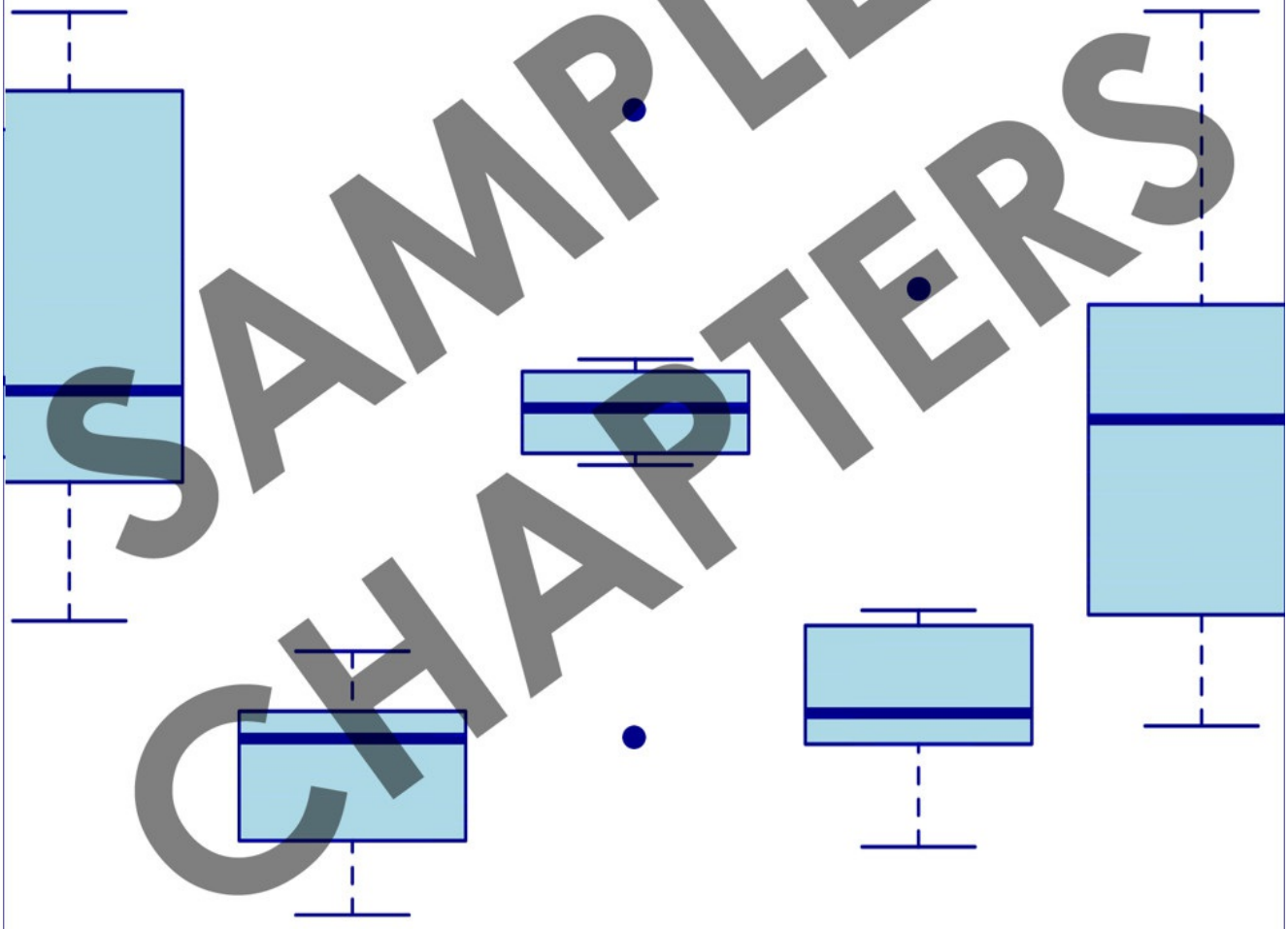
Introductory R

Robert J Knell

Fully revised and expanded version



A beginner's guide to data visualisation, statistical analysis and programming in R

Introductory R

A beginner's guide to data visualisation and statistical analysis and programming in R

Rob Knell School of Biological and Chemical Sciences Queen Mary University of London

Published by Robert Knell, Hersham, Walton on Thames United Kingdom KT12 5RH

First printed March 2013

ISBN 978-0-9575971-1-2

SAMPLE

Preface

I've been using R to analyse my data for about twelve years now, and about eight

years ago I started running an informal course for postgraduates, postdocs and on occasions academics at Queen Mary, University of London to teach them how to use it. More recently we have started trying to teach R to our undergraduates, with more success than I think we'd originally expected. This book has slowly arisen from a set of course notes that I originally wrote for the postgraduate course, and has since been field-tested on the undergraduate course. I've decided to self-publish it as an e-book for several reasons. Firstly, it's an interesting experiment. Secondly, it keeps the price down to roughly 20% of the cost of a book published in the traditional way. Thirdly, it leaves me with complete control over the content of the book, where it's available and what's included in it.

This book is written from the perspective of a biologist, and the examples given are mostly either biological or medical ones. The basic principles of the analyses presented are universal, however, and I've made an effort to explain the examples in such a way that I hope most people will understand them. Nonetheless, if you're more used to data from other fields then you might have some difficulty following some of the examples, in which case I'm sorry. Please drop me a line (r.knell@qmul.ac.uk) and let me know if you think one or more of the examples used are particularly opaque and I'll try to take your feedback into account.

This book was written with Sublime Text using R markdown . Conversion to html was done in R using knitr and final html editing and formatting was done in Sigil. For the kindle version .mobi conversion was done using Calibre. My thanks to those who have worked hard to make these outstanding tools available either for free or for very reasonable prices.

How to use this book

You can read this on anything you can read an ebook on, depending on the exact format of the copy you have bought. Because there are a lot of figures you might find some parts of it don't display particularly well on a small ebook reader, so the best thing to read this on is probably a tablet - an iPad or a reasonably sized Android tablet. These have decent sized high-res screens and the code and figures should all come out properly. If you try to read it on a smaller ebook

reader you won't be able to see the colours and some of the figures won't look too great, but the rest should be OK.

The R code is presented in a monospace font. For formats that aren't read on a Kindle code that is entered into R is coloured blue and presented with a pale blue background and output from R comes with a white background. Kindle formatting can't cope with the background, so input code is blue and output code is either black or blue-black depending on your reader.

```
This is code that is input into R
```

```
This is R output
```

Some of the R output is in fairly big tables or matrices which might be hard to read if you have your font size set too large because the lines will wrap round and make it difficult to read the table. If this happens you'll might like to drop the font size you're using to make the table behave. If you'd rather not then you should be able to make sense of the tables anyway but it might require a little thinking.

The structure of the book is, I hope, fairly self-evident. The first five chapters introduce and explain some of the fundamental concepts of R (e.g. basic use of the console, importing data, record Keeping, graphics).Chapters 6-13 are about statistics and how to analyse data in R. We start with basic concepts and go through a series of analyses from chi-squared tests to general linear model fitting. All of these are illustrated with examples using real data sets and the way to carry out these analyses in R is explained in detail, as is how to interpret the output - the latter is often more complicated than the former. Finally, chapters 14 to 19 are about programming in R, with the last two chapters being two reasonably complex worked examples.

The chapters which are most focussed on how to use R, rather than statistics, have exercises for you to work through yourself. These are Chapter 2, Chapter 5 and Chapters 14-17. The solutions to all of the exercises are in Appendix 4.

The data used for the examples is a mix of data from classic studies, data from recent surveys published by the government, data from my own research and data that’s been extracted from publications or made available publicly by the

authors of recent publications. I'd just like to take this opportunity to point out the fabulous resource that is the Dryad digital repository (http://datadryad.org) and encourage anyone who reads this to upload data from their publications to it. Because I've made an effort to use real data for many of the examples in this book you will see analyses worked out on warts-and-all data, with the kinds of problems and uncertainties that arise when we're dealing with data from real research. Some books will show you examples based on the analysis of made-up data which behaves perfectly. I do this to some extent but I try to avoid the danger of giving a false impression of the realities of data analysis: I think it's important to understand that it's rare to end up with a set of results that behaves nicely and plays properly with your analysis in every way.

If you are reading this on a tablet then remember that you can double-tap on a figure (or one of the output tables imported as a figure) to get a bigger version displayed.

There is a website for the book at http://www.introductoryr.co.uk where you can download datasets and scripts associated with the analyses presented here.

Acknowledgements

I'd just like to thank all the staff and students at QMUL who gave feedback on the various drafts of the manuscript, especially Mark Stevenson who took a lot of time to check through the code for a previous version, Dicky Clymo and Beth Clare for comments on the previous version or on draft chapters for this version, Lucy-Rose Wyatt who helped with the conversion of the previous version to the new format and Richard Nichols for ruthlessly weeding out any wording that could possibly be interpreted as implying that rejecting a null hypothesis might mean that the alternative hypothesis is true. Thanks also to everyone who pointed out to me that "Tilda" can refer to many things including a brand of rice and a railway station in India, but that the ~ symbol is not one of those things.

Chapter 1: A first R session

What is R?

In the beginning, Bell labs in the States developed a statistical analysis package called S. Nowadays this is most widely encountered as the commercially available package S-plus, distributed by Insightful corp. R is an open-source implementation of S, and differs from S-plus largely in its command-line only format. If you're not familiar with the concept, open source software is not produced and sold by commercial bodies in the way that software such as Microsoft Office is: rather, it is usually written collaboratively by teams of volunteers and then distributed for free. It's called open source because the source code is available and anyone can modify it and distribute it themselves if they conform to the requirements of the open source licence. R development is mostly done by the Core Development Team, who are listed here: http://www.r-project.org/contributors. Thanks everyone for all your hard work.

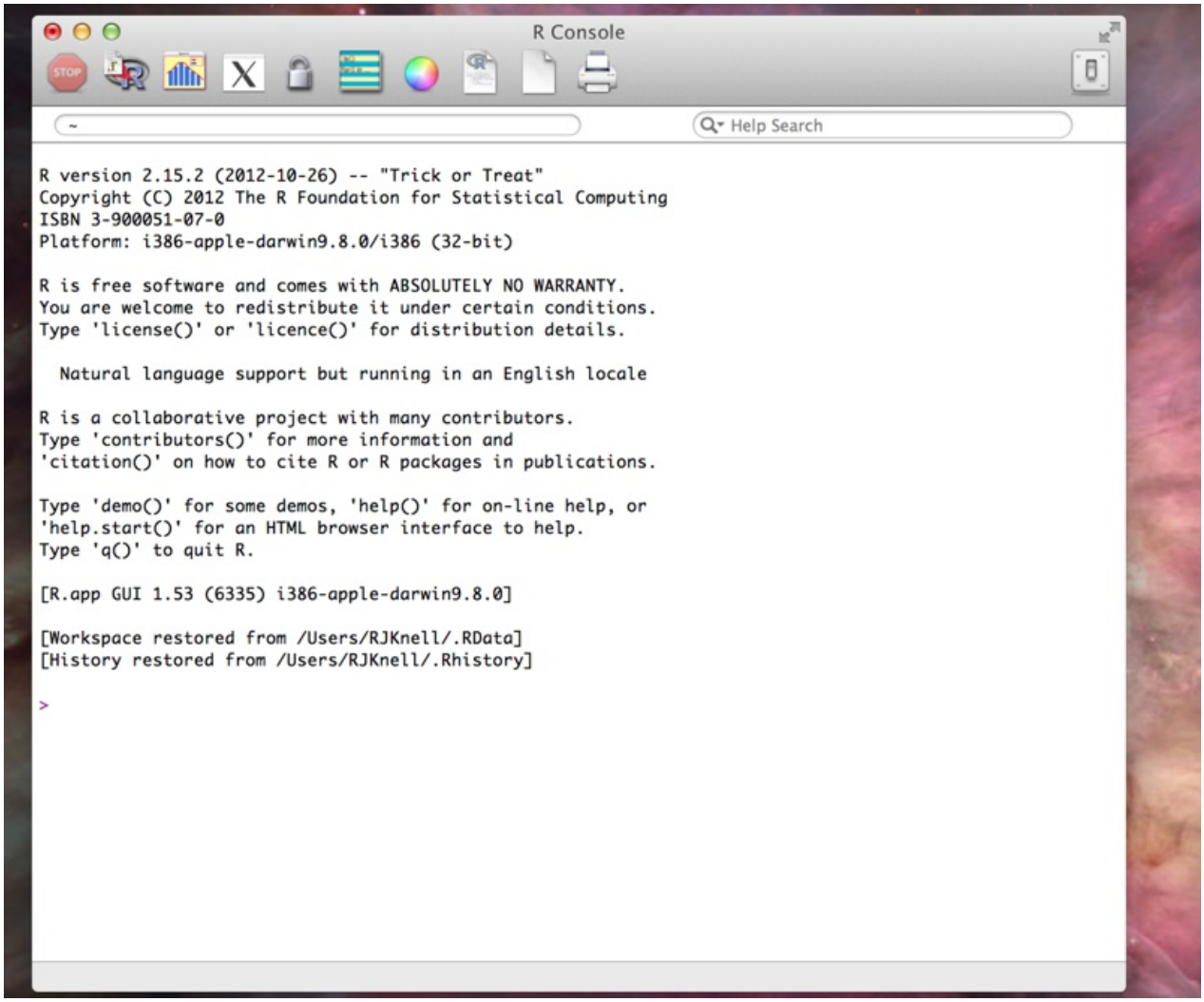
R has a broad set of facilities for doing statistical analyses. In addition, R is also a very powerful statistical programming language. The open-source nature of R means that as new statistical techniques are developed, new packages for R usually become freely available very soon after. There is R code available to do pretty much any sort of analysis that you can think of, and if you do manage to think of a new one you can always write the code yourself. R will carry out important analyses that are difficult or impossible in many other packages, including things like Generalized Linear and Generalized Additive Models, and Linear and Non-Linear Mixed Models. All of these are becoming increasingly widely used in biology, especially at the whole-organism end of things.

On top of the almost infinite variety of statistical tools available, R is also equipped with a broad range of graph-drawing tools, which make it very easy to produce publication-standard graphs of your data. Because R is produced by people who know about data presentation, rather than graphic design, the default options for R graphs are simple, elegant and sensible, contrasting somewhat with the default options for graphics in certain other well-known data manipulation packages. On top of the base graphics system there are a number of additional

graph packages available that give you whole new graphics systems and allow a further variety of graphics to be drawn easily. For a quick look at some of the variety of graphics that can be made in R have a look at some of the examples on show at http://rgraphgallery.blogspot.co.uk.

R is freely downloadable from http://cran.r-project.org/, as is a wide range of documentation. If you're using Windows or a Mac you can download it and run the installer. If you're on Linux then check the software management system: you might have to do a bit of jiggery-pokery like specifying the repository, depending on the specifics of the distro you're using. I'll assume that if you're using Linux then you can work all that out for yourself.

The main R window that you'll see when you start R is called the console. This is what it looks like when you are using a Mac, and on a PC it's similar.



A first R session

You've downloaded a copy of R, you've started the application and now you're faced with the command prompt without any nice buttons or menus to help you. This is the point where things usually start going pear-shaped for a good many people, so I'll talk you through a simple analysis in an attempt to show you that it's not as intimidatingly hard as you might think. Let's say you're doing a research project investigating the effect of spatial density on testosterone concentration in a species of lizard. You have a data set from a single population with measurements from 12 male lizards. For each of these you've recorded the nearest neighbour distance as a measure of the population density experienced by that animal, and you've caught the lizards, taken a blood sample and measured the amount of testosterone. Here are your data.

Lizard Number	Nearest Neighbour (*m*)	Plasma testosterone ($ng.ml^{-1}$)
1	4	22.2
2	7	26.1
3	3	21.5
4	5	23.8
5	8	23.8
6	3	20.5
7	7	24.6
8	4	22.6
9	3	19.9
10	5	23.9
11	5	19.8
12	4	21.3

Table: data for lizard example.

