L U R N

Let's Use  Now...

A. Jonathan R. Godfrey

March 26, 2017
ISBN 978-0-473-17650-1

Please include the following details when citing this publication as a single volume.

Author:	Godfrey, A. Jonathan R.
Year:	2010
Title:	LURN: Let's Use R Now
Publisher:	Institute of Fundamental Sciences, Massey University
Location:	Palmerston North, New Zealand
	ISBN 978-0-473-17650-1

wholly or in part, be done with all parties being aware that the work is made available to any third party under the same conditions as placed on this volume.

Contents

Chapter 1

What's this all about?

```
> Months = c("January", "February", "March", "April", "May", "June", "July", "August", "
```

R is a statistical programming language and environment. The up-front learning curve is somewhat steeper than other statistical software systems as R does not have the Graphical User Interface (GUI) that other packages have; browsing the menus for inspiration is not an option.

This means that R is best suited to users who know what they want, or in educational situations where students are encouraged to know what they want before they play around in a particular application. The difficulty most people will have when encountering R is that knowing what you want is different to knowing how to get what you want. This manual is aimed at developing a set of instructions for doing tasks that might be required in all introductory statistics courses — basic data analysis, manipulation and presentation. It is not intended to be a comprehensive textbook although some theory might be thrown in to help explain particular topics when this is necessary.

This collection should be useful as a reference guide; it might be useful as a series of tasks for undergraduate learning but is not specifically intended for that purpose. If there is something you think you want to do that isn't here let me know and I will endeavour to add it into the next iteration.

I've tried to keep the order of topics fairly linear, but to be honest it's difficult to talk about some things in such a constrained order. Most chapters will state what is assumed before the chapter is worked through. For example, if it's a simple matter of loading the data then it's done on the spot, assuming the reader can work out what is happening for

themselves. The index is therefore an invaluable resource for finding where commands were introduced or explained in greater detail. This means that jumping around should be possible. In the end, skipping chapters and sections of little interest is advisable; there is so much you could learn about using R. You should avoid getting bogged down by unimportant topics; note they are there and move on. There's always time to come back if you find you need to later.

1.1 Getting started

OK, lets assume that you've got this manual because you also have R and are ready to go. If you haven't installed R, then please do, and if you really know little about how to choose the best options for the questions that get asked during the installation, choose the default answer. I install new versions of R as they become available, and happily use the default settings without any downstream pain and suffering.

1.2 Versions of R

R is a collaborative project and development is ongoing. Most users do not need to update R with the release of every new version. I recommend staying with one R version for as long as you can. I only change versions to keep up with the latest version my students might be using and because some of the additional packages created by other R users are built on recent versions. I also ask that my students start a semester with the most current release so that they are on the same page as all of their classmates.

As it happens, this manual was compiled using version 3.4.0 which was released on 12 December, 2016. All code should work on other versions equally well. Please report any faults with the code to the author. Be warned though that if it works for me then it might be because you have an antiquated version of R!

Chapter 2

LURN... To Work R Blind

OK. Cards on the table here. I am blind. Many people know this and perhaps that's why you've obtained this document in the first place. I use R as my software of preference because I find it the most useful tool available to me; partly because I am blind, partly because of the work I do as a lecturer. I have used other packages like SAS and SPlus with almost as much success, Minitab and SPSS with differing levels of success (mostly depending on the version of that software), and have tried a number of other options with little or no success.

I do use Microsoft EXCEL at times, but not for doing the tasks described in this document. I do not regard EXCEL as statistical software but do use it to do some basic calculations and for editing large data sets. To this extent, EXCEL is just an example of a spreadsheet application and has no advantage or disadvantage over any other spreadsheet application. They are therefore all as equally useful/useless to me.

My software experiences have all been gained as a blind person. I had some useful vision for handling print while I was an undergraduate but I found using low-vision software was slower than operating as a blind person. Some of what follows might prove useful to people using software which enlarges the screen's contents but I have had little opportunity to test any low-vision software applications. I would, of course, be interested in learning how anyone using low-vision software gets on using R but the first two users I have worked with were running R quite easily. The low-vision user of R can choose whether operating as a sighted or blind user is better for them so reading through this chapter may prove important if they are to make the best choice.

The material that follows in this chapter has been tested by blind users of R on all three

operating systems mentioned here. I am grateful to the organizers of the second Summer University of the ICCHP, held in the Czech Republic from 29 July to 3 August 2011. This gave me an opportunity to showcase R to a wide audience of blind students from many European countries, and to verify the usefulness of the software on platforms other than Windows.

If you're a sighted user of R and want to know how your blind student is going to get on, you need to talk to them. It might help to read on, but it is going to be much more use if the blind student solves as many of their problems themselves — perhaps with your supervision and assistance.

2.1 Screen readers

Blind people use screen reading software to listen to what gets printed on the screen. Screen readers are good at processing text and generally less useful when a picture or graph needs to be described. I use a product known as JAWS, but other commercial screen readers should work equally well with R; I also test the usefulness of R with one open source screen reader called NVDA. JAWS is a Windows application. Few blind people are serious MacIntosh users, but some are taking up Linux as their operating system of choice. The screen reader in many Linux distributions is called ORCA, and Macintosh users have VoiceOver as part of the standard operating system.

This document assumes the blind user has a good working knowledge of their screen reader and operating system. Most of us know where to find help with our screen reader problems, and in any case the solution I have for using R as a blind person requires little more than the basic knowledge for either screen reader or the Windows operating system.

2.2 R and braille displays

A braille display provides the blind user with information in refreshable braille. The equipment that gets connected to the user's computer is certainly not cheap but can make access to otherwise printed information available to the blind user so that there can be little confusion about the syntax of commands or output. Many blind programmers I have spoken with, swear by their braille display for giving them true access to the printed word.

Sound from screen readers is wonderful, but there are opportunities to make mistakes when relying on sound alone, especially in a case-sensitive programming language like R.

At the 2011 Summer University I was able to see how students using a braille display could use R. The results were very pleasing indeed. The Windows users with braille displays relied on their screen reader to provide the information to the user via the braille display and as the screen readers were working well with R already, the access was further improved with braille.

A Linux user at the 2011 Summer University was able to make use of his braille display without explicitly using the synthetic speech from a screen reader. The interesting aspect to this discovery was that this means R is available to the deaf-blind user as well.

I still need to learn how well the braille display would work with R on a Mac, but expect the experiment would show similarly pleasing outcomes.

2.3 Setting up R as a blind Windows user

Most R users install the software and run it as the default installation allows. The main file that is executed is called “rgui.exe" but in spite of its name, little of R’s use is found within the graphical user interface (GUI). Most R users operate by either typing commands one at a time or by executing a whole bunch of commands together. In either case, the commands are all generated by typing standard text somewhere. Standard text is of course totally accessible to the blind user.

The standard R application window is not accessible to the blind user. We can hear a command being typed in but can’t hear what gets returned. This behaviour is somewhat strange as the sighted user sees what appears to be plain text on screen. Unfortunately it’s not plain text and it’s also not easy to explain. Please send an email if you want a more detailed explanation of the phenomenon. The most reassuring thing about the way a screen reader interacts with the “rgui.exe" application is that when you try to close the application using `ALT+F4`, JAWS will actually speak the dialogue perfectly well. It is in fact the only dialogue that works this well.

Obviously, it’s not all bad news. R does come with a terminal window system and this is accessible; it’s actually what the Linux user gets anyway! This terminal has the look and feel of a standard command prompt window that many users will be familiar with. This window is where all text is entered and where all non-graphical output will be printed.

Your screen reader will speak the contents of the terminal window as output is added.

The default installation of R places a shortcut on the desktop. Once you've installed R in the default way, you will need to edit this shortcut's properties. Find it, hit "ALT+Enter" and make sure you are on the dialogue box that is for the target file. The full path is given here and the last element is "rgui.exe". Change this last bit to read "rterm.exe" and hit OK.

Test it now. The shortcut should run R in a terminal window which looks like a DOS window. You will hear an introduction message about R and ultimately the prompt which is a greater than symbol. To get out of R, we need to type `q()` and answer a question about saving the workspace. This should all be heard if things are going correctly. Note: your answer needs to be followed by hitting the enter key.

Change in software is inevitable and R does undergo changes all the time. When version 2.10.0 came out the standard functionality of R's help was fundamentally altered. In version 2.9.2 and before, typing `?Mean` would open an internet browser with the help page. This was very readable by screen reader software. In version 2.10.0, this default behaviour changed. In version 2.10.1 however the developers fixed the problem. We expect this problem not to arise again, but just in case here is the solution.

If for some reason, the help functionality doesn't work as we suggest, you can force the html help pages to be displayed by performing the following task. You will need to find the folder where R was installed. It should be in the "Program Files" folder if you installed using the default settings when you installed R. We are looking for a file called `Rprofile.site`; it is in the "etc" folder. Open this file in your favourite text editor. We are looking for a line that contains the text `options(help_type="html")`. If there is a hash or number sign (call it what you will) at the beginning of the line, it will stop the command from being executed. Remove the character so that the line starts with `options`. Save the file.

If you've changed the shortcut on the desktop and checked the options for html help, then open R and type `?mean`. You should be able to see full html style help complete with the ability to link to other help pages.

2.3.1 Solution for the problem of the screen reader losing focus

I have encountered a problem where the screen reader loses focus when using the terminal window. This happened under both Windows Vista and Windows 7 using the JAWS screen reader. For more than two years, my only solution was to use the original RGui form of R, and learn to redirect your output to an external file. See Sections 2.6 and 9.4 which present a way of gaining access to the output. It is still a good idea to look at the suggested material, but a solution has come to hand.

In January 2014, Dr Robert Erhardt (Assistant Professor of Mathematics at Wake Forest University) contacted me with a solution. His suggestion is to hit the `ALT` key when the focus gets lost. I've tested this with JAWS 14 and NVDA with R 3.0.2 running under Windows 7 and it worked for me. It still works today.

2.3.2 Super fast default installation of R

In May 2014 I learnt how to install R from a command line. Having downloaded the installation file, I created a batch file with the single line `R-3.1.0-win.exe /silent` that I put in the same folder. This single line is what could be typed in at a command prompt to get R installed silently. Usin a batch file means that when the next version of R comes out, I will only need to change the version number in the filename.

2.4 Setting up R as a blind Mac user

If you're reading this section I assume you know how to work using a Mac. The following instructions were provided to me by Bram Duvigneau(a blind Mac user himself) at the Summer University in 2011, who had read this document and then given me some pointers to share.

Like Windows users, Mac users will find the screen reader for Mac works better if you use the terminal version of R instead of the GUI version. Having obtained the installer for the Mac from CRAN, follow the following steps:

1. Install R using the .pkg installer package you downloaded.
2. Ignore the shortcut that is put in `/Applications/R`, as this is the GUI version.
3. Launch Terminal: press `cmd+shift+u` in Finder and open the Terminal application.

4. Enter `R` (Note this is a capital R) to start the terminal version of R.

2.5 Setting up R as a blind Linux user

If you're reading this section I assume you know how to work in Linux. This means you know how you will obtain the installer for R yourself. Further I assume you know how your screen reader works.

Once you've downloaded the installer and installed R, you should be able to run it by typing "`R`" at the command prompt. Pretty simple really.

Many installations of Linux offer the user the opportunity to install directly so that the download step is not explicitly required.

I have witnessed (first-hand) a blind Linux user operate R without the speech synthesis turned on. His comfort in Linux with a braille display alone was sufficient.

There are a number of reasons why I'd like to be using Linux and R one day. Chief among these is the ability to set up each R script file as an executable file that will run without needing to actually open R itself.

Linux users have a much greater range of options available to them for improving the accessibility of any software. Re-direction of output from an R session is an example.

2.6 Getting the most out of the R window

In this case I actually mean getting anything out of R! Windows users of screen readers will find it difficult to copy and paste any output from R for insertion into a document. The solution is to create a text file that contains the output from the current session.

Use of the `sink()` command is explained in Section 9.4. For the blind user, this text file can be the means by which we gain access to R output as it is created, albeit in a different window. Better ways of working do exist. This suggestion is great for users who must work interactively with an open R session. Improved functionality for doing this is discussed in Section 15.1, but if a blind user can move to working with R in batch mode (one way or another) life will get even easier.

Once the sink is in place, the blind user opens the text file that is being created (and continuously updated) in their preferred browser. After a set of commands that would

normally print output in the terminal/console window have been issued, the blind user only needs to refresh the browser so that the latest output appears.

The result is that the output appears in the sink file if the command was successfully issued. Warning and error messages are printed in the terminal window and are read aloud by the screen reader. Users of the R console window will need to review the screen to see why nothing was added to the sink file.

2.7 More advanced ways of running R programs

Sometimes we will want to run a series of commands. Typing out lots of commands can prove tedious and to be honest no sighted R user does it by issuing commands via the command line one at a time.

R does have its own script window that the GUI version uses. In that window users edit the set of commands they want to run. Sometimes it's easier to obtain the code we want from another R program we used before, or by copying commands from an alternate source. R users often share programs and re-run them locally. Opening a R program in the script window of the GUI version of R, is an option available to the blind user but the outputs from running commands all gets pumped into the main console window where the screen reader may not be able to read it all.

The most advanced users of R who have large jobs to run will run them on another computer, and it is from this approach that my favourite method of running my R programs was developed.

All necessary commands are typed up in a standard text file. R command files are often given the single capital letter R as their extension but they are just plain text files, editable in any text editor program you choose.

For the moment, assume a valid set of commands has been placed in the file `testing.R` and this file has been placed in your working directory. You can see how the sighted R user runs batch jobs and the use of multiple working directories in Chapter 10 on what I think are good ideas.

My first approach uses what the geeks call "piping". We pipe our set of commands into R and pipe out all output to a text file. This is done using a batch file. Batch files are just text files themselves, with commands that are issued in a DOS window, sometimes known as the command prompt. All we need to do is create a new text file and give it the name

we want. This text file will have just one command in it which will look something like

```
"c:\program files\r\r-3.0.2\i386\bin\rterm.exe" --no-save < testing.R > testing.Rout
```

You'll need to edit this command to suit your installation, especially the full path to the executable file which will also depend on your operating system.

When you save this file, it can be called whatever you like but needs to have the `bat` extension. I tend to keep the files together with the same name so would recommend `testing.bat`Just clicking on it in windows explorer runs the file, its single command, and therefore the commands in the `testing.R` file. All output is then printed in the `testing.Rout` file including the commands you issued. The output file is just plain text and can be opened in a text editor or browser.

It's certainly not essential that you come to grips with using the batch file approach to run R in the background. It is fairly essential that you are comfortable writing up the R commands in a text file and pulling them into R using the `source()` command if you intend doing more than just simple jobs.

For the time-being, my recommendation is to run R using the terminal window until you gain some confidence in its operation. Come back to the batch file approach once you know that R is going to do what you need done.

Chapter 3

LURN... To Enter Data

The purpose of this chapter is to show the novice R user how R stores data by introducing the shortcuts that make data entry a fairly simple task.

Entering screeds of data is not fun in any software tool. It's more common for the R user to have the data they need already available from another source. This is covered in Chapter 4 on importing data from alternate sources.

Of course, if you really must enter data manually then you'd better read on; we can at least try to make it as painless as possible.

3.1 Using R as a simple calculator

R can be used to do basic operations whose results do not get stored as objects. We can also assign the answers to a variable. For example

```
> x=100/7
> x

[1] 14.29
```

This means we can use the variable by name later. For example

```
> 12*x

[1] 171.4
```

We can also use some basic mathematical functions such as the logarithm and square root via the `log()` and `sqrt()` commands — many other commands like this exist! For example

```
> x=sqrt(169)
> y=log(500)
> x*y

[1] 80.79
```

OK, these manipulations are trivial, but they can be used in conjunction with other data objects as we will see later. More detail on how to use R as a scientific calculator can be found in Chapter 17.

3.2 A simple set of numbers

Operating on single values is rare. We are usually faced with numbers that we wish to use as a set. Entering them as a set is therefore necessary. The most basic way of entering a set of numbers is using the `c()` command. For example

```
> y=c(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)
> y

 [1]   1   4   9  16  25  36  49  64  81 100
```

is the list of the squares of the first ten natural numbers. We can obtain the numbers from 1 to 10 faster by issuing

```
> x=1:10
> x

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

and therefore can obtain the desired set of the squares for the first ten natural numbers using shorter code based on a simple sequence.

```
> y=(1:10)^2
> y

 [1]   1   4   9  16  25  36  49  64  81 100
```

This is much more efficient than typing out the actual results as done previously. Note that the colon symbol is used for generating series of integers and that in terms of the order of mathematical operators, it comes after the exponent; the brackets around the sequence are essential. In this case, squaring a number is achieved through use of the carat symbol followed by a 2.

3.3 A simple set of text values

The `c()` command is good for entering any kind of data. We may need to enter a set of categories for example.

```
> Names = c("Jonathan", "Elizabeth", "Peter", "Jenna", "Callum", "Annabelle", "Cordelia"
> Names

[1] "Jonathan"  "Elizabeth" "Peter"     "Jenna"
[5] "Callum"    "Annabelle" "Cordelia"
```

Note two important features of this command. I capitalized the variable name here on purpose. The vast majority of R commands are in lower case and there is one called "names". I don't want to confuse my data and an existing R command so my preference is to use an upper case letter on the front of all my variable names that mean anything. The second point is that the character-valued data I entered were encapsulated by quote marks. This means that the actual names (mine, my last dog and some family members) were stored. If I had omitted the quote marks, R would have looked for variables with the appropriate names — not defined in this instance. You could remove a quote or two to see what happens if you must.

3.4 Logical indicators

R often uses logical indicators to tell us something. These variables take the values "TRUE" or "FALSE".

```
> Human = c(TRUE, TRUE, TRUE, FALSE, TRUE, TRUE, TRUE)
```

You might suspect that this variable tells you if the name given in the Names variable are for humans. The fourth name is therefore the dog. To extract the names of the humans we use the command

```
> Names[Human]

[1] "Jonathan"  "Elizabeth" "Peter"     "Callum"
[5] "Annabelle" "Cordelia"
```

Note here that the brackets used are square brackets. Use round brackets for functions, square for elements of an object.

It's pretty simple to extract the dog's name:

```
> Names[!Human]

[1] "Jenna"
```

In this situation the exclamation mark should be read as "not" and therefore picks up the elements where the logical variable is set to FALSE.

3.5 A note on subscripting

We've seen that we can find a subset of the set of names using the indicator variable, but it's frequently useful to be able to extract one or more elements by their location. For example

```
> Names[1]

[1] "Jonathan"
```

gives the first name, and

```
> Names[2:3]

[1] "Elizabeth" "Peter"
```

extracts the second and third names. The set of names entered is a *vector* and has only one subscript to monitor. We will see how to subscript elements within a *matrix* or *data.frame* later.

It's also often necessary to understand how R has stored an object. The `class()` command is useful, and so is the `str()` command. For example

```
> class(Names)

[1] "character"

> str(Names)

 chr [1:7] "Jonathan" "Elizabeth" "Peter" "Jenna" ...
```

We can get the names of all the people that aren't me by

```
> Names[-1]

[1] "Elizabeth" "Peter"     "Jenna"     "Callum"
[5] "Annabelle" "Cordelia"
```

which of course assumes you know my name was given first. The subscripts have used square brackets in these examples. The type (and number) of brackets is crucial. If you open a bracket it must be closed, and closed by a bracket of the same type. Nesting brackets is quite acceptable.

3.6 A patterned set of numbers

In many instances we need to generate series of values in a patterned way. Let's say we want to generate variables that represent the twenty working days over a four week period.

We want a list of the week number, and then a list of the weekday names. In both situations we will use the `rep()` command. It has three arguments; the list of values, the number of times `each` value is to be repeated, and the number of `times` the whole series should be repeated. The second and third of these arguments have default values so may not need to be stated explicitly.

```
> Week = rep(1:4, each=5)
> Week

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

> Day = rep(c("Mon", "Tue", "Wed", "Thu", "Fri"), times=4)
> Day

 [1] "Mon" "Tue" "Wed" "Thu" "Fri" "Mon" "Tue" "Wed" "Thu"
[10] "Fri" "Mon" "Tue" "Wed" "Thu" "Fri" "Mon" "Tue" "Wed"
[19] "Thu" "Fri"
```

These two variables can be brought together with the corresponding data using the `data.frame()` command illustrated later in this chapter.

3.7 Less pattern and more repetition

The `rep()` command is very flexible, and to be honest can either be a lot of fun to play with or just one big headache. Let's say we want to generate the series of numbers which has one 1, two 2's, three 3's, four 4's, and five 5's. Instead of using the constant for the number of times each element is repeated, we can choose the number of repeats for each element.

```
> rep(1:5, times=1:5)

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

Whatever you do you should observe the way your series is coming out. I would have expected the `each` to be used in the last example not `times` for example — always check.

3.8 An incomplete pattern

Let's say we want a set of numbers to be cycled, but know that we won't use the full cycle. The `rep()` has an argument that can stop the process early for us. Try using the `length.out` argument as follows

```
> rep(c(1,2,4), times=3, length.out=8)

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

3.9 Dates and times

The current time and date can be extracted using the `date()` command but this is not an object that can be manipulated — it is a character string only.

```
> date()

[1] "Sun Mar 26 11:05:12 2017"

> class(date())

[1] "character"
```

It might be useful to print this information as part of your documentation of an analysis. We can come back to an R session at a later time and to keep track of when we do things might prove useful. You can see from the output given here the exact time and date this document was compiled. This format is not to be confused with what we would store or manipulate; it is just a print out of the current time and date. The `Sys.Date()` command stores the same information as number.

```
> Sys.Date()

[1] "2017-03-26"

> class(Sys.Date())

[1] "Date"
```

This print out is different to what needs to happen when we want to store numerical values that represent the times and dates particular observations were taken. The base distribution of R does not cater for extracting the date and time in simple numeric terms. This can be achieved, but is beyond the scope of this chapter. It may prove best to store a date using its three constituent parts (day, month, and year) as separate numeric variables. Times should be stored using 24-hour format and be careful not to use a separator between the hour and minute values. Mathematical operations should not be done on these variables unless we convert the minutes to decimal fractions of an hour. In any situation you should decide what you will do with the data before choosing the format you wish to store it in.

An example for storing details of months might be useful here.

```
> Months = as.factor(c(3,6,9,12,3,6,9,12))
> Months

[1] 3  6  9  12 3  6  9  12
Levels: 3 6 9 12
```

The `as.factor()` command tells R that these numbers are to be thought of as non-numeric data. A *factor* also has an associated attribute called `levels`. We can edit the levels directly and this will change our entire variable.

```
> levels(Months) = c("Mar", "Jun", "Sep", "Dec")
> Months

[1] Mar Jun Sep Dec Mar Jun Sep Dec
Levels: Mar Jun Sep Dec
```

As it happens, we didn't need to actually explicitly state the month number when the variable was first created, but it is good practice to keep things logical!

3.10 Larger data objects

There are two data object types that are quite similar but not the same. A *matrix* is a two-dimensional array of values of the same type — numeric, character, or logical. A *data.frame* looks like a matrix but can have variables of different types embedded within it.

For example, we can create a new *data.frame* by combining the names and human status variables created earlier using

```
> data.frame(Names, Human)

      Names Human
1  Jonathan  TRUE
2 Elizabeth  TRUE
3     Peter  TRUE
4     Jenna FALSE
5    Callum  TRUE
6 Annabelle  TRUE
7  Cordelia  TRUE
```

We would usually assign the results of this command to a named object for storage.

```
> MyFirstDF = data.frame(Names, Human)
> str(MyFirstDF)

'data.frame': 7 obs. of  2 variables:
 $ Names: Factor w/ 7 levels "Annabelle","Callum",..: 6 4 7 5 2 1 3
 $ Human: logi  TRUE TRUE TRUE FALSE TRUE TRUE ...
```

Now we can see why we should not confuse `names` and `Names`. We can ask for the names of the variables in a *data.frame* using the `names()` command. For example

```
> names(MyFirstDF)

[1] "Names" "Human"
```

We can also now think about how we might extract the fourth name from the *data.frame* because this is the data structure we will work with the most. There are several alternatives.

```
> MyFirstDF[4,1]

[1] Jenna
7 Levels: Annabelle Callum Cordelia Elizabeth ... Peter

> MyFirstDF[4,"Names"]

[1] Jenna
7 Levels: Annabelle Callum Cordelia Elizabeth ... Peter

> MyFirstDF$Names[4]

[1] Jenna
7 Levels: Annabelle Callum Cordelia Elizabeth ... Peter
```

Notice that as well as the result we wanted, R has also printed the levels of the `Names` variable. This is because this variable has been determined to be a factor.

3.11 Appropriate data labelling

The construction of our first data.frame is slightly flawed. If the `MyFirstDF` data was going to be for all those beings I am in contact with, the details should be related to the individual concerned. In this situation, allowing the `Names` object to be data rather than a label was probably not the wisest move. Let's say that we want the year and month of birth for the individuals in the example, and that they should form a new *data.frame*.

```
> Year = c(1971, 1945, 1925, 2003, 2010, 2012, 2013)
> Month = c("October", "October", "July", "October", "April", "June", "June")
> MySecondDF = data.frame(Year, Month, Human, row.names = Names)
> str(MySecondDF)

'data.frame': 7 obs. of  3 variables:
 $ Year : num  1971 1945 1925 2003 2010 ...
 $ Month: Factor w/ 4 levels "April","July",..: 4 4 2 4 1 3 3
 $ Human: logi  TRUE TRUE TRUE FALSE TRUE TRUE ...
```

3.12 Other approaches

Data entry is tedious. Efficiency is therefore an important weapon in your armoury. When we plan experiments we are often interested in obtaining an observation (or multiple observations) for every combination of some factors. In this simple example, imagine there are three experimental factors, given the names H, W, and Sex, and that each factor can take either of two levels. The `expand.grid()` command is a useful way to construct a *data.frame*.

```
> MyThirdDF = expand.grid(h=c(60,80), w=c(100, 300), sex=c("Male", "Female"))
> MyThirdDF

  h   w    sex
1 60 100   Male
2 80 100   Male
3 60 300   Male
4 80 300   Male
5 60 100 Female
6 80 100 Female
7 60 300 Female
8 80 300 Female
```

I've used a full printout of the resulting *data.frame* instead of using the `str()` command to show what R has created because the `str()` command gives additional information that we do not require at this point.

Let's say we wish to add a new variable to this *data.frame*; a task common when designing an experiment. We can use the $ notation shown earlier in a new way. We add a set of eight random values extracted from a standard normal distribution here as an illustration using the `rnorm()` function.

```
> MyThirdDF$Response = rnorm(8)
> MyThirdDF

  h   w    sex Response
1 60 100   Male -0.53954
```

```
2 80 100   Male  0.09504
3 60 300   Male  0.16238
4 80 300   Male -0.40040
5 60 100 Female  0.07082
6 80 100 Female  0.06128
7 60 300 Female  0.22646
8 80 300 Female -0.66805
```

I like to create random data when planning an analysis. In this instance, the data are normally distributed which is not particularly important, but they are random which is important. There are many other functions that generate random data from other distributions; by convention, these functions all start with a letter “r" followed by a shortened form of the distribution’s name.

I do this because data that will be collected should be appropriate for the intended analysis to be conducted, and likewise the analysis should reflect the way data were collected. Creation of the random data means I can write the R commands that will generate the analysis, and once I have checked that the analysis is possible, I can then collect the data. When the data has been collected, I can then re-process the commands using the real data instead of the random data. This can save time during the analysis but more importantly, I can be confident that the data collection and analysis were planned well.

Chapter 4

LURN... To Import Data

This chapter covers the methods required to pull data from external sources into R. If you want to create data within R, you should be reading Chapter 3.

4.1 Data from external files

While I prefer to use files extracted from EXCEL with comma delimited values, R handles many common formats such as plain text files with space or tab delimiters. You need to know what format a file is, probably by opening it and actually seeing if it is as you expect. It is too easy to rename a file with various extensions which may have meanings in your operating system that have little relevance to R. A specific case in point is when you are presented with a file having the `txt` extension, which is commonly assumed to be a text file. We need to know if the first line of information in the file is actual data or the headings for the columns of data. We also need to know what symbol is used to separate the columns of the data; spaces, commas, or tabs are the most common options. Each of these options has a distinct R command associated with it, but all of these commands link back to the same `read.table()` command.

The various commands are as follows

Delimiter	R command	Common Extensions
space	`read.table()`	DAT, TXT
tab	`read.delim()`	TXT
comma	`read.csv()`	CSV, TXT

Note that the `txt` extension appears as possible extensions for all delimiter types. R will not assume any extension for these commands. You will need to explicitly state the full filename including extension when using these commands. Some extensions will have a default program associated with them by your operating system. For example, `txt` files will be opened in `Notepad` under Windows, and if Microsoft Office is installed on your machine, a `csv` file will be opened in Microsoft EXCEL.

To import a comma delimited file called `chickens.csv`, you would issue the following command

```
> Chickens = read.csv("chickens.csv")
```

In this most basic form, the `read.csv()` command will look for the `chickens.csv` file in the current working directory, and import it into R and store it as a *data.frame* called `Chickens`. The default settings of the `read.csv()` command are to have a `header` row in the file and to have no `row.names` attribute associated with the data in the file. If your data file already had a column for the names of the chickens as the first column, you would issue the command

```
> Chickens = read.csv("chickens.csv", row.names=1)
```

and if the data did not currently have any column headings you would issue the command

```
> Chickens = read.csv("chickens.csv", header=FALSE)
```

There are other settings to consider which you can investigate using the help for the `read.csv()` command by typing `?read.csv` at the command prompt. This help page is actually a combined help page for the family of commands described in this section.

The working above assumed you could put the data file into the correct working directory. So where was that? To find out where R thinks you are currently working, use the `getwd()` command. Note that the output may look a little strange to some users, especially Windows users. What do I mean? Look at the following:

```
> getwd()

[1] "C:/Users/ajgodfre/Documents/Research/Publications/LetsUseRNow"
```

The full path to the working directory where this chapter was processed has been displayed, starting with the letter associated with the hard drive, followed by a colon. Then the fun begins. The folder structure is represented using forward slash signs, not the backslash used in Windows operating systems even though the processing of this work is done using a Windows machine. The rest should be as expected and you could find the right folder by looking in the appropriate place on your hard drive. The reason for R's use of the forward slash is not entirely simple to explain, but in short it is because the standard backslash symbol has a special use in R. For the moment, the choice of slash versus backslash is not important. It is important when we need to type out the path to the location of a file for ourselves.

4.2 Data stored in another directory

I do try to keep each separate project in its own distinct directory, and moving the raw data file to that directory makes sense. It does not make sense to have multiple copies of a dataset though, so we need to know how to pull a file from a different working directory into R.

When I displayed the current working directory in the previous section using the `getwd()` command, we saw the way that R used forward slash symbols to denote the hierarchy of folders on our hard drive.

If we know the complete specification of the location of a file, right the way from the name of the hard drive down the directory tree to the actual location, the way we specify the location needs to match the way R has printed a path. That is, we use the forward slash symbol not a single backslash. If we do want to use a backslash symbol, we would need to use a double backslash, not just a single one. It is better to use a single forward slash symbol however. This is because the single forward slash presentation works for all operating systems and means our code can be shared to users of all operating systems. You might not plan to do this right now, but let's use good habits from the start.

It is common for data to be stored in a folder that is close to the one we are working in.

We might have a folder called `MyData` which is within the working directory (a subfolder), or it might be a folder at the same level in the directory tree as the current working directory. In either case, we don't need to specify the location using the full path. The term used to describe the location is a *relative path* because the reference to the location is relative to the current location having focus (our working directory).

If our data set is stored in the file `chickens.csv` in a subfolder called `MyData`, we can use

```
> Chickens = read.csv("MyData/chickens.csv")
```

to pull it into our current workspace. This is actually shorthand for the more complete form

```
> Chickens = read.csv("./MyData/chickens.csv")
```

where the current folder is denoted using the single period followed by the first slash. Personally, I would prefer to see the more complete form of this relative path being used but it is personal preference only.

If the `MyData` folder was not a subfolder, but was on the same level in the directory tree as our current working directory, we would use the shorthand symbol for the parent directory (one level up the directory tree). This is done using a double period.

