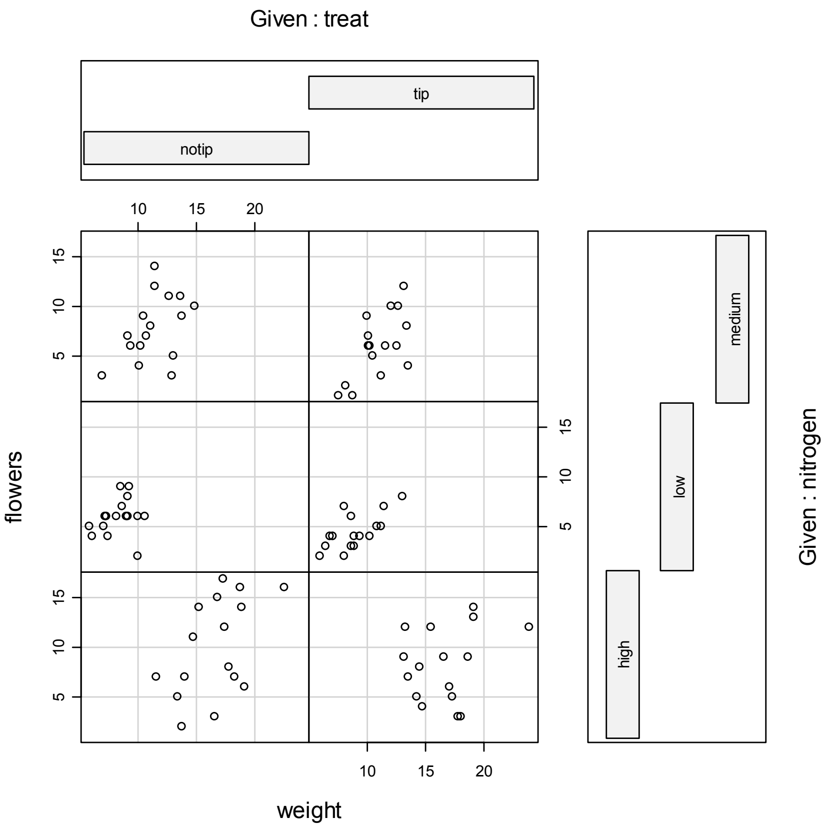


Figure 27: A coplot of two continuous variable conditioned by two categorical variables

Yet another method of plotting multiple variables is to use the plotting functions from the `lattice` package. The `lattice` package offers a wide variety of plotting methods which are extremely powerful and versatile. To access these functions you first have to load the `lattice` package into R's memory

```
> library(lattice)
```

To see a demonstration of the potential of `lattice` functions, type

```
> demo(lattice)        # hit return to start the demo and click on the graphic
                       # window to scroll through the examples
```

The most commonly used function in `lattice` is `xyplot()` which is used to plot panels of scatterplots. This function is somewhat similar to `coplot()`but offers much more versatility. A simple example of using `xyplot()` is to plot the number of flowers as a function of shoot area with a separate panel for nitrogen and treatment combinations (Figure 28)

```
> xyplot(flowers~shootarea|nitrogen*treat)
```

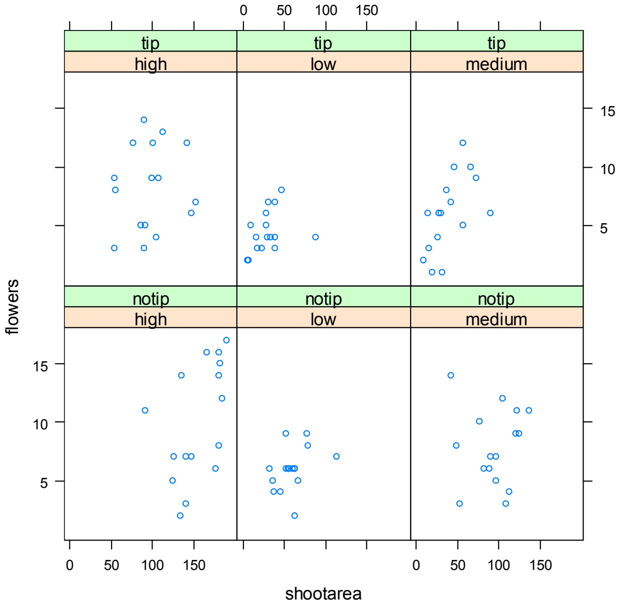


Figure 28: An xyplot of two continuous variables summarised by two categorical variables

You can also specify different symbols or colours for the data points in each plot which provide information about an additional grouping variable. For example, if we wanted to identify which data points were from block 1 or 2 in the petunia experiment (Figure 29)

```
> xyplot(flowers~shootarea|nitrogen*treat, groups=block,
auto.key=T)
```

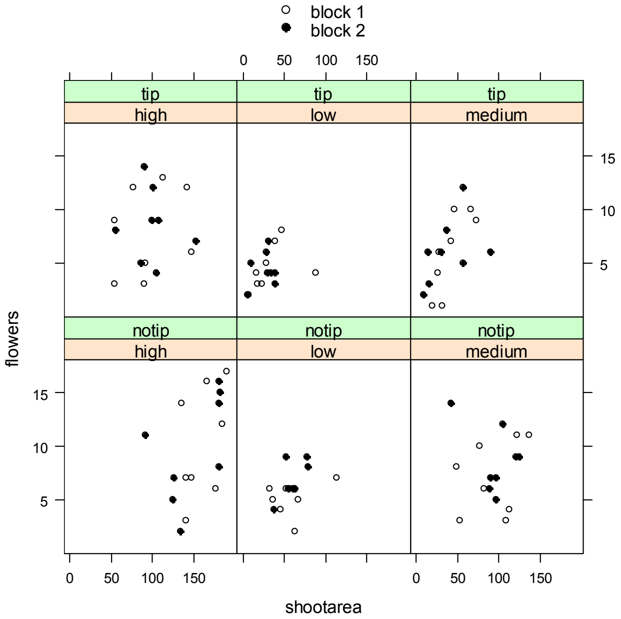


Figure 29: xyplot with a grouping variable

The auto.key=T argument includes a key specifying the grouping variable[1]. Remember to use help to find out more information on xyplot() as this is only the very briefest introduction.

4.2 Reformatting basic plots

All the graphs presented so far are suitable for data exploration. If however, you would like to make them a little prettier (for your thesis or publications for example) you can change many of the default settings to get them just the way you want. Many of the changes you can make are common to most of the graphing functions (except those in the package lattice) so it's worth mentioning a few now.