Before doing any sort of analysis we need to get our data loaded into the

programme. For a big data set you'd do this by entering the data into a spreadsheet or text file and importing it to R (see the "Importing data" chapter) but in this case, with a small dataset you can enter the data directly at the command prompt. First, the nearest neighbour data. Just type this in at the command prompt and press enter.

```
nearest.neighbour <- c(4, 7, 3, 5, 8, 3, 7, 4, 3, 5,
5, 4)
```

This has created a set of numbers (a vector) in R called nearest.neighbour. The "c" before the brackets tells R to put everything in the brackets together, and the arrow <- means "take what's to the right of the arrow and make an object with the name that's to the left of the arrow". You can check to make sure it's correct by just typing the name and pressing enter.

```
nearest.neighbour
```

```
 [1] 4 7 3 5 8 3 7 4 3 5 5 4
```

There are your data. Don't worry about the [1] at the start. Now you need to do something similar for the testosterone data.

```
testosterone <- c(22.2, 26.1, 21.5, 23.8, 23.8, 20.5,
24.6, 22.6, 19.9, 23.9, 19.8,
    21.3)
```

You can check your numbers here in the same way as before, by typing in the name of the object and pressing enter. Now your data are entered into R you can take a look at them. It's always a good idea to visualise your data before doing any analysis, and you can do this by asking R to plot the data out for you. Type this and you'll get a scatterplot.

```
plot(nearest.neighbour, testosterone)
```

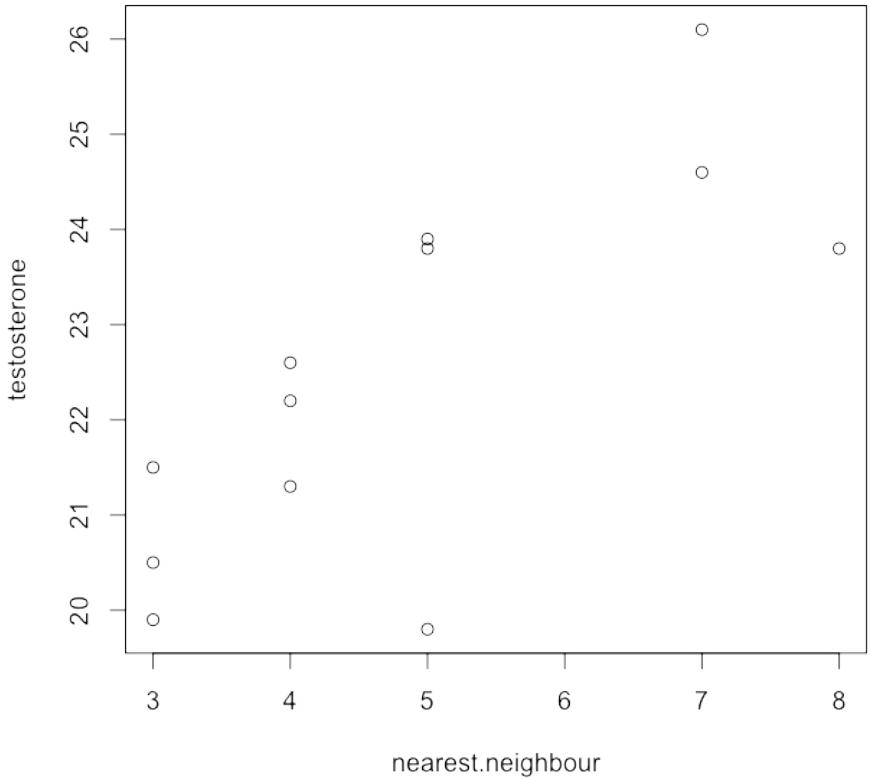


Figure 1: Testosterone titre plotted against the distance to the nearest neighbour

Not quite publication quality but not bad. You can look at the data and see that there does seem to be a positive relationship between nearest neighbour distance and testosterone concentration, and you can also see that there aren't any data points that are huge outliers that are likely to have a big effect on your analysis. Now you can go ahead and do some analysis: you want to know if there's a relationship between nearest neighbour distance and testosterone concentration, so you need to do a correlation analysis. More specifically, you want to know what the correlation coefficient *r* is for the two variables and whether the correlation coefficient is significantly different from zero (see chapter 7 for what statistical significance means, and chapter 10 for more on correlations). Type this in and press enter.

```
cor.test(nearest.neighbour, testosterone)
```

```
    Pearson's product-moment correlation

data:  nearest.neighbour and testosterone
t = 3.624, df = 10, p-value = 0.00466
alternative hypothesis: true correlation is not equal
to 0
95 percent confidence interval:
 0.3163 0.9267
sample estimates:
   cor
0.7535
```

This has everything you need. It tells you exactly what sort of analysis has been done, and the names of the variables you asked it to analyse. Then you get a significance test and a p-value, telling you that there is a statistically significant correlation between the variables. The last thing in the table is the actual estimate of the correlation coefficient (0.75) and you even get 95% confidence intervals for the estimate of r. That's it. You've input your data, drawn a graph and carried out a statistical test. This has illustrated some important points about how R works to you: you make it do things by typing commands and pressing enter; you can create data objects with your results in; there are commands that make R do things like carry out tests and draw graphs; and at least some of these commands have self-evident names like "plot". You've probably also noticed that there seem to be an awful lot of brackets and commas, and you might have found out that if you make mistakes in what you're typing the error messages are rarely helpful. You're halfway there already, but if you want to gain your black belt in command prompt-jitsu you really need to go to the very beginning and start with the basics, which is what the next chapter is all about.

Chapter 2: Basics

This chapter introduces a set of the very fundamental concepts that you need to know to be a confident user of R. It's a bit long and dry in places and if you're only interested in drawing graphs or doing an ANOVA on your project data you might be wondering why you need to worry about using subscripts to select entries in matrices. It's worth persevering because an understanding of the concepts that are explained here will make the later chapters much easier, and when you're doing your own work in R you will have a much better grasp of what's going on. There are a series of exercises that you can work through embedded in the text which might be useful, especially for the novice. Solutions are at the back of the book in Appendix 4.

Commands, objects and functions

Basic commands

You interact with R by typing something into the console at the command prompt and pressing enter. If what you've typed makes sense to R it'll do what it's told to do, if not then you'll get an error message. The simplest things you can get R to do are straightforward sums. Just type in a calculation and press enter.

```
6 + 3
```

```
[1] 9
```

```
3^3
```

```
[1] 27
```

There are several things to notice here. Firstly, you don't need to type an equals sign – just hit enter. Secondly, arithmetic operators are the same as they are in most other computer applications.

+ addition

- subtraction

* multiplication

/ division

^ raised to the power

Thirdly, why does the answer have a [1] in front? Because there's only one number in the answer. R is written to be good at dealing not only with single numbers but with groups of numbers: vectors and matrices. When it gives you an answer that contains more than one number it uses numbering like this to indicate where in the group of numbers each answer is. We'll see more of this as we go on.

When you're asking R to do calculations that are more complex than just a single arithmetic operation then it will carry out the calculations in the following order from left to right: first of all calculations in brackets, then exponents, then multiplication and division, then addition and subtraction. When in doubt use brackets to make sure that the calculation that you want to be done is the one that's actually done.

```
22 * 15 + 6
```

```
[1] 336
```

22 and 15 are multiplied, and then 6 is added to the result of that calculation.

```
22 * (15 + 6)
```

```
[1] 462
```

By adding a pair of brackets we change the order that the calculations are done in: this time 15 and 6 are added together, and 22 is then multiplied by the sum of

the two numbers.

```
22^2 + 7
```

```
[1] 491
```

For this calculation 22 is raised to the power 2 and 7 is then added to the result of that calculation.

```
22^(2 + 7)
```

```
[1] 1.207e+12
```

By adding a pair of brackets we make sure that the first calculation is to add 2 and 7, and 22 is then raised to the power of the sum of those two numbers. The result of this calculation is a number that's too big for R to display normally, so it's using scientific notation: 1.207e+12 is the way that R expresses 1.207×10^{12}, or 1,207,000,000,000.

You can use multiple brackets, and you can nest your brackets *ad infinitum*.

```
(2 * (17.2 + 5))/56
```

```
[1] 0.7929
```

Exercises 1: Calculations

Use R to carry out the following calculations (solutions are at the end of the chapter if you wish to check):

- 3 x 5
- 5^2
- Add 8 to 22 and then multiply the answer by 3

- Subtract 2.8 from 12 and then raise the answer to the power of 1.5 plus 1.3924
- Subtract 2.8 from 12, raise the answer to the power of 1.5 and then add 1.3924
- Divide 8 by 2.5 and then divide the answer by 3
- Subtract 2.5 from 8 and then divide the answer by 3
- Divide 2.5 by 3 and subtract the answer from 8

Objects

These simple calculations don't produce any kind of output that is remembered by R: the answers just appear in the console window but that's all. If you want to do further calculations with the answer to a calculation you need to give it a name and tell R to store it as an object.

```
answer <- 2 + 2
```

Tells R to add 2+2 and store the answer in an object called answer. To find out what is stored in answer, just type the name of the object:

```
answer
```

```
[1] 4
```

Take particular note of the middle symbol in the instruction above, <-. This is the allocation symbol, or the assign symbol, formed of a "less than" arrow and a hyphen <- and it looks like an arrow pointing towards "answer". It means "make the object on the left into the output of the command on the right". Mick Crawley (author of "The R Book" and several others) calls it the "Gets" symbol but I think that sounds silly so I call it the allocation symbol. You can call it what you want.

It also works the other way around: 2+2->answer. You can also use an equals sign for allocation but it's best not to do this because it can cause confusion with other uses of the equals sign.

In some older versions of R, and in S-Plus, the underscore character can also be used for allocation, but this is no longer available in recent releases of R. It's useful to know this so that when you try to use S-Plus code in R you can work out why it doesn't work.

You can use objects in calculations in exactly the same way you have already seen numbers being used above.

```
answer2 <- (3.5 + 1)^2
answer + answer2
```

```
[1] 24.25
```

You can also store the results of a calculation done with objects as another object.

```
answer3 <- answer/answer2
answer3
```

```
[1] 0.1975
```

When you first open R, there are no objects stored, but after you've been using it for a while you might have lots. You can get a list of what's there by using the ls() function (functions are explained in more detail in the next section).

```
ls()
```

```
 [1] "answer"              "answer2"             "answer3"
```

You can remove an object from R's memory by using the rm() function.

```
rm(answer2)
```

Notice that when you type this it doesn't ask you if you're sure, or give you any other sort of warning, nor does it let you know whether it's done as you asked. The object you asked it to remove has just gone: you can confirm this by using ls() again.

```
ls()
```

```
 [1] "answer"              "answer3"
```

It's gone, sure enough. If you try to delete an object that doesn't exist you'll get an error message. It's quite common when using R to be pleased when you type in a command and see nothing happen except for the command prompt popping up again. When nothing seems to happen that means that there haven't been any errors, at least as far as R is concerned, which implies that everything has gone smoothly.

Exercises 2: Objects

- Create an object called "X1" which is the number 73
- Create another object called "X2" which is the answer to the sum 101+36
- Multiply X1 and X2 together and store the object as another object called X3
- Subtract 1 from X3 and calculate the 4th root (NB you can calculate the fourth root of a number by raising it to the power of ¼)
- The answer should be 10

Functions

You can get so far by typing in calculations, but that's obviously not going to be much use for most statistical analyses. Remember that R is not so much a statistical package in the traditional sense, like Minitab, but is really a programming language designed to be good for carrying out statistical analyses. R comes with a huge variety of (mostly) (fairly) short ready-made pieces of code that will do things like manage your data, do more complex mathematical operations on your data, draw graphs and carry out statistical analyses ranging from the simple and straightforward to the eye-wateringly complex. These ready-made pieces of code are called functions. Each function name ends in a pair of brackets, and for many of the more straightforward functions you just type the name of the function and put the name of the object you'd like the procedure carried out on in the brackets.

```
log(27)
```

```
[1] 3.296
```

The natural log of 27

```
exp(3)
```

```
[1] 20.09
```

e raised to the power 3

```
sqrt(225)
```

```
[1] 15
```

Square root of 225

```
abs(-9)
```

```
[1] 9
```

Absolute (i.e. unsigned) value of –9

You can carry out more complex calculations by making the argument of the function (the bit between the brackets) a calculation itself:

```
sin(17 + answer)
```

will return the sine of 17 plus whatever the value of the object “answer” is: in this case 4.

You can use brackets within the function’s brackets to make sure that complicated calculations are carried out in the correct order:

```
exp((x * 2)^(1/3))
```

Will return the value of e raised to the power of whatever the value of x is, multiplied by 2, raised to the power 1/3 (in other words, e to the power of the cube root of 2x).

You can use functions in creating new objects:

```
D <- 1/sqrt(x)
```

creates an object called “D” that has the value of 1 divided by the square root of the value of the object x.

So far we’ve only looked at functions that have a single argument between the brackets. It’s often possible to control specific aspects of the way that the function operates, however, by adding further *arguments*, separated by commas. These extra arguments serve to do things like modify the way that the function is applied, or tell it to only use part of a dataset, or specify how the function should deal with missing datapoints: I could go on but I hope you get the idea. Here’s a straightforward example. The function `round()` rounds off a number to a certain

number of decimal places. You can tell it the number you want to round off by typing it in between the brackets after the function, and then you can tell it how many decimal places to round to by adding a second argument, *digits=*, with a comma separating it from the first argument.

```
round(14.5378, digits = 2)
```

```
[1] 14.54
```

```
round(14.5378, digits = 1)
```

```
[1] 14.5
```

Most R functions have default values specified for most of their arguments, and if nothing is specified the function will just use the default value. If we don't define a number of digits for `round()`, it will assume you want the number rounded off to no decimal places.

```
round(14.5378)
```

```
[1] 15
```

A few other examples.

```
logb(27, base = 3.5)
```

```
[1] 2.631
```

This calculates the logarithm of 27 to the base 3.5. In other words, it calculates the number that 3.5 has to be raised to to get 27.

```
signif(pi, digits = 3)
```

```
[1] 3.14
```

```
signif(pi, digits = 5)
```

```
[1] 3.1415
```

In this case we specified the arguments precisely. Very often in R the function can work out what the arguments mean simply by their position in the function call.

```
signif(pi, 3)
```

```
[1] 3.14
```

Exercises 3: Functions

- Calculate the log to the base 10 of x1 using the function `log10()`
- Calculate the square root of x2 using the function `sqrt()`
- Calculate the square root of x2 by raising it to the power 0.5: your answer should be the same as when you used the `sqrt()` function
- use the function `sum()` to add x1 and x2: you'll need to put both object names inside the brackets and separate them with a comma
- use the `mean()` function to calculate the arithmetic mean (i.e. the average) of x1 and x2
- Create an object called “x4” which is equal to 67.8953
- The function `round()` will round a number to a given number of decimal places. The default is zero so start by using this function to round x4 off to zero decimal places
- Now use the round function to give you the value specified by x4 rounded

off to three decimal places

- `floor()` and `ceiling()` can also be used to trim a number down to an integer: apply both of these functions to x4 and compare the output.

Vectors, Matrices and Data Frames

So far we've looked at individual numbers. When we're analysing experimental data, of course, we are likely to be working with lots of numbers, and R is especially good at dealing with objects that are groups of numbers, or groups of character or logical data. In the case of numbers these groups can be organised as sequences, which are called vectors, or as two dimensional tables of numbers, which are called matrices (singular matrix). R can also deal with tables that have some columns of numbers and some columns with other kinds of data: these are called data frames.