```
> Chickens = read.csv("../MyData/chickens.csv")
```

Relative paths are therefore quite useful because they avoid having to type out long paths. The full specification of apath might also become a problem when files for a project are moved from one storage device to another. For example, you might want to take your work to a friend, tutor, or colleague on a memory stick or other portable storage medium. Relative path referencing makes your work very transferable and transportable.

4.3 Data saved by other statistical software

Many statistical software packages use their own file types for storing data. R is no different actually! The chief problem we have is to find a way of transferring data from one

application to another. Like most other statistics programs, R doesn't handle all other file types. Some files can be imported into R using the commands in the *foreign* package, but it is probably best just to avoid the problems from the start.

In many instances it will prove easiest to use copy and paste functionality within your operating system to take the data from whatever original source it was given to you in and put it into a suitable spreadsheet program. Then save it using the comma separated values format and read the data into R using the commands given in the previous section.

I recommend trying to obtain the data in an easily imported file type rather than attempting to use the functions in the *foreign* package.

4.4 Data from files stored on the internet

Sometimes a data set is made available via the internet. If you can obtain the full URL for the downloadable data file then it can be entered into the `read.csv()` or `read.table()` commands. This exercise is seldom necessary except for data files that you know will be updated for distribution through the web. Some government agencies and financial database services do this.

4.5 Data contained in contributed packages

The base installation of R includes a package called *datasets*. These data sets are useful for testing code and writing examples for insertion into documents like this one. Data sets contained in the *datasets* package are actually ready and waiting to be accessed, but often we want to bring the data into our current workspace using a command such as

```
> data(airquality)
> str(airquality)

'data.frame': 153 obs. of  6 variables:
 $ Ozone  : int  41 36 12 18 NA 28 23 19 8 NA ...
 $ Solar.R: int  190 118 149 313 NA NA 299 99 19 194 ...
 $ Wind   : num  7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
 $ Temp   : int  67 72 74 62 56 66 65 59 61 69 ...
```

```
 $ Month  : int  5 5 5 5 5 5 5 5 5 5 ...
 $ Day    : int  1 2 3 4 5 6 7 8 9 10 ...

> ls()

[1] "airquality"
```

The `data()` command looks in the *datasets* package by default. If we wanted to get some data from another package we would need to state the name of the package explicitly. For example,

```
> data(anorexia, package="MASS")
```

(The *MASS* package is already installed by default.)

Often we will be using a particular data set because it is good for demonstrating functions within a particular package. If we are loading the package using the `library()` command, to get access to the functions, we will have also made the data available. This is why the data from the *datasets* is ready for use; this package is loaded by default whenever R is started.

4.6 Data cleanliness

Rather ironically, the suggestion for including this section came from my mother. All too frequently, data are entered by people who will not actually need to use the data themselves. Sometimes data are entered by different people and then compiled into a single dataset and the various codes that some sources of data may use are not necessarily in common with all other users. Take for example, the many ways we might enter data for the gender of individuals answering a survey. You might easily realise that an "`M`" means male, but R needs consistency.

It is extremely important that we have knowledge of the format of the data we import. Use of the `str()` command is a good start, and use of the `head()` command might also be useful.

Here are some pointers to look out for:

- Have you mixed the case of the text in your data? Remember that R is case sensitive and that “R" is not the same as “r" in R.

- Has R tried to be too smart and interpretted plain text information as if you wanted a *factor* instead? This occurs frequently with date values which are notorious for the variety of formats people choose. If you have a factor when you wanted just plain text, investigate the `as.character()` command which changes the format to being character valued data.

- Have a multitude of codes been used for variables like gender? If so, R will decide that the variable is a *factor* and list the different values the variable can take. The levels of the factor can be re-assigned using the `levels()` command. See how this is done on 18 and note that two elements of the assignment can be the same if the current codes mean the same thing.

- Have extra spaces crept into the codes for levels of factors somehow? This happens when an extra space occurs between words or is added on the end of a text string. Often, this problem is difficult to spot visually as we are talking about white space — R will find it though! Use the `levels()` command to fix this if the problem is small, but consider using a find/replace tool in a spreadsheet application as an alternative.

- Have numbers come into R as text for some reason? This happens for a number of reasons. A blank space being entered in the cell instead of a code for a missing value for example. R would see this as a character string “ ", not an empty cell. Another reason is that the missing value code is textual and not recognized by R. There are ways around this problem using either find/replace techniques in your spreadsheet application to delete the missing value codes, or by changing the way the `read.table()` or `read.csv()` command reads your data file. See the help for these commands if this is the case.

4.7 Other packages

It is all too common to need to import data from a spreadsheet application where the data is not conveniently placed in the upper-left corner of the first sheet. There are a number of such spreadsheet applications, but the most commonly used one is Microsoft Excel.

Data in spreadsheets is seldom ready for importing in the exact form we want it. If you need to extract a set of data that is in a named sheet or always appears in the same numbered sheet of a Microsoft Excel file, then you might investigate the *openxlsx* package. See Chapter 14 and the help page for the `read.xlsx()` command.

Chapter 5

LURN... To Manipulate your Datasets

This chapter explains how to rearrange your data to meet your needs. Basic sorting, data extraction and re-combination are discussed.

To get the most of this chapter, you should have worked through Chapter 3 on entering data manually. You do not need to have completely read through Chapter 4 on importing data as the examples here do not rely on its content. Ultimately, you will want to use the material in Chapter 4 in conjunction with what follows here.

This chapter starts using the functionality that is built into R, but we also take a look at an add-on package called *dplyr* that is becomeing more and more commonly used.

5.1 Sorting the data into a different order

The `sort()` command is illustrated in this section. It works for sorting a single vector of numeric values into ascending or descending order.

```
> x=round(rnorm(10),2)
> sort(x)

 [1] -0.89 -0.67 -0.54 -0.40  0.06  0.07  0.10  0.11  0.16
[10]  0.23

> sort(x, decreasing=TRUE)
```

```
 [1]  0.23  0.16  0.11  0.10  0.07  0.06 -0.40 -0.54 -0.67
[10] -0.89
```

We use the `order()` command to work with data that are non-numeric vectors, as well as data in *matrix* or *data.frame* format. For example, we first sort the names of me and my three children initially given in descending order of age.

```
> Home=c("Jonathan", "Callum", "Annabelle", "Cordelia", "Hershey")
> order(Home)

[1] 3 2 4 5 1

> Home[order(Home)]

[1] "Annabelle" "Callum"    "Cordelia"  "Hershey"
[5] "Jonathan"
```

So we see that the `order()` command doesn't actually do the re-ordering after all, but that we need to use the subscripts to extract the elements (the full set) from the pre-existing set.

We do need to take note of how the upper and lower case letters will be ordered.

```
> SomeLetters=c("A", "B", "a", "c", "b", "C")
> SomeLetters[order(SomeLetters)]

[1] "a" "A" "b" "B" "c" "C"
```

The `order()` command can prove a useful method of re-ordering experimental data if the run order has been stored as a variable. To illustrate this point we create a *data.frame* with two elements.

```
> x=round(rnorm(10),2)
> x

 [1] -0.35 -1.14  0.53 -0.46 -0.02 -1.13 -0.05  0.56  0.50
[10] -0.73

> RunOrder=sample(10)
```

Note that the `sample()` command used here has created a random permutation of the numbers one to ten.

```
> RunOrder

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

> Data=data.frame(x,RunOrder)
> Data[order(RunOrder),]

       x RunOrder
9   0.50        1
5  -0.02        2
4  -0.46        3
2  -1.14        4
7  -0.05        5
6  -1.13        6
10 -0.73        7
8   0.56        8
1  -0.35        9
3   0.53       10
```

5.2 Extracting data using information on one variable

To illustrate the extraction of a set of observations from a *data.frame*, we make use of one of the internal datasets provided with the base installation of R. We retreiv the `airquality` dat using:

```
> data(airquality)
```

An explanation of this command is given in Section 4.5. This data set has its own help page that can be viewed by typing `?airquality` at the R prompt.

We can ask for the rows of this data that match specified details using subscripting. Various operators exist for numerical comparisons; the simplest of which are < and >. For

the purposes of brevity in our examples that follow, we use the `nrow()` command to show how many observations in the data meet the specified conditions. The following commands extract hte number of days where the wind speed was less than ten miles per hour and then the temperature was less than 60 degF.

```
> nrow(airquality[airquality$Wind<10,])

[1] 81

> nrow(airquality[airquality$Temp<60,])

[1] 8
```

If we want to know how many observations have an exact value, we use == not a single equals sign because it has a different meaning in R. The double equals sign is the one used for comparisons. For example, the number of days when the temperature was recorded as 69 degF can be found using:

```
> nrow(airquality[airquality$Temp==69,])

[1] 3
```

5.3 Extracting data using information on more than one variable

To print out the rows of the `airquality` *data.frame* where the temperature was less than 60 degrees and the wind was less than ten miles per hour, we would use the `&` operator between the two conditions that identify which rows are required.

```
> airquality[airquality$Wind<10&airquality$Temp<60,]

   Ozone Solar.R Wind Temp Month Day
21     1       8  9.7   59     5  21
27    NA      NA  8.0   57     5  27
```

5.4 Use of dplyr for data manipulation

The *dplyr* package is an alternative to the basic R functionality shown in this chapter. The package creator (Hadley Wickham) wants his package to make data manipulation easier for the end-user, and more fficient with respect to computation time. He has simplified data manipulations to a small set of commands: `filter()`, `arrange()`, `sample_n()`, `sample_frac()`, `mutate()`, and `transmute()` are discussed here, while the others will be discussed in Section 8.5 where they are more relevant.

You'll need to install the package (see Section 14.1 for instructions) before running the function

```
> library(dplyr)


Attaching package:  'dplyr'
The following objects are masked from 'package:stats':
       filter, lag
The following objects are masked from 'package:base':
       intersect, setdiff, setequal, union
```

to gain access to the commands just listed. Note that the `head()` command is used in many of the following examples to ensure that only a handful of rows are printed out instead of the full result. An alternative is to create a new data structure using the `tbl_df()` command. Let's make a second copy of the air quality data that is the same as the original except for being made ready for use with the *dplyr* functions.

```
> airquality2 = tbl_df(airquality)
```

and then see the structure of the data using the `glimpse()` function that is a substitute for the `str()` command we've used previously.

```
> glimpse(airquality)

Observations: 153
Variables: 6
```

```
$ Ozone   <int> 41, 36, 12, 18, NA, 28, 23, 19, 8, NA, ...
$ Solar.R <int> 190, 118, 149, 313, NA, NA, 299, 99, 19...
$ Wind    <dbl> 7.4, 8.0, 12.6, 11.5, 14.3, 14.9, 8.6, ...
$ Temp    <int> 67, 72, 74, 62, 56, 66, 65, 59, 61, 69,...
$ Month   <int> 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, ...
$ Day     <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, ...
```

The `glimpse()` command works on a *data.frame* or the new structure called a *tbl_df* equally well.

The `filter()` and `arrange()` commands make many of the examples seen earlier in this chapter much easier. The `filter()` command is for extraction, while `arrange()` is for rearrangement. For example,

```
> filter(airquality, Wind<10&Temp<60)

  Ozone Solar.R Wind Temp Month Day
1     1       8  9.7   59     5  21
2    NA      NA  8.0   57     5  27
```

extracts the rows of the `airquality` data that meet the combinations of conditions specified. Use of `filter()` means we don't need the *data.frame* attached, or to gain access to the variables in it using the dollar notation; we also don't need the square brackets for common subscripting tasks. We could have stored the outcome in a new *data.frame* if we wished.

Perhaps the greatest gift of the *dplyr* package is its popularising a different way of combining commands, called the *pipe operator*. The last command issued can be re-written as:

```
> airquality %>% filter(Wind<10&Temp<60)

  Ozone Solar.R Wind Temp Month Day
1     1       8  9.7   59     5  21
2    NA      NA  8.0   57     5  27
```

This way of presenting commands is considered to be easier to read by many users which helps with the popularity of the *dplyr* package. The *pipe operator* comes from another package called *magrittr* so its use is not limited to the commands found in the *dplyr* package. This way of presenting multiple commands is called “chaining" in many resources. I like the simplicity that this offers so I have tried to use the *pipe operator* as often as I can from here on.

The `filter()` command is useful if we know the values a variable takes, but at times we just want to know more about the observations that are best or worst according to one (or more) variables. For example, if we wanted to find the hottest days, we can use the `arrange()` command instead of the `order()` command seen earlier.

```
> airquality2 %>% arrange(desc(Temp))

# A tibble: 153 × 6
   Ozone Solar.R  Wind  Temp Month   Day
   <int>   <int> <dbl> <int> <int> <int>
1     76     203   9.7    97     8    28
2     84     237   6.3    96     8    30
3    118     225   2.3    94     8    29
4     85     188   6.3    94     8    31
5     NA     259  10.9    93     6    11
6     73     183   2.8    93     9     3
7     91     189   4.6    93     9     4
8     NA     250   9.2    92     6    12
9     97     267   6.3    92     7     8
10    97     272   5.7    92     7     9
# ... with 143 more rows
```

We can add more variables to sort the data by if we like. We used the `desc` to get the temperatures to be in descending order in this example.

The temperatures just listed are measured on the Fahrenheit scale. I live in a country that uses the centigrade scale so a conversion is needed to make these results more interpretable. Enter the `mutate()` and `transmute()` commands.

```
> airquality %>% mutate(TempC = (Temp - 32) * 5 / 9) %>% head()

  Ozone Solar.R Wind Temp Month Day TempC
1    41     190  7.4   67     5   1 19.44
2    36     118  8.0   72     5   2 22.22
3    12     149 12.6   74     5   3 23.33
4    18     313 11.5   62     5   4 16.67
5    NA      NA 14.3   56     5   5 13.33
6    28      NA 14.9   66     5   6 18.89
```

will create a new column in the returned data. N.B. the new data was not stored in a new object so this command's effect was purely temporary.

Again, notice that we didn't need to do all that much to add a new column and that this code is perhaps a little simpler than the alternative:

```
> airquality$TempC = (airquality$Temp - 32) * 5 / 9
```

which we saw earlier. The main difference between the two commands is that this one will add the new colun to the original *data.frame*. In this instance the difference is probably fairly trivial, but when we use much larger data sets, the storage of results is important because of speed and memory usage concerns.

The command

```
> airquality2 %>% transmute(TempC = (Temp - 32) * 5 / 9)

# A tibble: 153 × 1
   TempC
   <dbl>
1  19.44
2  22.22
3  23.33
4  16.67
5  13.33
6  18.89
```

```
7  18.33
8  15.00
9  16.11
10 20.56
# ... with 143 more rows
```

creates the new variable as a stand alone object. It hasn't been stored as an object so is only temporarily available.

Finally, the *dplyr* package also gives users two simple ways to extract a random sample from a *data.frame*. The commands `sample_n()` and `sample_frac()` extract a sample of a predetermined size and a specified fraction respectively. For example:

```
> airquality %>% sample_n(4)

    Ozone Solar.R Wind Temp Month Day
145    23      14  9.2   71     9  22
122    84     237  6.3   96     8  30
38     29     127  9.7   82     6   7
18      6      78 18.4   57     5  18
```

and

```
> airquality %>% sample_frac(0.05)

    Ozone Solar.R Wind Temp Month Day
41     39     323 11.5   87     6  10
39     NA     273  6.9   87     6   8
108    22      71 10.3   77     8  16
111    31     244 10.9   78     8  19
110    23     115  7.4   76     8  18
101   110     207  8.0   90     8   9
31     37     279  7.4   76     5  31
68     77     276  5.1   88     7   7
```

The *dplyr* package is under active development and so we should expect more functionality to become available, but there is also work being done to make the processing of data even faster so that it can handle huge data sets.

Chapter 6

LURN... To Export a Dataset

This chapter assumes you've got some data to export, and I mean data; not results from analyses which is discussed in Chapter 9. You may also wish to use this information once you know how to tabulate various numerical summaries, discussed in Chapter 8.

6.1 Creating external files

This section is just the inverse action of importing data from an external file as discussed in Chapter 4. All the `read` type commands have corresponding `write` commands.

If you read Chapter 4 you'll know that my preference is to use files that are easily transferred and easily checked for their accuracy using other software. In particular, I prefer to create comma separated value (csv) files using the `write.csv()` command. If there was a *data.frame* called `Chickens` in my current workspace that I wanted to export for sharing with another user, I would issue the following command.

```
> write.csv(Chickens, file="chickens.csv")
```

If R had created rownames for the *data.frame*, I might not want to include them. A simple additional argument is all that is required.

```
> write.csv(Chickens, file="chickens.csv", row.names=FALSE)
```

If another file type is required then the help page for `write.csv()` may be useful. This page also gives the help on the `write.table()` command.

6.2 Exporting data for use in alternative software

The better statistical software options allow users to import data from a range of sources and file types. If for some (very strange) reason you need to create files in a specific format for importing into another statistics package then you will need to investigate the *foreign* package.

Chapter 7

LURN... To Create Simple Graphs

This chapter illustrates the construction of some basic exploratory data analysis graphs. More complex graphs are considered in Chapter 13, although at times we will see graphs presented in conjunction with particular statistical analysis techniques in other chapters. In this chapter, we concentrate on data taking continuous values initially, but a short section on graphs for discrete valued variables is also included.

There are many ways to create statistical graphs. The traditional method uses one of the base R packages, but in recent years, the *ggplot2* package has gained considerable attention. The presentation of both styles of graph are given throughout this chapter. You will need to ensure the *ggplot2* package is available for use in your R session by issuing the command:

```
> str(airquality)

'data.frame': 153 obs. of  6 variables:
 $ Ozone  : int  41 36 12 18 NA 28 23 19 8 NA ...
 $ Solar.R: int  190 118 149 313 NA NA 299 99 19 194 ...
 $ Wind   : num  7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
 $ Temp   : int  67 72 74 62 56 66 65 59 61 69 ...
 $ Month  : int  5 5 5 5 5 5 5 5 5 5 ...
 $ Day    : int  1 2 3 4 5 6 7 8 9 10 ...
```

It is common for R users to access the variables after issuing the command

```
> attach(airquality)
```

The `detach()` command undoes the `attach()` command. To remove the "attachment", you will issue the command `detach(airquality)`.

You could look at the help file for this data if you wanted to learn its complete story using `?airquality`. It tells us that the data are for daily readings of the following air quality values for May 1, 1973 (a Tuesday) to September 30, 1973.

- `Ozone`: Mean ozone in parts per billion from 1300 to 1500 hours at Roosevelt Island
- `Solar.R`: Solar radiation in Langleys in the frequency band 4000–7700 Angstroms from 0800 to 1200 hours at Central Park
- `Wind`: Average wind speed in miles per hour at 0700 and 1000 hours at LaGuardia Airport
- `Temp`: Maximum daily temperature in degrees Fahrenheit at La Guardia Airport.

This data suits our purposes for the majority of examples in this chapter but we will also need to look at another data set for the discussion of graphs for discrete valued variables.

7.1 Histograms

Like many graphical functions in R, the `hist()` command will attempt to make a suitably attractive histogram with the minimum of input from the user. Exhibit 7.1 shows what results from the simplest application of the `hist()` command.

Note that the figure created has default settings for the main title, axis labels and that the number of classes (also called bins) and the cutoffs between them have been chosen automatically. Various methods exist for these choices, but it is my recommendation that the user find out what happens when the default settings are chosen and then alter only what is actually necessary. For example, graphs in this document do not always need the default title inserted so we need to suppress the default action if we want to remove the title. We may also want more informative axis labels. Both of these alterations are done for the creation of Exhibit 7.2.

Exhibit 7.1 Histogram of Average wind speed in miles per hour at 0700 and 1000 hours at LaGuardia Airport. Obtained from the `air quality` data set.

```
> hist(Wind)
```

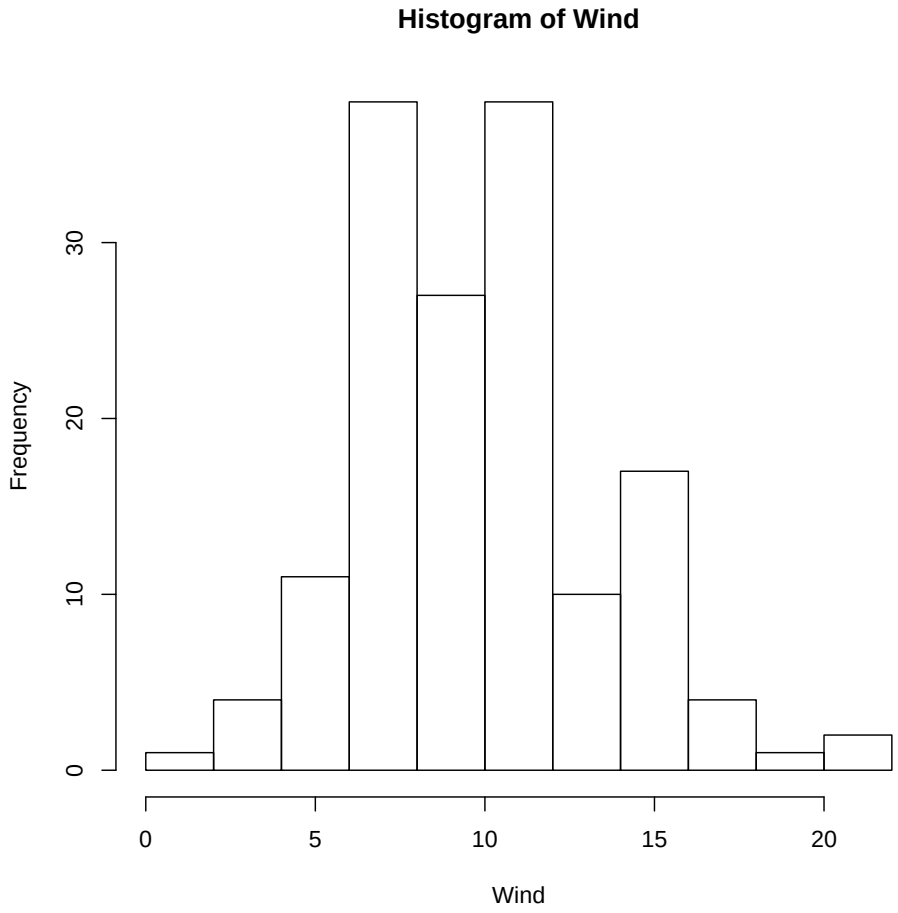
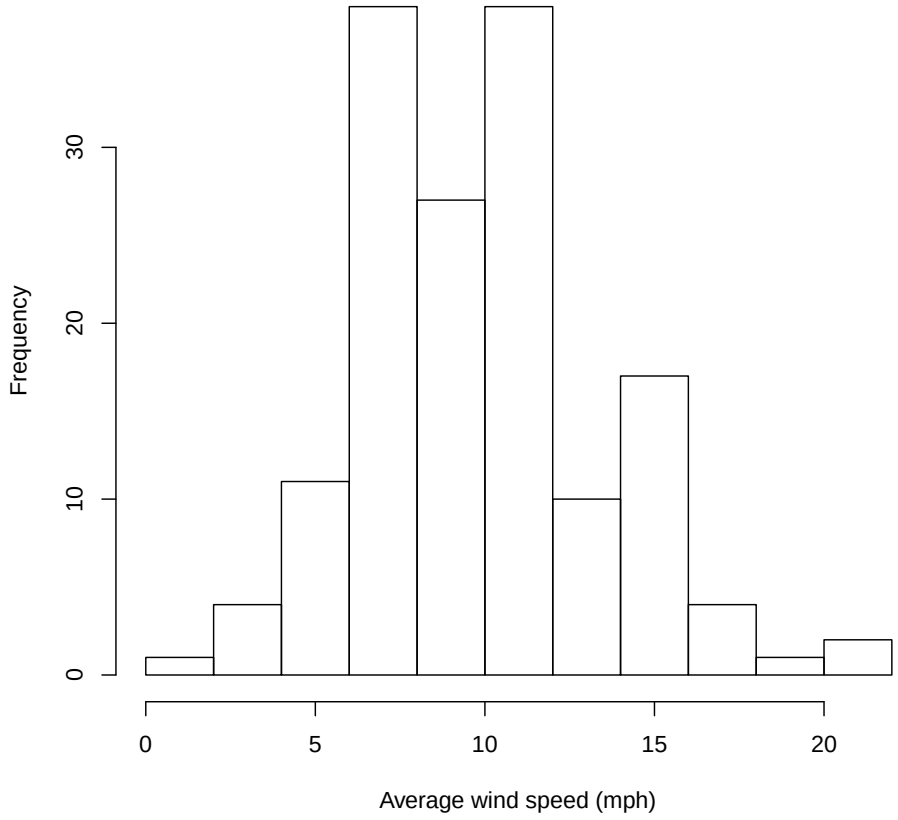


Exhibit 7.2 Histogram of Average wind speed at 0700 and 1000 hours at New York's LaGuardia Airport. Obtained from the `air quality` data set.