The plot of shoot area and weight of petunia plants on page 36 is a reasonable starting point for a graph, but it could do with a title, better axes labels, better axes scale and larger data points (Figure 30). To change the graph use

```
> plot(shootarea ~ weight, main="Relationship between shoot area
and weight of petunia plants", xlab="weight (g)", ylab="shoot
area (mm2)", xlim=c(0,25),ylim=c(0,200), pch=16,bty=”l”, cex=1.2)
```

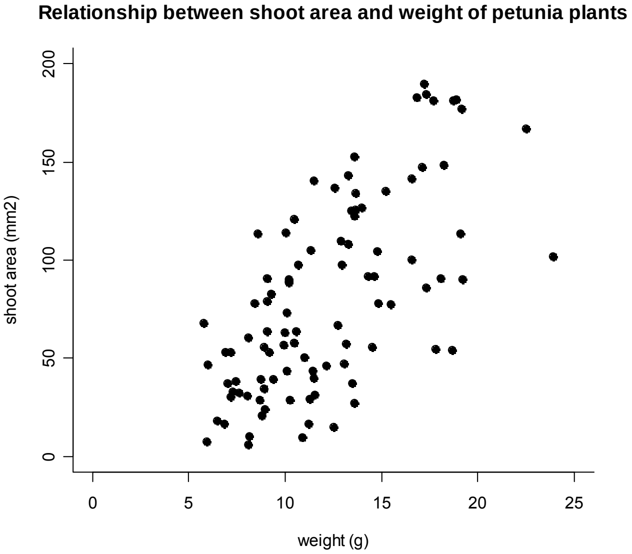


Figure 30: An example of a reformatted plot

[1] The code to produce this plot has been slightly simplified as the symbol and legend colours were changed from the default colour as this manual is printed in black and white. The actual code:

```
> symb <- c(1,16)
> xyplot(flowers~shootarea|nitrogen*treat, groups=block, col=”black”,
        pch=symb, key=list(points=list(pch=symb,
        col=”black”),text=list(c("block 1","block 2"))))
```

The command above may look a little intimidating at first, but all you need to do is break it down into its constituent parts. The arguments `main=””`, `xlab=””` and `ylab=””` add a main title, an x axis label and a y axis label[1] respectively. `xlim=` and `ylim=` sets the scale for the `x` and `y` axes. The option `pch=16` changes the type of symbol which is plotted (see Figure 31), `bty=”l”` controls the type of box drawn around the graph, which in this case is an L shape. `cex=1.2` controls the size of the text and symbols in the plot with the value corresponding to the change in size from the default (i.e. `cex=2` would double the size of the default).

A summary of useful graphical parameters you can use to customise your graphs is given in Table 1. Note, however, this is not an exhaustive list. Use `?help.default` to see other options.

Table1: Useful graphical parameters

Command	Description
`adj`	controls justification of the text (0 left-justified, 0.5 centred, 1 right-justified)
`bg`	specifies the background colour of the plot(i.e. : bg=”red”, bg=”blue”)
`bty`	controls the type of box drawn around the plot, values include: “o”, “l”, “7”, “c”, “u” , “]” (the box looks like the corresponding character); if bty=”n” the box is not drawn
`cex`	controls the size of text and symbols in the plotting area with respect to the default. Similar commands include: cex.axis controls the numbers on the axes, cex.lab numbers on the axis labels, cex.main the title and cex.sub the sub-title
`col`	controls the colour of symbols; as for cex there are: col.axis, col.lab, col.main, col.sub
`font`	an integer which controls the style of text (1: normal, 2: italics, 3: bold, 4: bold italics); as for cex there are: font.axis, font.lab, font.main, font.sub
`las`	an integer which controls the orientation of the axis labels (0: parallel to the axes, 1: horizontal, 2: perpendicular to the axes, 3: vertical)
`lty`	controls the line style, can be an integer (1: solid, 2: dashed, 3: dotted, 4: dotdash, 5: longdash, 6: twodash)
`lwd`	a numeric which controls the width of lines
`pch`	controls the type of symbol, either an integer between 1 and 25, or any single character within quotes “”
`ps`	an integer which controls the size in points of texts and symbols
`pty`	a character which specifies the type of the plotting region, “s”: square, “m”: maximal
`tck`	a value which specifies the length of tick-marks on the axes as a fraction of the width or height of the plot; if tck=1 a grid is drawn
`tcl`	a value which specifies the length of tick-marks on the axes as a fraction of the height of a line of text (by default tcl=-0.5)

[1] For simplicity the superscript for “mm2” has not been included here. To include use

```
ylab=expression(paste("shoot", " area"," (",mm^2,")"))
```

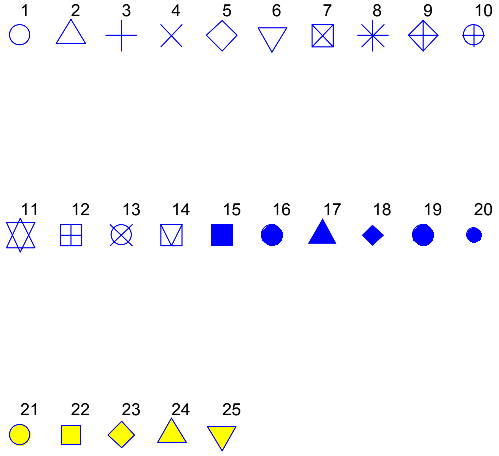


Figure 31: A summary of plotting symbols (`pch=1:25`)

In many cases it is often useful to build the graph in steps so you can add additional lines, data points and other useful information. It is important to realise that R will overlay subsequent commands on the same graph until you call a new graph using `plot()` etc. For example, if you wanted to plot weight against shoot area in the `petunia` dataframe, but use a different symbol and colour for each level of nitrogen (Figure 32) you could do it something like this

```
> plot(shootarea~weight, type="n", xlab="weight (g)",
ylab="shoot area (mm2)")
```

The `type="n"` option tells R to draw the axes but not to plot the data points. We can now add the data points using the `points()` function. We add each level of nitrogen at a time. So for data points in the group 'low' nitrogen we plot red circles

```
> points(shootarea[nitrogen=="low"]~weight[nitrogen=="low"],
  pch=1, col="red")
```

data points in the 'medium' nitrogen group we plot blue triangles

```
> points(shootarea [nitrogen=="medium"] ~ weight [nitrogen ==
 "medium"], pch=2, col="blue")
```

data points in the 'high' nitrogen group we plot black plus signs

```
> points(shootarea[nitrogen=="high"]~weight[nitrogen=="high"],
  pch=3, col="black")
```

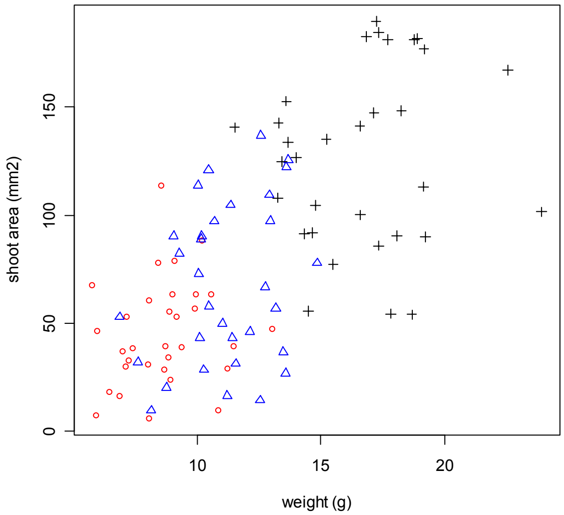


Figure 32: A scatterplot with different colours and symbols

An alternative method, although one that offers less control over the plotting symbols and colours, would be

```
> plot(shootarea~weight, xlab="weight (g)", ylab="shoot
area (mm2)", pch=as.numeric(nitrogen), col=as.numeric
(nitrogen))
```

Using the `as.numeric(nitrogen)` argument for `pch` and `col` converts the factor `nitrogen` to numeric codes which are used to represent the plotting symbols and colours.