Vectors

It's worth spending a little time looking at vectors and matrices, because a lot of the things you'll be doing in your analyses involve manipulating sequences and tables of numbers. Let's start by making a vector. We're going to do this using a function called `seq()`, which produces sequences of numbers. We could write out the command in full, with names for all the arguments, as `seq(from=1, to=10, by=1)`, but because we know that R knows that the first argument between the brackets corresponds to the *from=* argument, the second one to the *to=* argument, and the default value for *by=* is, we can write a much shorter instruction.

```
X <- seq(1, 10)
```

Which creates a new object called X (a vector in this case) containing a sequence of numbers counting up from 1 to 10. To see what X is just type its name.

```
X
```

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

There are other ways to set up vectors. One of the most important uses the `c()` function.

```
Y <- c(2, 4, 6, 8, 10, 12, 14, 16, 18, 20)
```

c stands for “concatenate”. Some people think it means “combine” but that’s nowhere near as exciting a word.

```
Y
```

```
 [1]  2  4  6  8 10 12 14 16 18 20
```

R will quite happily do arithmetic with vectors or matrices as well as simple numbers.

```
X + 3
```

```
 [1]  4  5  6  7  8  9 10 11 12 13
```

Adds three to every number in the vector. We can also do arithmetic with two vectors.

```
X * Y
```

```
 [1]   2   8  18  32  50  72  98 128 162 200
```

```
Y/X
```

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

Both of our vectors, X and Y, are the same length in this example, and R just

carries out the instruction on each number in the sequence in turn: the first number in X is multiplied by the first number in Y, then the second in X by the second in Y and so on until the last number. What happens if the two vectors are of different lengths? We can set up a vector half as long as X very simply.

```
X2 <- seq(1, 10, by = 2)
X2
```

```
[1] 1 3 5 7 9
```

Let's add this 5 number vector to our first 10 number vector and see what happens.

```
X + X2
```

```
 [1]  2  5  8 11 14  7 10 13 16 19
```

What's happened here? The first number in X (1) has been added to the first number in X2 (1) to give 2. The second numbers in each (2 and 3) have been added in the same way, and so on until we got to the last number in X2, 9, which was added to 5 to give 14. The next number in our answer is 7 - where did that come from? What happens when one vector is shorter than the other is that once all the numbers in the shorter vector have been used, you just start again at the beginning, so the sixth number of X (6) is added to the first number of X2 (1) to give the sixth number in our answer.

1+1=2
2+3=5
3+5=8
4+7=11
5+9=14
6+1=7
7+3=10
8+5=13
9+7=16
10+9=19

This still works if the length of the longer vector is not an exact multiple of the length of the shorter one, but you'll get a warning message.

```
X3 <- seq(1, 10, by = 3)
X3
```

```
[1]  1  4  7 10
```

```
X + X3
```

```
Warning: longer object length is not a multiple of
shorter object length
```

```
 [1]  2  6 10 14  6 10 14 18 10 14
```

Functions work just as well on vectors and matrices as they do on individual objects.

```
logX <- log(X)
```

This sets up a new object called logX which contains the natural log transformed values of X.

```
logX
```

```
 [1] 0.0000 0.6931 1.0986 1.3863 1.6094 1.7918 1.9459
2.0794 2.1972 2.3026
```

Note that this leaves the original variable, X, as it was. If you'd like to replace X with the transformed version then just type

```
X <- log(X)
```

and that'll do the trick. In general, it's better to make a new object than to replace one because it leaves more room for post-cockup recovery.

Exercises 4: Vectors

- Delete the objects x1, x2, x3 and x4 that you set up for the previous set of exercises using the rm() function
- Create a vector called "x1" consisting of the numbers 1,3,5 and 7
- Create a vector called "x2" consisting of the numbers 2,4,6 and 8
- Subtract x1 from x2
- Create a new vector called "x3" by multiplying vector x1 by vector x2
- Create a new vector called "x4" by taking the square root of each member of x3
- Use the `mean()` function to calculate the arithmetic mean of the numbers in vector x4
- Use the `median()` function to calculate the median value of the numbers in vector x3

Matrices

When we have data that are arranged in two dimensions rather than one we have a matrix, although not one that features Agent Smith. We can set one up using the `matrix()` function.

```
mat1 <- matrix(data = seq(1, 12), nrow = 3, ncol = 4,
dimnames = list(c("Row 1",
    "Row 2", "Row 3"), c("Col 1", "Col 2", "Col 3",
"Col 4")))
```

This is a complicated looking command. I've kept the argument names in to make it clearer what's going on. We can understand it better by going through each bit in turn.

```
mat1<-
```

Sets up an object called mat1.

```
matrix(data = seq(1, 12)
```

The object is a matrix, and the data in the matrix is a sequence of numbers counting from 1 to 12.

```
nrow=3, ncol=4
```

The number of rows in the matrix is 3 and there are 4 columns.

```
dimnames = list(c("Row 1", "Row 2", "Row 3"), c("Col 1", "Col 2", "Col 3", "Col 4"))
```

This gives the rows and the columns names. Note that the text is enclosed in quote marks. We'll talk about lists later on.

```
mat1
```

```
      Col 1 Col 2 Col 3 Col 4
Row 1     1     4     7    10
Row 2     2     5     8    11
Row 3     3     6     9    12
```

One thing to notice about this is that the default option in R is to fill a matrix up in column order rather than row order. This can be reversed by using the *byrow=TRUE* argument. Be careful about this when setting matrices up because it's easy to make a mistake.

We can carry out simple arithmetic with our matrix just as we can with a vector.

```
mat2 <- mat1/3
```

This divides every number in the matrix by three, and allocate the numbers to a new object called “mat2”.

```
mat2
```

```
       Col 1 Col 2 Col 3 Col 4
Row 1 0.3333 1.333 2.333 3.333
Row 2 0.6667 1.667 2.667 3.667
Row 3 1.0000 2.000 3.000 4.000
```

We can use matrices as arguments for functions.

```
sqrt(mat1)
```

```
      Col 1 Col 2 Col 3 Col 4
Row 1 1.000 2.000 2.646 3.162
Row 2 1.414 2.236 2.828 3.317
Row 3 1.732 2.449 3.000 3.464
```

I’m not going to go into the details of how matrices are added and multiplied but I’ll just tell you that R will do it without batting an eyelid.

```
mat1 + mat2
```

```
      Col 1 Col 2  Col 3 Col 4
Row 1 1.333 5.333  9.333 13.33
Row 2 2.667 6.667 10.667 14.67
Row 3 4.000 8.000 12.000 16.00
```

Exercises 5: Matrices

- Use the `rm()` function to delete the objects x1, x2, x3, x4 and ln
- Create a new vector called “V1” consisting of the following numbers 1,3,5,7,9,11
- Create a matrix called “mat1” using the following function: `mat1<-matrix(V1,nrow=2)`
- Create a matrix called “mat2” using the same function but add an extra argument *byrow=TRUE*
- Compare the two matrices and note how the data stored in the vector V1 are used to fill up the cells of the two matrices

Data types and Data frames

So far we’ve really only looked at data that are recorded as numbers: numerical, or quantitative data. Numeric objects can be further classified as, for example, *real* (real numbers), *complex* (complex numbers), *double* (double precision real numbers) or *integer* (integers). These distinctions really only become important if you go on to do some serious programming or maths that is a good way beyond what we'll be doing in this book - you might be relieved to know that we won't be thinking about complex numbers any more. Just in case you need to know you can find out if your vector is numeric using `mode()` and you can find out what kind of numeric it is using `class()`.

```
X1 <- 1:12
mode(X1)
```

```
[1] "numeric"
```

```
class(X1)
```

```
[1] "integer"
```

```
X2 <- sqrt(X1)
class(X2)
```

```
[1] "numeric"
```

R can also cope with strings of characters as objects. These have to be entered with quote marks around them because otherwise R will think that they're the names of objects and return an error when it can't find them.

```
X1 <- c("Red", "Red", "Blue", "Blue", "Green")
X1
```

```
[1] "Red"   "Red"   "Blue"  "Blue"  "Green"
```

```
class(X1)
```

```
[1] "character"
```

A special type of data in R is a **factor**. When we're collecting data we don't just record numbers: we might record whether a subject is male or female, whether a cricket is winged, wingless or intermediate or whether a leucocyte is an eosinophil, a neutrophil or a basophil. This type of data, where things are divided into classes, is called *categorical* or *nominal* data, and in R it is stored as a factor. We can input nominal data into R as numbers if we assign a number to each category, such as 1=red, 2=green and 3=blue and then tell R to make it a factor with the `factor()` function, but this can lead to confusion. Usually it's better to input data like this as the words themselves as character data and then tell R to make it a factor.

```
X1 <- factor(X1)
X1
```

```
[1] Red   Red   Blue  Blue  Green
Levels: Blue Green Red
```

You can see that now that we have classified X1 as a factor R tells us what the levels are that are found in the factor. Before you start any analysis it's a good idea to check whether you and R are both in agreement over which variables in your data set are factors, because sometimes you will find that you've assumed that something is a factor when it isn't. You can do this with the `is.factor()` function.

```
is.factor(X1)
```

```
[1] FALSE
```

```
is.factor(X2)
```

```
[1] TRUE
```

One thing to be aware of is that the mode of a factor will be *numeric* even if it's coded as characters. This is because R actually recodes variables and assigns a number to each level: you can see the numbers by using the `as.numeric()` function.

```
as.numeric(X2)
```

```
[1] 3 3 1 1 2
```

R codes the actor levels according to their alphabetical order, so "Blue" is coded as 1, "Green" as 2 and "Red" as 3. There's more on this in Chapter 12 when we think about ANOVA, factor levels and summary tables.

A final category of data is logical data. This is when we record something as true or false values and R can accept such data as “TRUE” vs “FALSE”.It can also cope with the abbreviations “T” and “F” but this is not advisable becuase R will

also let you allocate objects with the names “T” and “F” and if you then try to analyse some logical data coded as T or F this can lead to some very complicated problems. Stick with the words.

It’s normal to end up at the end of a study with a table of data that mixes up two or even three different sorts of data. If you’ve been studying how diet affects whether a species of cricket develops into winged or wingless morphs then for each individual you might have recorded Number (which animal was it) Diet (good or poor), Sex (male or female), Weight (mg), Fat content (mg) and wing morph (winged, wingless or intermediate), giving you a data table looking something like this:

Number	Diet	Sex	Weight	Fat.content	Morph
1	Poor	M	156	34	Winged
2	Poor	F	180	43	Winged
3	Good	M	167	40	Wingless
4	Good	F	190	43	Intermediate

Table 3.1 Example cricket data

We could input these data as a series of individual vectors, some numerical and some character, but that would lead to a lot of opportunity for confusion. It’s better to keep the whole data table as a single object in R, and there is a special type of object which we can do exactly this with. It’s called a Data frame, which is easiest to think of as being like a matrix but with different sorts of data in different columns. Most of the time when you’re using real scientific data in R you’ll be using a data frame. If you import a data table with multiple data types present then R will automatically classify your data as a data frame. Alternatively, if you want to input your data directly into R you can set up a data frame using the data.frame() function. Here's an example using the cricket data from above.

```
Number <- c(1, 2, 3, 4)
```

```
Diet <- c("Poor", "Poor", "Good", "Good")

Sex <- c("M", "F", "M", "F")

Weight <- c(156, 180, 167, 190)

Fat.content <- c(34, 43, 40, 43)

Morph <- c("Winged", "Winged", "Wingless",
"Intermediate")

crickets <- data.frame(Number, Diet, Sex, Weight,
Fat.content, Morph)

crickets
```

```
  Number Diet Sex Weight Fat.content        Morph
1      1 Poor   M    156           34       Winged
2      2 Poor   F    180           43       Winged
3      3 Good   M    167           40     Wingless
4      4 Good   F    190           43 Intermediate
```

Note that R assumes that the character vectors that are going into the new data frame are factors and makes them so.

```
is.factor(crickets$Diet)
```

```
[1] TRUE
```

Choosing data: subscripts and subsetting

As we saw in the last section, R is set up to deal with data in groups (vectors, matrices and dataframes), and most of the time that you're analysing real data you'll be dealing with data in one of these forms. For some simpler analyses you

might be happy just looking at all the data in a particular set of results, but a lot of the time you'll end up thinking "but what if we exclude males from the analysis?", or "does the result hold if we leave out that animal that might have been sick?" or "do we still see the trade-off if we only include the plants that flowered?". All of these can be done quite easily in R, and your chief weapons for such purposes are subscripts, which we'll deal with now, and the `subset()` command, which we've dealt with once we've dealt with subscripts.

Subscripts

Every number in a vector can be identified by its place in the sequence, and every number in a matrix can be identified by its row and column numbers. You can use subscripts to find individual numbers or groups within data structures. They're remarkably flexible and extremely useful.

```
Z <- rnorm(10, 2, 0.1)
```

This creates a vector called X made up of 10 random numbers drawn from a normal distribution with mean 2 and standard deviation 0.1. NB: if you try to do this your numbers won't be the same as mine, because they're drawn randomly each time.

```
Z
```

```
 [1] 2.039 2.065 2.034 1.920 2.066 2.003 1.929 2.048
1.931 1.945
```

If we want to find out what the fifth number in Z is we could just count along until we get there, and when R writes out a vector it helpfully puts a number at the start of each row which tells you where you are in the sequence. Just counting along a sequence of numbers can obviously get very unwieldy when we have larger datasets, and the potential for error by the counter is high even with each row being numbered. Fortunately we can just ask what the fifth number is by using a subscript, which at its simplest is just a number in square brackets after the name of our vector. R will go and look up whatever's at the position in the

vector that corresponds to the number and tell you what it is.

```
Z[5]
```

```
[1] 2.066
```

Subscripts do not have to be single numbers. The subscript can be an object.

```
p <- c(2, 5, 7)
```

This sets up a new object (*p*) which is a vector containing three numbers. We can now use this object as a subscript to find out what the second, fifth and seventh numbers in *Z* are.

```
Z[p]
```

```
[1] 2.065 2.066 1.929
```

The subscript can even be a function.

```
Z[seq(1:3)]
```

```
[1] 2.039 2.065 2.034
```

We know that seq(1:3) will return the numbers 1,2 and 3 so here we are asking what the first three numbers in *Z* are.

The subscript can also ask for the numbers in the vector excluding those specified in the subscript. This is particularly useful if you have some sort of dodgy data point that you want to exclude.

```
Z[-2]
```

```
[1] 2.039 2.034 1.920 2.066 2.003 1.929 2.048 1.931
```

```
1.945
```

This gives us all the numbers in Z except for the second one.

We can also include logical expressions in our subscript.

```
Z[Z > 1.95]
```

```
[1] 2.039 2.065 2.034 2.066 2.003 2.048
```

This returns all the numbers in Z that are greater than 1.95.

```
Z[Z <= 2]
```

```
[1] 1.920 1.929 1.931 1.945
```

This gives us all the numbers in Z that are less than or equal to 2.

You can use subscripts to find out useful things about your data. If you want to know how many numbers in Z are less than or equal to 2 you can combine some subscripting with the `length()` command.

```
length(Z[Z <= 2])
```

```
[1] 4
```

You can calculate other statistics as well. If you want to know the arithmetic mean of the numbers in Z that are less than or equal to 2 you can use a subscript.

```
mean(Z[Z <= 2])
```

```
[1] 1.931
```

This approach will work with just about any function. To find out the standard

deviation of the same set of numbers use this:

```
sd(Z[Z <= 2])
```

```
[1] 0.01055
```

and this gives the sum of the numbers in Z that are less than or equal to 2.

```
sum(Z[Z <= 2])
```

```
[1] 7.725
```

One thing to notice is that using subscripts gives you the values of the numbers that correspond to the criterion (NB for students in my second year tutorial group: this word is the singular of "criteria") you put in the square brackets but doesn't tell you where in the sequence they are. To do that we can use the function `which()`. To find out which numbers in Z are less than or equal to 2:

```
which(Z <= 2)
```

```
[1]  4  7  9 10
```

If we wanted to, we could then use these numbers in a subscript. Here I'm setting up an object that's a vector of these seven numbers.

```
less.than.2 <- which(Z <= 2)
```

Now I can use this object as a subscript itself.

```
Z[less.than.2]
```

```
[1] 1.920 1.929 1.931 1.945
```

The circle is complete. There is actually a serious point to this last part. There

are often several different ways of doing the same thing in R. It is often the case that there's an obvious "best way", but that isn't always the case: sometimes one way of doing something isn't noticeably easier or better than another, or sometimes doing something one way is better in one situation and doing it another way is better in a different situation. If someone else is doing something differently to you it doesn't necessarily mean that you are wrong: just check what they're doing and have a quick think about which method is better for what you're trying to do. If the answer is "my method", or if it's "I can't see any benefit to using the other method" then stick with what you're doing.

Exercises 6: Subscripts and vectors

The vectors x1,x2,x3 and x4 referred to here are the ones set up in exercises 4.