```
> hist(Wind, xlab="Average wind speed (mph)", main="")
```



7.2 Basic annotations to graphs

Variable names should be informative but aren't always what we want to see in graphs. Notice that in Exhibit 7.2, I've made the x-axis label more informative by indicating the units of measurement for the wind speed. As I already have a caption for my Exhibit, I have chosen to remove the default title by adding the argument `main=""` to the `hist()` command.

Aside from the alteration of the default axis label and the change in the main title for

our histograms, we could make quite a number of changes. The `hist()` function allows the user various options for the way the bars are filled in for example. It's often worth checking the default behaviour and then seeing if the resulting graph is what you want. If it isn't, then investigate your options by looking at the relevant help file; in this case type `?hist` to get to the help for the `hist()` function.

Other graphs we create will show points marked with small hollow circles; we may wish to make these circles smaller, joined by line segments, filled in, a different colour or combinations of all of these attributes. The `par()` function should be investigated to find out what is possible. Most graph functions allow the user the option of adding arguments that will be passed directly to the `par()` function to obtain the same behaviour. Investigate the graphical parameters at some stage using `?par` but be warned, there are lots of adjustments that can be made. Experimenting is really the only option when it comes time to get your graph looking perfect.

Additional text and/or lines can be added to some graph types. It may prove useful to show the line of best fit along with the data (illustrated in Chapter 12) for example. We'll see how to do the fancy things in Chapter 13.

7.3 Other univariate summary graphs

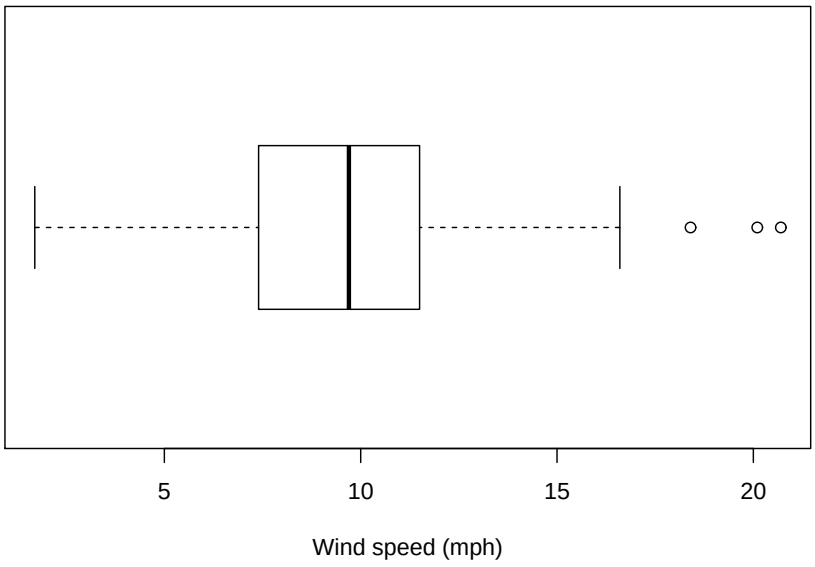
Histograms aren't the "be all and end all" for univariate summary graphs. As a case in point, they aren't at all useful for small sample sizes. Various other graphs appear in introductory statistics courses and that is why they appear here. I don't mean to support one over another at all — that's up to you to determine.

7.3.1 Boxplots

Boxplots show us quickly the shape of the distribution of a sample. They show the median, lower and upper quartiles, and the minimum and maximum of a sample. They will also identify points as outliers if these points are too far from the bulk of values in the sample.

Exhibit 7.3 shows us the boxplot for the average wind speed at LaGuardia airport. Notice that I have added an additional command to set up the size of the graph window. The `windows()` command has various aliases — `x11()`, `X11()`, `win.graph()` — none of which are required if the standard width and height of the graph window are acceptable.

Exhibit 7.3 Boxplot of Average wind speed in miles per hour at 0700 and 1000 hours at LaGuardia Airport. Obtained from the `air quality` data set.



You may wish to see why I've changed the height of the graph window by ignoring the `windows()` command from Exhibit 7.3

Notice that I've changed the default orientation of the boxplot by adding the argument `horizontal=TRUE` to the `boxplot()` command. I have also ensured the more informative axis label for wind speed is included using the `xlab` argument in the command.

7.3.2 Comparative boxplots

Boxplots are often useful for comparing several small samples at once. We must ensure that the same axis is used for the units of measure of interest and the easiest way to ensure this is to put the various samples into the same graphic with only one axis rather than having a series of single boxplots each having their own axis.

For the purposes of illustration, I want to show the distribution of the daily average wind speeds for the five months separately. The comparative boxplot is created using a formula to describe the relationship between the two variables that are referred to in our graph. The use of the tilde symbol between `Wind` and `Month` could be read as something

Exhibit 7.4 Comparative boxplots for the Average wind speed in miles per hour at 0700 and 1000 hours at LaGuardia Airport separated into groups for the months of May to September 1973. Data was Obtained from the `air quality` data set.

```
> boxplot(Wind~Month, xlab="Month", ylab="Wind speed (mph)")
```

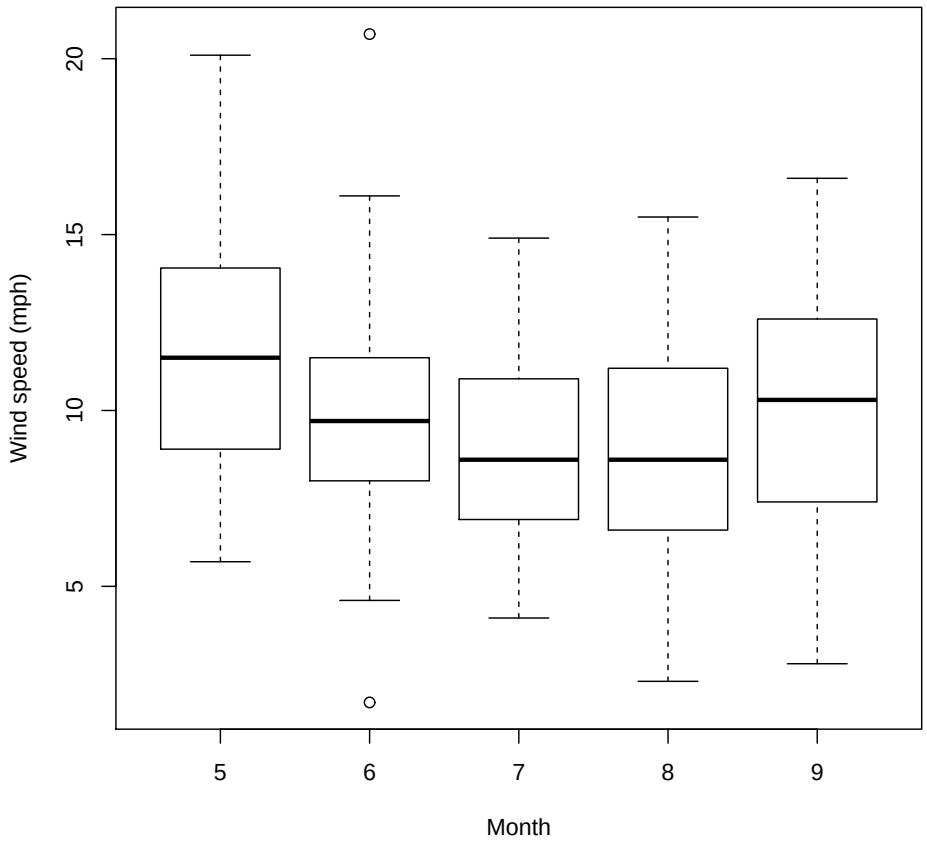
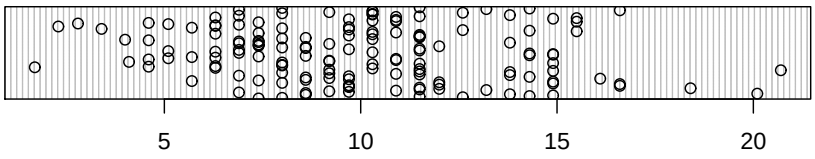


Exhibit 7.5 Dotplot of the Average wind speed in miles per hour at 0700 and 1000 hours at LaGuardia Airport. Obtained from the `air quality` data set.



like "average wind speed depends on the month" — well a theory that might be illustrated in our graph anyway. Certainly it is the potential for this relationship to exist that may be exposed through use of the comparative boxplot.

7.3.3 Dotplots

Dotplots aren't everyone's cup of tea but they are frequently offered as substitutes for boxplots and histograms. Again, I chose to alter the default window size for the example given in Exhibit 7.5 because I didn't like the way the regular graph window presented this graph.

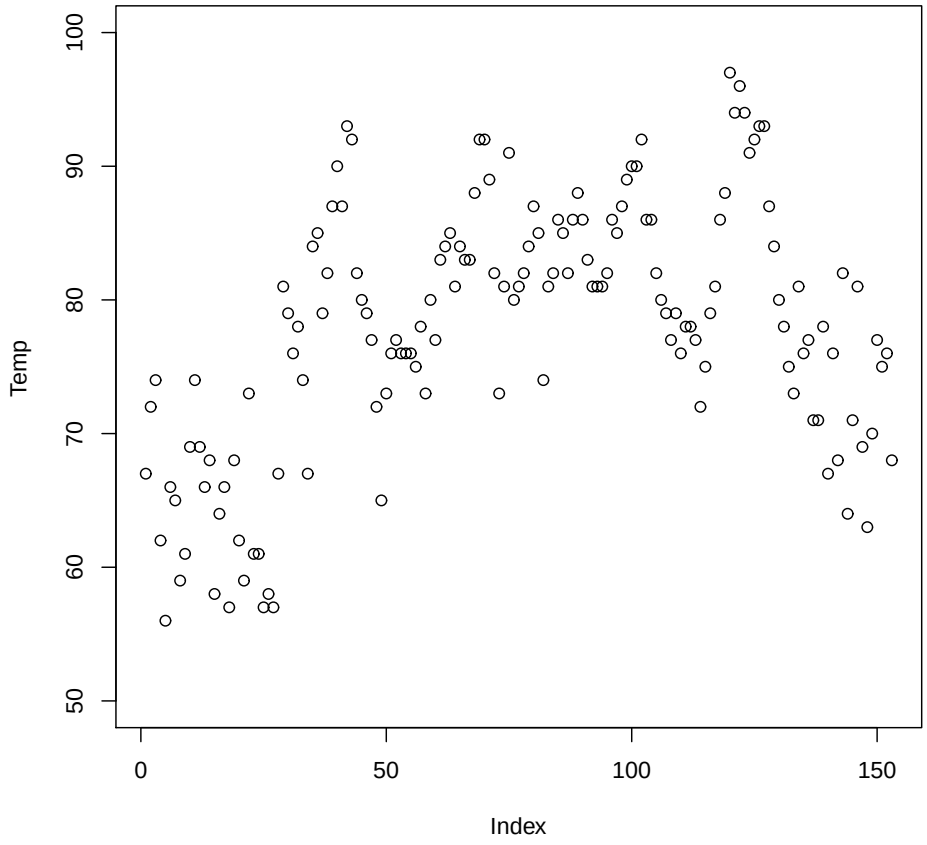
Some people find the standard dotplot that R creates a little artificial. The spacing between points horizontally and vertically is captured by the human eye, but this graph is a single dimensional representation. Adding jitter to the data allows points where there are ties to be represented by a pair of points on the graph rather than two perfectly overlaid points which look like a single point. The `jitter()` command can be embedded within the command for creation of a dotplot, but is not required for our example data as there are no ties. If there were a large number of ties, we would have used the command `dotchart(jitter(Wind))`.

7.3.4 Simple line plots

Occasionally it's useful to see how a measurement changes over the time data were collected. R does this very simply using the `plot()` command as shown in Exhibit 7.6 which is for

Exhibit 7.6 Line plot of the maximum daily temperatures from 1 May to 30 September 1973, measured in degrees Fahrenheit, at LaGuardia Airport. Data were obtained from the `air quality` data set.

```
> plot(Temp, ylim=c(50,100))
```

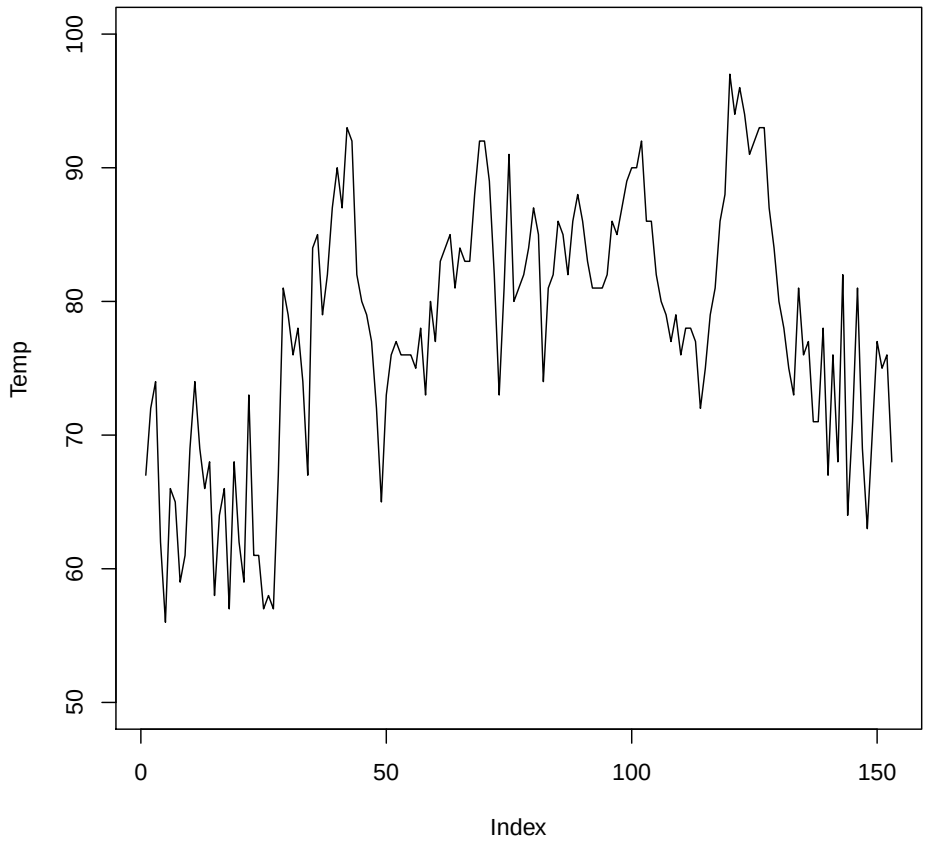


the maximum daily air temperatures at LaGuardia airport from 1 May to 30 September 1973. We can see periods of time where the maximum temperatures were fairly consistent and periods where it was fairly volatile. The middle of the graph is for the month of July which is the height of summer in New York and as a consequence we expect to see few points nearer the lower part of the graph during this period.

Notice that we have altered the range of values covered by the y-axis using a specific command. The `ylim` has a corresponding `xlim` to create limits for the x-axis. Adding

Exhibit 7.7 Line plot of the maximum daily temperatures from 1 May to 30 September 1973, measured in degrees Fahrenheit, at LaGuardia Airport. Data were obtained from the `air quality` data set.

```
> plot(Temp, ylim=c(50,100), type="l")
```



one more argument to the `plot()` command will change the plotting from points to lines (shown in Exhibit 7.7). Combinations of points and lines can be obtained (not shown); the user can also alter the style of the points and lines being printed.

There is a simpler way to generate time series plots which we demonstrate in Chapter **??**. It is easier to augment this line plot than the time series plot and in so doing, we will gain an insight into how other plots like the time series plot are created.

7.4 Quantile-quantile plots

The most common quantile-quantile plot we might wish to create is used to investigate the usefulness of the normal distribution to model a variable's distribution. Normal quantiles are created automatically for the normal quantile plot when it is generated using the `qqnorm()` command. The default plot for this is shown in Exhibit 7.8.

If these data were normally distributed, the points on the plot would lie on a straight line. The `qqline()` command adds the straight line to the plot to assist with determining the linearity of the points.

7.5 Scatter plots

Scatter plots are created using the `plot()` command by one of two methods. Exhibit 7.9 was created using

```
> plot(Wind, Temp)
```

but the same plot can be generated using what is known as a formula. In this case, only one argument is given to the `plot()` command but both variables of interest are in that argument.

```
> plot(Temp~Wind)
```

The tilde symbol is often read as "...is distributed as..." but we might simplify this to be read as "follows". This makes some sense as we generally create a scatter plot to see if one variable follows the other; in this case we are seeing if temperature depends on wind speed in some way.

Exhibit 7.8 Normal probability plot of the Average wind speed in miles per hour at 0700 and 1000 hours at LaGuardia Airport. Obtained from the `air quality` data set.

```
> qqnorm(Wind)
> qqline(Wind)
```

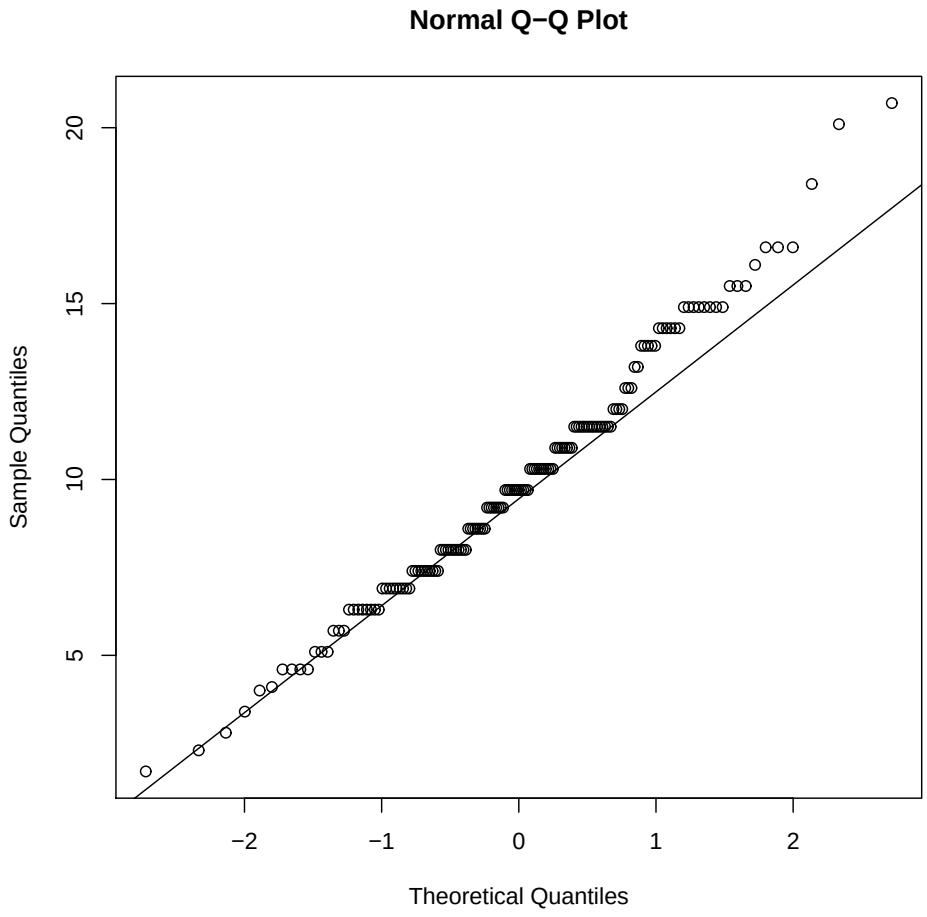
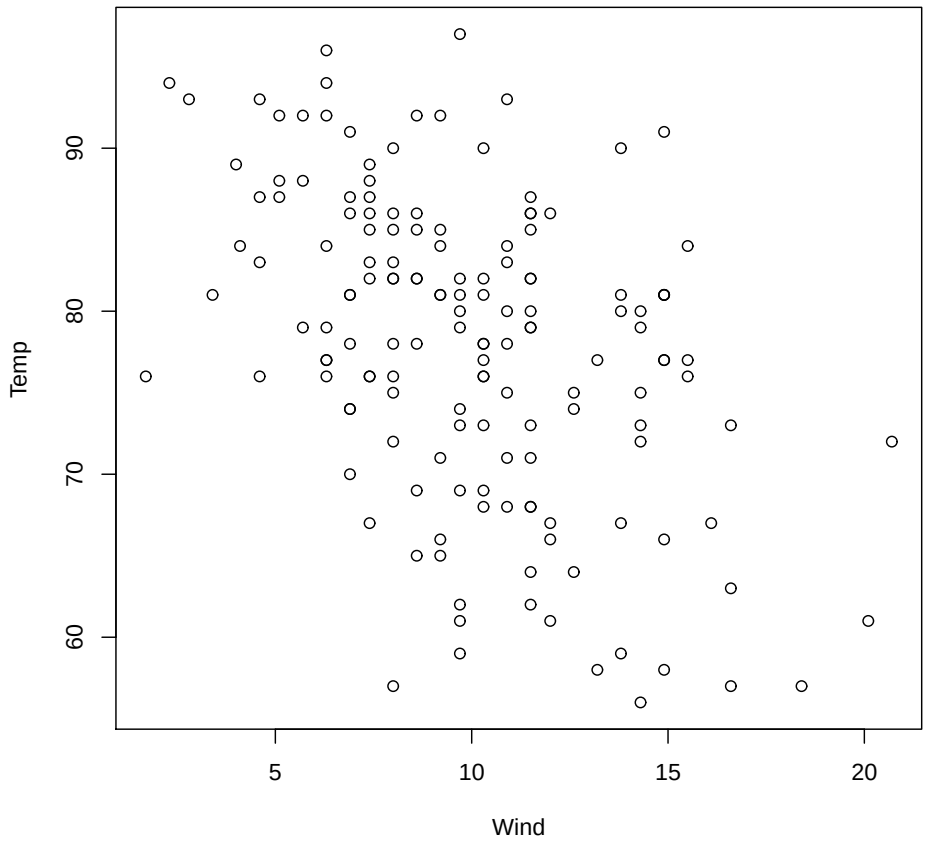


Exhibit 7.9 Scatter plot of the maximum daily temperature against the Average wind speed at 0700 and 1000 hours, both recorded at LaGuardia Airport. Data were obtained from the `air quality` data set.

```
> plot(Wind, Temp)
```



7.6 Scatter plot matrices

A scatter plot matrix is just a matrix of scatter plots where each variable is plotted against all other variables. This graphic is therefore very useful for a preliminary look at multivariate data. In R, we obtain this graphic using the `pairs()` command as illustrated in Exhibit 7.10.

Only four of the variables within the `air quality` data have been used in this example because the variables for the month and day take discrete values and therefore do not suit scatter plots — take a look for yourself if you must but it's probably better to think about why this is the case before you look at the graphs. To select the four variables of interest, I have created a *data.frame* with the variables I want included,; this `data.frame()` command is then nested within the `pairs()` command.

Note that the names of the variables appear in the spaces on the leading diagonal and that the graphs on either side of the diagonal plot the same data but with the axes reversed. Sometimes the human eye will pick up a relationship when the data are presented one way better than the other way.

7.7 Graphs for discrete valued variables

R does not contain many graphs for discrete valued variables.

7.7.1 Bar charts

If a variable is considered by R to be a *factor*, then the default action of the `plot()` command is to construct a bar chart. This is illustrated using a data set which is part of the default installation of R called `state.region` examined using

```
> str(state.region)

 Factor w/ 4 levels "Northeast","South",..: 2 4 4 2 4 4 1 2 2 2 ...

> levels(state.region)

[1] "Northeast"     "South"         "North Central"
[4] "West"
```

Exhibit 7.10 A scatter plot matrix of the numeric variables within the `airquality` data set.

```
> pairs(data.frame(Ozone, Solar.R, Wind, Temp))
```

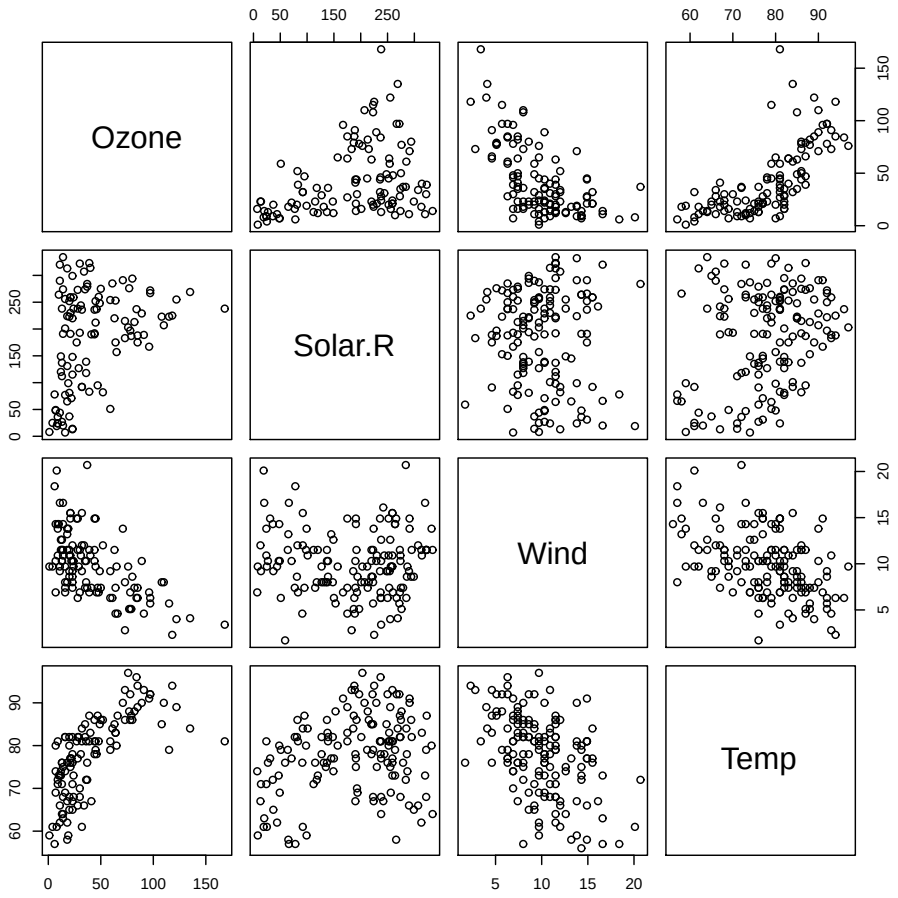
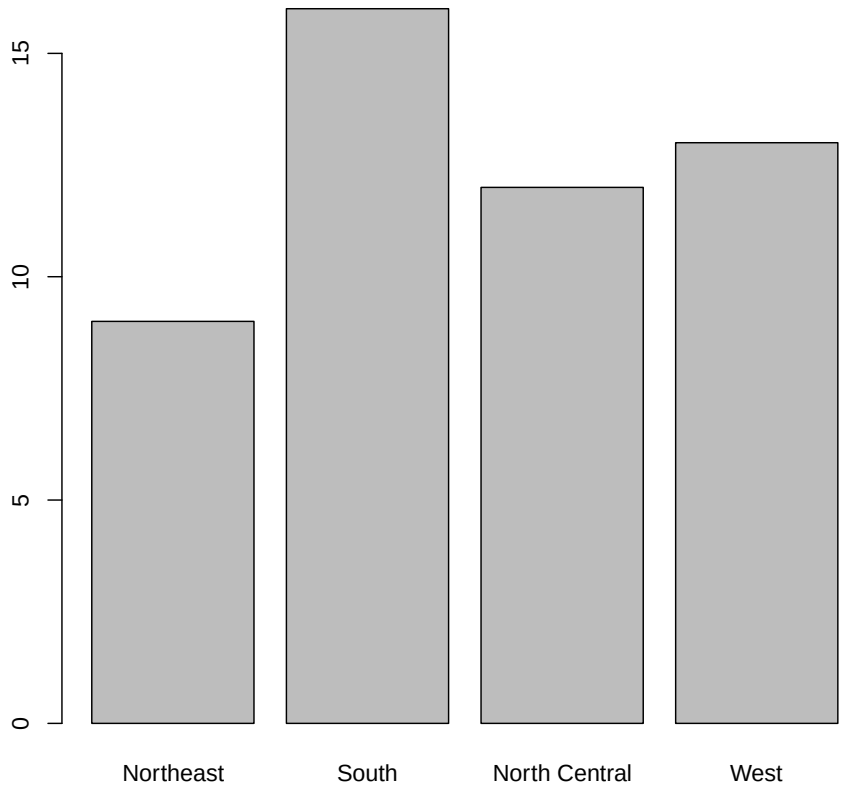


Exhibit 7.11 A bar chart showing which of the regions each of the fifty U.S. states belongs

```
> plot(state.region)
```



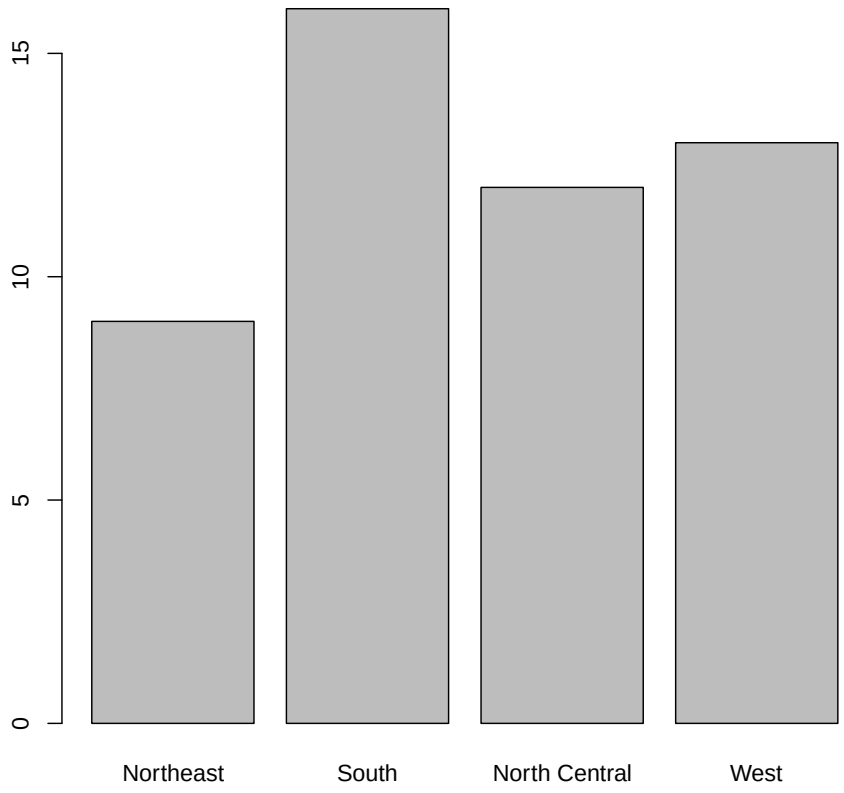
Given this variable is a *factor* with four levels it is well suited to presentation in a bar chart, as seen in Exhibit 7.11.

This figure was created from the raw data, that is the regions for each of the fifty states in the U.S. If we had summary data with counts for each of the categories, we would need to use the `barplot()` command. The `summary()` command shows us how many states fall into each category in this instance.