As we have used different colours and symbols to represent our data, it would be useful to include a key (Figure 33). To do this we have to first find the appropriate co-ordinates to position the legend using the `locator(1)` function. This function allows you to get the co-ordinates of one point (you can replace `1` by any number of points) by placing the mouse over the graphics window and positioning the crosshairs and left-clicking (top left in this case). The co-ordinates are printed in the R console window. This will be the position of the top left corner of the legend box.

```
> locator(1)
$x
[1] 5.608772

$y
[1] 192.1358
```

Once we have the position we can now use the `legend()` function to create the key. To keep things simple we first have to define a couple of vectors to describe our label text, points style and colour.

```
> labs <- c(“low”,”medium”, “high”)        # label text
> cols <- c(“red”, “blue”, “black”)        # colour of data points
> points <-c(1,2,3)                        # style of data points
```

And now to insert the key

```
> legend(5.6,192.1, labs, pch=points, col=cols)
```

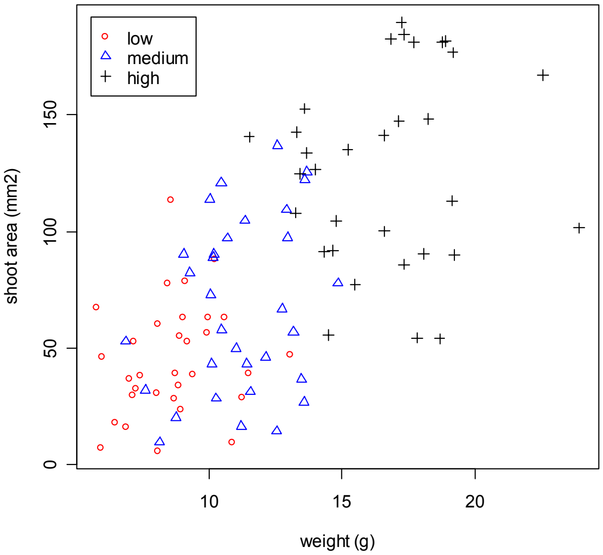


Figure 33: A scatterplot with a key

In the above command, the co-ordinates come first, then the vector `labs` which specifies the label text, followed by the vector `points` which defines the style of points and finally the vector `cols` to specify the colour of the points.

If you want to fit a regression line for these data you can use the `abline()` function (Figure 34). To do this we first have to perform a regression analysis using the `lm()` function and then plot the regression line using `abline()`. Don’t worry about the details of `lm()` at the moment, we will discuss this in more detail in section 5.3.

```
> petunia.lm <- lm(shootarea~weight, data=petunia)
> abline(petunia.lm,lty=1)
```

The first command tells R to perform a linear regression of `shootarea` and `weight` using the data `petunia` and store the result as `petunia.lm`. The second command plots the regression line of `petunia.lm` using a single continuous line (`lty=1`). The graph is shown below

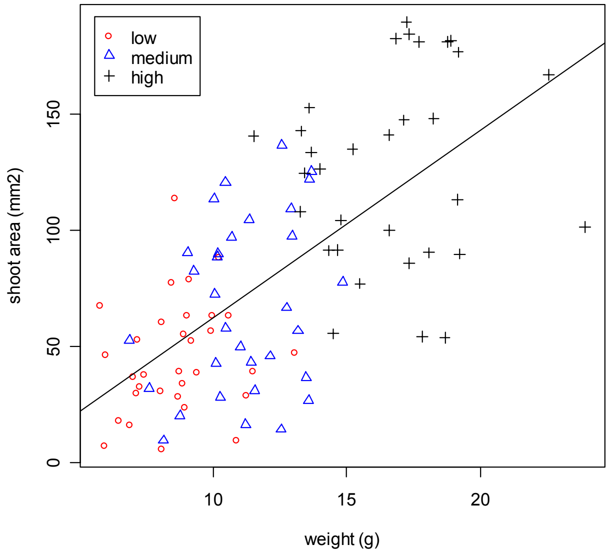


Figure 34: A scatterplot with a regression line

You can easily add text to a graph either on the plotting area by using `text()` or in the margins using `mtext()`. Suppose the graph above is one of a series plotted on the same page (more on this in section 4.3). It may be useful to place a letter on the graph so it can be identified in the figure title (Figure 35). As with adding a key, we first have to find the co-ordinates of the position of the centre of the text that we want to add using the `locator(1)` function.

```
> locator(1)
$x
[1] 22.50871

$y
[1] 187.414

> text(22.5,187.4, “(A)”, font=2)
```

The co-ordinates are listed first, then the text to be added contained within quotes and finally the `font=2` specifies boldface (1 corresponds to plain text, 3 to italics and 4 to boldface italics). Use `?text` and `?mtext` for more information.

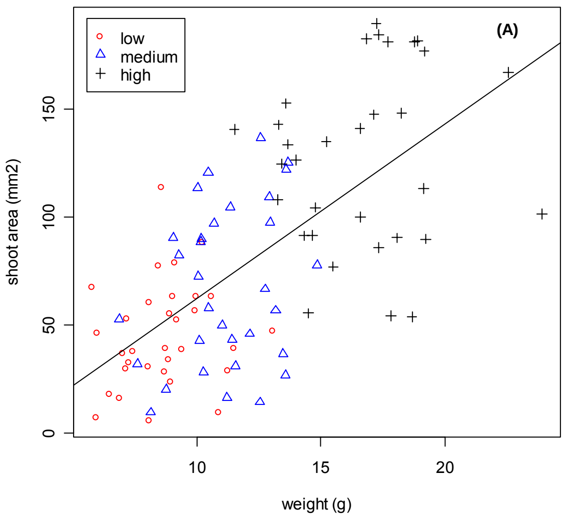


Figure 35: Adding text to a graph

4.3 Plotting multiple graphs

There are a number of methods for plotting multiple graphs in the same graphics window, some of which you have already met, i.e. (`pairs()`, `coplot()`, `xyplot()`). One of the most common methods is to use the main graphical parameter `par()` and change the number of graphs per screen using `mfrow=`. With this method, you have to specify the number of rows of plots you would like and then the number of plots per row. For example, to plot two graphs side by side then

```
> par(mfrow=(1,2)
> plot(shootarea,weight)          # plots the first graph (Figure 36)
```

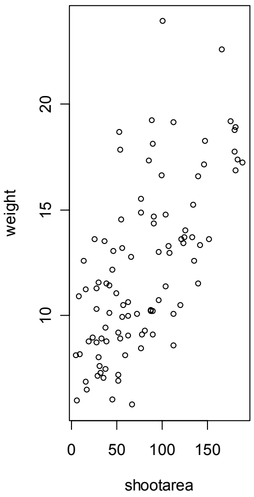


Figure 36: The first of a series of graphs plotted

To plot the next graph in the window you must issue another plotting directive (Figure 37)

```
> plot(nitrogen,shootarea,xlab="nitrogen",ylab="shootarea")
```