- Use a subscript to find out the value of the 3rd number in vector x1
- Use a subscript to find out the value of the numbers in vector x2 that aren't in the 3rd position
- Add the 1st number in vector x3 to the 4th number in vector x4
- Create a new vector called "In" which consists of the numbers 1 and 4
- Use subscripts and the "In" vector to create a new vector, x5, which consists of the sums of the 1st and the 4th numbers of x3 and x4
- Calculate the sum of all the numbers in x2 that are less than 7
- Calculate the mean of all the numbers in x1 that are greater than or equal to 3

Subscripts in matrices and data frames

Subscripts can also be used to get individual numbers, rows or columns from matrices and data frames in the same way as for vectors, except two numbers

are needed to identify an individual cell in these two dimensional data structures. The first number is the row number and the second is the column number. Here's another matrix.

```
mat4 <- matrix(data = seq(101, 112), nrow = 3, ncol =
4)
mat4
```

```
     [,1] [,2] [,3] [,4]
[1,]  101  104  107  110
[2,]  102  105  108  111
[3,]  103  106  109  112
```

To ask "What's the number that's in the third row and second column of mat2?", we put the row number first in the subscript, then a comma, then the column number. NB I always have to stop and think about this because some part of my brain always expects it to be column number then row number to make it like xy coordinates.

```
mat4[3, 2]
```

```
[1] 106
```

What are the numbers in the third row that are in the second, third and fourth columns?

```
mat4[3, c(2, 3, 4)]
```

```
[1] 106 109 112
```

To get hold of everything in a particular row, just put the row number followed by a comma and don't put in a number for the column. For example, if you just want the first row of the matrix use a 1.

```
mat4[1, ]
```

```
[1] 101 104 107 110
```

Likewise, if you want to get hold of a whole column then leave the row number empty.

```
mat4[, 3]
```

```
[1] 107 108 109
```

This gives us the third column of the matrix

```
mat4[, 1] + mat4[, 3]
```

```
[1] 208 210 212
```

This adds the first column of the matrix to the third column.

Exercises 7: Subscripts and matrices

These exercises use the objects previously set up in exercises 5.

- Multiply the second value in the first row of mat1 by the third value in the second row of mat2
- Create a new vector called “V2” which consists of the numbers in the first row of mat1 added to the numbers in the second row of mat2
- Create a new vector called “V3” which consists of the numbers in the second column of mat2 multiplied by the mean of the numbers in the second row of mat1
- Create a new matrix called “mat3” which consists of the first row of mat1 as the first column and then the first row of mat2 as the second column

Subset

The `subset()` function is useful when you want to extract part of a matrix or dataframe. It takes three main arguments, the first being the name of whatever you want to make a subset of, the second is a logical expression and the third tells R which columns you want to choose. It's best to show this with an example. Here's some data that were collected as part of an experiment on the effect of environmental temperature on leucocyte count in fish fry.

```
Counts <- read.csv("Counts.csv", header = T)
```

Let's look at the whole dataset to start with.

```
Counts
```

```
  Sex   Temp Weight Count
1   M   Hot    73.25   282
2   M   Hot    69.28   170
3   F   Hot    81.38   151
4   M   Hot    66.07   238
5   F Cold    83.32   136
6   F Cold    63.06   203
7   M Cold    78.48   312
8   M Cold    55.38   274
```

If we wanted to set up a second data frame containing only data from those fish that weighed 70mg or more, we can just specify the first two arguments.

```
Counts2 <- subset(Counts, Weight >= 70)

Counts2
```

```
  Sex   Temp Weight Count
1   M  Hot    73.25   282
3   F  Hot    81.38   151
5   F Cold    83.32   136
7   M Cold    78.48   312
```

What if we wanted to extract only the data on weights and leucocyte counts for male fish? For this we use the third argument as well, “select”.

```
Counts3 <- subset(Counts, Sex == "M", select =
c(Weight, Count))

Counts3
```

```
  Weight Count
1  73.25   282
2  69.28   170
4  66.07   238
7  78.48   312
8  55.38   274
```

One thing to notice here is that when we are specifying male fish only in the second argument we use the double equals sign (==). This is what’s used in R when we’re using logical expressions. The “M” is in inverted commas because it’s character data. It’s easy to forget and use a single equals sign, or miss out the inverted commas. If you do the latter you’ll get an error message.

```
Counts4 <- subset(Counts, Sex == M, select = c(Weight,
Count))
```

```
Error: object 'M' not found
```

If you only put a single equals sign in, however, you won’t get an error message. R will ignore the logical expression but it will select the columns specified and your new object will have data from both male and female fish. This could lead to

serious errors in your analysis, so always check.

```
Counts4 <- subset(Counts, Sex = "M", select =
c(Weight, Count))
```

See? No error message, but when you look at the output from this command you find that it hasn't been executed in the way you might wish.

```
Counts4
```

```
  Weight Count
1  73.25   282
2  69.28   170
3  81.38   151
4  66.07   238
5  83.32   136
6  63.06   203
7  78.48   312
8  55.38   274
```

`subset()` can also be used within other functions: if, for example, you only want to analyse part of a dataset but you don't want to set up a whole new object. We'll see some examples of this when we look at statistical model fitting in more detail.

Packages

This final section is a slight digression from the rest of the chapter but the use of R packages is such an important subject that it needs to be explained as early in your journey through the wonder that is R as possible. The basic R installation comes with a core set of functions that will allow you to do a wide range of analyses and draw a wide variety of graphs. If you want to go further, however, you'll come across one of the best features of R - the availability of packages that load new functions and other objects into your base R installation and enable you to use an almost infinite range of analysis techniques. Creating a new

package is straightforward, and because R is now so widely used in academia the majority of authors of publications describing new analysis techniques release an R package when they publish their new ideas. To give a flavour of how widespread this is, the figure below is a screenshot of part of the contents page from the December 2012 edition of Methods in Ecology and Evolution. The part of the contents page I've focussed on is the "Applications" section, where new statistical techniques tend to get reported in the journal. As you can see, of the seven papers published in this section, five are associated with R packages.

Figure 1. Screenshot of contents table from December 2012 Methods in Ecology and Evolution.

Some packages come with the base installation of R but are not automatically loaded when you start the software. These include:

- `lattice`, which includes functions for a range of advanced graphics,
- `MASS`, which is a package associated with the book "Modern Applied Statistics with S" by Venables and Ripley (2002, Springer-Verlag) and contains a variety of useful functions to do things like fit generalized linear models with negative binomial errors,
- `nlme` which lets you fit linear and non-linear mixed-effects models,
- `cluster` which brings a range of functions for cluster analysis and
- `survival` which has functions for survival analysis (surprise!). These will already probably be loaded onto your computer, but to make sure you can use the`installed.packages()` function. Just type it in with nothing between the brackets and you'll get more information about what's there than you really need. If you want to use one of them you can load them into R by using the `library()` function: so to load the cluster package you just need to type `library(cluster)` and it will be loaded.

If you want to know a bit more about what's in a particular package you can type `library(help=PACKAGE)` where `PACKAGE` is the name of, well, the package. This will get you some information about the package and a list of the various functions that are included in the package. If you want to know more then one of the easiest ways of finding out more information is to go to http://cran.r-project.org/web/packages/available_packages_by_name.html, which lists all 4000+ packages currently available for R. If you click on the name of a package you'll be able to navigate to a link for the package manual which should tell you everything you might ever want to know. It might be difficult to follow if you're a biologist or similar because it's likely to be written for consumption by statisticians, but you'll just need to persevere.

Most of the R packages out there aren't installed on your machine by default, of course. Let's say you're a community ecologist and you want to use the `vegan` package, which codes for a wide variety of functions to do things like calculate diversity indices and carry out ordinations. If you look on CRAN you can find the web page for vegan at http://cran.r-project.org/web/packages/vegan/index.html, which lets you look at the manual for the package and also provides links to a number of "vignettes" - documents giving details of how to carry out specific

analyses using the package. These can be very useful once you've got the package installed: when it's three o'clock in the morning and you're so desperate you would sell your cat's soul to Satan just to get that NMDS done those vignettes can preserve your sanity, but there's nothing obvious on the web page that you can click on to actually install it on your computer.

What you need to know before you install a package is that the central repository for R stuff, CRAN (Comprehensive R Archive Network) has a series of mirror sites around the World. When you download something you should choose a mirror site near to you so that you can avoid overloading the main CRAN site. A list of CRAN mirrors is available at http://cran.r-project.org/mirrors.html. Take a look at it and find some mirror sites near where you are.

Now that you've got an idea of which mirror sites you might want to use, you can set a mirror site by using `chooseCRANmirror()`, which will bring up a window that lets you choose the mirror site you'd like to use, and then you can install the package using the `install.packages()` function.

```
install.packages("vegan")
```

```
also installing the dependency 'permute'
trying URL
'http://www.stats.bris.ac.uk/R/bin/macosx/leopard/
contrib/2.15/permute_0.7-0.tgz'
Content type 'application/x-gzip' length 220346 bytes
(215 Kb)
opened URL
==================================================
downloaded 215 Kb

trying URL
'http://www.stats.bris.ac.uk/R/bin/macosx/leopard/
contrib/2.15/vegan_2.0-5.tgz'
Content type 'application/x-gzip' length 2353906 bytes
(2.2 Mb)
opened URL
```

```
==================================================
downloaded 2.2 Mb


The downloaded binary packages are in

/var/folders/qm/_szqszq95c34jn73g8b03bbh0000gn/T//RtmpmOBWXn/downlo
```

If you use the `install.packages()` function without specifying a mirror site you might well get a pop-up window asking you to choose a mirror site anyway, but it's more straightforward to do it first. Once the package is installed then load it by using the `library()` function and you're ready to go.

```
library(vegan)
```

```

```

Chapter 6: Basic Statistical Concepts

For the next 8 chapters our focus will shift from manipulating and visualising sets of numbers to analysing them, and as a starter this chapter has brief notes on some of the important fundamental concepts used to summarise and understand sets of data. If you're happy that you understand the difference between a population and a sample, and if (unlike at least 50% of academics in the biological sciences…) you know what a 95% confidence interval actually means then you might want just to skim through this chapter to pick up on the various R functions that are mentioned. If, on the other hand, your stats are a bit rusty or rudimentary this should act as a useful reminder, and the notes on how the various functions work in R should also be helpful. Please also have a look at the next chapter which is on statistical testing since this is a subject that lots of people are confused about even when they have had some stats training - to tell the truth, it's a subject that a lot of people are confused about *because* they've had some stats training, which is why I've put something in to try to help with this rather confusing issue.

This chapter and the next will show you the basics of data description and statistical testing but they aren't a substitute for proper training in statistics. If you are really a statistical novice then you might feel the need to look further afield for more information. There are an awful lot of books on statistics out there, and they range from the easily read and digested to the completely incomprehensible: I recommend that you seek out the former and avoid the latter. The exact choice of where you get your information will depend on what your particular interests are: for biologists I recommend Experimental Design and Data Analysis for Biologists by Quinn and Keough (2002, Published by CUP, ISBN 978-0521009768) but if you're in a different field then you'll need to shop around.

Populations and Samples

In statistics, the *population* consists of all of objects that you could potentially measure. This might be all of the lung cancer patients in the World, all of the malaria parasites infecting a particular species of lizard, all of the men in Europe between the ages of 15 and 25, all of the households in Texas with an income

greater than $100000 per annum, or all of the cars of a particular model. Most of the time, you won't be able to carry out whatever measurements you might wish to make on an entire population, which means that you have to *sample* from the population: you select a subset of the population, carry out your measurements on that subset and hope that the values that you get are a useful *estimate* of the true population values. Ideally this sampling will be done *randomly*, which means that every member of the population has an equal chance of being chosen and the probability of one member of the population being chosen is unaffected by whether another member has been chosen. It's often not easy to do straightforward random sampling of an entire population, in which case other techniques such as clustered sampling or stratified sampling will need to be used. There is a lot of literature out there on sampling if you need to know more. It's best to look for material that's specific to your field because the techniques associated with sampling vary a lot between fields: although the fundamental problem is the same for an ecologist sampling wasps from the canopy of a tropical forest and a psychologist sampling school-age children from households of a particular economic class the solutions to the problem are likely to be rather different. You can't sample children with a suction trap, but you don't need to get parental permission to catch wasps.

Population level *parameters* such as the mean and the variance are denoted by Greek letters: μ for the mean and σ^2 for the variance, whereas the *estimates* for these parameters that we calculate from our samples are denoted by Roman letters: $\bar{x}$ (x-bar) for the mean and s^2 for the variance.

Descriptive and exploratory statistics

Maximum and minimum

When you have done your experiment, carried out your survey or scraped your data off the website in question you need to be able to *describe* and *explore* your dataset to gain an understanding of what kind of numbers or other values you've aquired. Some of the most basic questions are what the largest and smallest numbers are: these can be found by using the `max()` and `min()` functions. The `range()` function will return both the minumum and the maximum of a vector.

Frequency distribution

Knowing the range of your data tells you something but you won't get much of a feel for where the majority of the measurements lie. To get a better idea of what's going on the best thing to look at is the frequency distribution, the count of how many times each value occurs in a data set. This is in lots of ways the most fundamental way of describing a set of data because it lets you see how spread out the data are and where the measurements tend to cluster. The overall shape of the frequency distribution is also very important in helping you understand the nature of your data. Frequency distributions are easiest to work out with integer data since the frequency distribution is literally the count of each value in the data set. For continuous data we divide up the range of values over which the data are found into a series of 'bins' and count how many values are in each bin. As an example, let's say we have the following values in our data set.

Measure1 = 8.47, 6.08, 9.57, 12.18, 7.60, 9.67, 9.39, 10.83, 10.46, 10.55, 8.37, 10.58, 9.42, 7.95, 11.86

We can put these into R:

```
M1 <- c(8.47, 6.08, 9.57, 12.18, 7.6, 9.67, 9.39,
10.83, 10.46, 10.55, 8.37, 10.58,
    9.42, 7.95, 11.86)

min(M1)
```

```
[1] 6.08
```

```
max(M1)
```

```
[1] 12.18
```

Our minimum is just over 6 and our maximum just over 12, so we might as well use the ranges 6.01-7,7.01-8 and so on until 12.01-13 as our bins, and we can

just count the number of values that go into each.

Bin	Count
6.01-7	1
7.01-8	2
8.01-9	2
9.01-10	4
10.01-11	4
11.01-12	1
12.01-13	1

Just looking at this table of counts you can get an idea of where the *central tendency* is in the data and what the shape of the frequency distibution looks like, but there are better ways of visualising the frequency distribution than just looking at a table. The method that most people use is to draw a histogram of the frequency distribution, which you do in R with the `hist()` function.

```
hist(M1, col = "grey", main = "")
```

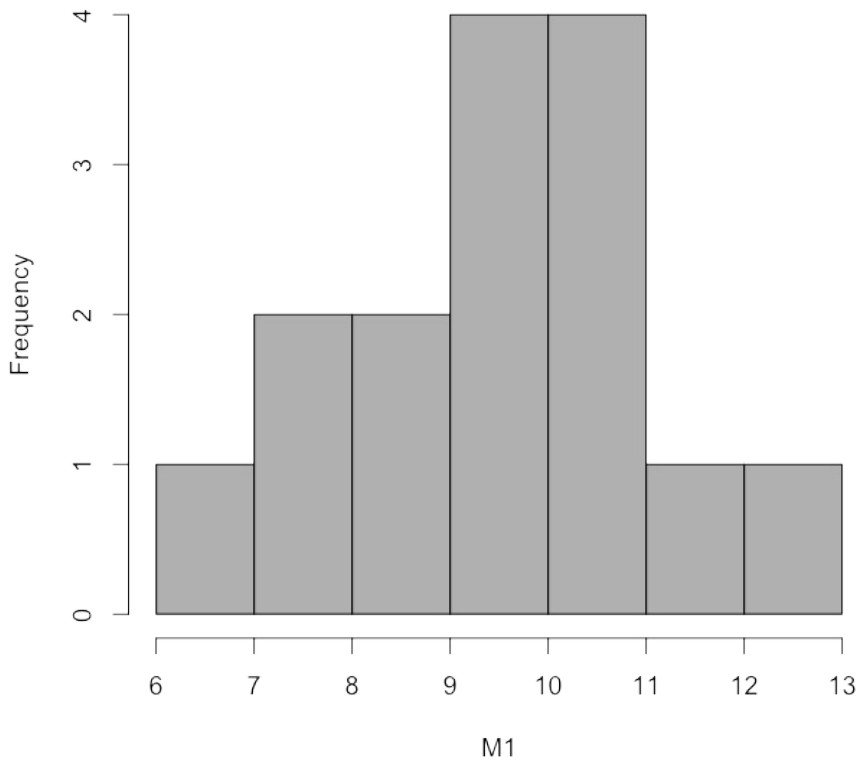


Figure 1: Histogram of the frequency distribution of vector M1

A second option is to draw a *boxplot*, otherwise known as a *box and whisker plot*. The most common version of these has a thick horizontal bar which indicates the *median* (see section on measures of central tendency), the top and bottom of the box indicates the *inter-quartile range* (see section on measures of dispersion) and the thin lines extending from the box (the *whiskers*) reach to the last data points within 1.5 times the interquartile range of the top or bottom of the box. If that last sounds a bit convoluted just think of them as a representation of the spread of data outside the interquartile range. Data points outside the range of the whisker are indicated individually: these are often described as *outliers* but be careful with this word: it's quite likely that they will actually be within the expected distribution of the data, and you certainly shouldn't discount data points just because they pop up on a box plot. You can draw boxplots in R with the `boxplot()` function.

```
boxplot(M1, col = "grey", xlab = "M1", ylab = "Value")
```