```
> summary(state.region)

    Northeast         South North Central          West
```

Exhibit 7.12 A bar chart showing which of the regions each of the fifty U.S. states belongs

```
> barplot(summary(state.region))
```



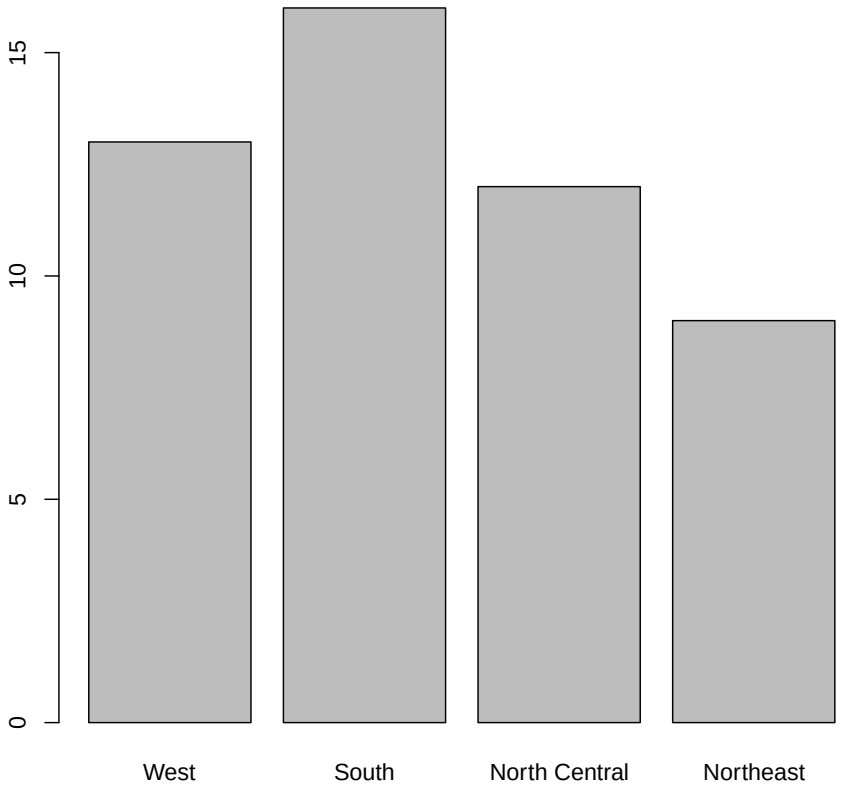

```
    9    16    12    13
```

These values are then plotted in Exhibit 7.12.

While this data set is rather trivial, it is useful for demonstrating one more feature. Note that both in the output above and in Exhibit 7.11, the Western region is the last category. To reorder the regions in the bar chart is actually fairly straight forward. All we need to do is make a small addition to the existing commands.

Exhibit 7.13 An improved bar chart showing which of the regions each of the fifty U.S. states belongs

```
> barplot(summary(state.region)[c(4,2,3,1)])
```

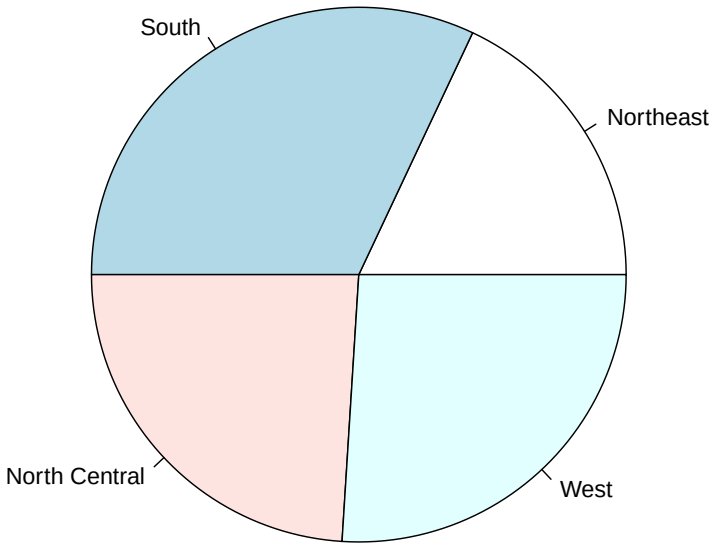



```
> summary(state.region)[c(4,2,3,1)]

        West         South North Central     Northeast
          13            16            12             9
```

and re-create the bar chart accordingly (see Exhibit 7.13).

Exhibit 7.14 A pie chart showing which of the regions each of the fifty U.S. states belongs

```
> pie(summary(state.region))
```



7.7.2 Pie charts

OK, if you must do so, make a pie chart using the `pie()` command. Even the help for this command says they are a bad representation for data, stating "Pie charts are a very bad way of displaying information. The eye is good at judging linear measures and bad at judging relative areas. A bar chart or dot chart is a preferable way of displaying this type of data."

The pie chart for the `state.region` data is given in Exhibit 7.14.

7.8 Closing

If you are carrying on working with R you might wish to remove direct access to the data sets we used in this chapter by issuing the following commands

```
> detach(airquality)
```

Chapter 8

LURN... To Examine Data Numerically

In this chapter we see how to obtain and present simple numerical summaries suitable for describing data. The same data set used in Chapter 7 on creating graphs is used again.

8.1 Obtaining basic numerical summaries of data

The `summary()` command is a very useful tool. It behaves differently for R objects depending on the nature of the object it is working on. The air quality data we used for Chapter 7 for demonstrating basic graphing techniques is a *data.frame* object as evidenced by asking R what class the object is using the `class()` command

```
> class(airquality)

[1] "data.frame"
```

Applying the `summary()` command to the `air quality` data gives:

```
> summary(airquality)

     Ozone           Solar.R         Wind
 Min.   :  1.0   Min.   :  7   Min.   : 1.70
 1st Qu.: 18.0   1st Qu.:116   1st Qu.: 7.40
```

```
 Median : 31.5   Median :205   Median : 9.70
 Mean   : 42.1   Mean   :186   Mean   : 9.96
 3rd Qu.: 63.2   3rd Qu.:259   3rd Qu.:11.50
 Max.   :168.0   Max.   :334   Max.   :20.70
 NA's   :37      NA's   :7
      Temp          Month            Day
 Min.   :56.0   Min.   :5.00   Min.   : 1.0
 1st Qu.:72.0   1st Qu.:6.00   1st Qu.: 8.0
 Median :79.0   Median :7.00   Median :16.0
 Mean   :77.9   Mean   :6.99   Mean   :15.8
 3rd Qu.:85.0   3rd Qu.:8.00   3rd Qu.:23.0
 Max.   :97.0   Max.   :9.00   Max.   :31.0
```

The `summary()` command finds the common numeric summary statistics for the continuous-valued variables and does a particulary useless job on the variables for month and day. Well it's not really R's fault. The month didn't need to be stored numerically, but in any case we got what we asked for! It's not R's fault we asked to be told what the minimum day number was!

Notice that the `summary()` command extracted the minimum, maximum, median and mean for the variables in the *data.frame*. These are all available using corresponding `min()`, `max()`, `median()`, and `mean()` commands separately but they work in different ways on data frames. For example we might have thought to use the `mean()` command on a *data.frame*, but it is better practice to use the more specific `colMeans()` command instead.

```
> colMeans(airquality[,1:4])

  Ozone Solar.R    Wind    Temp
     NA      NA   9.958  77.882

> colMeans(airquality[,1:4], na.rm=TRUE)

  Ozone Solar.R    Wind    Temp
 42.129 185.932   9.958  77.882
```

```
> mean(airquality[,1], na.rm=TRUE)

[1] 42.13
```

Note that I've told R which columns of the `air quality` *data.frame* to work with using the appropriate subscripting code. I've also needed to tell R to ignore the missing data for the two variables `Ozone` and `Solar.R` using the `na.rm=TRUE`. The output for the `summary()` command did tell us how many values for each variable were missing. Recall that missing values are represented by a "`NA`".

The `min()` and `max()` commands all work on the data by assuming we want the statistic for the whole list of numbers over all columns. For example

```
> #median(airquality[,1:4], na.rm=TRUE)
> min(airquality[,1:4], na.rm=TRUE)

[1] 1

> max(airquality[,1:4], na.rm=TRUE)

[1] 334
```

It used to be possible to ask R to use the `median()`, in a command like this, but as of version 2-14-0, an error message was returned instead of an answer.

We will need to see how to get the relevant medians, minima, and maxima for the columns separately later in this chapter, but for now let's see how the relevant summary measures can be obtained for a single column of the `air quality` data. The `attach()` command allows us direct access to the variables by name.

```
> attach(airquality)
```

```
> min(Temp)

[1] 56

> max(Wind)
```

```
[1] 20.7

> median(Solar.R)

[1] NA

> median(Solar.R, na.rm=TRUE)

[1] 205
```

Notice that a consequence of missing values in our data is that some functions will return the `NA` value. If we want the quantity returned to be estimated using the available data we simply add the argument `na.rm=TRUE` which removes the values denoted with the `NA`.

The first and third quartiles are not so easily extracted however. We can get the first and third quartiles from the five number summary by extracting elements of the output returned by the `fivenum()` command. For example

```
> fivenum(Ozone)

[1]   1.0  18.0  31.5  63.5 168.0
```

There are some minor differences between the methods used by the `summary()` and `fivenum()` commands for the first and third quartiles. See the relevant help page using `?fivenum` for an explanation.

Knowing how these commands work on simple sets of numbers is very useful for the more elegant presentations in the following sections.

8.2 Obtaining slightly more elegant summaries

When preparing reports it is unlikely that we would want a single statistic all that often. Rather, we are usually interested in exposing patterns or differences within the data. We do this graphically but we may need to extract the exact quantities plotted in the graph as well.

The `tapply()` and `aggregate()` commands are very useful for creating tables of results, sometimes referred to as *pivot tables*. The `tapply()` command has three arguments; the

main variable of interest, the grouping variable(s), and the function we want performed on each group within the data. In contrast, the `aggregate()` command has a slightly different way of relating the response variable to one or more grouping variables. It is of course, easier to show how these commands work in action so we'll use the air quality data again and present various alternatives and the slightly different resulting output.

Let's say we want to know the `mean()` Wind speed over the five months data were collected. We could use

```
> tapply(Wind, Month, mean)

     5      6      7      8      9
11.623 10.267  8.942  8.794 10.180
```

or

```
> aggregate(Wind~Month, data=airquality, mean)

  Month   Wind
1     5 11.623
2     6 10.267
3     7  8.942
4     8  8.794
5     9 10.180
```

These commands do the job very nicely; choose the one that delivers the output in the way you want it. We could ask for other quantities such as the minimum, maximum, or standard deviation within each month by referring to the relevant R command in the third argument. Note that the `aggregate()` command allows specification of the *data.frame* in which to find the variables in the *formula*; this approach is commonplace when we fit models for our data and avoids the need to make use of the `attach()` and `detach()` commands.

If we had multiple response variables of interest, we would need to repeat the command for each one. If we had more than one way of asking for the grouping though, the `tapply()` command is very useful indeed as it presents a two-way pivot table. The air quality data doesn't have a second factor we can use for this illustration, so I've created a second variable

which asks if the day of the month is in the second half of the month. In this case, the `tapply()` and `aggregate()` commands generate quite different outcomes.

```
> Day15 = Day > 15
> tapply(Wind, list(Month, Day15), mean)

   FALSE   TRUE
5 11.313 11.912
6 10.787  9.747
7  9.360  8.550
8  8.907  8.688
9  9.547 10.813

> aggregate(Wind ~ Month + Day15, data=airquality, mean)

   Month Day15   Wind
1      5 FALSE 11.313
2      6 FALSE 10.787
3      7 FALSE  9.360
4      8 FALSE  8.907
5      9 FALSE  9.547
6      5  TRUE 11.912
7      6  TRUE  9.747
8      7  TRUE  8.550
9      8  TRUE  8.688
10     9  TRUE 10.813
```

If we wanted more than one statistic of interest, we would need to repeat the command and change the function being referenced in the third argument. We can use the `summary()` command here though as it gives us more quantities.

```
> tapply(Wind, Month, summary)

$`5`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
```

```
    5.7     8.9    11.5    11.6    14.1    20.1

$`6`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    1.7     8.0     9.7    10.3    11.5    20.7

$`7`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   4.10    6.90    8.60    8.94   10.90   14.90

$`8`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   2.30    6.60    8.60    8.79   11.20   15.50

$`9`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   2.80    7.55   10.30   10.18   12.32   16.60
```

This printout isn't particularly attractive, but it is quick and informative. If presentation was more important than getting results efficiently, we would look to improve the appearance — this means more difficult coding though.

There is one more feature of the `tapply()` command that must be pointed out. The simple commands shown thus far use the default settings of the relevant functions. In particular, the `mean()` command does not ignore missing values by default.

```
> tapply(Ozone, Month, mean)

 5  6  7  8  9
NA NA NA NA NA
```

We now know that there are missing `Ozone` observations in each month. We can add a fourth argument to the command which will be passed on to the function mentioned in the third argument.

```
> tapply(Ozone, Month, mean, na.rm=TRUE)

    5     6     7     8     9
23.62 29.44 59.12 59.96 31.45
```

gives us the means of the observed data.

8.3 Getting things printed how we want them

We will often need to watch the way R chooses to print out results. When a direct reference to data is sought, the order of the data is maintained. In our example using the `tapply()` command above, the months were numeric values. The order was probably the most logical one as it was in chronological order.

If we had a long list of results and it was the extremes we wanted to focus attention towards, the `sort()` and `rev()` commands would be most useful. For example, say we want to order the monthly average wind speeds found above in ascending order, then we would use

```
> sort(tapply(Wind, Month, mean))

     8      7      9      6      5
 8.794  8.942 10.180 10.267 11.623
```

So we now know that August has the smallest average wind speed and that May is the windiest month.

To get the data listed in reverse chronological order we would use:

```
> rev(tapply(Wind, Month, mean))

     9      8      7      6      5
10.180  8.794  8.942 10.267 11.623
```

and we can combine both commands to get the months ordered from windiest to least windy using:

```
> rev(sort(tapply(Wind, Month, mean)))

     5      6      9      7      8
11.623 10.267 10.180  8.942  8.794
```

8.4 Correlation structure within a data set

The summaries given thus far in this chapter all refer to variables as single objects. Often we have multivariate data and want to know more about it. Looking for inter-relationships among variables can be achieved using correlation coefficients; this gives us a numerical equivalent to the scatter plot matrices shown in Section 7.6.

In fact, the commands used here resemble those used to generate the scatter plot matrix. I like to do this to save effort. My ability to copy and paste is better than my typing!

```
> cor(data.frame(Ozone, Solar.R, Wind, Temp))

        Ozone Solar.R   Wind   Temp
Ozone       1      NA     NA     NA
Solar.R    NA       1     NA     NA
Wind       NA      NA  1.000 -0.458
Temp       NA      NA -0.458  1.000
```

Having said this, it is now obvious that the missing values in some variables is causing grief. An additional argument is required.

```
> cor(data.frame(Ozone, Solar.R, Wind, Temp),
   use = "pairwise.complete.obs")

          Ozone  Solar.R     Wind    Temp
Ozone    1.0000  0.34834 -0.60155  0.6984
Solar.R  0.3483  1.00000 -0.05679  0.2758
Wind    -0.6015 -0.05679  1.00000 -0.4580
Temp     0.6984  0.27584 -0.45799  1.0000
```

Often, R will allow arguments to be abbreviated. Unfortunately, this isn't the case for this additional argument. It is the case for choosing the preferred correlation measure to be found though. The default is to calculate Pearson's correlation coefficient. My personal preference is to use both this measure and Spearman's rank correlation coefficient as it does not require the relationship to be linear, nor does it require the samples to be normally distributed. As well as finding Spearman's measure, I've decided that printing correlations to five decimal places is unnecessary so have restricted the output using the `round()` command.

```
> round(cor(data.frame(Ozone, Solar.R, Wind, Temp),
   use = "pairwise.complete.obs", method = "s"), 3)

         Ozone Solar.R   Wind   Temp
Ozone    1.000   0.348 -0.590  0.774
Solar.R  0.348   1.000 -0.001  0.207
Wind    -0.590  -0.001  1.000 -0.447
Temp     0.774   0.207 -0.447  1.000
```

8.5 Use of dplyr for data summarisation

In Section 5.4, we saw how the *dplyr* package offers an alternative to base R functionality for manipulating data. We now see that this package offers commands for summarising data. In particular, the commands: `group_by()`, `summarise()`, and `count()`.

You'll need to install the package (see Section 14.1 for instructions) before running the function

```
> library(dplyr)
```

to gain access to the commands just listed. We will also make use of the *dplyr* data structure called a *tbl_df* instead of the common *data.frame* using:

```
> airquality2=tbl_df(airquality)
```

The `summarise()` and `group_by()` commands can be used instead of the `tapply()` command introduced earlier in this chapter. To get the overall average wind speed from the `airquality` data, we could use:

```
> airquality %>% summarise(mean(Wind, na.rm = TRUE))

  mean(Wind, na.rm = TRUE)
1                    9.958
```

which is more long-winded than just using the `mean()` command alone. The value of `summarise()` becomes more evident when we seek the means for groups within our data, such as:

```
> airquality%>% group_by(Month) %>% summarise(mean(Wind, na.rm = TRUE))

# A tibble: 5 × 2
  Month `mean(Wind, na.rm = TRUE)`
  <int>                      <dbl>
1     5                     11.623
2     6                     10.267
3     7                      8.942
4     8                      8.794
5     9                     10.180
```

We could use other functions seen in this chapter for alternative summary statistics within these functions. A common task is to check on how much data we have for each group of interest. We could use `summarise()` and `length()` to count up the data, but the *dplyr* package gives us the `count()` function to make this even more efficient.

```
> airquality%>% count(Month)

# A tibble: 5 × 2
  Month     n
  <int> <int>
1     5    31
```

```
2     6    30
3     7    31
4     8    31
5     9    30
```

As an illustration of the real power of the *pipe operator*, say we wanted to show the hottest months by their average temperature within the air quality data. One way to do this using commands from the *dplyr* package is to:

```
> Grouped = group_by(airquality, Month)
> Summarised = summarise(Grouped, AveTemp=mean(Temp, na.rm = TRUE))
> arrange(Summarised, desc(AveTemp))

# A tibble: 5 × 2
  Month AveTemp
  <int>   <dbl>
1     8   83.97
2     7   83.90
3     6   79.10
4     9   76.90
5     5   65.55
```

Which has stored data at each step. Note in particular that a new variable was created by name in the `summarise()` command; this helps in the subsequent `arrange()` step. We might want to store the results at each step, but if we didn't want to, then we could use piping like this:

```
> airquality2 %>%
   group_by(Month) %>%
   summarise(AveTemp = mean(Temp, na.rm = TRUE))  %>%
   arrange(desc(AveTemp))

# A tibble: 5 × 2
  Month AveTemp
  <int>   <dbl>
```

```
1     8   83.97
2     7   83.90
3     6   79.10
4     9   76.90
5     5   65.55
```

Note that commands that might be nested on one line in the standard way of combining commands can be split out onto multiple lines to make it even easier to see what has been managed by each command. This leaves room for comments to be added at the end of lines to help anyone else read the code being written.

8.6 Closing

If you are carrying on working with R you might wish to remove direct access to the data sets we used in this chapter by issuing the following commands

```
> detach(airquality)
```

Chapter 9

LURN... To Save Results for Later Inspection

In this chapter we see how we can save our results, both graphical and numerical, in files for later use. I find it useful to be able to generate tables and figures for documents as individual files because I use the LaTeX system for typesetting documents. I like the advantage this approach has because it means I do not need to sift through larger documents to delete or repair old tables and figures if data is updated or errors in my working are found. In fact the document you are reading is prepared in exactly this way. All tables and figures generated by R were stored as separate files for later use.

This chapter is not about saving data. If you want to know how to save your data for sharing or storing then you need Chapter 6 on exporting data.

This chapter also assumes you already know how to create a graph.

9.1 Copying and pasting graphs

OK you can do it if you must. If you are using a word processor then this is the simplest way to get a graph into your document. You need to go through all the steps again if the figure turns out to be substandard in some way so watch what you're doing!

R will open a new window if required, or a pane in the existing graph window for each graph created, regardless of the mode you choose to run R. The menus offer the choice of copying the current window to the clipboard and indicate the hot-key for doing this. You can then place the graph in your word processed document where you want it by pasting

it as required.

9.2 Saving a graph using the menus

R does give the user a range of options for saving graphs within the pull down menus. You should choose the file type that best suits your needs. The problem with this way of saving graphs, and the copy and paste approach, is that it is labour intensive. When it comes time to develop a program that will generate lots of graphs, you will want to have them saved automatically.

9.3 Saving a graph using commands within your R program

There are several ways to save a graph created within your R program. I offer one suggestion that I think provides some degree of flexibility, especially if you need to change the file type for some reason. First of all, recall that R will open a new graph window if necessary or add a new graph to the existing graph window device. I make sure that I close all graph windows immediately after creating and saving them in my R scripts. To close a graph window we use the `dev.off()` command. This keeps things tidy and means we don't have lots of graph windows to worry about. Opening graphs is a slightly different story.

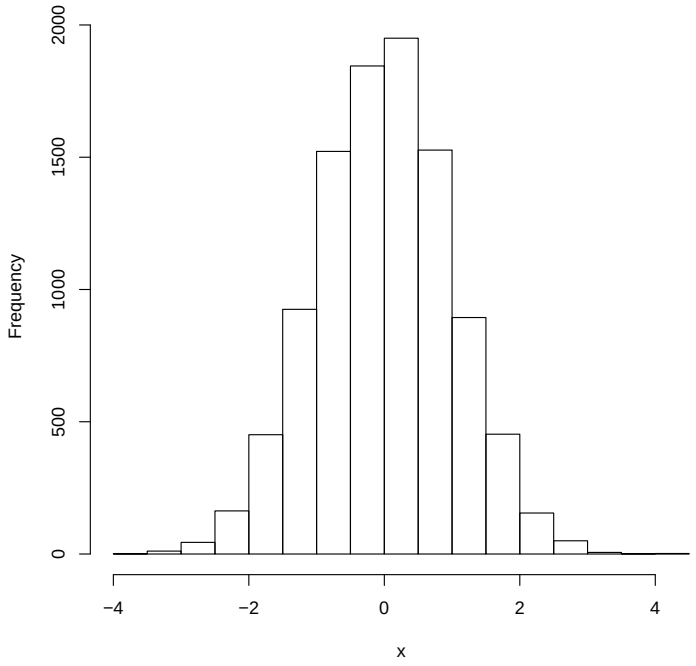
For a basic graph such as a histogram where the default size of the window suits my purposes, I use the `x11()` command to open a new graph window; we'll see how to change this soon. I then create my graph as required and save it using the `savePlot()` command. Providing an example now seems appropriate.

The following code will result in a graph similar to that given in Exhibit 9.1 — similar because random data are used and you will get something slightly different.

It should be fairly obvious what each command does so the only one that really needs further explanation is the `savePlot()` command. The first argument is the intended filename for our graph. The filename may need some further explanation. It is in several parts — `.\\figures\\HistRandValsStandNorm.eps` has a path, a filename and the extension.

The `.\\figures\\` is the path that shows R to save the file in a subfolder off the current working directory. I've called the file `HistRandValsStandNorm.eps` because I want to use

Exhibit 9.1 Histogram of 10,000 random numbers drawn from a standard normal distribution.



LaTeX to compile these notes into a presentable document. Note that I have explicitly stated the extension in the complete filename to be the same as the second argument for the entire command, "eps" in this instance. You can change this to suit other file types that you might need using `find/replace` techniques in your script files. Other file types available include "jpg", "bmp", and "pdf". See the help on the `savePlot()` command for a complete listing.

The `x11()` command can be replaced by a number of others which are all aliased with one another. The example above doesn't supply any arguments to this command, but width and height quantities can be supplied. The default size is 7×7 inches but apparently this doesn't always behave properly under the Windows operating system. You might be advised to re-run the code above with the first command replaced by `x11(7,5)` or similar, changing the second number to alter the aspect ratio of the resulting figure window. Once you get the size you want for certain types of graphs you're bound to look back to your old code and replicate the same size graph windows every time. A boxplot is a key example

where the default window size is often inappropriate.

9.4 Saving output in a text file

One simple way to redirect the output from the main R window is to use the `sink()` command. In this case, you will type in or execute commands as per normal but not see any of the output as this is appended to a separate text file. Try the following code on for size.

```
> x=1:20
> x

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

```
> sink(".\\other\\JustPrintX.txt")
> x
> ## sink()
```

This creates a text file called `JustPrintX.txt` in a subfolder called `other`, within the current working directory (which I created earlier). The file is worth looking at because it should show exactly what would have been printed in the R window if the sink were not active. Note that the second use of `sink()` stops the sink from being active.

Just typing the name of an object implicitly calls the relevant `print()` function for that object type. This does not work when a sink is active however, so we must use the `print()` command explicitly to get the output into our sink file. For example

```
> sink(".\\other\\JustPrintX2.txt")
> print(x)

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

> ## sink()
```

Check this file out and note the contents are what we might expect. It's cleaner to use the `cat()` command in place of the `print()` command here though. For example

```
> sink(".\\other\\JustPrintX3.txt")
> cat(x)

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

> ## sink()
```

An even quicker method useful for a single result is to use the `cat()` command's full potential:

```
> cat(x, file=".\\other\\JustCatX.txt")
```

9.5 Saving the entire R Console

Windows users of R who use the RConsole have an option to save the entire console using the menus. At present, however, this is not an option that can be invoked via the command line.

It is possible to save the history of commands using the `savehistory()` command, and of course your data and command history can be saved at the end of a session.

You'll soon see that you do not need to save a copy of the R console as it takes so little time to re-run a set of commands and re-generate the output desired.

Chapter 10

LURN... To Become More Efficient in your Workplace

This chapter is aimed at the efficient use of R. To make use of the functionality described here you must have read the material on storing your results in Chapter 9.

10.1 Managing R when you have multiple projects on the go

One of the greatest frustrations for novice users when using software like R is the management of the various datasets and projects they have. The temptation to give objects short names that mean something now and are utterly mystifying in six months time (perhaps when you go to do a clean out) is a common trap. If it is something temporary, give it a useless uninformative name; if it's actually going to be important, give it a useful informative name when you first create it.

It's also common to just start working in the main working directory and just keep on adding various data sets and analysis objects until such time as the working directory has hundreds of objects and becomes unwieldy. If you plan to use R for one or two analyses and then move on this might work, but this is not good practice.

My recommendation is to use different folders for the different projects you wish to work on. I make folders and subfolders for my R work in the same way I would for my word processed documents. By changing the working directory, we end up with a command

history and stored workspace specific to the one project. It also means your graphs are saved here and are then easier to find. It might even prove useful to keep the R work and the associated word processed documents in the same folder.

If you change working directory, either using the menus or the `setwd()` command and save the workspace when quitting R, you will see that files called `.RData` and `.Rhistory` have been created. When you open R and change to this working directory, you will have access to the relevant data and command history.

The following is useful as a Windows user of R. As an additional step towards efficiency, I copy the shortcut used to start R into each working directory. It will need slight editing to make sure the shortcut doesn't quite take the default action by deleting the contents of the "Start in" dialogue box for the shortcut's properties. In this way, I can use Windows Explorer to move to the folder for a certain project and click the shortcut for R to get started on a project at the point where I left off.

10.2 Batch processing commands

When your R program gets to have a lot of commands that must be processed in order, it's a good idea to use the R script window or a text editor to create a file with the commands in it. Keeping your commands in a text file means you can share them with colleagues very easily.

If you choose to use the R script window, then the commands are typed up and are not executed until you specifically request for them to be executed. This is achieved using the pull down menus.

If you have saved the commands in a text file, and this text file is placed in your working directory, you will be able to execute the commands using the `source()` command.

As a simple experiment, type up some commands in a new script window. Execute them using the pull down menus. Now save the commands in a text file, and go to the R console and type `source("myfile.R")` — note that I used the extension "R" in the filename here. This is common practice among R users.

If you're not using the main Windows version of R, you will be entering commands in a terminal window. In this case, it's probably easiest to use your favourite text editor software to type up and save your R commands. It doesn't matter which version of R is being used, the `source()` command will behave consistently.

10.3 Running an R script without opening R under Windows

Running your R programs on another computer is a great idea because it means you can get on with another project while a long program is running. You'll probably need some support from your local network administrator to achieve that, but in the meantime you can see how it's done by running some jobs in the background on your own machine.

In the previous section I discussed the ability to use the `source()` command to import a text file of commands into an R session.

To continue with the rest of this section we need to know where to find R on our local machine. On my computer this was

`C:\Program Files\R\R-3.4.0`

because I installed this version of R using the default settings. This folder is often called R_HOME. If you haven't installed R using the default settings, you will need to find out where your R_HOME is and remember it in the instructions that follow.

What we actually want is to find a file called `rterm.exe` which is in the `bin` subfolder. The full filename including its path is going to be

`C:\Program Files\R\R-3.4.0\bin\Rterm.exe`

Note that this is the file that will run R in a terminal window, not `rgui.exe` which is for the standard R console. Note also that as from version 2.12.0, R running under Windows had an extra element in the folder structure which depends on the machine you are using. Many Windows users are now moving to 64 bit operating systems and software, while others still use the older 32 bit software. Check your path carefully.

We now need to create a batch file. This is just a text file with the extension `bat` instead of the normal `txt`. If you open a text file it is readable in a text editor like notepad; a batch file is a set of commands that are executed when you run the file. On Windows, this is as simple as pressing `<Enter>` when the filename is highlighted in the Explorer window, or double clicking as if you were opening the file.