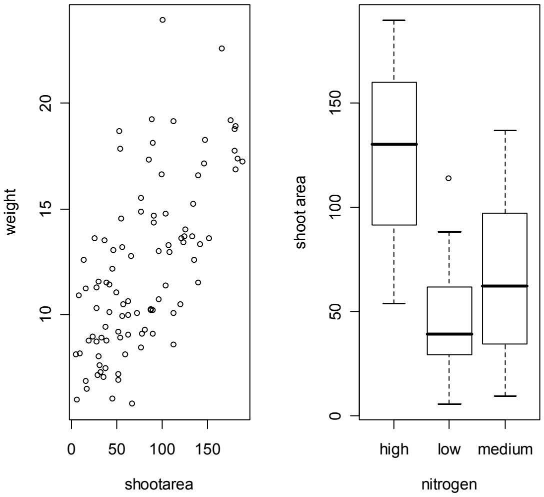


Figure 37: Two graphs plotted in the same graphics window

As you can see from the above example, you can mix different types of graph which is very useful when exploring your data. When you have finished, don’t forget to return to the old layout

```
> par(mfrow=c(1,1))
```

Whilst we are on the subject of graphical parameters, it is worth noting that you can use many of the plotting parameters discussed in section 4.2 with the `par()` function. For example, you can change the background of all plots by

```
> par(bg="lavender")
```

This will give all plots a lavender background colour until specified otherwise. Use `?par` to find out more information on this very powerful function.

5.0 Basic statistics

In addition to R’s powerful graphic facilities, R includes a host of procedures which you can use to analyse your data. Many of these procedures are included with the base installation of R, however, even more can be installed with packages available from the CRAN website. All of the procedures described below can be carried out without installing additional packages.

5.1 One and two sample tests

The two main functions for these types of tests are the `t.test()`and `Wilcox.test()` that perform *t* tests and Wilcoxon’s signed rank test respectively. Both of these tests can be applied to one- and two- sample analyses as well as paired data.

As an example of a one sample *t* test we will use the `trees` dataset which is included with R. To access the dataset

```
> data(trees)
> attach(trees)
> names(trees)
[1] "Girth"  "Height" "Volume"
> summary(trees)
     Girth           Height       Volume
 Min.   : 8.30   Min.   :63   Min.   :10.20
 1st Qu.:11.05   1st Qu.:72   1st Qu.:19.40
 Median :12.90   Median :76   Median :24.20
 Mean   :13.25   Mean   :76   Mean   :30.17
 3rd Qu.:15.25   3rd Qu.:80   3rd Qu.:37.30
 Max.   :20.60   Max.   :87   Max.   :77.00
```

If we wanted to test whether mean height of black cherry trees in this sample is 70 ft or not assuming these data are normally distributed

```
> t.test(Height, mu=70)

        One Sample t-test

data:  Height
t = 5.2429, df = 30, p-value = 1.173e-05
alternative hypothesis: true mean is not equal to 70
95 percent confidence interval:
 73.6628 78.3372
sample estimates:
mean of x
   76
```

The above summary has a fairly logical layout and includes the name of the test that we have asked for (`One Sample t-test`), which data has been used (`data:  Height`), the *t* statistic, degrees of freedom and associated *p* value (`t = 5.2429, df = 30, p-value = 1.173e-05`). It also states the alternative hypothesis (`alternative hypothesis: true mean is not equal to 70`) which tells us this is a two sided test (not equal to), the 95% confidence interval for the mean (`95 percent confidence interval:73.6628 78.3372`) and also an estimate of the mean (`sample estimates:mean of x 76`). In the above example, the *p* value is very small and therefore we would reject the null hypothesis and therefore the mean height of our sample of black cherry trees is not equal to 70.

The function `t.test()` also has a number of additional arguments which can be used for one-sample tests. You can specify that a one sided test is required by using either `alternative=”greater”` or `alternative=”less”` which tests the null alternative that the sample mean is greater or less than the mean specified. For example, to test whether our sample mean is greater than 70 ft.

```
> t.test(Height, mu=70, alternative="greater")

        One Sample t-test

data:  Height
t = 5.2429, df = 30, p-value = 5.866e-06
alternative hypothesis: true mean is greater than 70
95 percent confidence interval:
 74.05764      Inf
sample estimates:
mean of x
       76
```

You can also change the confidence level used for estimating the confidence intervals using the argument `conf.level=0.99`. In this case the new confidence interval would be 99%.

Although *t* tests are fairly robust against small departures from normality you may wish to use a ‘distribution free method’ such as the Wilcoxon’s signed rank test. In R, this is done in almost exactly the same way as the *t* test but using the `Wilcox.test()` function

```
> wilcox.test(Height, mu=70)

     Wilcoxon signed rank test with continuity correction

data:  Height
V = 419.5, p-value = 0.0001229
```

```
alternative hypothesis: true location is not equal to 70

Warning messages:
1: cannot compute exact p-value with ties in:
wilcox.test.default(Height, mu = 70)
```

Don't worry too much about the warning message, R is just letting you know that your sample contained a number of values which were the same and therefore it was not possible to calculate an exact *p* value. This is only really a problem with small sample sizes. You can also use the arguments `alternative ="greater"` and `alternative="less"`.

It is always a good idea to examine your data for departures from normality, rather than just assuming everything is ok. In addition to the functions you have already come across (`hist()`, `boxplot()`, `summary()` etc), perhaps the simplest test of normality is the 'quantile-quantile plot'. This graph plots the ranked sample quantiles from your distribution against a similar number of ranked quantiles taken from a normal distribution. If your sample is normally distributed then the plot of your data points will be in a straight line. Departures from normality will show up as a curve or s-shape in your data points. Judging just how much departure is acceptable comes with a little bit of practice.

To construct a Q-Q plot (Figure 38) you need to use both the `qqnorm()` and `qqline()` functions

```
> qqnorm(Height)
> qqline(Height, lty=2)
```

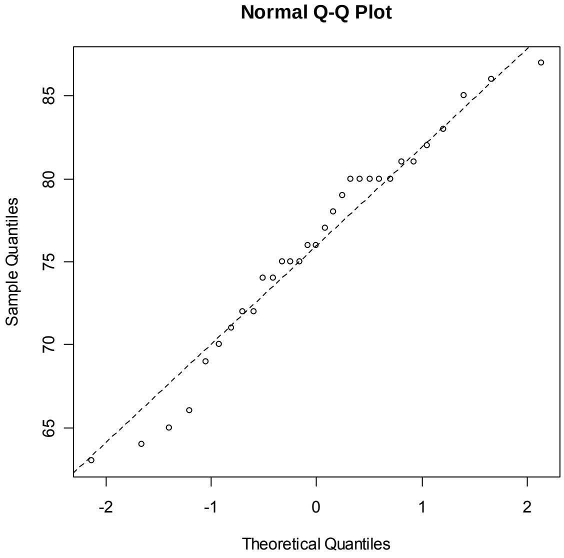


Figure 38: Q-Q plot of the height data

If you insist on performing a specific test for normality you can use the function `shapiro.test()` which performs a Shapiro – Wilk test of normality

```
> shapiro.test(Height)

        Shapiro-Wilk normality test

data:  Height
W = 0.9655, p-value = 0.4034
```

In the example above, the *p* value = 0.4034 which suggests that there is no evidence to reject the null hypothesis and we can therefore assume these data are normally distributed.