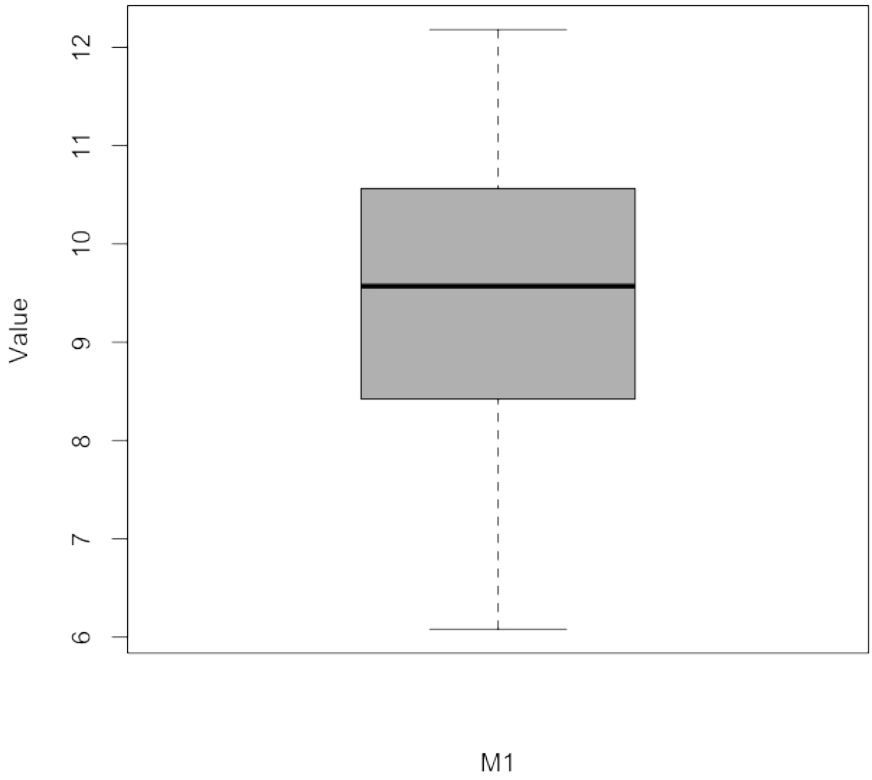


Figure 2: Boxplot of vector M1

Let's add another data point to M1 so see what a histogram and a boxplot look like when we have a data point that's a bit different from the rest.

```
M2 <- c(M1, 17.3)
```

We can use the `mfrow` graphical parameter to draw two plots side-by-side.

```
par(mfrow = c(1, 2))

hist(M2, col = "grey", main = "")

boxplot(M2, col = "grey", xlab = "M1", ylab = "Value")
```

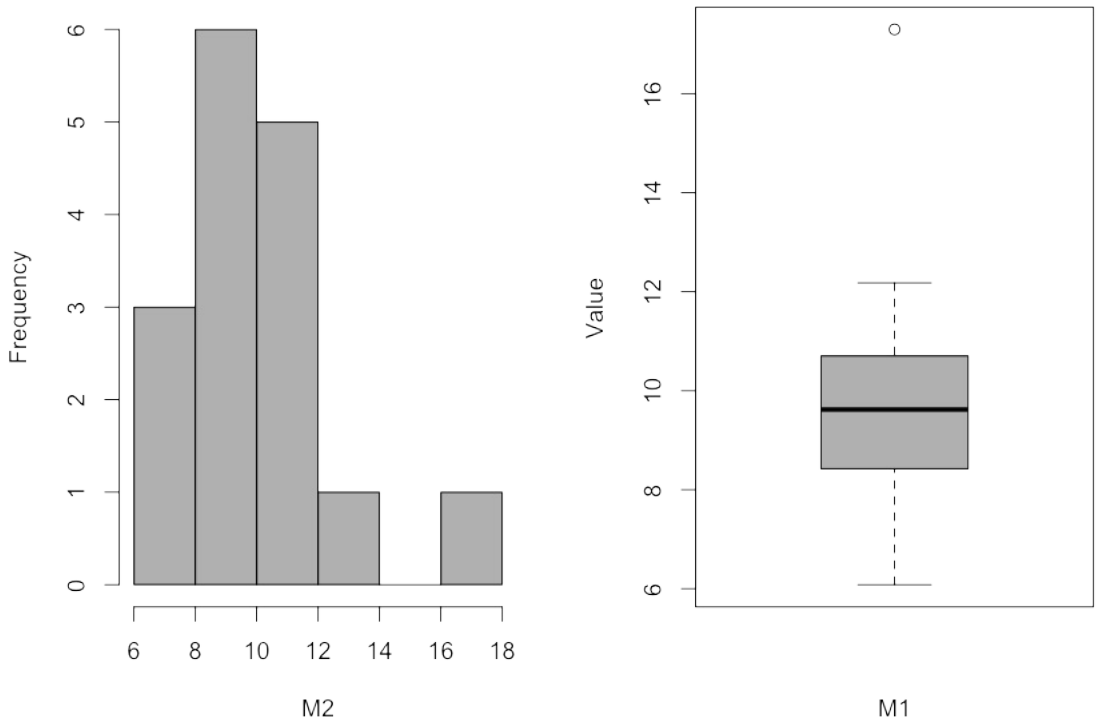


Figure 3: Comparison of histogram (left) and boxplot(right) for vector M2

In general people tend to use a frequency histogram when they are looking at one or a few vectors of data, and boxplots when they're looking at several different vectors, or one vector divided up by the levels of one or more factors: see the first plot in chapter 12 for an example of the latter.

Finally, another quick way to visualise a vector of data is to use a *stem and leaf diagram* which in its simplest form puts the values before the decimal point on the left hand side of a line and the values after the decimal point on the right, or the tens on the left and the units on the right, or something similar depending on the exact nature of the data. The R function `stem()` will draw you a stem and leaf diagram, but sometimes the default options aren't quite what you want so in this case we're going to use the *scale* argument to make the diagram exactly as we'd like it.

```
stem(M1, scale = 2)
```

```
The decimal point is at the |

 6 | 1
 7 | 6
 8 | 045
 9 | 4467
10 | 5668
11 | 9
12 | 2
```

Some of the values are in slightly different rows with this diagram than we might expect because the function is, for example, rounding 7.95 up to 8.0. Nonetheless, you can see the shape of the frequency distribution quite nicely, and you can see that the stem and leaf plot retains all of the information about your data set: you can see the value of each number in the vector M1 while also looking at the distribution. If there are any values that need further examination you'll know exactly what they are.

```
stem(M2, scale = 3)
```

```
The decimal point is at the |

 6 | 1
 7 | 6
 8 | 045
 9 | 4467
10 | 5668
11 | 9
12 | 2
13 |
14 |
15 |
16 |
```

```
  17 | 3
```

When you look at the diagram you can see not only that there is one data point that's a bit different from the others, but also what the value for that datapoint is. A caveat about using stem and leaf plots in R is that it's sometimes hard to work out exactly what a number is because the R function will often use bins that spread over several values on the left side of the plot, and then doesn't give any indication of which value the one on the right side should be associated with. This is why I've been using the *scale* argument to make these plots clear. If we don't use it we get this.

```
stem(M2)
```

```
  The decimal point is 1 digit(s) to the right of the
|

  0 | 6888899
  1 | 00011122
  1 | 7
```

Which isn't so informative.

Kinds of frequency distribution

There are a great many possible shapes of frequency distribution. The most common is the famous *normal distribution*, the symmetrical bell-shaped distribution. Next most common, at least in biology, are various *skewed* distributions, with long tails extending either to the right of the main body of the distribution (*positive skew*) or to the left (*negative skew*). Skewed distributions can arise for a variety of reasons: *multiplicative* rather than *additive* processes (such as bacterial growth when there's plenty of nutrients, where the number of bacteria in one generation is determined by multiplying the number in the previous generation rather than just by adding something to it) can create things like *lognormal* distributions, and count data often follow a *Poisson distribution*

which will have positive skew unless the mean is large. Frequency distributions with two peaks instead of one are called *bimodal*. We can use some of R's built in functions for drawing random numbers from different distributions to illustrate these.

```
par(mfrow = c(2, 2))

hist(rnorm(1000, 5, 2), col = "grey", main = "Normal",
breaks = 15)

hist(exp(rnorm(1000, 0, 0.5)), col = "grey", main =
"Lognormal", breaks = 15)

hist(rpois(1000, 2), col = "grey", main = "Poisson")

hist(c(rnorm(500, 0, 2), rnorm(500, 8, 2)), col =
"grey", main = "Bimodal", breaks = 15)
```

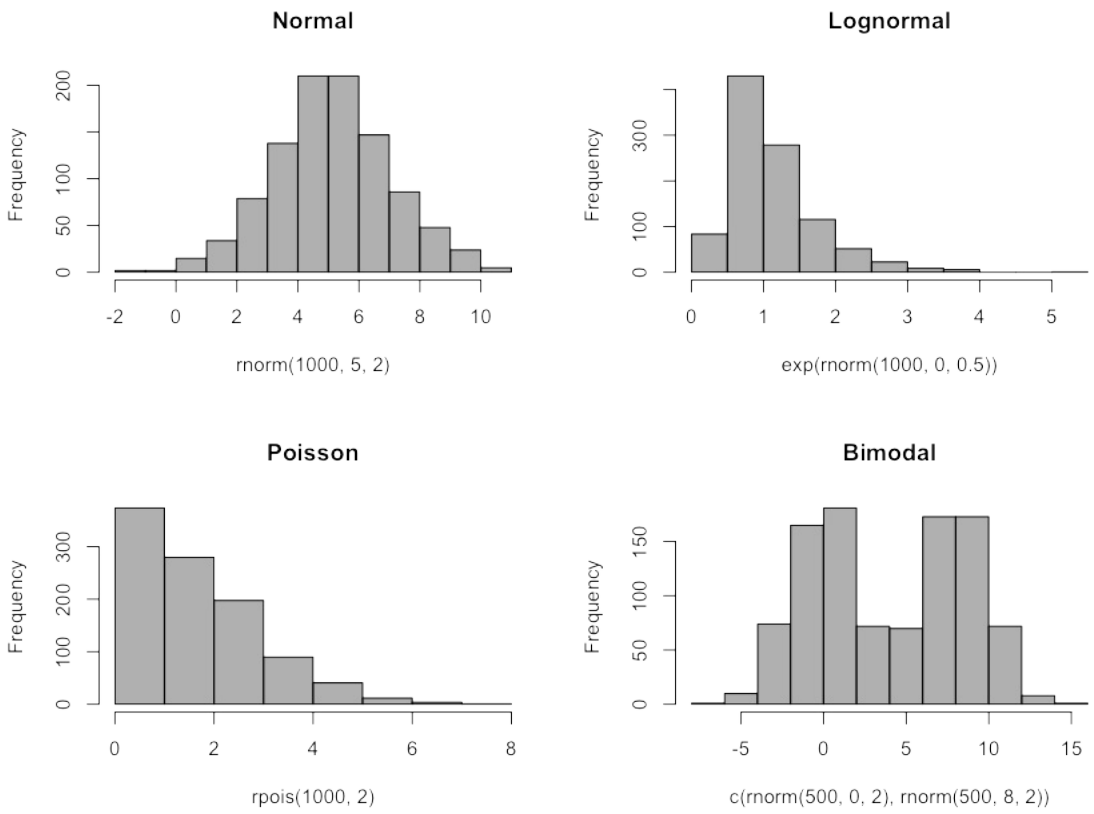


Figure 4: Frequency histograms of 1000 numbers sampled at random from populations with four different distributions

A cautionary word about frequency distributions

Knowing the shape of a frequency distribution is very important: it really tells you a lot about the data that you're dealing with, and can guide you when you're deciding which analysis to use. There is however, a certain amount of misinformation around on how to think about frequency distributions, however - in particular, there are still people being taught that they should carry out statistical tests to see if a frequency distribution is different from a normal distribution, and that they should only use standard parametric statistical techniques if their data does not deviate from a normal distribution. If you've been taught this then sorry, this is wrong and you shouldn't do it. The assumption behind most standard parametric statistics is that the *errors* are normally distributed, not that the data themselves are, so you should be looking at diagnostic plots of residuals vs. predicted values, or qq-plots of residuals (see the chapteron linear regression for more on these) rather than worrying about the raw data.

Measures of central tendency

Looking at frequency histograms, boxplots or stem and leaf plots can tell us a lot about a set of data, but we also want to be able to summarise some of the important features of our data, in particular the *location*: where it is and the *spread* or dispersion: how far from the centre of the data set the values tend to be found. Let's look at measures of *location* first, often called *measures of central tendency*. These tell us where the middle of the frequency distribution is. The ones that you'll see most often are the *mean* and the *median*. You will also see the *mode* in many elementary statistics textbooks but in practice it's not used as often as the other two.

Mean

The mean is the arithmetic mean of the data set, often called the “average”, calculated by dividing the sum of all the numbers by the number of values in the dataset.

```
mean(M1)
```

```
[1] 9.532
```

Be careful with means. They are really only useful when describing data drawn from symmetrical frequency distributions (such as a normal distribution). If there is skew in the data the value of the mean will be pulled away from the main bulk of the data and won't really tell you much that's useful.

```
mean(M2)
```

```
[1] 10.02
```

You can see here how the mean value for our example set of data changes when we add a single data point that's a bit outside the range of the rest.

Median

```
median(M1)
```

```
[1] 9.57
```

The *median* is simply the number in the middle. If you *rank* your data - put them in order according to their value - and take the middle value, that's the median. If you have an even number of data points then take the average of the two points in the middle. For non-symmetrical frequency distributions the median is a rather better measure of where the centre of the data set is than the mean, since it's less influenced by outliers than the mean.

```
median(M2)
```

```
[1] 9.62
```

Compare this value with the mean value for *M2* calculated above.

Mode

The *mode* of a dataset (or the *modal value*) is the most common value in the dataset. I'm only mentioning it because it appears in every elementary stats text so you might be expecting to see it. As mentioned above, it's rare to see anyone using the mode in real life and if you think about it the mode is really only meaningful for discrete variables: for a continuous variable every data point is likely to be unique so every data point is a mode. There isn't even a straightforward way of finding a mode for a discrete variable in R: the `mode()` function actually returns the *storage mode* of an object.

```
mode(X1)
```

```
[1] "numeric"
```

Just use the median. If you have to find a mode, a quick and somewhat dirty way to calculate a mode for a continuous variable is to use the maximum value of a *kernel density estimate*, as follows. Alternatively, put “finding the mode” into RSeek.org and enjoy the read.

```
dens <- density(M1)

dens$x[which(dens$y == max(dens$y))]
```

```
[1] 9.881
```

Measures of dispersion

While measures of central tendency like the mean and the median tell you where the centre of a distribution is, measures of dispersion tell you how spread out the data tend to be around that centre. The most common ones you'll see are the *variance* and the *standard deviation*, which are mathematically very closely related. Two others that are often used are the *inter-quartile range*, or *IQR*, and the *range*.