So let's start by creating a text file in the chosen working directory. Do this using the menus in Windows Explorer. Open the file so that we are able to add the following text.

```
"C:\Program Files\R\R-3.4.0\bin\Rterm.exe" < mycommands.R > myoutput.txt
```

This single command has three parts; the first part in quotes is the full path to the program we want to run, and the quotes are essential. The second is the set of commands we want executed in that program, and the last is the file where we want the results to be saved. The symbols between the three parts are essential and the direction is important. This process is often called "piping". The contents of the commands file are piped into R., and the results are piped out to another file.

Save the text file and change its filename so that the extension is `bat`. Run the file. If the full path to the `rterm.exe` was correct for your machine, and the file of commands `mycommands.R` exists in the current directory, you should see that a new text file called `myoutput.txt` has been created. You can open this file to see what happened.

A word of warning

This approach works well for programs that have no bugs. Finding where a bug exists is not impossible but can be frustrating if there are too many bugs to fix. The program will stop if a line of code contains an error of any kind. Unfortunately, experimentation is probably the only way to know if things will work properly.

You could re-run your programs frequently in order to ensure that you are getting what you want. If it proves slow and frustrating to do this, it's probably because you have a large data set or some computationally intensive elements in your program. If this is the case, you could use a subset of your data while you test your programs.

10.4 Using file associations in Windows

When I want to edit an R script file, I want it to open in my text editor of choice. Under Windows operating systems, this can be achieved by ensuring a file association is created. The simplest way to do this is to select the default program to be use when you want to open a file with the extension "`.R`".

If you do this, then the next logical thing to want to do is to have that R script processed using R itself. You could use the approach given earlier in this chapter but an even better way exists that uses file associations again.

We need to make a batch file again, just like in the example on piping. This batch file will have just one line in it and it needs to be put in a folder that is on the system path.

The single line is

```
"C:\Program Files\R\R-3.4.0\bin\R.exe" CMD BATCH --vanilla --quiet %1
```

Now look for an R script and select the default program to open it again. This time, browse for the batch file you just created. Now every time you click on an R script, it will be processed by R. All output including commands will be put into a file of the same name with an extension that has the three extra letters "`out`" on the end. Use the process for file associations again to get this file opened automatically; use the viewer of your choice, whether that be a text editor or a browser.

The use of the two arguments `vanilla` and `quiet` ensure you have a fresh workspace and that the normal announcements of version number etc are not included in the output file. You might want to include these welcome messages so remove the `quiet` argument if you want. Another argument that might be useful is the `save` argument. Add that in if you want to be able to open R again in interactive mode later and be able ot continue from where you left off.

You might want to go back now and re-assign the default program for an R script to be the text editor, and use the "open with" menu item to have it processed using your batch process.

10.5 Don't re-invent any wheels

Sometimes it is obvious that you will need to write out some quite detailed and complex sets of commands, but when you can't find the functionality within the base distribution of R, you might be tempted to write the code yourself. Before you do that though, I suggest you stop and think if the task is something that someone else is likely to have done before.

If the answer is yes, then you should look at the vast list of contributed packages available via the Comprehensive R Archive Network (CRAN) accessible under

http://CRAN.R-project.org

be warned though that this is not necessarily a successful way of finding the right code. I prefer to do an internet search and hope the supporting documentation for any contributed packages has the same key words as you are thinking about. The data for many textbooks is also available as contributed packages so don't be tempted to enter data unnecessarily.

Chapter 11

LURN... To Do Basic Inference

This chapter is all about finding confidence intervals and performing hypothesis tests. While many textbooks separate these activities there is little point doing so when it comes time to using R as both are done at the same time in the majority of cases.

11.1 Confidence intervals and hypothesis tests for the mean of one population

The *datasets* package contains a set of annual precipitation values (in inches) for 70 U.S. cities. If we could assume this set to be a random sample (and we will even if it is not true), then we can find a confidence interval for the mean annual precipitation for U.S. cities on the whole.

You could use a manual approach to find the 95% confidence interval for the population mean (μ) yourself using the `mean()`, `sd()`, `length()`, and `qt()` commands as follows:

```
> mean(precip)

[1] 34.89

> sd(precip)

[1] 13.71

> length(precip)
```

```
[1] 70

> qt(0.975, length(precip)-1)

[1] 1.995
```

Note the use of the `qt()` command here. It has two arguments; one for the level of confidence (95% confidence relates to an upper tail area of 0.025) and the degrees of freedom (based on the sample size). The 95% confidence interval is therefore found using

```
> mean(precip) - qt(0.975, length(precip)-1) * sd(precip) / sqrt(length(precip))

[1] 31.62

> mean(precip) + qt(0.975, length(precip)-1) * sd(precip) / sqrt(length(precip))

[1] 38.15
```

or, with a nicer (quicker) command:

```
> mean(precip) + c(-1,1) * qt(0.975, length(precip)-1) * sd(precip) / sqrt(length(precip

[1] 31.62 38.15
```

These calculations include all the commands needed for a hypothesis test, except the need to find the specific p-value of a hypothesis test. Let's test the notion that the annual precipitation is actually greater than 30 inches per annum. This is a one-sided hypothesis test where the null hypothesis is that $\mu = 30$.

```
> TestStat= (mean(precip)-30)/(sd(precip) / sqrt(length(precip)))
> pt(TestStat, length(precip)-1, lower.tail=FALSE)

[1] 0.001976
```

Note that the `pt()` command required us to think about the degrees of freedom and that the upper tail area was important to us. We now see that the null hypothesis is rejected as the p-value is very small.

These calculations are fine, but there is a quicker way! The `t.test()` command does it all for us — both confidence interval and hypothesis test.

```
> t.test(precip)


One Sample t-test

data:  precip
t = 21, df = 69, p-value <2e-16
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 31.62 38.15
sample estimates:
mean of x
    34.89
```

This gives a confidence interval with the default level of confidence (95%), while

```
> t.test(precip, mu=30, alternative="greater")


One Sample t-test

data:  precip
t = 3, df = 69, p-value = 0.002
alternative hypothesis: true mean is greater than 30
95 percent confidence interval:
 32.15   Inf
sample estimates:
mean of x
    34.89
```

gives us the results for a one-sided hypothesis test and the corresponding confidence interval.

The `t.test()` command is used for many t-tests, notably for comparing two population means, whether the data are paired or not.

11.2 Confidence intervals and hypothesis tests for the difference of two population means

The first question you must ask yourself when working with two sets of values you wish to compare is if they are two independent samples or two measurements taken on one sample. If the latter is the case, then we have paired samples and will need to add an extra argument to our command to allow for the link. In both cases, the difference of the two population means is denoted $\mu_1 - \mu_2$, although as we will see, the linked sample case is really a reduction of the paired values to a one-sample t-test.

First, we see how to do a basic two sample t-test using the ... data.

11.2.1 Two paired samples

The `sleep` data that is in the *datasets* package has two recordings for each of ten individuals. In an experimental design sense, we would think of the individuals as a blocking factor. In fact we will see how this is done in Section **??**.

The data set lists the twenty observations in a single column. This is a little unusual for paired data that will be analyzed using the `t.test()` command. Caution must be expressed because the variable that shows how the observations are paired is not stated. If the observations are not ordered the same for each group, then the paring will be done incorrectly. If for any reason you cannot be sure of the ordering of the data then the use of the `aov()` command as discussed in Section **??** is strongly advised.

Using the `attach()` command to be able to refer to the variables directly speeds up this process.

```
> attach(sleep)
```

We employ the `t.test()` command with the `paired` argument set to `TRUE`. We can use either of two approaches. Either we separate the two groups apart using subscripting

```
> t.test(extra[group=="1"], extra[group=="2"], paired=TRUE)


Paired t-test

data:  extra[group == "1"] and extra[group == "2"]
t = -4.1, df = 9, p-value = 0.003
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -2.4599 -0.7001
sample estimates:
mean of the differences
                  -1.58
```

which is the way we would normally work when we have the two measurements stored using two distinct variables; or, use the formula approach:

```
> t.test(extra~group, paired=TRUE)


Paired t-test

data:  extra by group
t = -4.1, df = 9, p-value = 0.003
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -2.4599 -0.7001
sample estimates:
mean of the differences
                  -1.58
```

You could of course alter the analysis by adding arguments to these commands to suit your needs. Experiment a little before detaching the `sleep` data using the `detach()` command:

```
> detach(sleep)
```

11.3 Confidence intervals and hypothesis tests for a proportion from one population

11.4 Confidence intervals and hypothesis tests for the difference of two population proportions

11.5 Hypothesis tests and confidence intervals for Correlation coefficients

In Section 8.4 we examined the correlation of the variables in the `airquality` data set using the `cor()` command. If we require a formal hypothesis test for the significance of the correlation coefficient we will need to use the `cor.test()` command. The difference between the two commands is that we could offer the entire data set as the object to work on with the `cor()` command while we need to specify two vectors for the `cor.test()` command.

There are problems with cherry-picking one pair of variables from a set to perform a post hoc test like this. We should probably consider finding the significance of all correlations of interest and adjusting the way we think about the likelihood of getting the set of hypothesis tests we observe as a combination using a *Bonn Ferroni adjustment* in our analysis; that is beyond the scope of this exposition so for the moment just assume we are (and were before data was collected) interested in only the correlation that exists between a single pair of variables. We use the `Ozone` and `Wind` variables from the `airquality` data.

To test the notion that there is zero correlation between the two variables we would probably consider using Pearson's correlation coefficient which measures the strength of the linear relationship between a pair of variables, but use of Spearman's rank correlation coefficient will measure the strength of any monotonic relationship that might exist. I prefer to use both at the same time in many circumstances.

```
> attach(airquality)
> cor.test(Ozone, Wind)


Pearson's product-moment correlation

data:  Ozone and Wind
t = -8, df = 110, p-value = 9e-13
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.7064 -0.4709
sample estimates:
    cor
-0.6015

> cor.test(Ozone, Wind, method="spearman")

Warning in cor.test.default(Ozone, Wind, method = "spearman"):  Cannot compute
exact p-value with ties


Spearman's rank correlation rho

data:  Ozone and Wind
S = 410000, p-value = 3e-12
alternative hypothesis: true rho is not equal to 0
sample estimates:
    rho
-0.5902

> detach(airquality)
```

So it seems the two variables are correlated and have a monotonic relationship that implies that as wind increases the amount of ozone decreases and that this relationship is fairly linear. We will look at this relationship again when we look at linear regression in

Chapter 12.

11.6 Testing the independence of two categorical variables

using the chisq.test command. Refer graphical representation to the *vcd* package.

11.7 Testing the normality of a distribution

norm test package here.

We can use the `ks.test()` function for the Kolmogorov-Smirnov test. It is suitable for comparing two samples, or a single sample against a theoretical distribution.

Chapter 12

LURN... To Perform Regression Analyses

This chapter presents the most basic regression models. To really get the most out of R and regression techniques (such as those taught in second or later statistics courses) you will need to look for guidance from a suitable textbook, many of which incorporate use of R as the preferred software tool.

12.1 Data and suitable exploratory graphs

The easiest way to fit regression models is using data that are contained in a *data.frame*. This means we can use the `attach()` command to get at the separate components if we need them. An alternative is to have direct access to each variable independently within our current workspace. For neatness, I prefer to keep data in the *data.frame* format.

We can use the `air quality` data described in Chapter 7. Recall that it is available from your current workspace as it is contained within the *datasets* package. Typing

```
> data(airquality)
```

will make the data explicitly available to you in your current R workspace.

We can look for relationships among the variables within this data set using the techniques described in Chapter 7. Note in particular the scatter plot matrix presented in Exhibit 7.10.

12.2 The simple regression model

To obtain the equation for a straight line relationship of the form

$$y = \alpha + \beta x + \varepsilon \tag{12.1}$$

we use the `lm()` command. The first argument for this command is a *formula* which is the main component of any `lm()` command you issue. It is of the form `y~x` where the y is the response variable and the x is the predictor variable. We will see how to modify the right hand side of the *formula* in subsequent sections. The ε is the error term for our model; we look at that aspect more in Section 12.4 below.

A second common argument is the `data` statement. This means that the model formula can be stated using variable names from within the *data.frame* mentioned in the `data` statement. Models to explain the amount of `Ozone` using `Wind` and subsequently `Temp` are as follows.

```
> Ozone.lm1 = lm(Ozone~Wind, data=airquality)
> Ozone.lm1


Call:
lm(formula = Ozone ~ Wind, data = airquality)

Coefficients:
(Intercept)         Wind
      96.87        -5.55

> summary(Ozone.lm1)


Call:
lm(formula = Ozone ~ Wind, data = airquality)

Residuals:
   Min     1Q Median     3Q    Max
```

```
-51.57 -18.85  -4.87  15.23  90.00

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)    96.87       7.24   13.38  < 2e-16 ***
Wind           -5.55       0.69   -8.04  9.3e-13 ***
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 26.5 on 114 degrees of freedom
  (37 observations deleted due to missingness)
Multiple R-squared:  0.362,Adjusted R-squared:  0.356
F-statistic: 64.6 on 1 and 114 DF,  p-value: 9.27e-13

> anova(Ozone.lm1)

Analysis of Variance Table

Response: Ozone
           Df Sum Sq Mean Sq F value  Pr(>F)
Wind        1  45284   45284    64.6 9.3e-13 ***
Residuals 114  79859     701
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

> Ozone.lm2 = lm(Ozone~Temp, data=airquality)
> summary(Ozone.lm2)


Call:
lm(formula = Ozone ~ Temp, data = airquality)
```

```
Residuals:
   Min     1Q Median     3Q    Max
-40.73 -17.41  -0.59  11.31 118.27

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -146.995     18.287   -8.04  9.4e-13 ***
Temp           2.429      0.233   10.42  < 2e-16 ***
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 23.7 on 114 degrees of freedom
  (37 observations deleted due to missingness)
Multiple R-squared:  0.488,Adjusted R-squared:  0.483
F-statistic:  109 on 1 and 114 DF,  p-value: <2e-16

> anova(Ozone.lm2)

Analysis of Variance Table

Response: Ozone
           Df Sum Sq Mean Sq F value Pr(>F)
Temp        1  61033   61033     109 <2e-16 ***
Residuals 114  64110     562
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

There are very good reasons for creating the new objects in this example. Aside from the use of the `summary()` method and `anova()` command to extract more useful information about the model, we will refer back to the object using other commands in subsequent sections of this chapter. It also means we can investigate the model object, and extract some quantities on their own. For example, it's common to want to extract the R^2 for a

model. We can't do this from the model object itself, but we can from the `summary()` of the model.

```
> names(Ozone.lm1)

 [1] "coefficients"  "residuals"     "effects"
 [4] "rank"          "fitted.values" "assign"
 [7] "qr"            "df.residual"   "na.action"
[10] "xlevels"       "call"          "terms"
[13] "model"

> names(summary(Ozone.lm1))

 [1] "call"          "terms"         "residuals"
 [4] "coefficients"  "aliased"       "sigma"
 [7] "df"            "r.squared"     "adj.r.squared"
[10] "fstatistic"    "cov.unscaled"  "na.action"

> summary(Ozone.lm1)$r.squared

[1] 0.3619
```

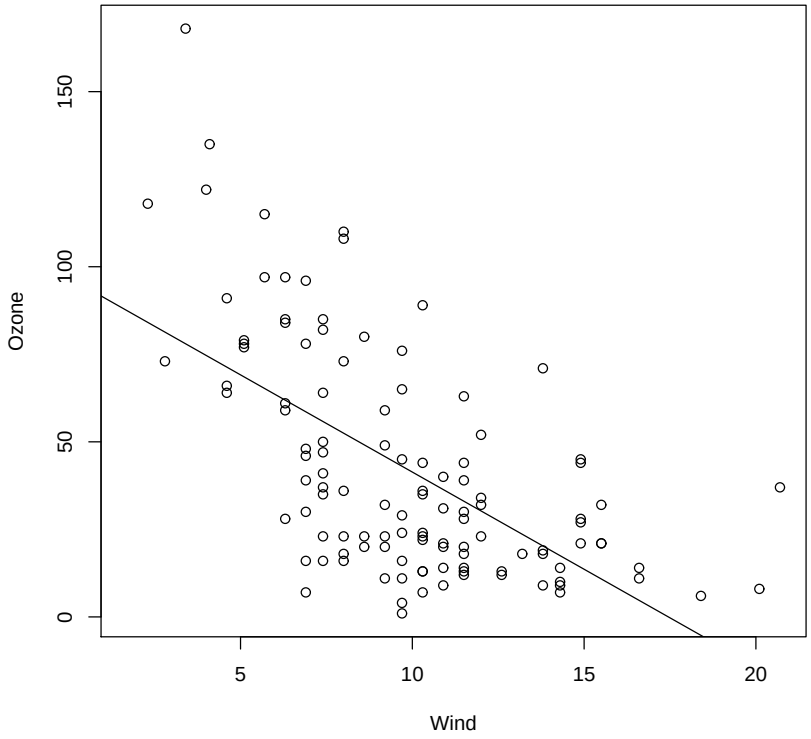
12.3 Presenting the straight line model's suitability in a graph

Adding a fitted line to the scatter plot when a simple relationship has been fitted is actually a very simple task. Note that when we saw the outcome of the simple model above, we obtained just the intercept and slope coefficient values until we employed the `summary()` method to extract more useful information. These two values will be used by the `abline()` function to add the straight line to an existing plot. See Exhibit 12.1 for example.

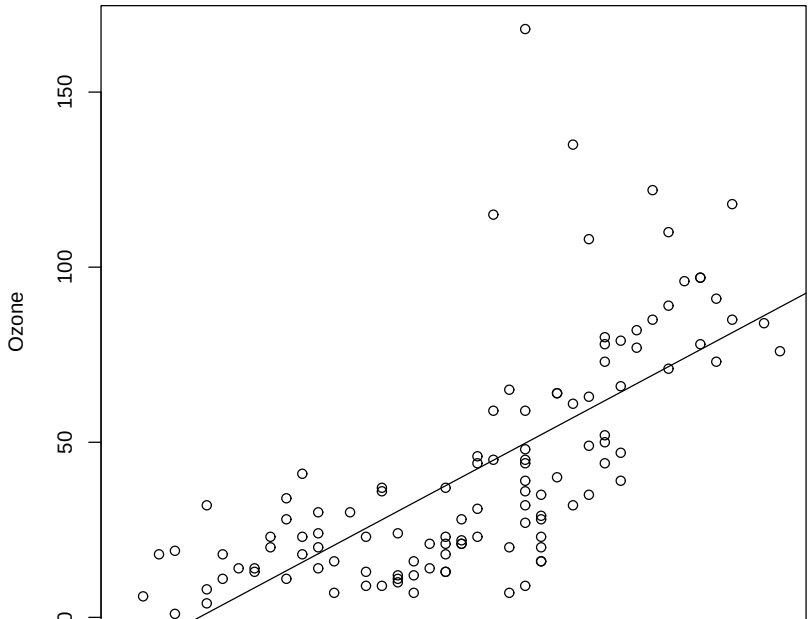
We can see from these graphs that our straight line model might be appropriate for explaining `Ozone` using `Temp` as the predictor, but that the straight line model using `Wind` as the predictor is not a good idea as there is fairly obvious curvature in the relationship. As it happens neither of these models are perfect and more work is required.

Exhibit 12.1 Two simple regressions with the straight line added onto the scatter plots.

```
> attach(airquality)
> plot(Wind, Ozone)
> abline(Ozone.lm1)
```




```
> plot(Temp, Ozone)
> abline(Ozone.lm2)
```



12.4 Model validation using diagnostic plots

R has many useful built-in methods for doing common tasks efficiently. Obtaining residual plots for a model is a great example. The `plot()` command acts on a *lm* object by generating a series of plots. The most commonly used of these plots are the plot of residuals vs the fitted values and the normal probability plot of residuals. These plots are generated by many statistical applications but the other two presented by default in R are not always given by other programs. R also provides a Scale-Location plot of the square root of the absolute value of residuals against the fitted values, and a plot of residuals against the leverages. These approaches for diagnosing problems in a regression model are seldom taught in introductory statistics courses. The scale vs location plot is another means of determining if the residuals have constant variance. Leverage is a measure of how much influence an observation has on determining the model. A rule of thumb says that a leverage of more than twice the average leverage is a problem.

The diagnostic plots for the inadequate model for `Ozone` being predicted by a linear function of `Wind` are presented in Exhibit 12.2. These plots are enhanced by R to add more information than the basic user is familiar with. Additional lines are added to three of the plots to assist in diagnosing problems. We can see from this plot that the model is inadequate as there is a nonlinear relationship between the response and predictor, and further that the residuals might not have very constant variance.

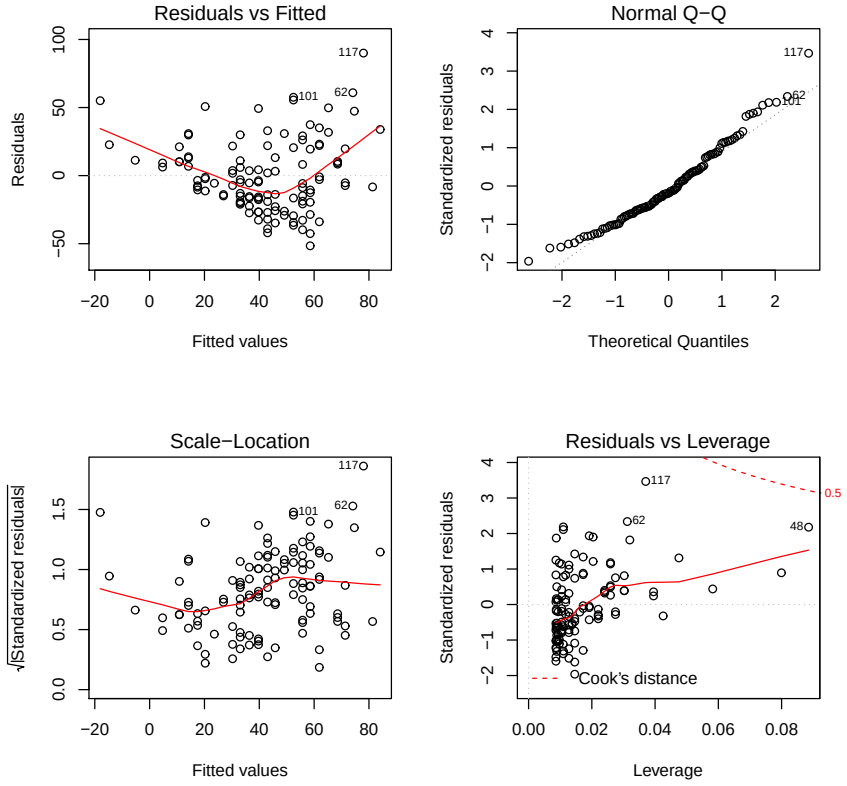
It might be easier to extract just the information we want, so we can build simpler plots ourselves. We need to find the residuals, fitted values and leverages for the model. These are found using the `resid()`, `fitted()`, and `hatvalues()` commands respectively.

```
> Ozone.resid1 = resid(Ozone.lm1)
> Ozone.fitted1 = fitted(Ozone.lm1)
> Ozone.lev1 = hatvalues(Ozone.lm1)
```

The various plots can be constructed using the `plot()`, and `qqnorm()` commands as needed, but if we want the information on the leverages in text form we will need to do tasks like

Exhibit 12.2 Residual plots for the simple regression using Wind to predict the Ozone level.

```
> par(mfrow=c(2,2))
> plot(Ozone.lm1)
```



```
> mean(Ozone.lev1)

[1] 0.01724

> max(Ozone.lev1)

[1] 0.08854

> Ozone.lev1[Ozone.lev1>2*mean(Ozone.lev1)]

      9      18      22      48     117     121     126
0.07994 0.05822 0.03951 0.08854 0.03703 0.04753 0.04256
    148
0.03951
```

The last command in this block has printed out the 8 observations that have excess leverage on this model. We should probably see if these are days that had extremely high wind. This is not done as this model has already been shown to be poor at explaining the amount of `Ozone` in the atmosphere.

12.5 Polynomial regression models

Given the obvious curvature in the relationship between `Ozone` and `Wind` appears monotonic, we can probably try both transforming the variables, and polynomial regression to explain the relationship. We take advantage of the fact that R has an in-built way of producing polynomial terms for insertion into models, and use the `poly()` function in this instance. This command needs to know which variable to work with and the degree of the polynomial desired. For example, to fit the quadratic model we would use the commands

```
> Ozone.poly2 = lm(Ozone~poly(Wind,2, raw=TRUE), data=airquality)
> summary(Ozone.poly2)


Call:
lm(formula = Ozone ~ poly(Wind, 2, raw = TRUE), data = airquality)
```

```
Residuals:
   Min     1Q Median     3Q    Max
-52.29 -16.00  -4.66  12.44  64.04

Coefficients:
                            Estimate Std. Error t value
(Intercept)                  162.572     14.185   11.46
poly(Wind, 2, raw = TRUE)1  -19.444      2.735   -7.11
poly(Wind, 2, raw = TRUE)2    0.649      0.124    5.22
                            Pr(>|t|)
(Intercept)                  < 2e-16 ***
poly(Wind, 2, raw = TRUE)1  1.1e-10 ***
poly(Wind, 2, raw = TRUE)2  8.3e-07 ***
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 23.9 on 113 degrees of freedom
  (37 observations deleted due to missingness)
Multiple R-squared:  0.486,Adjusted R-squared:  0.477
F-statistic: 53.4 on 2 and 113 DF,  p-value: <2e-16
```

Note that in order to obtain a model with the correct coefficients for the polynomial terms in the model, we need to use the `raw` argument, setting it to `TRUE`.

We usually justify the use of a quadratic form by determining there is no practical benefit in making the model more complicated. To do this, we need to investigate the model for the cubic form of the relationship using

```
> Ozone.poly3 = lm(Ozone~poly(Wind,3, raw=TRUE), data=airquality)
```

Rather than continuously investigate the summaries of the various models, we can employ the `anova()` command to compare the models.

```
> anova(Ozone.lm1, Ozone.poly2, Ozone.poly3)

Analysis of Variance Table

Model 1: Ozone ~ Wind
Model 2: Ozone ~ poly(Wind, 2, raw = TRUE)
Model 3: Ozone ~ poly(Wind, 3, raw = TRUE)
  Res.Df   RSS Df Sum of Sq     F  Pr(>F)
1    114 79859
2    113 64360  1     15499 27.62 7.1e-07 ***
3    112 62858  1      1502  2.68     0.1
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

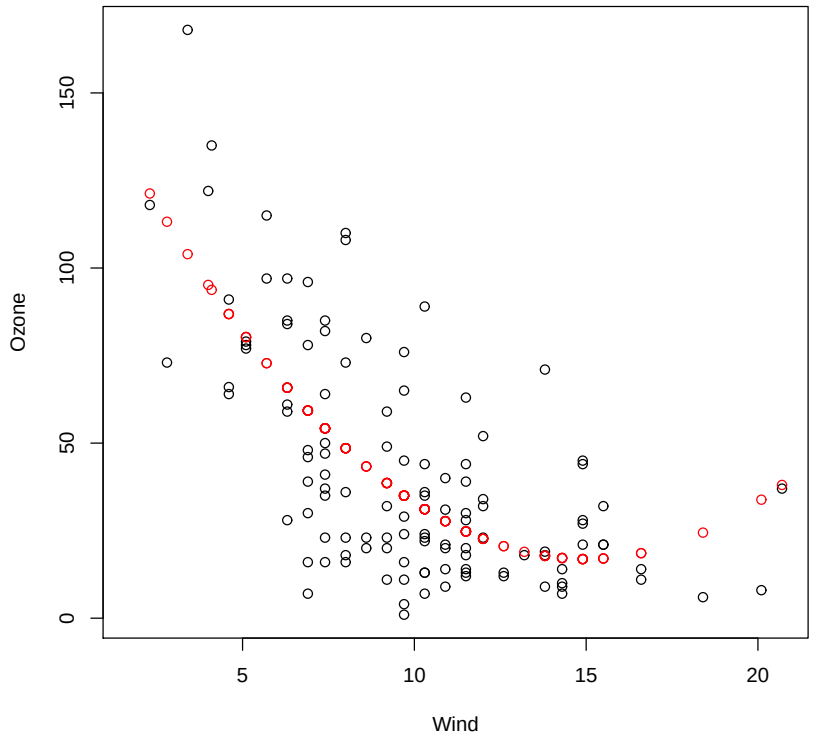
We can compare all three models created thus far because they are a series of nested models. We could not include the model using `Temp` as the predictor in this command for example.

On the basis of the output from our `anova()` command, we might assume that the quadratic form was sufficient to explain the relationship between `Ozone` and `Wind` because there is little to be gained by adding the cubic term to our model. Validation of the chosen model via the residual analysis is suggested as a next step.

12.6 Presenting the polynomial model's suitability in a graph

The `abline()` command demonstrated earlier is only useful for straight lines. We have seen that the quadratic function is better at explaining the relationship between `Ozone` and `Wind`. One solution is to store the fitted values from this model and plot them against the `Wind` variable, but this will only show the series of points not a smooth curve.

```
> plot(Ozone~Wind, data=airquality)
> points(airquality$Wind[!is.na(airquality$Ozone)], fitted(Ozone.poly2), col=2)
```



Note that the `points()` command used here includes the `col` argument to change the colour of the points to red — the second colour in the list of colours.

Also note that there are missing values in the records for `Ozone`. The fitted values from the model are only calculated for the observations where `Ozone` was recorded, so we have employed the `!()` logical operator and `is.na()` command to include only complete cases for `Ozone` and `Wind`.