In addition to one-sample tests, both the `t.test()` and `Wilcox.test()` functions can be used to test for differences between two samples. A two sample *t* test is used to test the hypothesis that the two samples come from distributions with the same mean. For example, a study was conducted to test whether 'seeding' clouds with dimethylsulphate altered the moisture content of the clouds. Ten random clouds were 'seeded' with a further ten 'unseeded'. The dataset can be found in the 'atmosphere.txt' file

```
> atmos<-read.table("D:\\Aberdeen R-Course\\atmosphere
.txt", header=T)
> attach(atmos)
> names(atmos)
[1] "moisture" "treatment"
> atmos
   moisture   treatment
1     300.6   seeded
2     302.4   seeded
3     298.6   seeded
4     315.9   seeded
5     306.9   seeded
……
16    299.5 unseeded
17    304.6 unseeded
18    298.2 unseeded
19    296.3 unseeded
20    301.4 unseeded
```

As with our previous dataframe (`petunia`), these data are stacked. The column 'moisture' contains the moisture content measured in each cloud and the column 'treatment' identifies whether the cloud was 'seeded' or 'unseeded'. To perform a two-sample *t* test simply enter

```
> t.test(moisture~treatment)

        Welch Two Sample t-test

data:  moisture by treatment
t = 2.5404, df = 16.807, p-value = 0.02125
alternative hypothesis: true difference in means is not
equal to 0
95 percent confidence interval:
  1.446433 15.693567
sample estimates:
  mean in group seeded mean in group unseeded
                303.63                 295.06
```

Notice the use of the formula method (`moisture~treatment`, which reads as moisture described by treatment) to specify the test. You can also use other methods depending on the format of the dataframe. Use `?t.test` for further details. The details of the output are similar to the one-sample *t* test. The Welch's variant of the *t* test is used by default and does not assume that the variances of the two samples are equal. If you are sure the variances in the two samples are the same, you can specify this using the `var.equal=T` argument

```
> t.test(moisture~treatment, var.equal=T)

        Two Sample t-test

data:  moisture by treatment
t = 2.5404, df = 18, p-value = 0.02051
alternative hypothesis: true difference in means is not
equal to 0
95 percent confidence interval:
  1.482679 15.657321
sample estimates:
  mean in group seeded mean in group unseeded
                303.63                 295.06
```

To test whether the assumption of equal variances is valid you can perform an *F*-test on the ratio of the group variances using the `var.test()` function.

```
> var.test(moisture~treatment)

        F test to compare two variances

data:  moisture by treatment
F = 0.5792, num df = 9, denom df = 9, p-value = 0.4283
alternative hypothesis: true ratio of variances is not
equal to 1
```

```
95 percent confidence interval:
 0.1438623 2.3318107
sample estimates:
ratio of variances
         0.5791888
```

As the *p* value is greater than 0.05, there is no evidence to suggest that the variances are unequal at the 95% level. Note however, that the *F*-test is sensitive to departures from normality and should not be used with data which is not normal. See the `car` package for alternatives.

The non-parametric two-sample Wilcoxon test (also known as a Mann-Whitney U test) can be performed using the same formula method

```
> wilcox.test(moisture~treatment)

        Wilcoxon rank sum test

data:  moisture by treatment
W = 79, p-value = 0.02881
alternative hypothesis: true location shift is not equal to
0
```

You can also use the `t.test()` and `wilcox.test()` functions to test paired data. Paired data are where there are two measurements on the same experimental unit (either individual, site etc) and essentially tests for differences between the paired observations. For example, the `pollution` dataset gives the biodiversity score of aquatic invertebrates collected using kick samples in 17 different rivers. These data are paired because two samples were taken on each river, one upstream of a paper mill and one downstream.

```
> pollution <- read.table("D:\\Aberdeen R-Course\\pollution
.txt", header=T)
> attach(pollution)
> names(pollution)
[1] "down" "up"
```

Note, in this case, the data are not stacked with upstream and downstream values in separate columns (you can use the formula method on stacked data if you wish). To conduct a paired *t* test use the `paired=T` argument

```
> t.test(down,up, paired=T)

        Paired t-test

data:  down and up
```

```
t = -3.0502, df = 15, p-value = 0.0081
alternative hypothesis: true difference in means is not
equal to 0
95 percent confidence interval:
 -1.4864388 -0.2635612
sample estimates:
mean of the differences
                 -0.875
```

The output is almost identical to that of a one-sample *t* test. It is also possible to perform a non-parametric matched-pairs Wilcoxon test in the same way

```
> wilcox.test(down,up, paired=T)

        Wilcoxon signed rank test with continuity
correction

data:  down and up
V = 8, p-value = 0.01406
alternative hypothesis: true location shift is not equal to
0

Warning messages:
1: cannot compute exact p-value with ties in: wilcox.test.
default(down, up, paired = T)
```

The function `prop.test()` can be used to compare two or more proportions. For example, a company wishes to test the effectiveness of an advertising campaign for a particular brand of cat food. The company commissions two polls, one before the advertising campaign and one after, with each poll asking cat owners whether they would buy this brand of cat food. The results are given in the table below

	before	after
would buy	45	71
would not buy	35	32

From the table above we can conclude that 56% of cat owners would buy the cat food before the campaign compared to 69% after. But, has the advertising campaign been a success?

The `prop.test()` function has two main arguments which are given as two vectors. The first vector contains the number of positive outcomes and the second vector the total numbers for each group. So to perform the test we first need to define these vectors

```
> buy <- c(45,71)    # creates a vector of positive outcomes
> total <-c((45+35),(71+32))   # creates a vector of total numbers
> prop.test(buy,total)         # perform the test

        2-sample test for equality of proportions with
continuity correction

data:  buy out of total
X-squared = 2.598, df = 1, p-value = 0.107
alternative hypothesis: two.sided
95 percent confidence interval:
 -0.27865200  0.02501122
sample estimates:
   prop 1    prop 2
0.5625000 0.6893204
```

There is no evidence to support that the advertising campaign has changed cat owners opinions of the cat food (*p* = 0.107). Use `?prop.test` to explore additional uses of the binomial proportions test.