Variance and standard deviation

These closely related statistics both measure the average difference between the mean of a set of numbers and the individual numbers in that set. For a given group of numbers, we can calculate a mean and then calculate the difference between each number in the data set and the mean. Let's do that for our M1 numbers.

```
M1
```

```
 [1]  8.47  6.08  9.57 12.18  7.60  9.67  9.39 10.83
10.46 10.55  8.37
[12] 10.58  9.42  7.95 11.86
```

```
meanM1 <- mean(M1)

M1 - meanM1
```

```
 [1] -1.062 -3.452  0.038  2.648 -1.932  0.138 -0.142
1.298  0.928  1.018
[11] -1.162  1.048 -0.112 -1.582  2.328
```

```
sum(M1 - meanM1)
```

```
[1] 8.882e-16
```

This adds up to something very very very close to zero[1], of course, because some of the numbers are below the mean and some are above the mean, so what we do is to take the *square* of the differences between the individual numbers and the mean. Squaring a negative number gives a positive answer so the negatives no longer cancel the positives. If we add all the squared differences (or to put it in statistical-ese the *squared deviations from the mean*) together we get the *sum of squares* for our set of numbers.

```
sum((M1 - meanM1)^2)
```

```
[1] 37.8
```

The sum of squares is useful in the calculation of ANOVA and GLMS but not so useful as a simple measure of spread because it will increase as we increase the number of measurements, so we standardise the measure by dividing by the *degrees of freedom*, which in this case is *n-1*. We divide by the *df* and not just by *n* because we are estimating a mean from a sample, and if we just used *n* then if our sample size were 1 we would estimate the variance of the *population* that our sample came from as zero.

```
sum((M1 - meanM1)^2)/(length(M1) - 1)
```

```
[1] 2.7
```

Alternatively, we can just use the `var()` function.

```
var(M1)
```

```
[1] 2.7
```

Formally, we can write the variance down as

$$s^2 = \frac{\Sigma(x - \bar{x})^2}{n - 1}$$

where s^2 represents the variance and the *x* with a line over it (usually called "x-bar") represents the mean. The higher the variance the more spread out the distribution of data, as can be seen in figure 5 below.

```
x1 <- seq(-4, 4, length = 100)

y1 <- dnorm(x1, mean = 0, sd = 1)
```

```
y2 <- dnorm(x1, mean = 0, sd = 2)

plot(y1 ~ x1, type = "l", col = "darkgreen", xlab =
"x", ylab = "P(x)")

points(y2 ~ x1, type = "l", col = "steelblue")

legend("topright", legend = c("sd=1, variance=1",
"sd=2, variance=4"), lty = 1, col = c("darkgreen",
    "steelblue"))
```

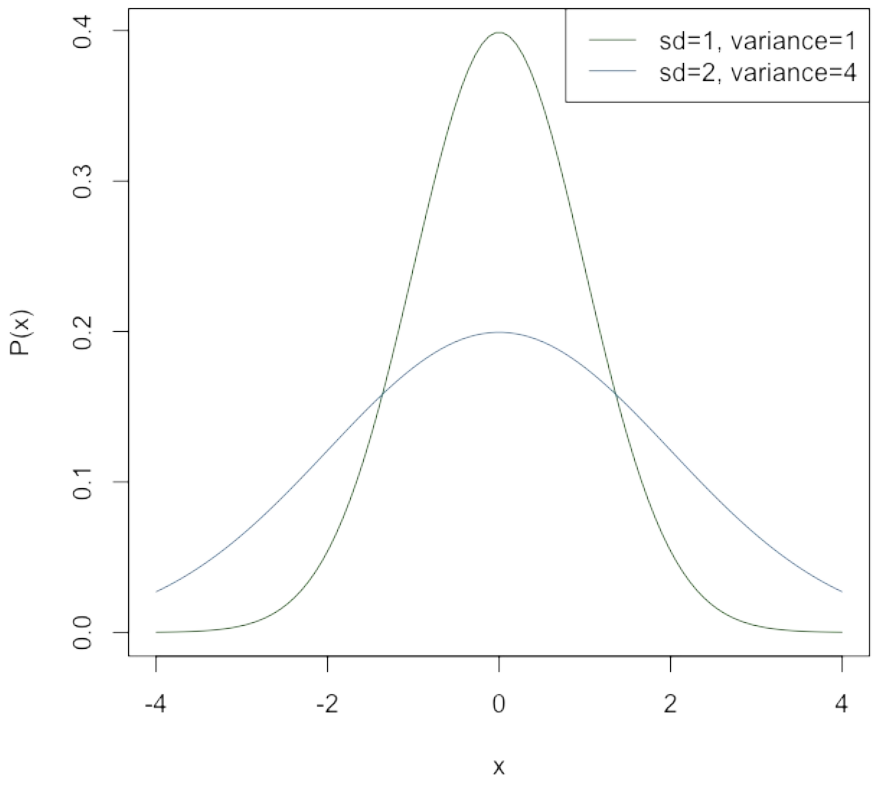


Figure 5: Probability densities for two normal distributions, both with means of zero but with differing variances.

Because the variance is calculated from the squared deviations from the mean, it's sometimes hard to relate it back to the actual data you're looking at. The square root of the variance, or the *standard deviation*, on the other hand, is easy to understand and, if your data are drawn from a normal distribution, tells you useful things to know about the data. If you draw the *probability density* of a

normal distribution (the “bell curve”) then one standard deviation is the distance from the centre of the distribution (the mean and also, in this case, the median) to the point of *inflexion* in the curve: the place where the slope of the line starts to become less steep. As shown in figure 6, for a large set of data sampled from a randomly distributed population, 68% of the values will be within one standard deviation of the mean, 95% will be within 1.96 standard deviations of the mean (and 96% will be within 2) and 99% of the values will be within 2.58 standard deviations.

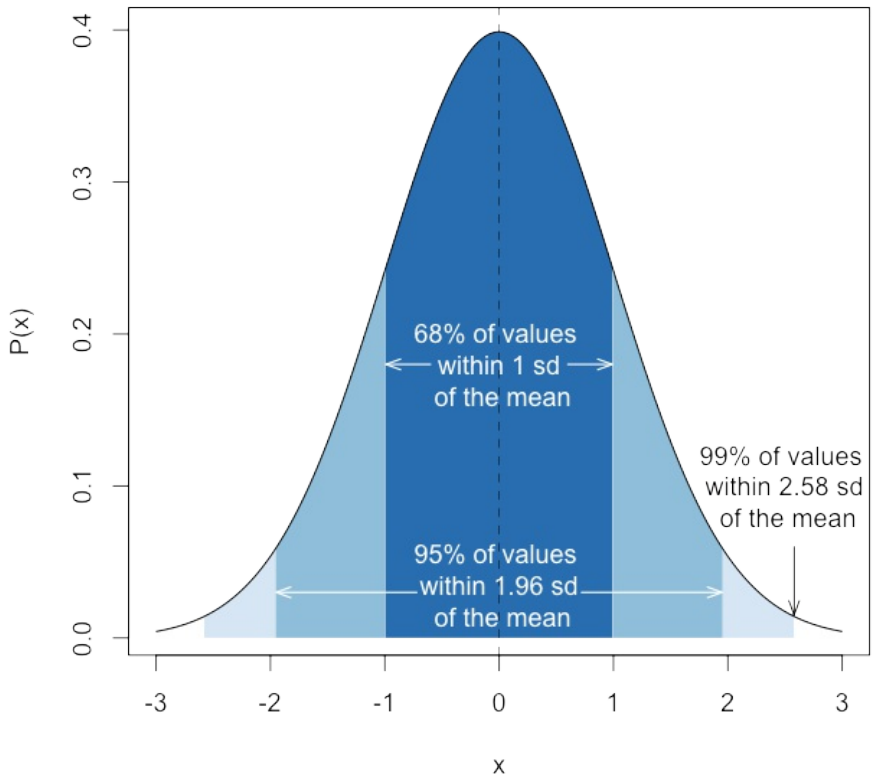


Figure 6: Probability density of a standard normal distribution with mean=0 and standard deviation=1 showing the areas defied by the mean plus or minus 1, 1.96 and 2.58 standard deviations. The code for this figure is given at the end of the chapter[2].

Range and interquartile range

Although the variance and the standard deviation are useful for describing well behaved normal or approximately normal distributions of data, if your frequency

distribution is strongly skewed or otherwise deviant it's better to use other measures of dispersion that don't assume that your data are symmetrically distributed around the mean. One option is just to use the *range*, defined by the minimum and maximum values, but this is obviously very sensitive to extreme values, so what people tend to use is the *interquartile range* or *IQR*. The *quartiles* of a distribution are calculated by ranking your data and then dividing it into four equal groups - the first quartile is the value that divides the first group from the second (in other words, the value halfway between the first value and the median), the second quartile is the median, the third quartile is the value halfway between the median and the maximum value and the fourth quartile is the maximum value. The IQR is the range of numbers defined by the first and third quartiles, and can be found using the `IQR()` function.

```
IQR(M1)
```

```
[1] 2.145
```

Alternatively, if you use the `summary()` function on a vector of numbers it will give you the first and third quartiles as well as quite a lot of other useful statistics including the mean, the median and the maximum and minimum values.

```
summary(M1)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
   6.08    8.42    9.57    9.53   10.60   12.20 
```

Standard errors and confidence intervals

When we estimate a parameter such as a mean it is useful to also have an indication of how accurate that estimate is likely to be. A mean estimated from a small sample drawn from a population with a high variance is more likely to be inaccurate than one estimated from a large sample, or one drawn from a population with a low variance. The *standard error* and the *confidence intervals* (or *confidence limits*) are two ways of quantifying how good our estimates are

likely to be.

The standard error (SE) of a mean is best understood by imagining a situation where you repeatedly sample at random from a population, with the same sample size each time, and calculate a mean for each sample. The standard error is the standard deviation of that set of means. We can use R to do exactly this to make the point. First let's use the `replicate()` and `rnorm()` functions to generate 1000 samples, each of 25 numbers, drawn at random from a normal distribution with mean 25 and standard deviation 2.

```
X1 <- replicate(1000, mean(rnorm(n = 25, mean = 5, sd
= 2)))
```

Let's have a look at our distribution of means.

```
hist(X1, col = "grey")
```

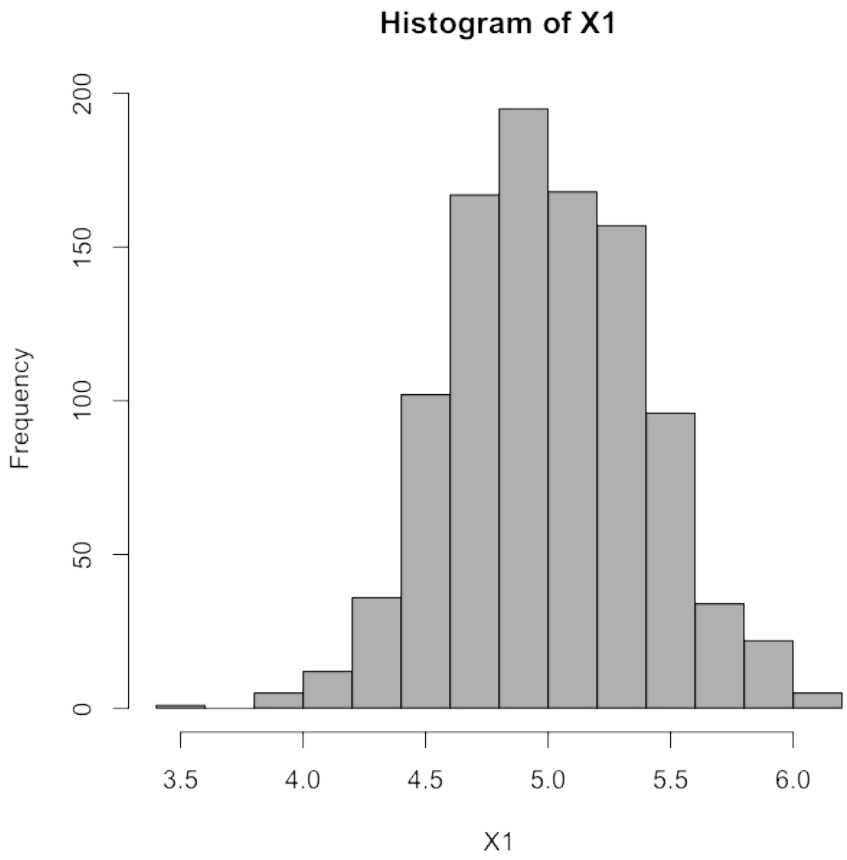


Figure 7: Frequency histogram for the means of 1000 samples

drawn from a normal distribution with mean 5 and standard deviation 2

You can see that when we sample repeatedly and calculate a mean for each sample, our sample means are themselves normally distributed. In fact, they would be normally distributed no matter what the shape of the underlying distribution they were sampled from was as a consequence of something called the *central limit theorem*. Our estimated means are clustered around the true value of the mean, and most of them are reasonably close, but some are rather further from the mean and a few are quite a distance from the true value. The standard deviation of this distribution of means is the standard error.

```
sd(X1)
```

```
[1] 0.3971
```

In practice, we can estimate the standard error from the standard standard deviation of a single sample by using this equation.

$$SE_{mean} = \frac{s}{\sqrt{n}}$$

If you recall the section on standard deviations, you'll remember that we know what proportion of values drawn from a normal distribution will lie within a given number of standard deviations of the mean, and in particular 95% of values will be within 1.96 standard deviations from the mean. Remember that the standard error is the standard deviation of the distribution of means that you would expect if you sampled repeatedly. This means that if you calculate a single mean and a standard error for that mean, then 95% of the time the true population value of the mean will lie in the range of the mean ± 1.96xSE. This range is the 95% confidence interval for the mean.

If you're dealing with small samples (in practice, anything less than 100) this neat formula doesn't quite work out because our estimates of the standard deviation get a bit biased when we have a small sample size, so to correct for this instead of just using 1.96 to calculate our confidence intervals we use a value of *t* (see

the chapter on statistical testing for an explanation of *t* if you haven't met it before) on *n-1* degrees of freedom, so our formula for the 95% confidence interval of a mean becomes

$$95\%CI = \bar{x} \pm t_{n-1} \frac{s}{\sqrt{n}}$$

where *t* is the value of *t* which would encompass 95% of the values in a t-distribution. See the examples given in the various chapters on programming for how to calculate this using R.

Footnotes

1) It's not exactly zero because of a little bit of floating point error: see the First Circle of the R Inferno at http://www.burns-stat.com/pages/Tutor/R_inferno.pdf

2) Code used to draw figure 6

```
X1 <- seq(-3, 3, length = 300)
Y1 <- dnorm(X1)
plot(X1, Y1, type = "n", xlab = "x", ylab = "P(x)")
abline(v = 0, lwd = 0.5, lty = 2)

x0 <- min(which(X1 >= -2.58))
x1 <- min(which(X1 >= -1.96))
x2 <- min(which(X1 >= -1))
x3 <- max(which(X1 <= 1))
x4 <- max(which(X1 <= 1.96))
x5 <- max(which(X1 <= 2.58))

polygon(x = c(X1[c(1, 1:x0, x0)]), y = c(0, Y1[1:x0],
0), col = "white", border = NA)
polygon(x = c(X1[c(x0, x0:x1, x1)]), y = c(0,
Y1[x0:x1], 0), col = "#deebf7", border = NA)
polygon(x = c(X1[c(x1, x1:x2, x2)]), y = c(0,
```

```
Y1[x1:x2], 0), col = "#9ecae1", border = NA)
polygon(x = c(X1[c(x2, x2:x3, x3)]), y = c(0,
Y1[x2:x3], 0), col = "#3182bd", border = NA)
polygon(x = c(X1[c(x3, x3:x4, x4)]), y = c(0,
Y1[x3:x4], 0), col = "#9ecae1", border = NA)
polygon(x = c(X1[c(x4, x4:x5, x5)]), y = c(0,
Y1[x4:x5], 0), col = "#deebf7", border = NA)
polygon(x = c(X1[c(x5, x5:300, 300)]), y = c(0,
Y1[x5:300], 0), col = "white", border = NA)


points(X1, Y1, type = "l")

abline(v = 0, lwd = 0.5, lty = 2)

text(0, 0.18, "68% of values \n within 1 sd \n of the
mean", cex = 1, col = "white")
arrows(0.6, 0.18, 0.99, 0.18, length = 0.1, angle =
20, col = "white")
arrows(-0.6, 0.18, -0.99, 0.18, length = 0.1, angle =
20, col = "white")


text(0, 0.035, "95% of values \n within 1.96 sd \n of
the mean", cex = 1, col = "white")
arrows(0.72, 0.03, 1.95, 0.03, length = 0.1, angle =
20, col = "white")
arrows(-0.72, 0.03, -1.95, 0.03, length = 0.1, angle =
20, col = "white")

text(2.5, 0.1, "99% of values \n within 2.58 sd \n of
the mean", cex = 1)
arrows(2.58, 0.06, 2.58, 0.015, length = 0.1, angle =
20)
```

Chapter 8: The Chi-squared Test and a Classic Dataset on Smoking and Cancer

When it comes to crucial studies in public health, it's hard to find one more important than a paper that was published in the British Medical Journal in 1950 by Richard Doll and A. Bradford Hill, entitled "Smoking and Carcinoma of the Lung: a Preliminary Report". Doctors had been noticing a huge rise in the incidence of an unpleasant and lethal disease, lung cancer, for a number of years: the figures quoted in the Doll and Hill paper are that in 1922 there were 612 deaths from the disease in the UK, but in 1947 there were 9,287, a roughly 15 times increase in a 25 year period. The two possible explanations that most people considered likely were firstly that a general increase in atmospheric pollution was the cause, and secondly that smoking, and especially cigarette smoking, was causing the increase in lung cancer. Doll and Hill carried out a survey of people diagnosed with lung cancer in hospitals in London. Each time a patient in a hospital in London was diagnosed with lung cancer the investigators were notified, and the patient was interviewed and asked a series of questions, including details of his or her smoking history. The interviewer then found a second patient in the same hospital who didn't have lung cancer but who matched the original patient in terms of sex and age, and asked them the same questions. Each lung cancer patient was therefore matched with a second patient who didn't have lung cancer but who was similar to the first patient in most important aspects - what we would nowadays call a case-control study. Using these data we can look at two of the questions that Doll and Hill addressed in their publication. Firstly, if we just consider the smokers in the sample, are cancer patients more likely to be heavy smokers than controls? Secondly, are patients with lung cancer more likely to be smokers than patients without?