Another solution that plots a curve instead of points, is demonstrated in Section **??**. It is much more elegant, especially given the missing data problem encountered in this example.

12.7 Multiple regression models

The addition of multiple terms on the right hand side of the formula in the `lm()` command is very simple. We can put both `Wind` and `Temp` into a model as predictors using

```
> Ozone.lm3 = lm(Ozone~Wind+Temp, data=airquality)
```

If we want to combine the models for the quadratic form for `Wind` and the linear form of `Temp` we would

```
> Ozone.lm4 = lm(Ozone~poly(Wind,2)+Temp, data=airquality)
```

This then allows us the opportunity of using the `anova()` command as demonstrated above to compare these two models and one constructed earlier which used only `Temp` as a predictor.

```
> anova(Ozone.lm2, Ozone.lm3, Ozone.lm4)

Analysis of Variance Table

Model 1: Ozone ~ Temp
Model 2: Ozone ~ Wind + Temp
Model 3: Ozone ~ poly(Wind, 2) + Temp
  Res.Df   RSS Df Sum of Sq    F  Pr(>F)
1    114 64110
2    113 53973  1     10137 25.7 1.6e-06 ***
3    112 44213  1      9760 24.7 2.4e-06 ***
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The creation of interaction variables is also simple in R Changing the "+" sign in the model to a "*" sign tells R that we want the two main variables and their interaction to be included. The interaction term uses a ":" between the variable names in the output.

```
> Ozone.lm5 = lm(Ozone~poly(Wind,2)*Temp, data=airquality)
> summary(Ozone.lm5)


Call:
lm(formula = Ozone ~ poly(Wind, 2) * Temp, data = airquality)

Residuals:
   Min     1Q Median     3Q    Max
-44.70 -10.54  -3.25  10.60  82.56

Coefficients:
                    Estimate Std. Error t value Pr(>|t|)
(Intercept)          -91.113     18.779   -4.85  4.1e-06
poly(Wind, 2)1       108.281    262.970    0.41     0.68
poly(Wind, 2)2        99.235    177.521    0.56     0.58
Temp                   1.688      0.237    7.13  1.1e-10
poly(Wind, 2)1:Temp   -3.301      3.290   -1.00     0.32
poly(Wind, 2)2:Temp   -0.147      2.273   -0.06     0.95

(Intercept)         ***
poly(Wind, 2)1
poly(Wind, 2)2
Temp                 ***
poly(Wind, 2)1:Temp
poly(Wind, 2)2:Temp
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 20 on 110 degrees of freedom
  (37 observations deleted due to missingness)
Multiple R-squared:  0.65,Adjusted R-squared:  0.634
```

```
F-statistic: 40.9 on 5 and 110 DF,  p-value: <2e-16
```

Note that this model includes two interaction terms, and on the face of it, neither term looks like it is contributing towards explaining the amount of ozone in the atmosphere. This can be tested using

```
> anova(Ozone.lm5, Ozone.lm4)

Analysis of Variance Table

Model 1: Ozone ~ poly(Wind, 2) * Temp
Model 2: Ozone ~ poly(Wind, 2) + Temp
  Res.Df   RSS Df Sum of Sq    F Pr(>F)
1    110 43801
2    112 44213 -2      -412 0.52    0.6
```

Multiple regression models can be checked using the approaches described in Section 12.4 above, but it may also prove useful to plot residuals from the current model against each current and potential predictor variable in the data set.

12.8 Indicator variables

When working with a variable that can take one of two values, such as gender, many statistical packages need the user to create an indicator variable if this effect is to be incorporated into a regression model.

An indicator variable takes the values zero or one, where a “1" indicates one of the two possible values. Normally, the software will create an indicator for each of the values of the original variable, and when the original variable takes three levels, three indicator variables are made.

The advantage of the indicator is that the model fitted has a coefficient for the indicator variable that reflects the constant difference between the two groups within the data implied by the original variable.

R does not need explicit creation of indicator variables as it will see the form of the variable and create indicator variables in the background. The output for an indicator

variable in the regression summary is only ever so slightly different in that you will see `GenderM` where you might have thought to see just `Gender`. This is because R tells you that the indicator variable created, and therefore the coefficient printed in the output, is for the Male level of Gender. For reasons not explained here, you will not see both `GenderF` and `GenderM` in the output unless you explicitly ask R not to fit the intercept term.

As an example,

Chapter 13

LURN... To Create Complex Graphs

This chapter assumes an understanding of the material in Chapter 7 on creating simple graphs. That material included adding simple elements of good graphing practice, like meaningful titles and axis labels. In this chapter we see colour, multiple graphs in one window, multiple sets of points on a single set of axes, adding extra points or lines, and introduces colour, different symbols and the legends that will then become necessary additions.

13.1 More than one graph in a single window

We have already seen how to combine a set of graphs to be collected into one window so that they form a single object for inserting into documents. The `par()` command controls a large number of graphical parameters, including the layout of the graph window. One of its common uses is to split up the graph window quickly.

Exhibit 12.2 shows the residual analysis plots for a regression model. Four separate graphs form that residual analysis by default. If we do not combine them into a single window using the `par()` command we need to manipulate the four graphs separately when it comes time to include them in a report.

The `mfrow` argument in the `par()` command tells R how to divide up the graph window. Note that it divides the existing window so that the aspect ratio of the separate panels may change from what you expect unless you either change the overall shape of the

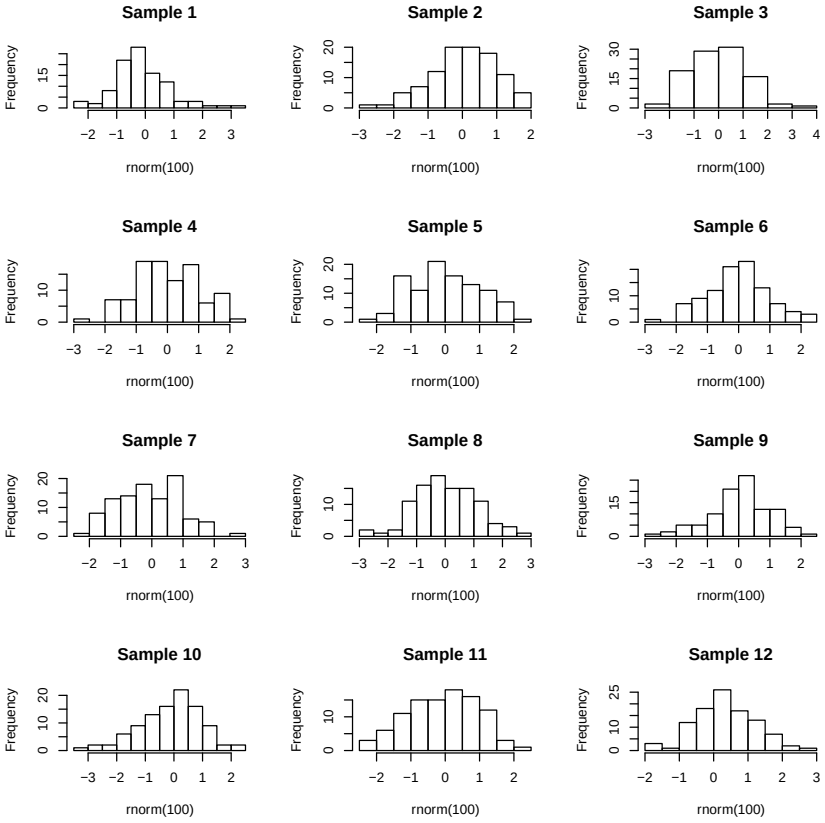
graph window (see Exhibit 7.3 where we did this for a boxplot) or you divide the window up equally in both vertical and horizontal directions (see Exhibit 12.2). In all examples presented thus far, the window has been divided up into four equal sections. R then inserts successively created graphs into those sections until the window has four graphs in it. A fifth graph would appear on a new window that had four sections in it; this would fit graphs five to eight, with the ninth going onto a new sheet and so on. You can split the window into what ever size grid you like. Keeping the aspect ratio of the graphs is a good idea, but making them slightly wider than they are high is better than the reverse. That is, five rows of four graphs is better than four rows of five graphs, unless you are prepared to change the overall height and width of the window.

To illustrate creation of a lot of graphs and the efficiency of managing them as a single graph window, we will create the histograms of twelve distinct random samples of size 100, taken from the standard normal distribution.

The final graph appears in Exhibit **??**, but there are a couple of functions that need explaining. First, the `par()` command with the `mfrow` argument opens a graphing window and splits it into twelve sections. The second line uses the `for()` command to create a loop which is just a set of commands that are grouped together and run over and over until the end of the loop. In this case, the loop is run a total of twelve times with an arbitrary lable given to the counter used. A loop's counter is often not used in the processing of the code so is given a meaningless name; in this instance, the word `Counter` is used to point out where the counter appears in the code. The third line creates a histogram with the `hist()` command, of a set of 100 random values taken from the standard normal distribution, found using the `rnorm()` command. Note the code also adds a title to each of the distinct graphs. This is done using the `paste()` command which joins some text with the counter number to form the title that will, as a consequence, change for each graph. If there had been two or more commands inside the loop, they would need to be grouped together using curly braces. Ignoring the braces would mean only the first line of the set would be issued. The braces are included here as an example of good practice.

Exhibit 13.1 Twelve samples of size 100 drawn from the standard normal distribution, plotted using separate histograms in a single window

```
> par(mfrow=c(4,3))
> for(Counter in 1:12){
   hist(rnorm(100), main=paste("Sample", Counter))}
```



13.2 Allowing different sized graphs to be included in a single window

The `layout()` command is very powerful, but can be a nightmare to sort out. Before demonstrating how to implemet the command, I need to show you what is possible by explaining how to think about the way the command works.

Regardless of the size and shape of the particular graphs and how you want them arranged, you need to think of the graphing winow as a rectangle that can be split into any number of rows and columns. The shape of the sections of the window are dependent on the number of rows and columns of course so for the time being let's work with a 4×4 grid. This means the 16 sections are the same shape as the overall window.

We tell the `layout()` command how to use up the sections by giving each section a label. The sections that get used for the first graph will be given a "1", sections for the second graph get a "2" and so on. For illustration, let's use up nine of the 16 sections for a large graph (a scatter plot), and place three smaller ones down the side of it; below those, we will add a pair of boxplots which each need two sections of the window.

The `layout()` command needs to interpret these labels as a matrix. It would probably help to write out what you want for the arrangement, create the matrix of labels, check that the two agree, and then worry about the code for the various graphs. So for my description, I want a 4×4 matrix, with a total of six different numbers in it. Nine of them will be ones, a two, a three, a four, two fives, and two sixes. They will be laid out as follows:

$$\begin{matrix} 1 & 1 & 1 & 2 \\ 1 & 1 & 1 & 3 \\ 1 & 1 & 1 & 4 \\ 5 & 5 & 6 & 6 \end{matrix}$$

Getting this matrix in R means identifying its contents, naming exactly how many rows or columns are needed and how to insert the values into the defined structure. We use the `matrix()` command to create the object for our arrangement, and the `c()` command to combine the labels into a single object. Arguments to the `matrix()` command we need are either the `nrow` or `ncol` (one will do) and the `byrow` switch that tells the `matrix()` command to fill up the matrix by row until it is full.

```
> Arrangement.mat = matrix(c(1,1,1,2,1,1,1,3,1,1,1,4,5,5,6,6), byrow=TRUE, nrow=4)
> Arrangement.mat

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

Having established the success of the code for the arrangement structure, we just open the graph window using the `layout()` command, and add plots in the order our labels specify. The full set of instructions is given in Exhibit 13.2. Note that the graphs are chosen for the illustration of the `layout()` command, not for the purposes of showing anything of interest in any of the graphs created. The `quakes` data set distributed with the base installation of R is used in this illustration.

Note that the way the points have been plotted and the way the text appears in these graphs, differs. The addition of the `cex` argument in the third and fourth `plot()` commands forces the size of the plotted symbol to be reduced. This is known as *character expansion*, and can be applied to the text that appears on the graphs as well. Individual elements of text are adjusted using the `cex.main`, `cex.sub`, `cex.lab`, and `cex.axis` arguments. See the help for the `par()` command for further details on their use.

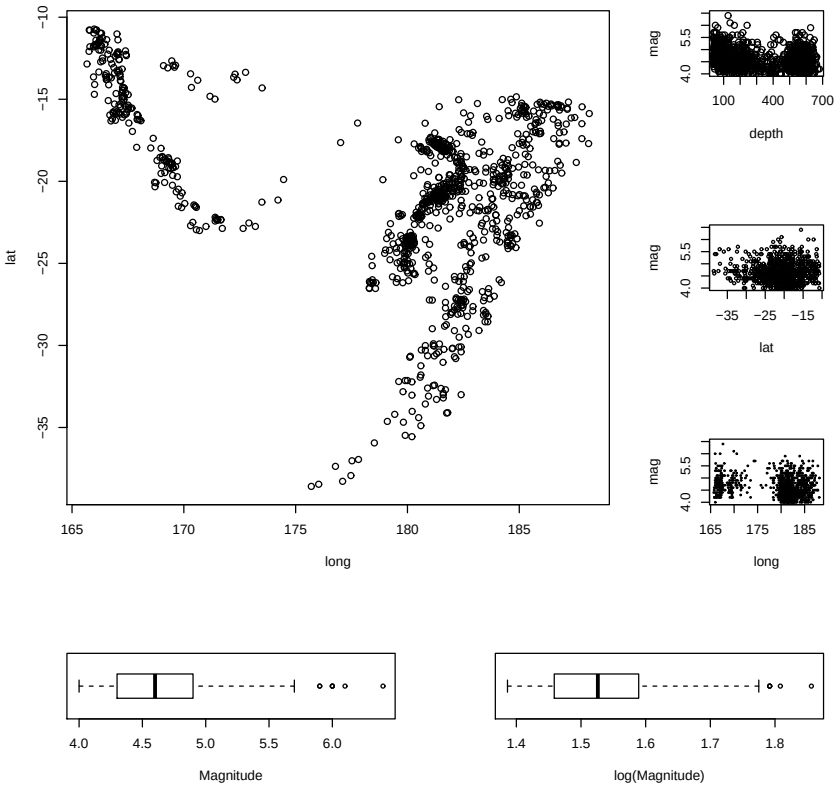
13.3 Using colour or different symbols

We show how to create a graph with different colours and symbols without passing judgement on the suitability of the colours or symbols used. The important feature to remember when you are about to create such a graph is that the reader will not necessarily see the difference in colour as you do. Similarly, the selection of symbol should be thought out carefully. We do not want a solid symbol to appear bigger than a hollow one or vice versa.

The `iris` data supplied with R for this example. It has five variables in all; four are for numeric measurements and the fifth is the species. In Exhibit 13.3 we use three colours for the species and in Exhibit 13.4 we use different symbols. There is no reason why you might not use both arguments in combination to use different colours for the different symbols.

Exhibit 13.2 A selection of differing size graphs combined into one graph window

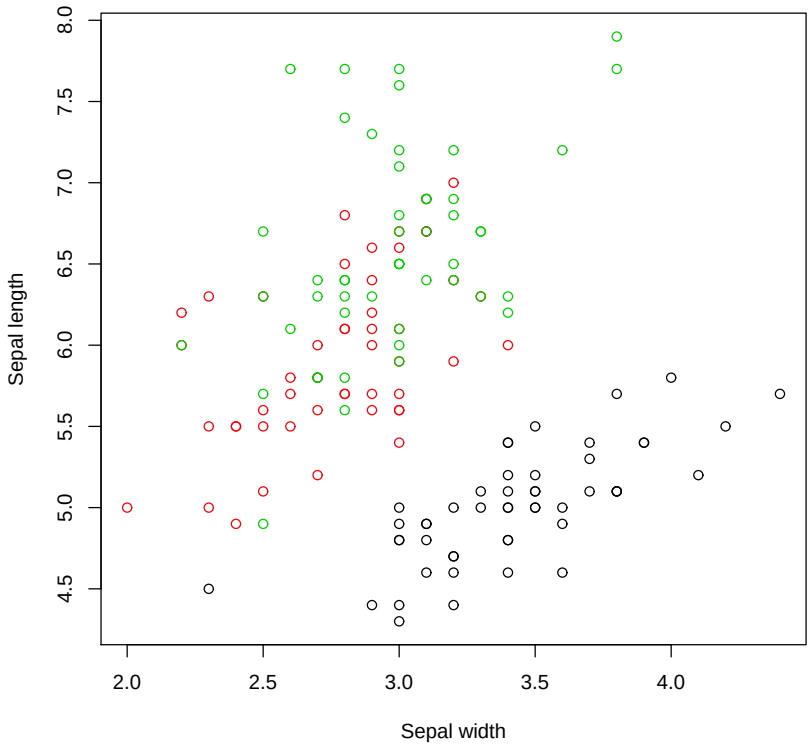
```
> attach(quakes)
> layout(Arrangement.mat)
> plot(long, lat)
> plot(depth, mag, ylim=c(4,6.5))
> plot(lat, mag, ylim=c(4,6.5), cex=0.5)
> plot(long, mag, ylim=c(4,6.5), cex=0.25)
> boxplot(mag, xlab="Magnitude", horizontal=TRUE)
> boxplot(log(mag), xlab="log(Magnitude)", horizontal=TRUE)
```




```
> detach(quakes)
```

Exhibit 13.3

```
> plot(Sepal.Length~Sepal.Width, data=iris, xlab="Sepal width", ylab="Sepal length", col
```



Note that we needed to convert the species names to numbers using the `as.numeric()` command to get the symbols to be plotted.

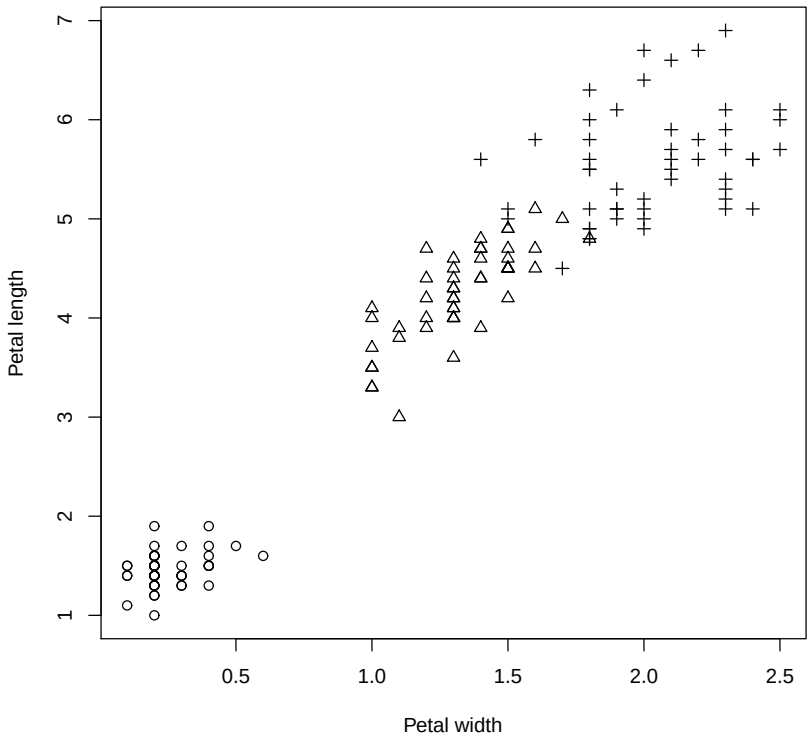
R seesm to have converted the species names to numbers when we called for different colours, starting with 1 for black, 2 for red, and 3 for green. As an aside, the colour can be specified by name or by a number. There are well over 600 named colours to choose from so the full list is therefore not printed here. Type

```
> colors()
```

if you are really keen to know what they all are. If this list is still not sufficient to meet your needs, you can design your own colour using a combination of red, green, and blue components. This is explained under the help for the `par()` command.

Exhibit 13.4

```
> plot(Petal.Length~Petal.Width, data=iris, xlab="Petal width", ylab="Petal length", pch
```



13.4 Adding points to an existing graph

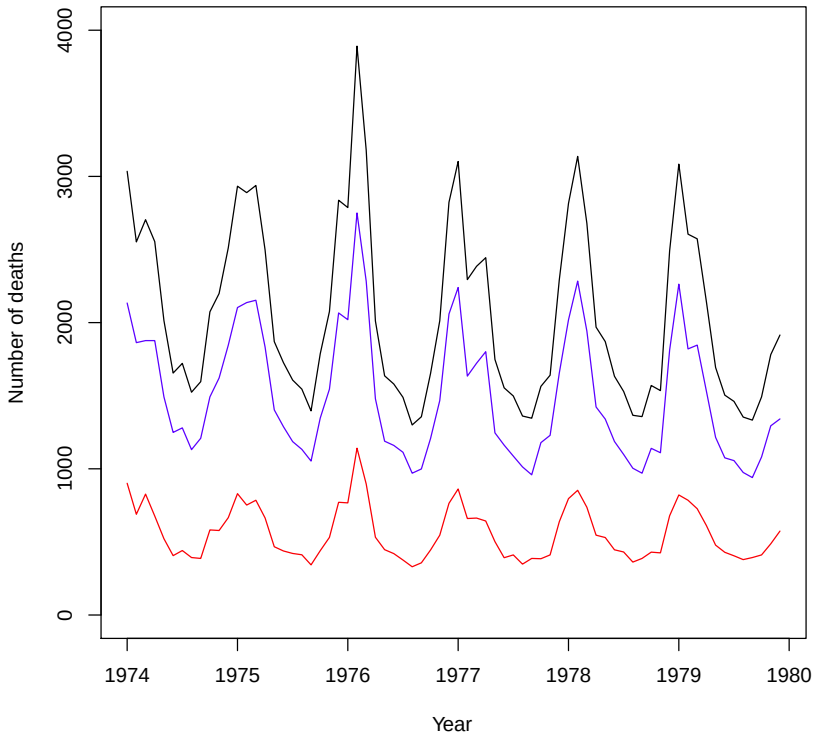
13.5 Using lines instead of points

The default action for any scatter plot is to mark each point. If the set of x-values are ordered we can create a line plot. This is often seen as a time series plot. We create a time series plot very easily using the `plot()` command on an object that has been stored as a time series. There are many examples of time series i the *datasets* package. We use the `ldeaths` series which counts the number of deaths due to lung diseases in the United Kingdom, and its associated series for male and female deaths.

In Exhibit 13.5, we see a set of three commands. The first plots a time series on a set of axes and the axis labels. The second and third each add a set of line segments from the

Exhibit 13.5 Deaths from lung diseases in the United Kingdom

```
> plot(ldeaths, ylim=c(0,4000), xlab="Year", ylab="Number of deaths")
> lines(fdeaths, col="red")
> lines(mdeaths, col="blue")
```



other two series using the `lines()` command.

We will see that the plot can be improved by adding a legend in Exhibit 13.7.

13.6 Adding lines to an existing graph

In Section 12.3 we saw how to add a fitted line from a simple regression model to a plot of the data used to fit the model, using the `abline()` command. This command has other uses that are explained here. This section is only interested in the addition of a single straight line at a time; a series of line segments joined together was covered in Section 13.5.

Perhaps the easiest tasks of adding lines to a plot is to add either a horizontal or a vertical line. In Exhibit 13.6, we see the locations of the quakes marked by their latitude and longitude. On this occasion, the graph window has been left square and the spacing on the graph itself would be different for the latitude and longitude if we did not force the axes to have similar length for each direction. The `xlim` and `ylim` arguments create the axes, but note the need to correctly define the axis for latitude from a large negative number to the less negative number so that the orientation of north and south is kept consistent with convention. Fiji is in the southern hemisphere after all!

The `plot()` command creates the axes and plots the points for the earthquakes in Exhibit 13.6. We then add two vertical and two horizontal lines to the plot, both using the `abline()` command. Notice that the four commands used to add the four extra lines state the horizontal or vertical placement with a single letter, and that the line type has been chosen in three instances. It turns out that line type 1 (a solid line) is the default. More types do exist but those given are the most commonly chosen. The thickness of the lines can be altered also, using the `lwd` argument. The default line width is 1.

It is also possible to add a straight line that is niether vertical or horizontal using the `abline()` command. To achieve this we need to know the slope and intercept of the line. This is the form of this command used when we added the fitted line for a simple linear regression model to the plot in Section 12.3. The plotted line will have the equation $y = a + bx$ and we need only supply the values of a and b to the `abline()` command. Exhibit **??** shows an example using the

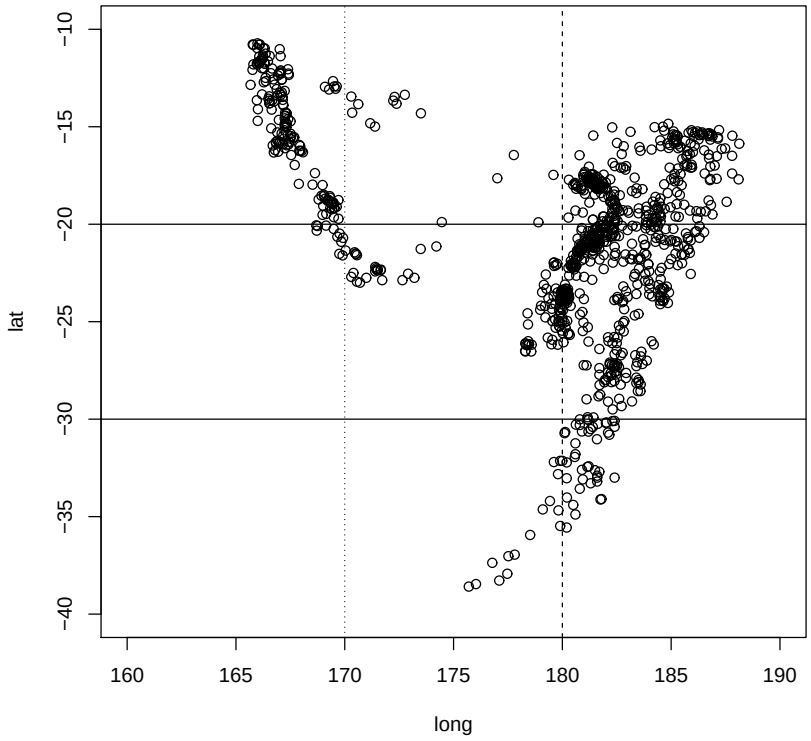
13.7 Adding a curve to a graph

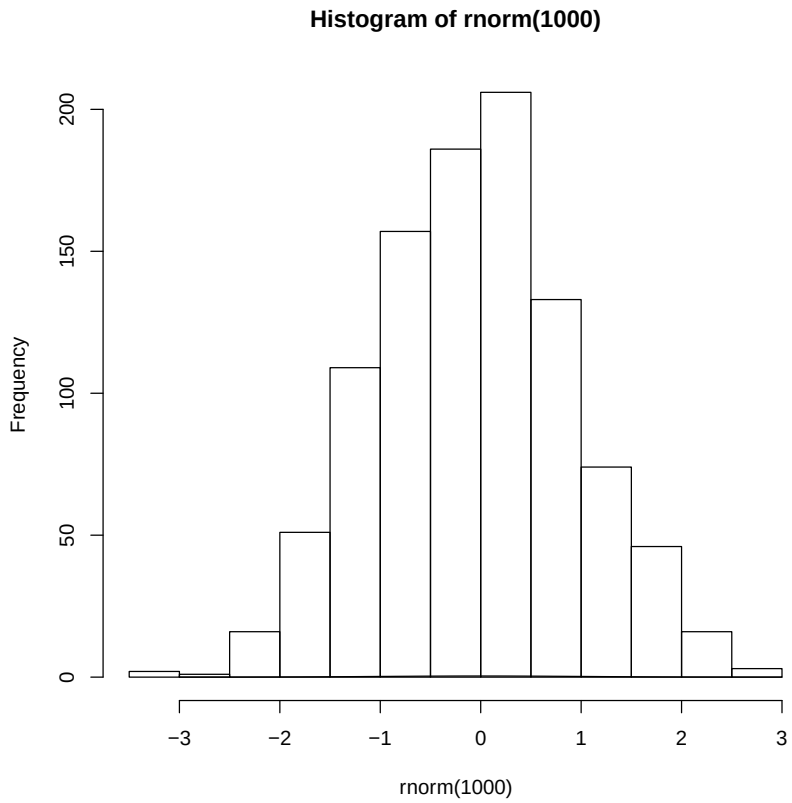
Use of `abline()` is great if you have a straight line to add to a plot, but as soon as the line needs to be curved the problems begin. We will see that it isn't actually too difficult if we use the `curve()` command. First, we add a curve representing the normal probability density function to a histogram of random values.

```
> hist(rnorm(1000))
> curve(dnorm(x), -3,3, add=TRUE)
```

Exhibit 13.6 Horizontal and vertical lines added to a scatter plot

```
> plot(lat~long, data=quakes, xlim=c(160,190), ylim=c(-40,-10))
> abline(h=-30)
> abline(h=-20, lty=1)
> abline(v=180, lty=2)
> abline(v=170, lty=3)
```

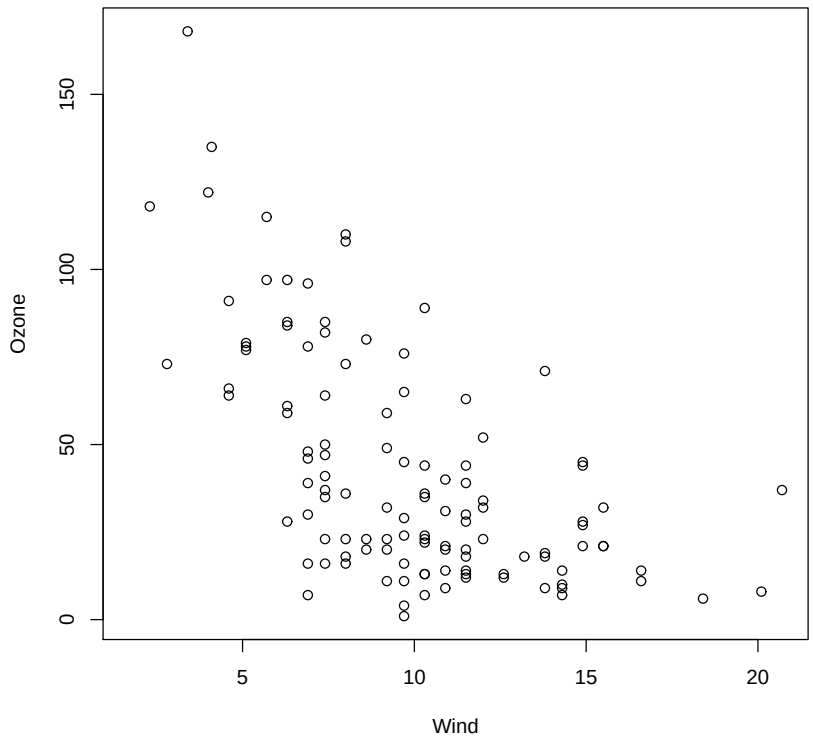




The `curve()` function has four arguments in this example. First is the function being plotted, which in this case is `dnorm()` —the density function for the normal distribution. We then supply the range of values over which this function is to be evaluated using two arguments, and finally we ask that the curve be aded to the existing plot. There are other arguments that we could add, including `n`, the number of points to be evaluated over the specified range.