We could also analyse the count data in the above example as a Chi-square contingency table. The simplest method is to convert the tabulated table into a 2x2 matrix using the `matrix()` function (note, there are many alternative methods of constructing a table like this)

```
> buyers <- matrix(c(45,35,71,32),nrow= 2)
> buyers
     [,1] [,2]
[1,]   45   71
[2,]   35   32
```

Notice that you enter the data column wise into the matrix and then specify the number of rows using `nrow=`

We can also change the row names and column names from the defaults to make it look more like a table (you don’t really need to do this to perform a Chi-square test)

```
> colnames(buyers) <- c("before", "after")
> rownames(buyers) <- c("buy", "notbuy")
> buyers
       before after
buy        45    71
notbuy     35    32
```

You can then perform a Chi-square test to test whether the number of cat owners buying the cat food is independent of the advertising campaign using the chisq.test() function. In this example the only argument is the matrix of counts

```
> chisq.test(buyers)

        Pearson's Chi-squared test with Yates' continuity
correction

data:  buyers
X-squared = 2.598, df = 1, p-value = 0.107
```

There is no evidence (*p* = 0.107) to suggest that we should reject the null hypothesis that the number of cat owners buying the cat food is independent of the advertising campaign. You may have spotted that for a 2x2 table, this test is exactly equivalent to the prop.test(). You can also use the chisq.test() function on raw (untabulated) data and to test for goodness of fit (see ?chisq.test for more details).

5.2 Correlation

In R, the Pearson's product-moment correlation coefficient between two continuous variables can be found using the cor() function. Using the trees data set again, we can determine the correlation coefficient of the relationship between tree height and volume

```
> data(trees)
> attach(trees)
> names(trees)
[1] "Girth"  "Height" "Volume"
> cor(Height,Volume)
[1] 0.5982497
```

or we can produce a matrix of correlation coefficients for all variables in a dataframe

```
> cor(trees)
           Girth    Height    Volume
Girth  1.0000000 0.5192801 0.9671194
Height 0.5192801 1.0000000 0.5982497
Volume 0.9671194 0.5982497 1.0000000
```

Note that the correlation coefficients are identical in each half of the matrix. Also, be aware that, although a matrix of coefficients can be useful, a little commonsense should be used when using cor() on dataframes with numerous variables. It is not good practice to trawl through these types of matrices in the hope of finding large coefficients without having an *a priori* reason for doing so.

If you have missing values in the variables you are trying to correlate, cor() will return an error message (as will most basic statistical tests in R). You will either have to remove these observations or tell R what to do when an observation is missing. A useful argument to use in this situation is use=”complete.obs”

```
> cor(trees, use="complete.obs")
```

The function cor() will return the correlation coefficient of two variables, but gives no indication whether the coefficient is significantly different from zero. To do this you need to use the function cor.test()

```
> cor.test(Height, Volume)

        Pearson's product-moment correlation

data:  Height and Volume
t = 4.0205, df = 29, p-value = 0.0003784
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.3095235 0.7859756
sample estimates:
      cor
0.5982497
```

Two non-parametric equivalents to Pearson correlation are available within the cor.test() function; Spearman’s rank and Kendall’s tau coefficient. To call these simply include the argument method=”spearman” or method= ”kendall” depending on the test you wish to use. For example

```
> cor.test(Height, Volume, method="spearman")

        Spearman's rank correlation rho

data:  Height and Volume
S = 2089.598, p-value = 0.0006484
alternative hypothesis: true rho is not equal to 0
sample estimates:
      rho
0.5787101

Warning message:
Cannot compute exact p-values with ties in:
cor.test.default(Height, Volume, method = "spearman")
```

5.3 Simple linear regression

To fit a linear regression (“least-squares”) model to your data, we use the lm() function (linear model). This can be used to fit simple linear (single continuous explanatory variable), multiple linear (multiple continuous explanatory variables) and polynomial regression models. The structure of the main argument when using the lm() function is specified using model formula

response variable ~ explanatory variable(s)

You have already come across this type of model specification in connection with some plotting functions (plot(), boxplot() etc) and also with the *t* and Wilcoxon’s tests. It is simply read as ‘the response variable described by (~) the explanatory variable(s)’. So a linear regression of *y* on *x* would be written as

```
> lm(y~x)
```

multiple regression with two explanatory variables (x_1 and x_2)

> lm(y~x_1+x_2)   # fits a regression plane

multiple regression with an interaction term

> lm(y~x_1*x_2)

quadratic regression

> lm(y~I(x)^2)   # the function I() tells R to treat the variable ‘as is’ and not
# to compute the actual quantity

It is important that you get to grips with model formulae (and the above is only the briefest of explanations) as this is the main format used by R for many different types of statistical analyses, including, ANOVA, generalised linear models, mixed effects models and generalised additive models.

Ok, time for an example. The dataframe smoking summarises the results of a study investigating the possible relationship between mortality rate and smoking across 25 occupational groups in the UK. The variable occupational.group specifies the different occupational groups studied, smoking is an index of the average number of cigarettes smoked each day (relative to the number smoked across all occupations) and the variable mortality is an index of the death rate from lung cancer in each group (relative to the death rate across all occupational groups).

```
> smoke <- read.table("D:\\Aberdeen R-Course\\smoking.txt",
header=T)
> attach(smoke)
> names(smoke)
[1] "occupational.group" "smoking"           "mortality"
```

If we ignore occupational group, a scatterplot of these data is shown in Figure 39

```
> plot(mortality~smoking)
```

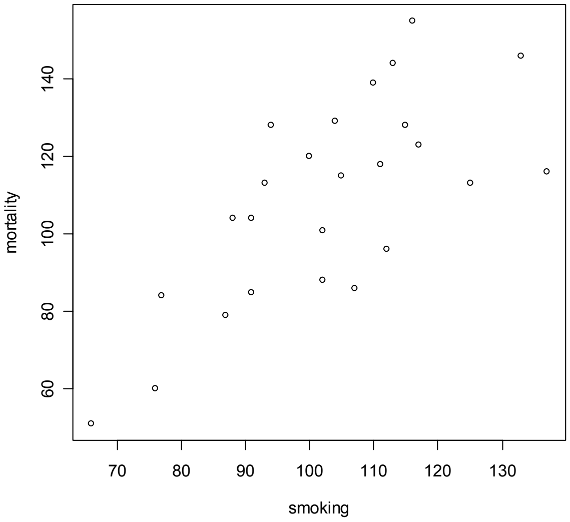


Figure 39: The relationship between smoking and mortality rate

To fit a linear regression to these data

```
> smoke.lm <- lm(mortality~smoking, data=smoke, na.action=
na.exclude)
```

What we have done here is to conduct a linear regression and stored the results as `smoke.lm` (you can call it what you want). We have also included the argument `data=smoke` which tells R that the data for the analysis is contained within the dataframe `smoke`. This is not really necessary here as we have already attached smoke to the workspace. The argument `na.action=na.exlude` tells R to exclude missing values (NA's). This is important if we want to extract information from the model such as residuals and fitted values.

Perhaps somewhat confusingly (at least at first) it appears that nothing has happened, you don't automatically get the voluminous output that you normally get with other statistical packages. In fact, what R does, is store the output of the analysis in what is known as a *model class object* (which we have called `smoke.lm`) from which you are able to extract exactly what you want using extractor functions. To see what elements are contained within the object use the `attribute()` function (or alternatively `names()`)

```
> attributes(smoke.lm)
$names
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"      "qr"         "df.residual"
 [9] "xlevels"       "call"          "terms"
"model"

$class
[1] "lm"
```

To extract an element from the object, simply type the name of the object followed by a dollar sign ($) and then the name of the element. For example, to extract the coefficients of the regression

```
> smoke.lm$coefficients
(Intercept)     smoking
  -2.885319    1.087532
```

The above summary gives the intercept (-2.885) and the slope (1.087) of the linear regression model.