Are lung cancer patients who smoke more likely to be heavy smokers?

Here is part of one of the tables of data from the Doll and Hill paper.

Disease Group	No. Smoking Daily				
	1 Cig.–*	5 Cigs.–	15 Cigs.–	25 Cigs.–	50 Cigs.+
Males: Lung-carcinoma patients (647)	33 (5·1%)	250 (38·6%)	196 (30·3%)	136 (21·0%)	32 (5·0%)
Control patients with diseases other than cancer (622) . .	55 (8·8%)	293 (47·1%)	190 (30·5%)	71 (11·4%)	13 (2·1%)

This table reports the amount of tobacco consumed daily before the onset of the current disease in male smokers both with and without lung cancer. Tobacco smoked in ways other than cigarettes has been converted to cigarette equivalents. One thing to notice is that the total number of patients is not the same in each group - there are more cancer patients than controls, a consequence of there being more controls who were classified as non-smokers and not included in this particular analysis. Looking at these data as a table we can see some differences between groups, but it will be easier to visualise it if we draw a graph. To do this we need to enter these data into R, and the best way to input these data is as a matrix.

```
cancer <- matrix(data = c(33, 250, 196, 136, 32, 55,
293, 190, 71, 13), nrow = 2,
    ncol = 5, byrow = T)
```

Let's break that instruction down so that we know what's going on.

```
cancer<-
```

Make a new object called "cancer". It should be whatever is on the other side of the allocation symbol.

```
matrix(data=c(33,250,196,136,32,55,293,190,71,13)…
```

What goes into the object “cancer” is a matrix, containing ten data points which I have just read off the table in the Doll and Hill paper.

```
…,nrow=2, ncol=5, byrow=TRUE)
```

After the “data” argument for the matrix function we have three more. nrow= tells R how many rows the matrix should have and ncol= gives the number of columns. byrow=TRUE tells R to read the data into the matrix by filling up the first row with numbers and then the second row. If byrow was set to FALSE it would fill it up starting with the first column.

The whole instruction therefore tells R the following: Set up a new object called “cancer”. It should be a matrix with two rows and five columns, filled with the following data. Read the data in by row, rather than by column, please. It would be nice to have proper names for our rows and columns, and we can do this with the `dimnames()` (think “dimension names”) function.

```
dimnames(cancer) <- list(c("Lung cancer", "Control"),
c("1 to 4", "5 to 14", "15 to 24",
    "25 to 49", "50+"))
```

The reason we’re using a list here is that we are giving the function two separate vectors of character data. Lists are a way of putting together sets of any sort of object in R: in this case one vector of row names (“Lung cancer” and “Control”) and one vector of column names. Let’s check that worked properly.

```
cancer
```

```
            1 to 4 5 to 14 15 to 24 25 to 49 50+
Lung cancer     33     250      196      136  32
Control         55     293      190       71  13
```

Our data matrix looks fine. Let’s draw ourselves a graph so that we can see any patterns in the data. A bar graph is the best way to represent frequencies like these, and we can use the barplot() function.

```
barplot(cancer, beside = T)
```

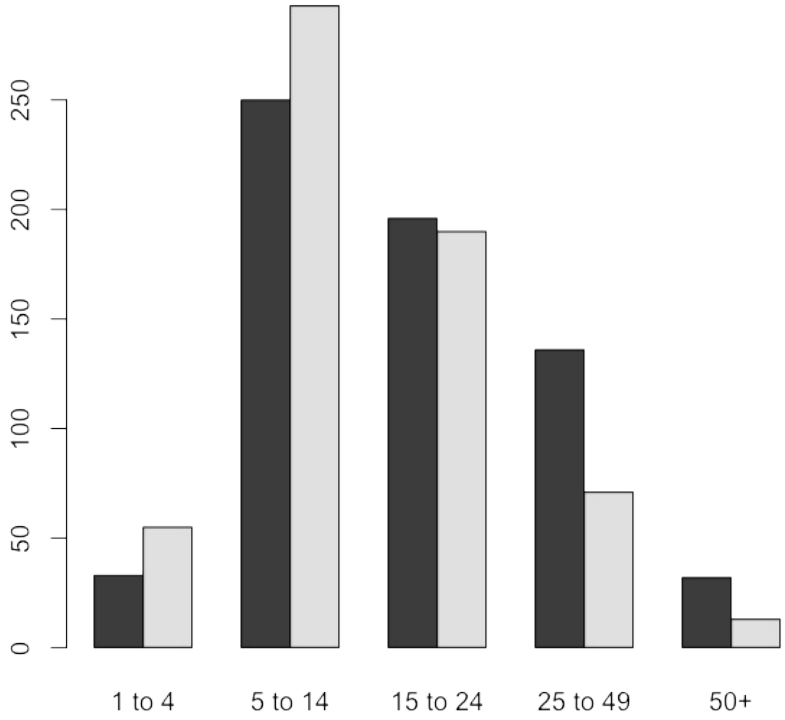


Figure 1: Barplot showing the tobacco consumption in cigarette equivalents per day for lung cancer patients and controls

The barplot() function is happy to take our matrix and if we tell it to plot the two rows beside each other (beside=T) it will draw us a nice graph.

It would be nice to have a legend and some axis labels just to make everything really clear. The barplot() function will automatically add a legend if you ask it to (legend.text=TRUE) and the xlab= and ylab= arguments let us specify the labels for the x- and y-axes.

```
barplot(cancer, beside = T, legend.text = T, xlab =
"Number of cigarettes smoked",
    ylab = "Number of cases")
```

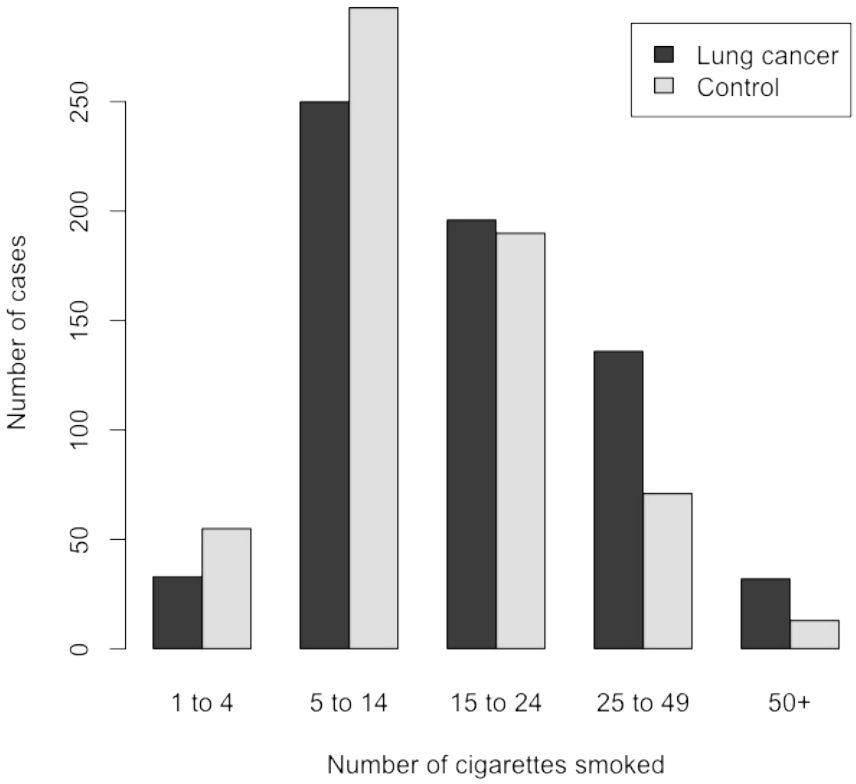


Figure 2: Barplot showing the tobacco consumption in cigarette equivalents per day for lung cancer patients and controls, now with axis labels and a legend

Looking at the graph we can see some differences between the frequencies of cancer patients and controls in the different groups: more controls reported smoking fewer than 15 cigarettes per day, whereas more lung cancer patients reported smoking more than 25 - in other words, lung cancer patients were more likely to be heavy smokers. The question we need to answer now is whether that apparent difference is likely simply to be the result of random error during sampling, or whether it's likely to reflect a genuine difference between the lung cancer patients and the controls. In other words, we want to know the probability of seeing a pattern like this if we were to sample at random from two populations where the distribution of smoking was actually the same. We can answer this question by carrying out a chi-square test on our data using the chisq.test() function.

The Chi-squared test

The Chi-squared distribution is a statistical probability distribution which is widely used in a large number of statistical tests, all of which are technically chi-squared tests. The most common of these is the *Pearson's Chi-Squared test*, and when people refer to a chi-squared test they are usually talking about the Pearson's version.

One of the main uses of Pearson's chi-squared test (hereafter just the chi-squared test) is to compare observed frequencies with the frequencies expected under some null hypothesis, and since this is the simplest way to use it we'll look at this first. As an example, consider an experiment on the genetics of flower colour in snapdragons. You know that the flowers can be red, pink or white and you suspect that flower colour is controlled by a single locus (a single gene for non-biologists) with two varieties, or alleles, which code for red and white respectively. You also think that the pink flowers are heterozygotes, with one copy of each allele, and the red and white flowers are homozygotes, with two copies of the same allele. You have a supply of pink flowering plants and you cross them and grow 500 of the seeds into flowering plants and score them for flower colour. If the colour of the flowers is controlled in the way you suspect then you'd expect 25% of the offspring to have white flowers, 50% to have pink, and 25% to have red, so your *expected frequencies* are 125 white, 250 pink and 125 white. In fact, what you get when you do your experiment - your *observed frequencies* - are 106 white, 252 pink and 142 white. That's different from what was expected in that there are rather fewer white flowers and rather more red flowers than we expected to see. Could this just be a consequence of random chance? This is where the chi-squared test can be used.

The test statistic for a chi-square test is calculated as:

$$\chi^2 = \sum \frac{(O-E)^2}{E},$$

where O is the frequency that is observed and E is the expected frequency under the null hypothesis. This is calculated for each set of values and then they're all

added together (which is what the capital sigma means).

In the case of our snapdragons, here is our calculation of the test statistic.

Flower colour	Expected	Observed	$(E - O)^2$	$(E - O)^2 / E$
White	125	106	361	2.888
Pink	250	252	4	0.016
Red	125	142	289	2.312
Sum				**5.216**

This gives us a test statistic of 5.216, and we can now ask the question of how likely we are to see this value or greater given the null hypothesis. We can do this by using the `pchisq()` function in R. Since this returns the probability of a value less than or equal to the number give we have to subtract it from one. The *degrees of freedom* we need to use is the number of categories that we have counts for minus 1: in this case 3-1=2.

```
1 - pchisq(5.216, 2)
```

```
[1] 0.07368
```

This gives us a p-value which is slightly greater than the p=0.05 cutoff for statistical significance. This is therefore a non-significant result, and we do not have grounds to reject the null hypothesis. The p-value is close to significance, however, so all we can really say is that there is a lot of uncertainty about the genetics of flower colour in snapdragons. If we want to get a more definitive answer we should probably repeat the experiment with more snapdragons[1]. Back to the greenhouse.

To do the same test in R without having to do the calculations yourself use the `chisq.test()` function. For a chisquared test for *goodness of fit*, which is what this example is, you need to give the function a vector of observed counts and a vector of the probabilities for each count.

```
observed <- c(106, 252, 142)

expected <- c(0.25, 0.5, 0.25)

chisq.test(observed, p = expected)
```

```
    Chi-squared test for given probabilities

data:  observed
X-squared = 5.216, df = 2, p-value = 0.07368
```

Chi-squared test with contingency tables

The other main way that Pearson's chi-squared test is used is as a test of *independence* when you have counts related to two or more variables, as we do in our comparision between the numbers of lung cancer and control patients smoking different numbers of cigarettes a day. The basics of how this test works are exactly the same way as the previous version, but the details are slightly more complicated because we have to calculate the expected frequencies as well as the observed one. When we have counts related to more than one variable the data are traditionally presented in a *contingency table*, with the counts related to each variable presented as a separate row or column - in our example the variables are whether or not the patient had lung cancer (presented as a row) and the number of cigarettes smoked per day (presented as a column). Once we have our contingency table then we can calculate the expected values for each *cell* in the table as the row total (the sum of all the values in the row that the cell is in) times the column total divided by the overall total, which is the sum of all the cells in the table. Once we have these expected frequencies we can calculate the test statistic as before.

Here's the table for the Doll and Hill data, with the row and column totals.

Number of cigs	1-4	5-14	15-24	25-49	50+	Row total
Lung cancer	33	250	196	136	32	647
Control	55	293	190	71	13	622
Column totals	88	543	386	207	45	1269

The expected value for the top left cell is (647 x 88) / 1269 = 44.87

The $(O - E)^2 / E$ for this cell is therefore $(33-44.87)^2 / 44.87 = 3.14$

If we repeat this for each cell in the table and sum them we get we get 36.95 as our test statistic. The degrees of freedom for a chi-squared test for independence is (R-1) x (C-1) where R and C are the number of rows and columns in the table respectively (excluding the row and column for the totals if you've included them, so we have (5-1) x (2-1) or 4 df.

```
1 - pchisq(36.95, 4)
```

```
[1] 1.845e-07
```

The number that R returns for our p-value is given in the notation that R uses for very big or very small numbers: this is the equivalent of writing 1.842×10^{-7}, so our p-value is 0.0000001845. This is a very small number, indicating that we would be very unlikely to see a test statistic with a value as great, or greater than this if the patterns in our data had arisen by random chance. On this basis we would reject the null hypothesis, that the distribution of counts of people smoking different numbers of cigarettes is independent of whether they are a lung cancer patient or a control, and cautiously accept the alternative, that the distribution of counts of numbers of cigarettes smoked is different between lung cancer patients and controls.

Of course, the whole point of using R is that you don't have to do your chisquared tests from first principles every time so it's quicker to use the `chisq.test()` function. If you feed this a matrix as its first argument it will assume that you want to carry out a chisquared test for independence on the values in the matrix.

```
chisq.test(cancer)
```

```
    Pearson's Chi-squared test

data:  cancer
X-squared = 36.95, df = 4, p-value = 1.842e-07
```

This gives us our calculated test statistic, the degrees of freedom for the test and the p-value. The latter is very similar to the one we calculated before, with a slight difference in th last significant figure only which arises from R doing its calculations to rather more significant figures than we have done.