The second example shown here is the addition of the quadratic fitted line for a regression model. We use the example given in Section 12.5.

```
> Ozone.poly2 = lm(Ozone~poly(Wind,2, raw=TRUE), data=airquality)
> Coeffs=coef(Ozone.poly2)
> plot(Ozone~Wind, data=airquality)
```




```
> curve(Coeffs[1]+Coeffs[2]*x+Coeffs[2]*x^2, min(Wind), max(Wind), add=TRUE)

Error in curve(Coeffs[1] + Coeffs[2] * x + Coeffs[2] * x^2, min(Wind), :  object
'Wind' not found
```

13.8 Adding text to an existing graph

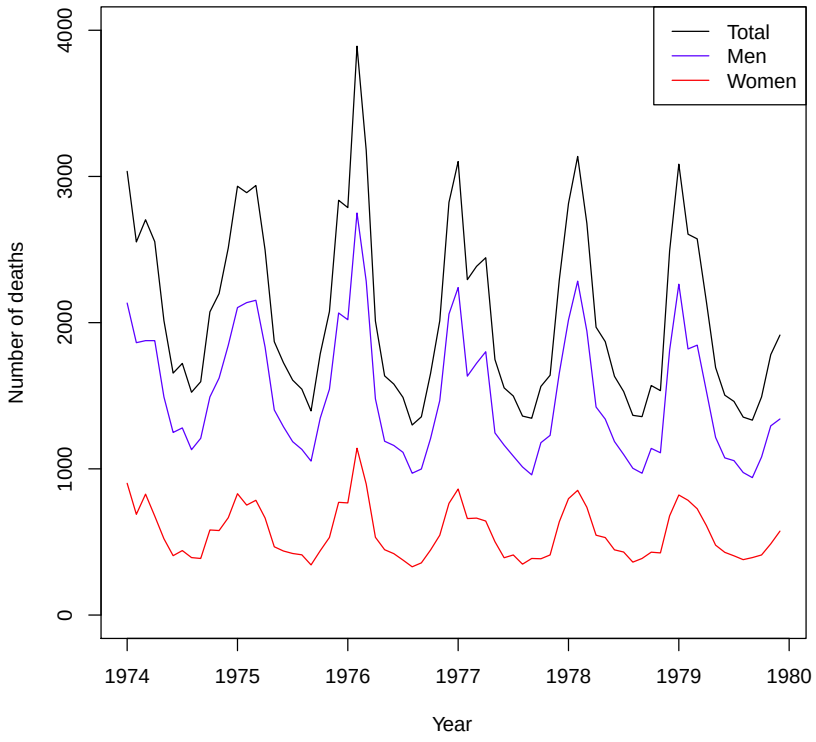
13.9 Adding a legend for different information within a graph

13.10 More complex bar charts

A bar chart, as seen in Section 7.7.1, has a bar showing the frequency of each level of a single factor. A *cluster bar chart* has a bar for each combination of two factors. The

Exhibit 13.7 Deaths from lung diseases in the United Kingdom (with legend added)

```
> plot(ldeaths, ylim=c(0,4000), xlab="Year", ylab="Number of deaths")
> lines(fdeaths, col="red")
> lines(mdeaths, col="blue")
> legend("topright", legend=c("Total", "Men", "Women"), col=c("black", "blue", "red"), l
```



clustering indicates how the bars are grouped together. A *stack bar chart* on the other hand has a single bar for each level of one factor split into sections for each level of the second factor; it therefore appears to stack the bars on top of one another.

To create an example, we will first create a list of frequencies of outcomes and the two categorical variables required. The cluster bar chart and stack bar chart then appear in Exhibits 13.8 and 13.9 respectively. Our data consist of the counts of males and females of ages 17 to 20.

```
> Frequency=c(12,19, 22, 18, 11,13, 13, 12)
> Gender=rep(c("Male", "Female"), 4)
> Age=rep(c(17,18,19,20), each=2)
```

To create the two types of bar chart, we need to create a matrix for the counts to be used in the graphs. In this instance, this is done using the `tapply()` command The actual plotting is done using the `barplot()` command which recognizes it is working on this matrix and acts accordingly, including the ability to generate a legend for the graphs.

13.11 Contour plots

A scatter plot indicates the presence of an observation at pairs of x and y values. When a third variable is also measured, we run into some trouble when comparing it to the values of x and y in a single graph. Three-dimensional displays can be employed but require the interpretation of the creator and the reader to turn the two-dimensional graph on paper or the screen into the three-dimensional object actually being displayed.

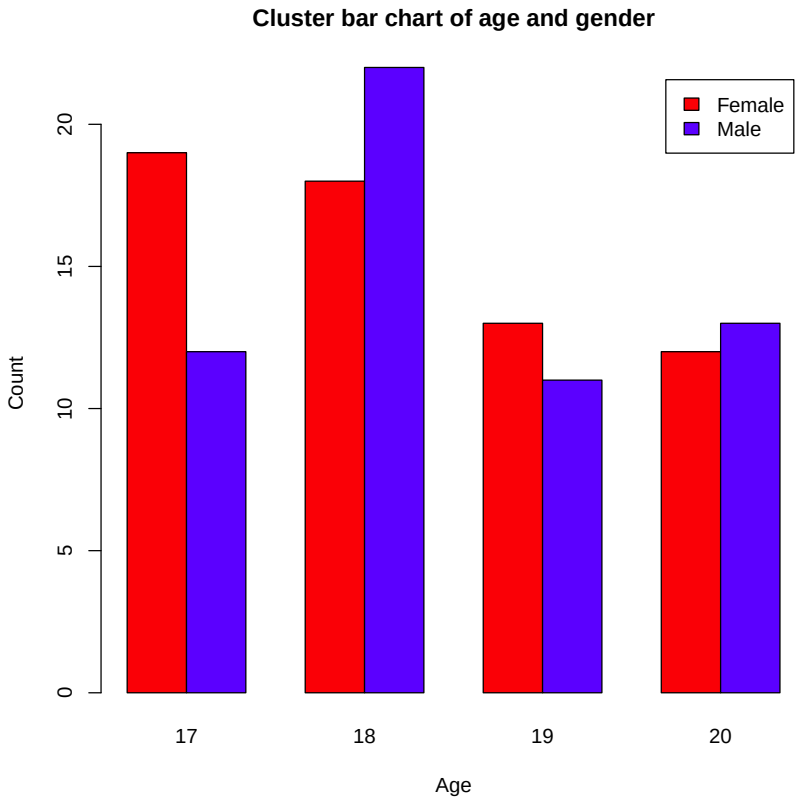
A contour plot is a two-dimensional display. It shows a set of curves that represent common values of a variable plotted against two other variables. They are commonly seen in weather charts where barometric pressure lines (contours) are plotted on the map of a country or region (latitude and longitude are the two other variables).

Exhibit 13.10 shows the contour lines for the altitude of a volcano in Auckland, New Zealand. Mt Eden is also known by its Maori name Maunga Whau and the data we need for this illustration is provided in the datasets supplied with base installations of R.

This provides a black and white image that has the contours marked with the value of the third variable. A coloured version of the plot is available using the `filled.contour()`

Exhibit 13.8 An example of a cluster bar chart

```
> barplot(tapply(Frequency, list(Gender, Age), sum), col=c(2,4), legend=TRUE, beside=TRU
> title("Cluster bar chart of age and gender", ylab="Count", xlab="Age")
```



command in place of the `contour()` command in Exhibit 13.10. An even more impressive version of this plot can be obtained by looking at the example that accompanies the help for the `volcano` data. Typing:

```
> example(volcano)
```

generates the plot using the code given in the help file.

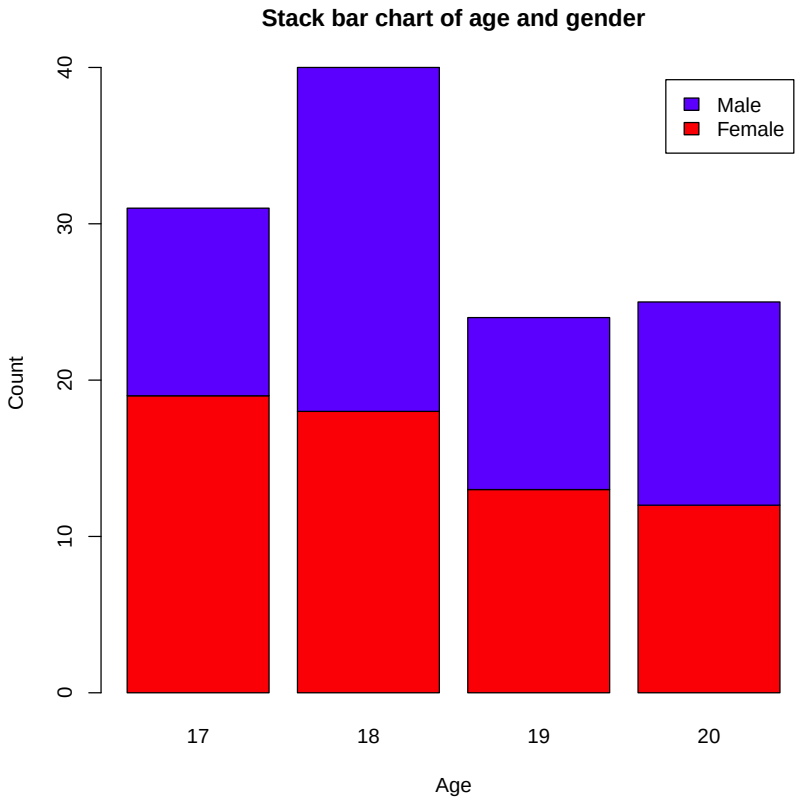
The `volcano` data is stored as a matrix of altitudes. The `contour()` command uses this to interpolate the coordinates of the points where the altitude is constant.

A further example is generating the contours for a function. Let's plot the function

$$z = 0.75x^2 + 0.5y^2 - 0.4xy + x - y - 1$$

Exhibit 13.9 An example of a stack bar chart

```
> barplot(tapply(Frequency, list(Gender, Age), sum), col=c(2,4), legend=TRUE)
> title("Stack bar chart of age and gender", ylab="Count", xlab="Age")
```



over the range $-5 \leq x, y \leq 5$. All necessary codes are given in Exhibit 13.11 which includes the contour plot of the function.

Note the `contour()` function needed to know the unique values of the x and y variables we created, but that the z matrix needed these values repeated enough times so that the function was evaluated over the entire set of combinations of x and y.

Exhibit 13.10 Contour plot of Mt Eden, a volcano in Auckland, New Zealand.

```
> contour(volcano)
```

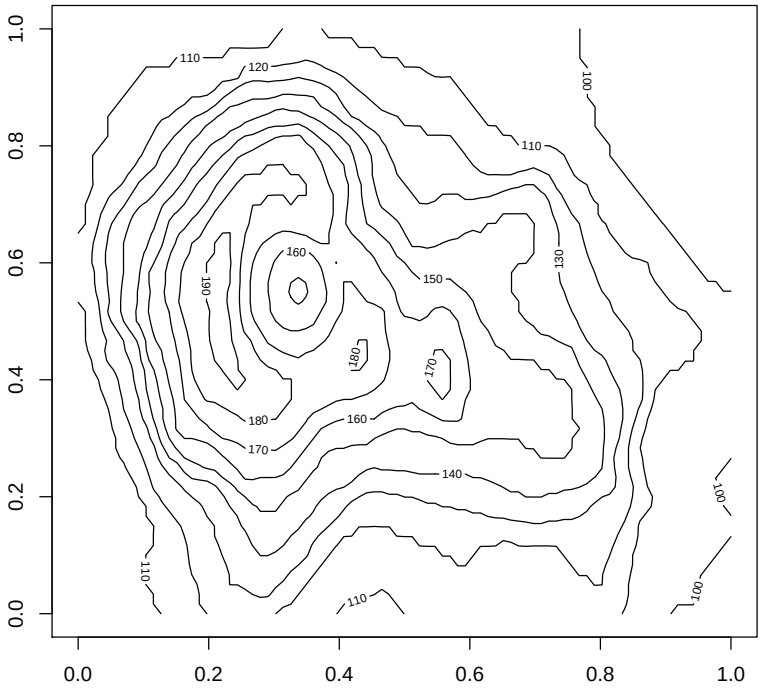
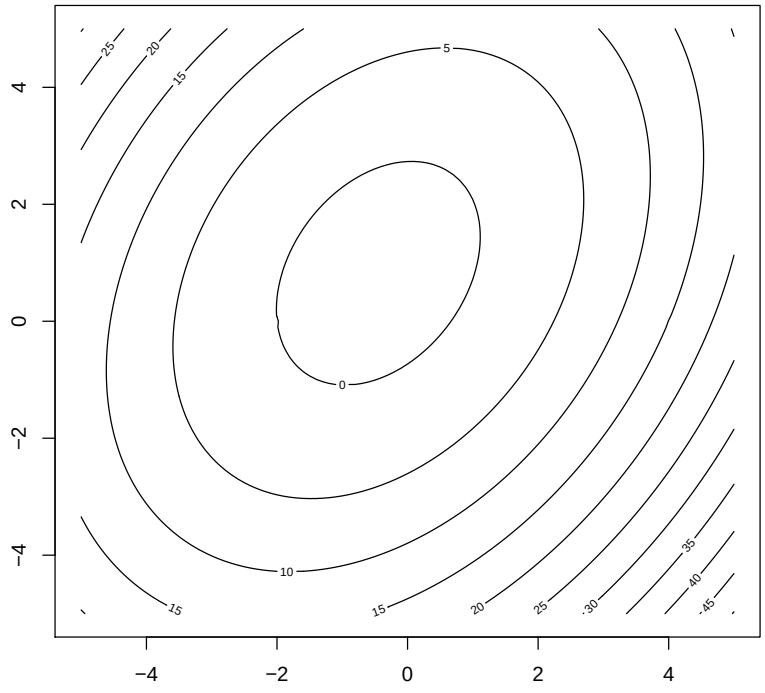


Exhibit 13.11 Contour plot showing values of z, where $z = 0.75x^2+0.5y^2-0.4xy+x-y-1$

```
> xUnique=((0:100)-50)/10
> yUnique=((0:100)-50)/10
> x=rep(xUnique, 101)
> y=rep(yUnique, each=101)
> z=matrix(0.75*x^2 + 0.5*y^2-0.4*x*y+x-y-1, nrow=101)
> contour(xUnique, yUnique, z)
```



Chapter 14

Extending R beyond the base installation

This chapter explains how you do some tasks that are not part of the basic installation of R. This includes instructions on how to add extra functionality via add-on packages and pointers to some user interfaces.

14.1 Installing additional packages

Many R users find that the vast range of functions that are built into the basic R installation are not enough to meet their specific needs. R is relatively easy to adapt to meet their needs because additional functions are easily created to perform specific tasks.

In order to let others use their creations, R users often make their functions available in add-on packages that can be downloaded by any R user. The network of servers that store these packages is known as CRAN, which is short for Comprehensive R Archive Network. There are over 8000 add-on packages currently available via CRAN as well as others that do not meet the CRAN criteria. To make it to CRAN, an add-on package must meet standards of presentation, quality of documentation, and be available for all operating systems. Add-on packages that do not meet these criteria do exist, especially those that are specific to one operating system.

Installing an add-on package requires two actions. First, you must ask R to install a package of a certain name, either using the `install.packages()` command or the Packages pull-down menu item.

The second task is necessary only once in each R session. When you go to install a package from CRAN, R will ask you to select a *CRAN mirror*. This just wants you to choose which one of the servers scattered around the world is the one you should point your internet connection towards.

These tasks are most commonly achieved by selecting the menu items and associated dialogue boxes. If you do not want to use the dialogue box approach for selecting a CRAN mirror, there is a text based alternative. If you type

```
> chooseCRANmirror(FALSE)
```

you will be given a list of the available CRAN mirrors and a different prompt. Type in the number linked to the CRAN mirror you prefer and press the Enter key. If you want to choose the first item on the list (which is a cloud-based server anyway) then you could avoid reviewing the list using:

```
> chooseCRANmirror(ind=1)
```

Installation of individual packages can be achieved using the `install.packages()` command. Investigate its help page to ensure you get the most out of its use.

One key argument for this command is `dependencies`. There are many packages that rely on the work of other packages for their success. It's probably a good idea to get into the habit of installing all dependent packages. For example:

```
> install.packages("Dodge", dependencies=TRUE)
```

which will install the *Dodge* package. If this package used another package that was not currently installed on your machine, that package would also have been downloaded and installed.

14.2 Updating add-on packages

If you want to be sure you have the latest version of the packages you have downloaded are being used, then every once in a while you should issue the command

```
> update.packages(ask=FALSE)
```

Note that the `ask` argument is set to `FALSE` here. The default action is to ask the user if they want a particular package updated. I find this frustrating, especially if it's been a while since I updated my packages.

You do not need to do this task very often. I would recommend doing it after installing a new version of R, or if you see a warning message about the version of a package you use.

Finally, note that if you have not established a connection with a CRAN mirror in the current R session, then you will be prompted to do so before anything is downloaded.

14.3 Using an enhanced graphical interface

The base installation of R doesn't offer much statistical functionality in its menu system (assuming you use one!). Various projects are under development that enhance the way users can interact with R.

Perhaps the most well-known is the R Commander project. This interface has been given a lot of development over a number of years now and can support the needs of most introductory level statistical analyses. Its implementation is via the *Rcmdr* add-on package downloadable from CRAN.

I don't use this package as it is more suited to novice to intermediate R users. It also creates a host of new functions that improve on similar functions included in base R that I don't need. The idea of working with mouse clicks is counter-intuitive to reproducible research ideas that most medium to advanced R users like.

14.4 Use of an integrated development environment (IDE)

The best integrated development environment (IDE) for R users is R Studio. Some people find this way of working to be a bewildering experience. Others love the convenience of having each project contained in a structured environment.

My advice for anyone wanting to use RStudio or any new IDE is to find someone to give you a hands-on demonstration. Perhaps watching videos from the internet are a useful alternative, but you need to know that this tool is what you need before investing time in its use.

Be warned: RStudio is a very powerful and useful tool for even the most advanced R users. As such, it can be too much for the novice R user.

Chapter 15

LURN... To use the BrailleR add-on package

This chapter explains how the *BrailleR* add-on package can be used by blind people to get many tasks done quickly and effectively. In general, the package does three things:

1. creates a copy of hte R console/terminal in a text file including all standard output.
2. creates text descriptions of some graphs. These tools should help the blind user create graphs (with confidence) for the sighted world we must work in.
3. provides convenience functions for novice users who do not know enough R commands to survive. Sighted people have the benefit of graphical user interfaces (GUI) to help them, but none of them are accessible by screen reading software.

Initially, the package was built on top of functionality I wrote to support my own work practices, but additions to meet the needs of other blind R users have been made on request.

You will need to install the *BrailleR* package before continuing on with the examples in this chapter. It is available from CRAN. Use the examples in Chapter 14 to help install this package. Then use the `library()` command to make sure the package is ready for action.

```
> library(BrailleR)

The BrailleR.View,  option is set to FALSE.
```

```
Attaching package:  'BrailleR'
The following objects are masked from 'package:graphics':
       boxplot, hist
The following object is masked from 'package:utils':
       history
```

This chapter was created using version 0.25.6 of the *BrailleR* package. This version of the package has a number of other packages that it depends on for functionality. When you make the *BrailleR* package available, you may see a few introduction messages that tell you about the other packages being loaded. You will be told about some default settings for the package that can be altered later to suit your needs.

15.1 Creating a copy of the R console window

In Chapter 2 I mentioned the need to save the content of the console window to a text file to be able to copy and paste its content to a report or document. One major feature of the BrailleR package is that this functionality can be done easily and efficiently.

There are three types of content printed in the console window.

1. The commands we type,
2. The output valid commands generate, and
3. Any error or warning messages that arise from our commands.

The `sink()` command does not currently record the commands or the errors and warning messages. It is therefore a valuable tool for those R users that are confident about the validity of their scripts.

Saving the commands typed into an R session can be achieved using the `savehistory()` command. For example

```
> savehistory("WhatIDidToday.txt")
```

will save all the issued commands into a text file called `WhatIDidToday.txt` in the working directory.

If I just want to have everything I do during a session recorded for posterity, I want all contents of the console window to be saved. A plain text file is sufficient but it is possible to use add-on packages to create other file formats.

I have included a convenience function in the *BrailleR* package that starts a text file with the date and time in its filename. It gets saved in the current working directory. To get this file started, just type

```
> txtOut()
```

In fact this command starts several files, one for the complete console and one for the issued commands. The command actually runs the `txtStart()` command but specifies particular options for the user's convenience. If you are happy to use the defalt filename then you just need to hit <enter>, but if you want to specify the filename, you type the name, and then hit <Enter> twice (once for your file and once to get out of the filename requesting prompt).

See the help page for the `txtOut()` command as it includes the history of the command and its relatives. The other related commands add comments to the text file or stop the processing entirely. Also note that the help page mentions how to get the packages needed for other file formats.

You can open the text file that is being created and modified as you work in any browser. Once you have the file open and have done some work, you must refresh your browser to see the latest changes. Keep the browser open as you work and remember to refresh often to keep on top of what is happening with complete access via your screen reader.

15.2 Text interpretation of graphs

Statistics is an inherently visual discipline. Many statistical analyses lead to a graphic representation of data that is easy for the sighted world to interpret. Through use of R we can create functions that can interpret the graphs using text descriptions. We'll start with a histogram.

In most circumstances, the user will create a histogram using the `hist()` command. This command takes data and creates a number of values needed to construct the histogram. We can see what has been created by storing these values in an object. For example:

```
> MyHist = hist(airquality$Wind, xlab="Average wind speed (mph)", main="", plot=FALSE)

Warning in hist.default(x = airquality$Wind, xlab = "Average wind speed (mph)",
:  arguments 'main', 'xlab' are not made use of
```

The `plot` argument stops the graph being plotted. We can get the graph anytime by issuing the `plot()` command but we are only interested in seeing what R has stored in the `MyHist` object at the moment.

```
> MyHist

$breaks
 [1]  0  2  4  6  8 10 12 14 16 18 20 22

$counts
 [1]  1  4 11 38 27 38 10 17  4  1  2

$density
 [1] 0.003268 0.013072 0.035948 0.124183 0.088235 0.124183
 [7] 0.032680 0.055556 0.013072 0.003268 0.006536

$mids
 [1]  1  3  5  7  9 11 13 15 17 19 21

$xname
[1] "airquality$Wind"

$equidist
[1] TRUE

$main
[1] ""

$xlab
```

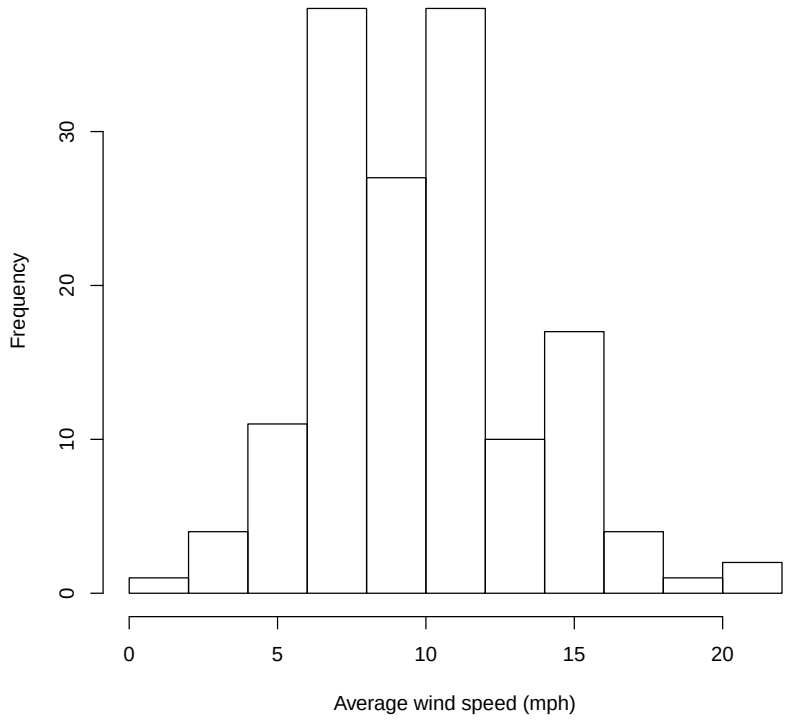
```
[1] "Average wind speed (mph)"

$ylab
[1] "Frequency"

$NBars
[1] 11

$xaxp
[1] 0 1 5

$yaxp
[1] 0 1 5

$xTicks
[1] 0.0 0.2 0.4 0.6 0.8 1.0

$yTicks
[1] 0.0 0.2 0.4 0.6 0.8 1.0

$sub
[1] ""

attr(,"class")
[1] "Augmented" "histogram"
> str(MyHist)
List of 15
 $ breaks  : num [1:12] 0 2 4 6 8 10 12 14 16 18 ...
 $ counts  : int [1:11] 1 4 11 38 27 38 10 17 4 1 ...
 $ density : num [1:11] 0.00327 0.01307 0.03595 0.12418 0.08824 ...
 $ mids    : num [1:11] 1 3 5 7 9 11 13 15 17 19 ...
 $ xname   : chr "airquality$Wind"
```

```
 $ equidist: logi TRUE
 $ main    : chr ""
 $ xlab    : chr "Average wind speed (mph)"
 $ ylab    : chr "Frequency"
 $ NBars   : int 11
 $ xaxp    : num [1:3] 0 1 5
 $ yaxp    : num [1:3] 0 1 5
 $ xTicks  : num [1:6] 0 0.2 0.4 0.6 0.8 1
 $ yTicks  : num [1:6] 0 0.2 0.4 0.6 0.8 1
 $ sub     : chr ""
 - attr(*, "class")= chr [1:2] "Augmented" "histogram"

> class(MyHist)

[1] "Augmented" "histogram"
```

The full print out has everything we need but is not in a friendly format. The `str()` command gives us a condensed version of the data being stored but isn't sufficient, and the `class()` command just tells us that this information is stored in an object of type *histogram*. The class of the object is the most important as it is what tells R how to work on the object when we use the `plot()` command on the `MyHist` object. The `plot()` command is actually a family of commands, one of which is purposely designed for histograms. The `plot.histogram()` command is actually what constructs the graph when the user calls the `plot()` command on an object of type *histogram*. This family of commands is known as a *method*.