You can obtain more information about the fitted regression using the `summary()` extractor function

```
> summary(smoke.lm)

Call:
lm(formula = mortality ~ smoking, data = smoke)

Residuals:
    Min      1Q  Median      3Q     Max
-30.107 -17.892   3.145  14.132  31.732

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  -2.8853    23.0337  -0.125    0.901
smoking       1.0875     0.2209   4.922 5.66e-05 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 '
' 1

Residual standard error: 18.62 on 23 degrees of freedom
Multiple R-Squared: 0.513,      Adjusted R-squared: 0.4918
F-statistic: 24.23 on 1 and 23 DF,  p-value: 5.658e-05
```

This shows you everything you need to know about the parameter estimates, their standard errors and associated *t* tests and *p* values. It also gives you an idea of the distribution of the residuals which can be used to check for the assumptions of normality

```
Residuals:
    Min      1Q  Median      3Q     Max 
-30.107 -17.892   3.145  14.132  31.732 
```

and the R^2, adjusted R^2 , *F* statistic, associated degrees of freedom and *p* value (tests the hypothesis that the regression coefficient is zero).

If you would prefer to see the ANOVA table rather than the parameter estimates then you can use the `summary.aov()` function

```
> summary.aov(smoke.lm)
            Df Sum Sq Mean Sq F value    Pr(>F)    
smoking      1 8395.7  8395.7  24.228 5.658e-05 ***
Residuals   23 7970.3   346.5                      
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' 
' 1 
```

Finally, we can add the fitted regression line to the scatterplot (Figure 40) using the `abline()` function

```
> abline(smoke.lm)   # alternatively abline(-2.885,1.087)
```

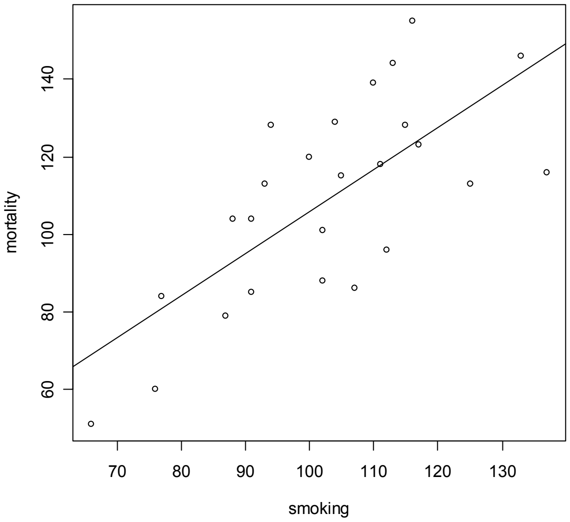


Figure 40: Relationship between mortality and smoking with fitted line included

Before accepting the results of the regression analysis it is important to check the assumptions of constancy of variances and normality of errors. To check for constancy of variances we can construct a graph of residuals versus fitted values (Figure 41) using the `resid()` and `fitted()` functions

```
> plot(resid(smoke.lm) ~ fitted(smoke.lm))
> abline(y=0)          # includes a horizontal line at y=0 for reference
```

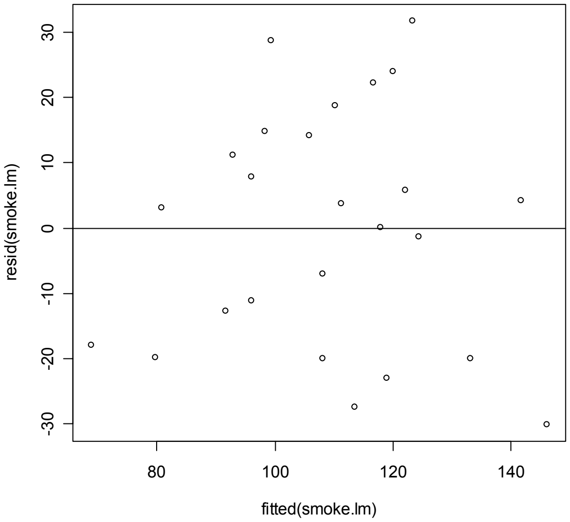


Figure 41: residuals versus fitted values from the model `smoke.lm`

It takes a little practice to interpret these types of graph, but what you are looking for is **no** pattern or structure in your data points. What you definitely **don't** want to see is the scatter increasing as the fitted values get bigger (this has been described as looking like a trumpet or a wedge of cheese).

To check for normality of errors we can use the Q-Q plot (Figure 42) which was introduced in section 5.1.

```
> qqnorm(resid(smoke.lm))
> qqline(resid(smoke.lm))
```

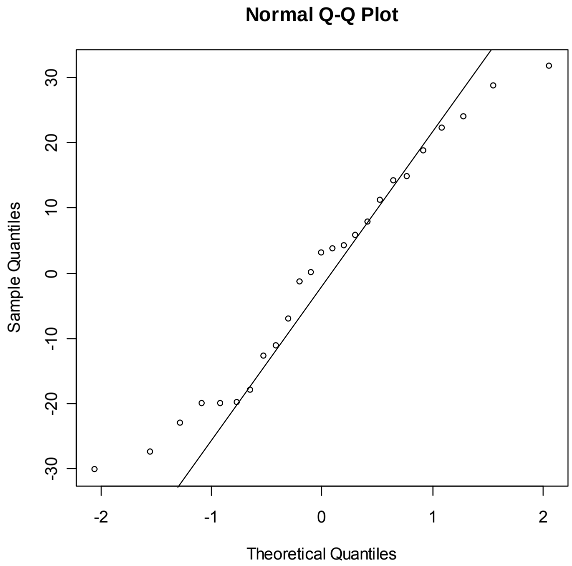


Figure 42: Q-Q plot of the residuals of the model `smoke.lm`

Alternatively, you can get R to do most of the hard work for you by using the `plot()` function on the model itself. Before we do this we should tell R that we want to plot four graphs in the same plotting window

```
> par(mfrow=c(2,2))  # plots 2 graphs in 2 rows
> plot(smoke.lm)     # produces 4 diagnostic plots
```

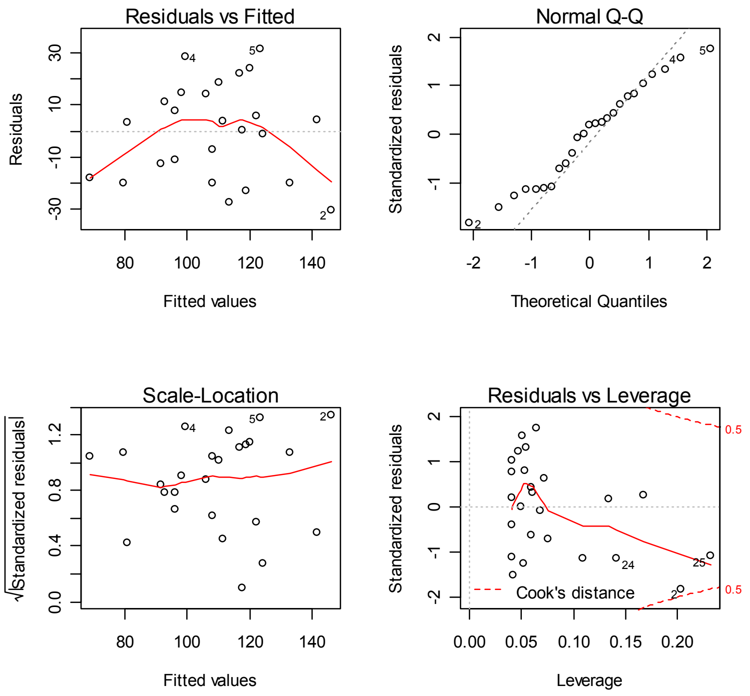


Figure 43: model diagnostic plots produced using the `plot()` function

The first two graphs (top left and top right) are the same residual versus fitted and Q-Q plot we produced before. The third graph (bottom left) is the same as the first but produced on a different scale (the square root of the standardised residuals) and again you are looking for no pattern or structure in the data points. The fourth graph (bottom right) gives you an indication whether any of your observations are having a large influence (Cook's distance) or leverage on your regression coefficient estimates. From the above graphs you can see that points 2 and 25 appear to have the most leverage and also a Cook's distance close to 0.5 and would warrant closer examination. You can access what these values represent by

```
smoke[2,]
       smoking mortality
Miners     137       116
> smoke[25,]
              smoking mortality
Professionals      66        51
```

What you do about influential data points or data points with high leverage is up to you. If you would like to examine the effect of removing one of these points on the parameter estimates you can use the `update()` function. To remove data point 2 (miners, mortality = 116 and smoking = 137) and store the results of the regression in a new object called `smoke.lm2`

```
> smoke.lm2 <- update(smoke.lm, subset=-2)
> summary(smoke.lm2)

Call:
lm(formula = mortality ~ smoking, data = smoke, subset =
(mortality !=
    116), na.action = na.exclude)

Residuals:
     Min       1Q   Median       3Q      Max
-29.7425 -11.6920  -0.4745  13.6141  28.7587

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -20.0755    23.5798  -0.851    0.404
smoking       1.2693     0.2297   5.526 1.49e-05 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 '
' 1

Residual standard error: 17.62 on 22 degrees of freedom
Multiple R-Squared: 0.5813,     Adjusted R-squared: 0.5622
F-statistic: 30.54 on 1 and 22 DF,  p-value: 1.488e-05
```

There are numerous other functions which are useful for producing diagnostic plots. For example, `rstandard()` and `rstudent()` returns the standardised and studentised residuals. The function `dffits()` expresses how much an observation affects the associated fitted value and the function `dfbetas()` gives the change in the estimated parameters if an observation is excluded, relative to its standard error (intercept is the solid line and slope is the dashed line in the example below). The solid bold line in the same graph represents the Cook's distance. Again, all three graphs indicate that observation two is having a large effect on the parameter estimates. Examples of how to use these functions are given below and Figure 44

```
> par(mfrow=c(2,2))
> plot(dffits(smoke.lm), type="l")
> plot(rstudent(smoke.lm))
> matplot(dfbetas(smoke.lm), type="l", col="black")
> lines(sqrt(cooks.distance(smoke.lm)), lwd=2)
```

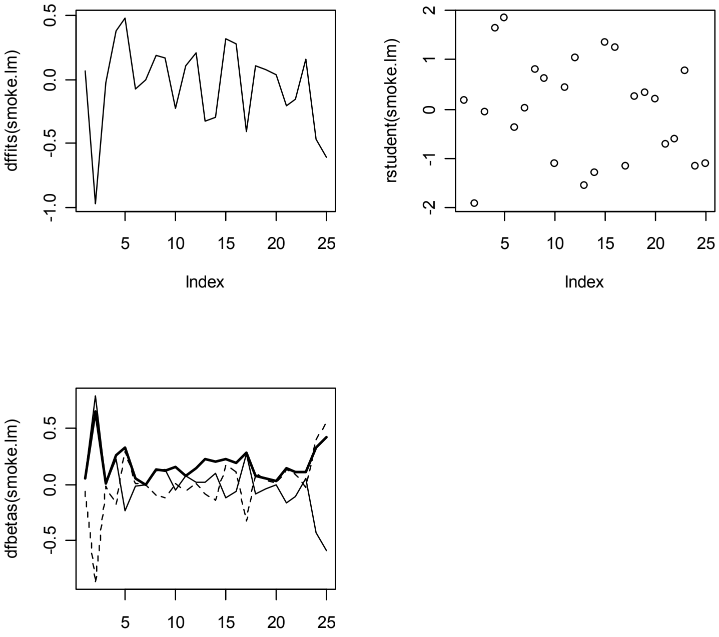


Figure 44: Further regression diagnostics

A list of other useful functions for model simplification and validation is shown in the table below

`add1`	tests successively all the terms that can be added to a model
`drop1`	tests successively all the terms that can be removed from a model
`step`	selects a model with AIC (calls add1 and drop1)

`anova`	computes a table of analysis of variance or deviance for one or several models
`predict`	computes the predicted values for new data from a fitted model
`update`	re-fits a model with a new formula or new data

5.4 Other statistical tests

As with most things R related, a complete description of the variety and flexibility of different statistical analyses you can perform is beyond the scope of this introductory text. Further information can be found in any of the excellent documents referred to on page 10. A table of some of the more common statistical functions is given below, most of which are used in a similar fashion to `lm()`

`aov`	fits an Anova type model to your data
`glm`	fits a generalised linear model with a specific error structure specified using the `family=` directive (poisson, binomial, gamma)
`gam`	fits a generalised additive model
`lme` & `nlme`	fits linear and non-linear mixed effects models. The package nlme must be installed
`kruskal.test`	performs a Kruskal-Wallis rank sum test
`friedman.test`	performs a Friedman's test
`ks.test`	performs a Kolmogorov-Smirnov test

6.0 A final word

We hope that after reading this guide, completing the practical exercises and attending the workshop we have equipped you with some of the basic skills to enable you to start using R for your own data handling and analysis – or at least made you aware of some of the possibilities of what you can do! It has probably been an intense few days, but again, hopefully enjoyable. Don't worry if you can't remember everything you have learned, just refer back to the notes and scripts you have made during the course and in time it will get easier. As is natural in any short course, there are far too many things to cover and not enough time to include all of them. For example, we have not included any information on writing your own functions (although an example is given on page 44) which can be very useful when performing specific undocumented analyses, automating analytical routines or producing custom graphics. Therefore, we would strongly encourage you to invest in a good book or download some of the many excellent free guides available on the web which contain a wealth of information on this subject and many others. Finally, if you mention R in a publication, you must cite the original reference (use `citation()`):

R Development Core Team (2006). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org.

Index of functions and arguments