The *BrailleR* package includes a *method* for creating text interpretations in a similar way. The `VI()` command is actually a family of commands that includes the `VI.histogram()` command. When we issue the `VI()` command on the `MyHist` object, the `VI.histogram()` function does the work. Let's see what we get:

```
> VI(hist(airquality$Wind, xlab="Average wind speed (mph)", main=""))
```

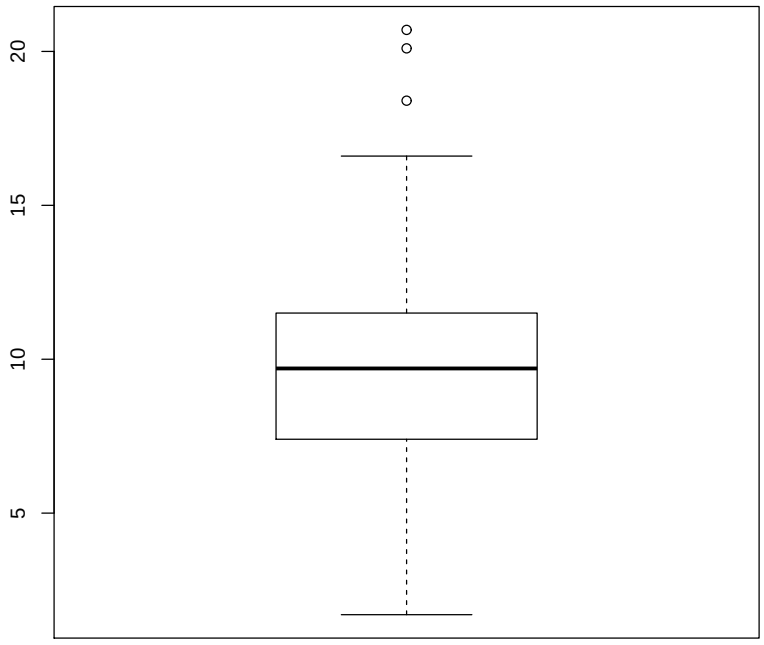



```
This is a histogram, with the title:
"Average wind speed (mph)" is marked on the x-axis.
Tick marks for the x-axis are at: 0, 5, 10, 15, and 20
There are a total of 153 elements for this variable.
Tick marks for the y-axis are at: 0, 10, 20, and 30
It has 11 bins with equal widths, starting at 0 and ending at 22 .
The mids and counts for the bins are:
mid = 1  count = 1
mid = 3  count = 4
mid = 5  count = 11
mid = 7  count = 38
mid = 9  count = 27
mid = 11  count = 38
```

```
mid = 13  count = 10
mid = 15  count = 17
mid = 17  count = 4
mid = 19  count = 1
mid = 21  count = 2
```

Other functions exist for boxplots and dotplots. Try:

```
> VI(boxplot(airquality$Wind, xlab="Average wind speed (mph)"))
```




```
This graph has a boxplot printed vertically
but has no title
"Average wind speed (mph)" appears on the x-axis.
"" appears on the y-axis.
Tick marks for the y-axis are at: 5, 10, 15, and 20
This variable has 153 values.
```

```
An outlier is marked at: 20.1 18.4 20.7
The whiskers extend to 1.7 and 16.6 from the ends of the box,
which are at 7.4 and 11.5
The median, 9.7 is 56 % from the lower end of the box to the upper end.
The upper whisker is 0.89 times the length of the lower whisker.
```

and

```
> VI(dotplot(x))

Error in dotplot(x):  object 'x' not found
```

The `dotplot()` command is not part of base R which uses the `stripchart()` command instead. The `hist()` and `boxplot()` commands used here are actually commands from the *BrailleR* package that use the standard commands indirectly. At this point in time, I haven't worked out how to get the `stripchart()` command to work as a *BrailleR* command so I needed to use a different name for my version. The *BrailleR* versions of these three commands store the additional information we need to get the text interpretation that is not possible from the standard version of the commands.

15.3 Basic descriptions of variables

The `UniDesc()` function is designed to quickly generate a text summary of a single variable. It also saves several graphs. It can also save the text in a file and use the `VI()` command from the *BrailleR* package to add text descriptions to the output file.

One obvious issue is that saving lots of files could become problematic. Default filenames are used and all files can be put into a folder of the user's choosing.

An example will help. Try

```
> data(airquality)
> Ozone=airquality$Ozone
> UniDesc(Ozone, Folder="AirQuality")
```

Take a look in the `AirQuality` folder that has been created in your working directory. It has a number of files that all start with "Ozone" — the name of the variable being

analysed. You will also have three files stored in the current working directory which have the filename `Ozone-UniDesc.*` where the star is for the extension which is one of `html`, `R`, and `Rmd`.

The most useful one at first is the `Ozone-UniDesc.html` file. Open this file in a browser to see what analyses have been included. This file is specifically formatted for use with screen readers. I hope you find it valuable. If you are reading this document on the web, then the link `Ozone-UniDesc.html` will open it in your browser. The link will not work if you are using the `pdf` version of LURN.

The commands used to generate parts of this HTML document are given in the R script file; the third file is called an R markdown document. This is the file that gets converted into the other two files. It might prove useful to learn how to write this kind of file for yourself one day.

In the `airquality` subfolder you will find that graphs have been saved in a variety of formats. They each have their uses and hopefully the filetype you need is there. Special attention is made for those users of LaTeX who need specific formats for graphics. Some more advanced information is presented in LaTeX formatted tables and put into files with the `tex` extension. For example, the content of Table 15.1 were presented in the `Ozone-UniDesc.html` document and included in this document using the LaTeXversion of that table.

	Statistic	P Value
Shapiro-Wilk	0.8787	0.0000
Anderson-Darling	4.5211	0.0000
Cramer-von Mises	0.8033	0.0000
Lilliefors (Kolmogorov-Smirnov)	0.1480	0.0000
Pearson chi-square	73.7241	0.0000
Shapiro-Francia	0.8786	0.0000

Table 15.1: Tests for normality: Variable is Ozone.

Whenever the `VI()` function is employed on an object that does not yet have an explicit function tied to it, then you will be told that this is the case. This package is a work in progress after all. Please feel free to send an email if you'd like any extra functionality.

15.4 Altering R output to make it easier to read

Some R output is arranged in nice tables that are easy for the sighted user to read, but are difficult to understand if using synthesised speech to read the information back to you. The `VI()` command is used again here but is now employed on a *data.frame* object. We'll compare it with the `summary()` command used frequently.

```
> summary(airquality)
     Ozone           Solar.R         Wind
 Min.   :  1.0   Min.   :  7   Min.   : 1.70
 1st Qu.: 18.0   1st Qu.:116   1st Qu.: 7.40
 Median : 31.5   Median :205   Median : 9.70
 Mean   : 42.1   Mean   :186   Mean   : 9.96
 3rd Qu.: 63.2   3rd Qu.:259   3rd Qu.:11.50
 Max.   :168.0   Max.   :334   Max.   :20.70
 NA's   :37      NA's   :7
      Temp           Month            Day
 Min.   :56.0   Min.   :5.00   Min.   : 1.0
 1st Qu.:72.0   1st Qu.:6.00   1st Qu.: 8.0
 Median :79.0   Median :7.00   Median :16.0
 Mean   :77.9   Mean   :6.99   Mean   :15.8
 3rd Qu.:85.0   3rd Qu.:8.00   3rd Qu.:23.0
 Max.   :97.0   Max.   :9.00   Max.   :31.0


> VI(airquality)


The summary of each variable is
Ozone: Min. 1   1st Qu. 18   Median 31.5   Mean 42.1293103448276   3rd Qu. 63.25   Max.
Solar.R: Min. 7   1st Qu. 115.75   Median 205   Mean 185.931506849315   3rd Qu. 258.75
Wind: Min. 1.7   1st Qu. 7.4   Median 9.7   Mean 9.95751633986928   3rd Qu. 11.5   Max.
Temp: Min. 56   1st Qu. 72   Median 79   Mean 77.8823529411765   3rd Qu. 85   Max. 97
Month: Min. 5   1st Qu. 6   Median 7   Mean 6.99346405228758   3rd Qu. 8   Max. 9
Day: Min. 1   1st Qu. 8   Median 16   Mean 15.8039215686275   3rd Qu. 23   Max. 31
```

The output generated by the `summary()` command is difficult to follow as variables are represented in columns while rows that are read aloud by a screen reader are for the sample statistics.

The `VI()` method has given us a summary of the variables one by one. This should prove easier to navigate and interpret as either a braille or screen reader user.

15.5 Reading a scatter plot

One challenge facing blind people is the inability to glance at a graph and understand what is intended just as sighted people can. The most difficult graphs to deal with are multidimensional representations of data. Two dimensional scatter plots are the first example of such graphs.

When a sighted person looks at a scatter plot, they are looking for a number of things. A relationship between the variables is the most common thing to look for, but there is also a need to identify points that are unusual in the context of the data presented.

One possible way to describe how the data points are summarised at a glance is that the graphing window is broken up into areas and a rough guess at the density of points within each region is evaluated. If this evaluation of density can be guessed, it can be counted more exactly by a purposely written function. The `WhereXY()` function does this counting for us. It assumes equal-sized rectangular regions in the graph window. We can choose how many regions but the illustrations here all use a 5×5 grid.

Let's work with a number of examples, all based on a set of predetermined x-values and various possible y-values.

```
> x=(0:100)/100
> y1=5*x+rnorm(101)
> y2=(2*x-1)^2+rnorm(101)/6
> y3=11*x+rnorm(101)
> y4=y3; y4[100]=0
```

What we might guess from the above is that a graph of (x, y_1) pairs would show quite a lot of noise; (x, y_2) pairs are a parabola with some noise; (x, y_3) pairs are for a straight line with some noise; and, (x, y_4) pairs are the same as the previous straight line, but one point is some distance from the line.

The `WhereXY()` command for each plot is now given.

```
> WhereXY(x,y2, grid=c(5,5))

     1  2  3  4  5 Sum
5    6  0  0  0  3   9
4    3  1  0  0 12  16
3   11  6  0  6  5  28
2    1 11 13 12  0  37
1    0  2  7  2  0  11
Sum 21 20 20 20 20 101

> WhereXY(x,y2, grid=c(5,5))

     1  2  3  4  5 Sum
5    6  0  0  0  3   9
4    3  1  0  0 12  16
3   11  6  0  6  5  28
2    1 11 13 12  0  37
1    0  2  7  2  0  11
Sum 21 20 20 20 20 101

> WhereXY(x,y3, grid=c(5,5))

     1  2  3  4  5 Sum
5    0  0  0  3 12  15
4    0  0  4 13  8  25
3    0  7 15  4  0  26
2   14 12  1  0  0  27
1    7  1  0  0  0   8
Sum 21 20 20 20 20 101

> WhereXY(x,y4, grid=c(5,5))

     1  2  3  4  5 Sum
5    0  0  0  3 11  14
```

```
4    0  0  4 13  8  25
3    0  7 15  4  0  26
2   14 12  1  0  0  27
1    7  1  0  0  1   9
Sum 21 20 20 20 20 101
```

We might also alter the number of cells in the grid to see how sensitive we want to make our investigation. Personal preferences will determine which combination of rows and columns works best.

15.6 What else?

Future developments in the BrailleR package will be made to meet demand. If you do want some additional functionality that you think is likely to be wanted by others then please do add to the wish list. Current ideas on the wish list include:

- Easier to interpret multiple regression output.
- Easier to interpret principal component analysis output.
- More graph types being described using the `VI()` command.
- Easier to interpret factor analysis output.

Chapter 16

LURN... To create maps

This chapter explains how to create simple maps using some add-on packages. You will need to have installed the *maps*, *mapdata*, and *ggmap* packages to complete the examples in this chapter. If you have not installed an add-on package before, you should take a look at Chapter 14 first.

After installing the packages, load them into your R session using:

```
> library(maps)
> library(mapdata)
> library(ggmap)

Loading required package:  ggplot2
```

16.1 Creation of maps

The creation of maps is actually a simple task in R, as long as the database for the desired land mass is available. There are a number of these data bases available as downloadable packages from CRAN.

I first created a map when I wanted a map of New Zealand (my home country) for a presentation I gave when I was overseas. I also wanted to show a portion of the country to highlight how far my home town is from the nation's capital. The rectangle that appears on the map of New Zealand was created using the `polygon()` command. The `box()` command

places the frame around the map which is created using the `map()` command. Exhibit 16.1 shows the first map, while Exhibit 16.2 shows the central part of the country.

The maps in these graphs are based on latitude and longitude data. The `map()` function ensures the aspect ratio of the resulting plot is in keeping with the actual view you would have from space if you were directly above the centre of the map being plotted. Note that this rule doesn't apply for the world map options that exist.

16.2 Adding cities to a map

The `map.cities()` function extracts data from a database of the world's cities. Judicious selection of the criteria for population was required to get Palmerston North marked, and the `capitals` argument was used to get the point for Wellington. I have used some colours to distinguish the different cities for my plot.

One challenge in making this graphic presentable for this document was assuring myself that the aspect ratio was correct. Latitudes and longitudes do not take up equal space especially as you move further from the tropics. R will get the aspect ratio correct but being sure is another matter.

The graph I first created for this example was 5.7cm high and just over 10cm wide. Was it in proportion though? If each degree of latitude in the map used 1cm of space, we would expect each degree of longitude to take up at most 1 cm of space. An approximation for this is found by multiplying the latitude space by $\cos(\theta)$ where θ is the latitude. So each degree of longitude in my plot should take up about 0.76 of the space of each degree of latitude. I've plotted three degrees of latitude and seven degrees of longitude, so I expect my 5.7cm high graphic to be

$$\frac{5.7}{3} \times 7 \times 0.76 == 10.1$$

centimetres in width. This was about right.

You can see the way we would do this calculation using R in Chapter 17.

Exhibit 16.1 Map of New Zealand with a rectangle marked on it that bounds the area used in a subsequent map.

```
> map("nzHires", ylim=c(-48,-34), xlim=c(166,179), fill=T, col=3)
> box()
> polygon(x=c(172, 179, 179, 172), y=c(-42,-42, -39,-39), border=1, lty=2)
```

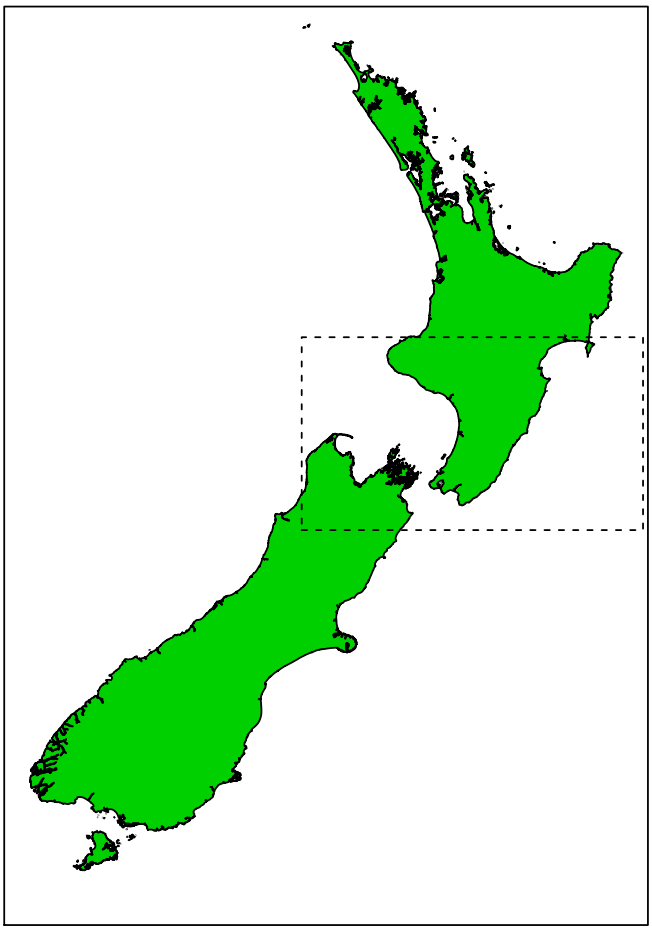
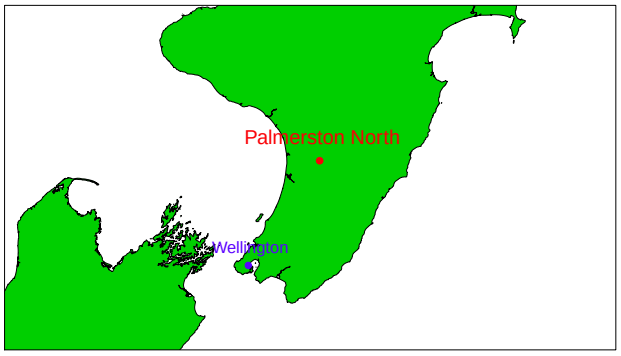


Exhibit 16.2 Map showing the central part of New Zealand in order to show two cities in the lower North Island.

```
> map("nzHires", ylim=c(-42,-39), xlim=c(172,179), fill=T, col=3)
> box()
> map.cities(minpop=70000,maxpop=80000, pch=16, col=2, cex=2)
> map.cities(capitals=1, col=4, pch=19)
```

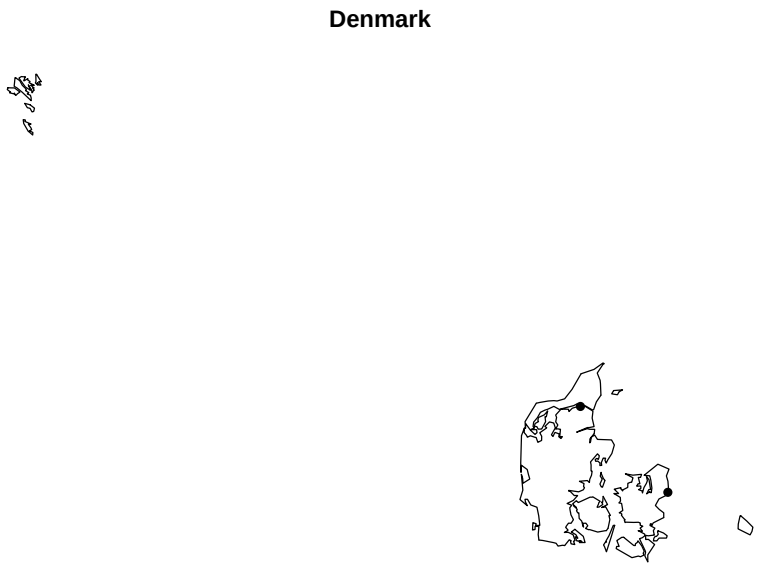


16.3 Using Google's services to map locations of interest

The ability to search Google for specific locations is really useful, especially if you intend plotting maps with points of interest such as universities rather than just cities. Our first example does look for cities though, so that it builds on examples from previous sections in this chapter.

In mid-2015, I will attend the UseR conference in Aalborg, which is in Denmark. I know Denmark has a coastline and is bordered by Germany, but actually I don't know much else (especially where individual cities are!). I want ot know where the city of Aalborg is in relation to Copenhagen (the capital of Denmark). Witht he help of the *ggmap* package tapping into Google maps, I can find the latitute and longitude for both cities and then plot them on the map of Denmark as follows:

```
> Places = geocode(location=c("Aalborg", "Copenhagen"))

Information from URL : http://maps.googleapis.com/maps/api/geocode/json?address=Aalbor
Information from URL : http://maps.googleapis.com/maps/api/geocode/json?address=Copenh

> Places

     lon   lat
1  9.922 57.05
2 12.568 55.68

> map('worldHires', 'Denmark')
> title('Denmark')
> points(Places, pch=16)
```



Chapter 17

LURN... To Use R as a Scientific Calculator

This chapter explains how to use R as a scientific calculator, including its use of trigonometric functions. We'll also plot some of the functions for illustrative purposes.

17.1 Trigonometric functions

R has a series of commands that all perform quite similarly. The only real issue is to find out what actual commands get what we want. First of all, we need to know how R will work with degrees or radians.

```
> pi

[1] 3.142

> sin(pi/6)

[1] 0.5

> sin(30)

[1] -0.988
```

So our experiment shows us that R is working in radians.

The functions `sin()`, `cos()`, and `tan()` give the sine, cosine, and tangent respectively. To find the inverse sine, sometimes called the arcsine, we use the `asin()` command. The hyperbolic functions associated with these functions also exist; use `sinh()`, `cosh()` and `tanh()` and for the inverse, just add an extra `a` before the command named here.

17.2 Graphs of trigonometric functions

Chapter 18

LURN... To Use R for Basic Calculus Tasks

This chapter explains how various tasks in introductory calculus can be undertaken using R.

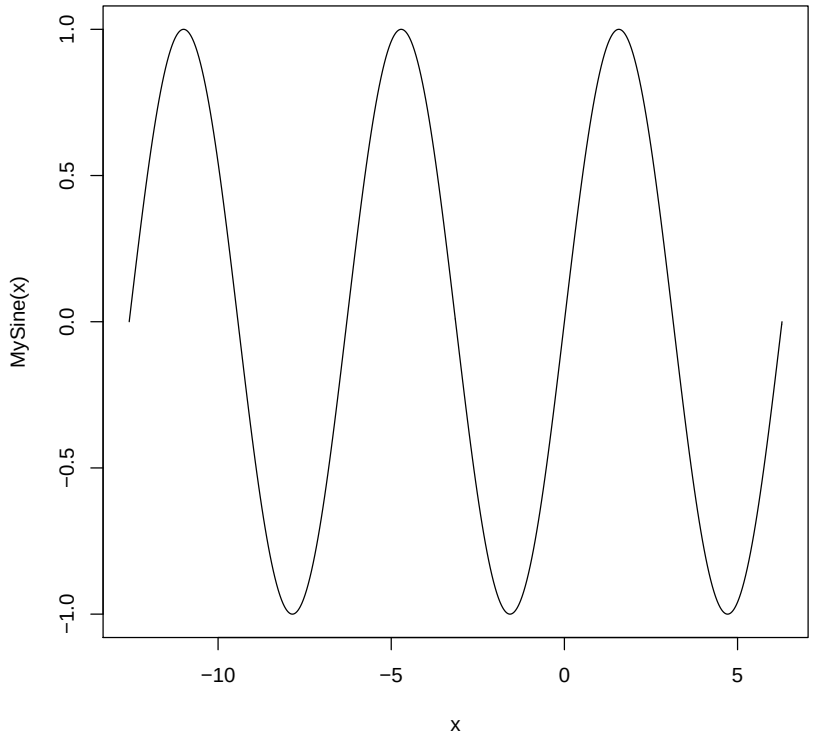
18.1 Creating mathematical expressions

Before we embark on doing those fun things we want to do, we need to learn how to store a mathematical expression as an object in R. There are many keywords that appear in these expressions which are translated to symbols when seen printed in a graph window but remain in ordinary text in R commands. Such symbols include Greek letters, hats and bars, all of which have a meaning.

18.2 Plotting functions

This can be a lot of fun if you set things up right to start with. We create a *function* to represent the mathematical expression using the `function()` command, then plot it using either the `plot.function()` or `curve()` commands.

```
> MySine <- function(x){ sin(x)}
> curve(MySine, -4*pi, 2*pi, n = 2001)
```



You can use any valid mathematical expression in the *function* you create. You might find it useful to test your idea= values for your inputs that are easy to work with, such as zero and one. You could then use integers, including negative integers, but it could be easier to test your functions using values that lead to identifiable results such as multiples of π when working with trignometric functions. For example: «EvalMySine» MySine(c((0:4)*pi/2)) @

18.3 Differentiation

The internal code used for differentiation in R is not as comprehensive as other mathematical software. It can handle arithmetic operators $+$, $-$, $*$, / and ^, and the single-variable functions `exp()`, `log()` (natural logarithm), `sin()`, `cos()`, `tan()`, `sinh()`, `cosh()`, `sqrt()`, `pnorm()`, `dnorm()`, `asin()` (inverse sine), `acos()` (inverse cosine), `atan()` (inverse tangent), and various `gamma()` functions. The functions `pnorm()` and `dnorm()` are restricted to the standard normal distribution.

The easiest way to find the derivative of an expression is to store the mathematical function using the `expression()` command, and then use the `D()` or `deriv()` commands.

```
> y=expression(exp(2*x))
> D(y,"x")

exp(2 * x) * 2
```

The second argument to the `D()` function needs quote marks around it.

Chapter 19

LURN... To Use R for Linear Algebra

This chapter explains how some tasks commonly taught in introductory algebra courses can be undertaken using R. The functionality is built into R because so many tasks in statistical analyses require manipulation of vectors and matrices, although much of this work is hidden behind other commands and functions.

19.1 Basic notes on storage of vectors and matrices

We will concern ourselves with only numeric valued vectors and matrices in this chapter. Other types of data can be stored in vectors and matrices (as well as other data structures) but they have little relevance for linear algebra at this time.

In linear algebra, a vector is a one dimensional set of numbers. It is usually referred to as a row vector or a column vector to indicate its orientation. This orientation does matter for the ability to perform certain tasks when working with two or more vectors and matrices.

A matrix is a two-dimensional array of numbers, having a number of rows and columns. Many mathematical software applications do not explicitly distinguish a vector from a matrix because they store a row vector as if it were a matrix having just one row. Similarly, a column vector is stored as a matrix having just one column.

R does distinguish betweena vector and a matrix, but it does not state whether a vector is a row or column vector. This does have its advantages, and probably some disadvantages

as well. If for any reason you need to force the orientation of a vector, then create it as a matrix with one of its dimensions set equal to one. For example, we create a simple vector:

```
> x=1:4
> x

[1] 1 2 3 4

> class(x)

[1] "integer"
```

We can convert this to a column matrix using

```
> X=matrix(x, ncol=1)
> class(X)

[1] "matrix"

> X

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

Notice that I've used a capital letter for the matrix version of the same set of numbers here which is possible because R is case sensitive. The `ncol` argument can be replaced by using the `nrow` argument if a row vector is required.

19.2 Creating some simple matrix structures

The *identity matrix* is a diagonal matrix so we use the `diag()` command to create it. For example

```
> I3=diag(3)
> I3

     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

We often need to find the transpose of a matrix. The `t()` command does this.

```
> M=matrix(1:6,nrow=2)
> M

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

> t(M)

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

A *Hilbert matrix* can be created using a formula. The `row()` and `col()` commands use the size of the argument to create objects of the same size.

```
> row(I3)

     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    2    2    2
[3,]    3    3    3

> Hilb3=row(I3)+col(I3)-1
```

Notice that the Hilbert matrix is created by adding two matrices together and then a constant is subtracted from all elements of the result.

19.3 Matrix and vector calculations

We saw a simple addition of elements of two matrices of the same size as well as a subtraction in the previous section. R uses the standard arithmetic operators on pairs of matrices (or vectors) on an element-by-element basis.

Addition of a constant to a vector or matrix is effected on all elements

```
> x+2

[1] 3 4 5 6

> 2+x

[1] 3 4 5 6
```

Similarly, we can multiply a vector or matrix by a scalar using:

```
> 5*x

[1]  5 10 15 20

> x*5

[1]  5 10 15 20

> 5*X

     [,1]
[1,]    5
[2,]   10
[3,]   15
[4,]   20
```

which retains the size of the vector or matrix. Also note that pre-multiplying by a scalar is equivalent to post-multiplying by a scalar, as should have been expected. This is of course different to matrix multiplication.

Matrix operations use different symbols from simple arithmetic operators to distinguish the different results that might be desired. For example,

```
> A=matrix(1:4, nrow=2)
> B=matrix(7:10, nrow=2)
> A*B

     [,1] [,2]
[1,]    7   27
[2,]   16   40
```

creates a matrix with elements based on simple multiplication of the paired elements, not those based on proper matrix multiplication.

If we want to multiply two matrices, the order is important.

```
> A%*%B

     [,1] [,2]
[1,]   31   39
[2,]   46   58

> B%*%A

     [,1] [,2]
[1,]   25   57
[2,]   28   64
```

Multiplying a matrix by a vector is easily achieved, and R will orient the vector to conform to the rules of matrix multiplication as required.

```
> y=c(3,4)
> A%*%y
```

```
     [,1]
[1,]   15
[2,]   22

> y%*%A

     [,1] [,2]
[1,]   11   25
```

19.4 Inverting a matrix

The inverse of a matrix is used in many statistical procedures. The `solve()` command inverts a square matrix.

```
> A

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

> solve(A)

     [,1] [,2]
[1,]   -2  1.5
[2,]    1 -0.5
```

For matrices that are not square, a generalised inverse can be found using the `ginv()` command found in the *MASS* package that is part of the standard R distribution.

```
> M

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

```
> require(MASS)
> ginv(M)

        [,1]    [,2]
[1,] -1.3333  1.0833
[2,] -0.3333  0.3333
[3,]  0.6667 -0.4167
```

Of course, the product of the matrix and its generalised inverse should be an identity matrix.

```
> M%*%ginv(M)

          [,1]      [,2]
[1,] 1.000e+00 -2.22e-16
[2,] 6.661e-16  1.00e+00
```

Like many other applications, R does not recognize the number zero very well. This is due to the way decimal numbers are stored.

19.5 Solving a set of linear equations

If we want to solve the set of equations, represented in matrix form as $Ax = b$, we use the `solve()` command

```
> A

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

> b=c(5,7)
> solve(A,b)

[1] 0.5 1.5
```

19.6 Determinants and traces

The `det()` command is used for finding the determinant of a matrix.

```
> A

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

> det(A)

[1] -2
```

The trace of a matrix is not so easily found. There is a command that is called `trace()` but this is not an algebraic function. If you want the trace of the matrix, you will need to sum the eigenvalues of the matrix.

19.7 Eigenvalues and eigenvectors

Calculation of eigenvalues and their associated eigenvalues is fairly simple, but extracting the elements required might pose a little difficulty. Let's create a 4×4 Hilbert matrix and find its eigenvalues:

```
> H4=matrix(-1,nrow=4, ncol=4)
> H4=H4+row(H4)+col(H4)
> H4

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

> eigen(H4)
```

```
eigen() decomposition
$values
[1]  1.717e+01  4.441e-16 -3.522e-16 -1.165e+00

$vectors
        [,1]    [,2]    [,3]     [,4]
[1,] -0.3147 -0.5477  0.0000  0.77521
[2,] -0.4275  0.7303  0.4082  0.34244
[3,] -0.5402  0.1826 -0.8165 -0.09032
[4,] -0.6530 -0.3651  0.4082 -0.52309
```

If we only wanted the eigenvalues, we could add the argument `only.values=TRUE`; if the eigenvectors were of interest, then we would need to extract the second element of the above result:

```
> eigen(H4)$vectors

        [,1]    [,2]    [,3]     [,4]
[1,] -0.3147 -0.5477  0.0000  0.77521
[2,] -0.4275  0.7303  0.4082  0.34244
[3,] -0.5402  0.1826 -0.8165 -0.09032
[4,] -0.6530 -0.3651  0.4082 -0.52309
```

Index