If we want to look at the output of the analysis in a bit more detail then we can save an object containing the results of the chi-squared test.

```
cancer.chisq <- chisq.test(cancer)
```

If we do this then when we press enter we don't get any output, but if we type the name of the object we get the same output that we had before:

```
cancer.chisq
```

```
    Pearson's Chi-squared test

data:  cancer
X-squared = 36.95, df = 4, p-value = 1.842e-07
```

That doesn't give us any new information, but there is quite a bit of useful material summarised in the cancer.chisq object that we can ask to see. Our object (cancer.chisq) is in fact stored as a list of several different pieces of information with different names:

```
names(cancer.chisq)
```

```
[1] "statistic" "parameter" "p.value"   "method"
"data.name" "observed"
[7] "expected"  "residuals" "stdres"
```

If we just type the name of the object it tells us the method ("method"), the dataset used ("data.name"), the calculated test statistic ("statistic"), the df ("parameter") and the p-value ("p-value"). These have been selected as the pieces of information that someone doing a test like this is likely to want to see, and if you have a stored object with the results of a different sort of analysis you might be given a different set of information by default. The stored material that we don't automatically get for a chi-squared test are "observed" - the observed data, "expected" - the expected values if there's no difference between groups, "residuals" - the Pearson's residuals for each value in our matrix, calculated as (observed-expected)/√(expected) and "stdres" - the standardised residuals. The residuals will tell us how far each cell in the matrix was from the value expected if the null hypothesis were true.

```
cancer.chisq$residuals
```

```
            1 to 4 5 to 14 15 to 24 25 to 49    50+
Lung cancer -1.772  -1.614 -0.05718    2.965  1.891
Control      1.807   1.646  0.05832   -3.024 -1.928
```

These correspond to what we've already seen from the barplot. When we look at the light smokers (fewer than 15 cigarettes per day) the lung cancer patients have negative residual values and the controls have positive ones - in other words, there are fewer lung cancer patients and more controls reporting relatively light smoking habits. For the 15-24 cigarettes per day group the residuals are close to zero for both groups, but then for the heavy smokers we have positive residuals for the lung cancer patients and negative residuals for the controls, telling us that there are more lung cancer patients reporting heavy smoking habits than expected.

Assumptions and limitations of the chi-squared test

Like most statistical tests, the chi-squared test assumes that each data point is *independent*, meaning that the probability of getting a particular value for one data point is not affected by the values of any other data points. An example of non-independence would be when there are multiple counts from the same individual: if we reared 100 snapdragon plants in the first example, and scored five flowers on each for colour, then the colour of one flower on a plant would be correlated with the colour of the other flowers, meaning that the assumption of independence would be violated and the “real” sample size would be 100, not 500.

A limitation of the chi-squared test is that when the counts in some of the cells of a contingency table are small it becomes less reliable. A rule of thumb is that if any of the cells have counts less than 5 you shouldn't really rely on a chi-squared test and it's advisable to use a *Fisher's exact test* instead.

Are lung cancer patients more likely to be smokers than the controls?

At first glance this seems to be the more obvious analysis of the two, and so looking at these data second might be thought of as a little strange, but in fact these data are rather less straightforward to analyse. This is because, to our 21st Century eyes, there is an astonishingly low number of patients classified as non-smokers in this study. Doll and Hill defined a smoker as being someone who at some point in their life had smoked at least one cigarette a day for a period of at least a year, and of the 1298 male patients included in this study only 29, or 2.2%, were classed as non-smokers. The smokers and non-smokers were not evenly spread between lung cancer patients and controls: as can be seen from this data table from the paper, only two of the lung cancer patients were non-smokers, whereas 27 of the controls were identified as being in this group.

Disease Group	No. of Non-smokers	No. of Smokers
Males:		
Lung-carcinoma patients (649)	2 (0·3%)	647
Control patients with diseases other than cancer (649) ..	27 (4·2%)	622

The table's a bit wonky because the only copy I could get of the paper was a rather bad scan, which does give it a nice antique feel. Back ot the data. We can see a possible pattern in these data, with lung cancer patients being less likely than controls to be non-smokers. What we don't know is how likely this pattern is to have arisen by chance. The counts of non-smokers are only a small fraction of the overall counts, and it's difficult to know whether we might be likely to get such a pattern just by drawing at random from two similar populations with similar, low percentages of smokers.

We could analyse these data using another chi-squared test, but as we noted above the chi-squared test is not really suitable when we have a contingency table with low values in one or more cell, as is the case here. Fortunately we have the option of using a test that takes this into account, namely Fisher's exact test.

Fisher's exact test

Fisher's exact test is an alternative to the chi-squared test which can be used for contingency tables when there are counts below 5 or even of zero in some of the cells in the table. It relies on some fairly complex calculations which we won't go into detail about, and with large counts the test becomes difficult to calculate and may be unreliable, so when the frequencies you're dealing with are fairly large use a chi-squared test and when they're small use a Fisher's exact test. The R function you want is `fisher.test()` which takes a matrix as an argument in just the same way that `chisq.test()` does.

First we need to set up a matrix of our data:

```
smokers <- matrix(data = c(2, 647, 27, 622), byrow =
```

```
T, nrow = 2, ncol = 2)

dimnames(smokers) <- list(c("Lung Cancer", "Control"),
c("Non smoker", "Smoker"))
```

```
smokers
```

```
            Non smoker Smoker
Lung Cancer          2    647
Control             27    622
```

Now we can test whether smoking is distributed homogeneously across lung cancer patients and controls.

```
fisher.test(smokers)
```

```

    Fisher's Exact Test for Count Data

data:  smokers
p-value = 1.281e-06
alternative hypothesis: true odds ratio is not equal
to 1
95 percent confidence interval:
 0.008173 0.285788
sample estimates:
odds ratio
   0.07129
```

We get a little more in the output for the Fisher's exact test than for the chi-square test. Most of our extra information centres around the "odds ratio" - this is ratio of the odds of a patient being a non-smoker in the lung cancer group to the odds of a patient being a non-smoker in the control group. The estimated value from the Fisher's exact test (0.07128) is very close to what we get if we just calculate an odds ratio ourselves.

```
(2/647)/(27/622)
```

```
[1] 0.07121
```

The question that Fisher's exact test is asking is whether the odds ratio is significantly different from one, and you can see that the p-value of 1.281e-06, or 0.000001281 indicates that we should discard our null hypothesis and accept the alternative. In other words, we can be confident that the differences in the frequencies of smokers and non-smokers in the two groups are unlikely to have arisen by random sampling error.

If we look at the analysis in the original Doll and Hill paper, they seem to have arrived at a slightly different answer. Here is the table we saw earlier with the analysis result as well.

Disease Group	No. of Non-smokers	No. of Smokers	Probability Test
Males:			
Lung-carcinoma patients (649)	2 (0·3%)	647	P (exact method) = 0·00000064
Control patients with diseases other than cancer (649) ..	27 (4·2%)	622	

You can see that their p-value is different from ours. Why is this? One thing you can immediately notice when you compare their p-value with ours is that theirs is exactly half of ours. To the experienced statistical detective this is a complete give-away and tells us that while we have used a two-tailed test (alternative hypothesis is that the odds-ratio is not equal to 1), Doll and Hill used a one-tailed test (alternative hypothesis that the odds-ratio is less than one). Nowadays we would regard this as being a bit naughty without good justification, and something that should certainly be explained properly in the paper, but given how long ago it was, and that history has proven Doll and Hill to have been spectacularly right, I think we can let them off the hook and not get the knighthoods posthumously withdrawn. Rest easy guys.

We can do a one-tailed test just to check using the "alternative=" argument in fisher.test().

```
fisher.test(smokers, alternative = "less")
```

```
    Fisher's Exact Test for Count Data

data:  smokers
p-value = 6.403e-07
alternative hypothesis: true odds ratio is less than 1
95 percent confidence interval:
 0.0000 0.2445
sample estimates:
odds ratio
   0.07129
```

Giving us a p-value of 6.403e-07, or 0.00000064, the same as the original paper.

Footnotes

1) Repeating the experiment with a larger sample size is not necessarily going to give a lower p-value because the p-value can increase with a larger sample size as well as decrease. What repeating the experiment with a larger sample size will do is to decrease the amount of uncertainty about whether there is an effect or not. It is sadly quite common to hear people say things like "this isn't quite signficant because the sample size is a bit small" or even worse 'this isn't significant yet but it will be once I've got a few more measurements". Saying something like that is the statistical equivalent of writing "Know-nothing bozo" on your forehead in marker pen.

Chapter 16: Loops

If you want R to do something several times, one of the easiest ways to make it happen is to use a loop: a piece of code which will be repeated a set number of times (it starts at the top, runs through to the bottom and then goes back up to the top - a loop). These have the basic format

```
for (i in 1:X) {

    Some instructions

}
```

What this does is to set up a temporary object called *i* (you can use other names as well but *i* is the most common). The first time the loop is run *i* has the first number in the *(i in 1:X)* argument allocated to it. In this case that number is 1. R will then carry out the set of instructions in the braces, and when it's finished it will add 1 to *i*, and then repeat the instructions and add another 1 to *i*, which acts as a counter. Once *i* has counted up to the number represented by *X* then R knows that it's time to stop repeating the instructions in the loop, and it will go on to evaluate the next set of instructions, if there are any. In other words, R will loop through the set of instructions given in the braces X times.

Here's a simple example. Let's say we have a vector of numbers *X1* which has 10 numbers (3,5,8,3,1,1,12,7,2,4) and we wish to calculate the *cumulative sums* for the vector[1]. In other words, for each number we wish to calculate the sum of that number and all of the preceding numbers, so that our new vector *X2* would be 3, 3+5, 3+5+8, and so on. We can do this by using a loop of code which calculates this figure for each number in the vector in turn.

```
X2 <- numeric(10)

for (i in 1:10) {

    X2[i] <- sum(X1[1:i])

}
```

```
X2
```

```
 [1]  3  8 16 19 20 21 33 40 42 46
```

One of the useful things to know about these loops is that you can refer to *i* as an object in the code that's between the braces. You can see how this works in the code above, where we use *i* as a subscript to make sure that our numbers are going in the right places. We already have a vector set up for our result, called *X2*. For the first time the loop is run, *i*=1 and so the instructions in the loop are the equivalent of

```
X2[1] <- sum(X1[1:1])
```

The first number in X2 is set to be equal to the sum of the first number in X1. For the second time the loop runs *i* has had 1 added to it and so it's now equal to 2, and so the second run of the loop sets the value of the second number in X2 to be equal to the sum of the first and second numbers in X1.

```
X2[2] <- sum(X1[1:2])
```

For the third run of the loop *i* is set to 3 and this time it's the third number in X2 which is set to the sum of the first 3 numbers in X1.

```
X2[3] <- sum(X1[1:3])
```

This continues until *i*=10, after which R will stop going back to the beginning of the loop. If there are any more instructions after the loop it will go on to execute them, as in the case of our final instruction above which is to tell us what's in the X2 object, otherwise it'll stop.

For another example let's assume that you want to illustrate how the mean of a sample of numbers converges on the true population mean as sample size increases. The example below will start with *i*=1, meaning that it will generate one random number drawn from a population with mean 0 and standard deviation 1, calculate the mean of that number and store the result as the first number in a

vector called *output*. It will then add one to the counter, meaning that *i* is now equal to 2. It will loop through the expression again with *i*=2, so it will generate two random numbers from a normal distribution with mean zero and standard deviation 1, calculate the mean of the two numbers and store it as the second number in *output*. Because it's looped through the instructions in braces again *i* will get one added again so *i*=3, three random numbers will be generated and the mean stored as the third number in “output”. This will continue until *i*=2000, after which R will stop.

```
output <- numeric(2000)
```

As before, we have to set up a vector for our results before we run our loop.

```
for (i in 1:2000) {

    output[i] <- mean(rnorm(i, 0, 1))
}
```

We can now plot a graph to show how the mean of the sample converges on the mean of the population from which the sample was drawn. Because we only have one vector of numbers R will plot them on a scatterplot with the x-variable being their position in the vector, which in this case happens to be the same as the sample size because of the way we set the loop up. We can add a line indicating the “true” population mean that the data were drawn from using the abline() function.

```
plot(output, pch = 16, cex = 0.4, xlab = "Sample
size", ylab = "Mean", font.lab = 2)

abline(0, 0, lwd = 0.5)
```

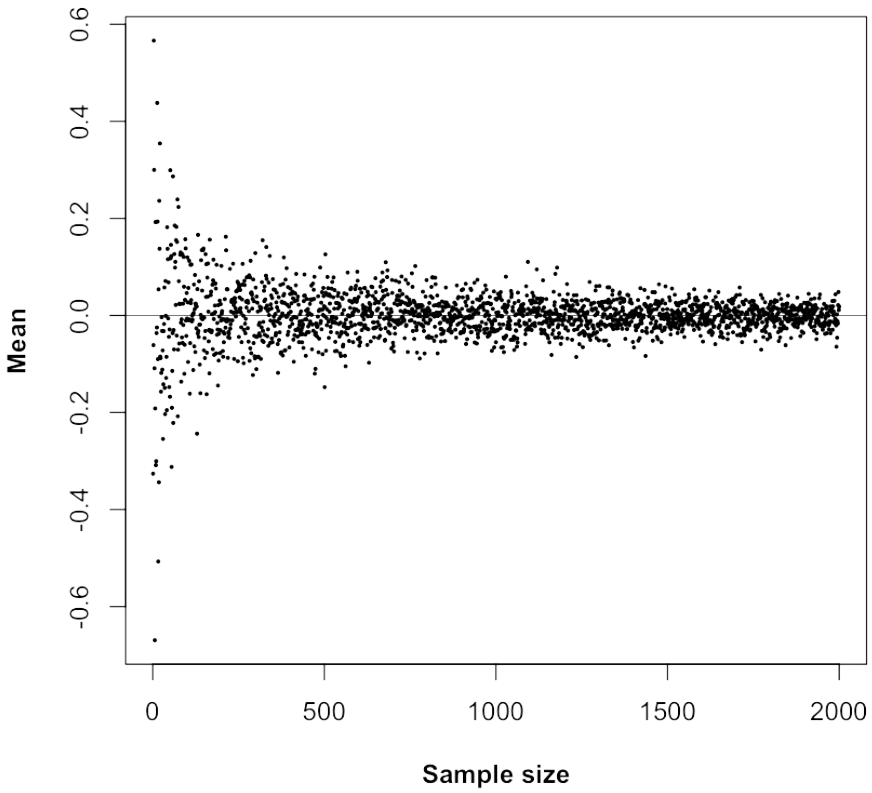


Figure 1: How the estimated mean converges on the true mean of a sampled population as the sample size increases

A more complicated for() loop

If you want to draw not just the means but also the 95% confidence intervals it doesn't take a great deal of adjustment to the code. Since we're generating two numbers per iteration we need a matrix to store the output rather than a vector, and the largest sample is 100 rather than 2000 this time just to keep the plot at the end readable.

```
output <- matrix(data = NA, nrow = 100, ncol = 3)

colnames(output) <- c("N", "Mean", "CI")
```

The `for()` loop is now a bit more complicated. For each step we're setting up an object to store the randomly drawn sample with sample size *i*, and then we're calculating the mean and 95% confidence intervals separately. As we've seen in

previous chapters, one function which we'll need is the `qt()` function which returns the quantiles of the t-distribution which we need to get our confidence limits.

```
for (i in 1:100) {

    X1 <- rnorm(i, 0, 1)

    output[i, 1] <- i

    output[i, 2] <- mean(X1)

    output[i, 3] <- (sd(X1)/sqrt(i)) * qt(0.975, i -
1)
}
```

What happens here is that each time R runs through the loop the first thing it meets is this instruction

```
X1 <- rnorm(i, 0, 1)
```

which it carries out by drawing *i* random numbers from a standard normal distribution, and allocating those numbers as an object called *X1*. Note that *X1* will get overwritten with a new set of numbers each time the loop is run so think of it as just a disposable temporary store for our sampled numbers. The next instruction tells R to write the sample size (which is equal to *i*) in row *i* of the first column of the output matrix.

```
output[i, 1] <- i
```

The mean of *X1* is then written to the second column of the output matrix, on the row corresponding to *i*.

```
output[i, 2] <- mean(X1)
```

The final expression in the loop is the calculation of the 95% confidence interval for the mean of the sample of size *i*, which gets written to row *i* in the third

column of the output matrix. The calculation of the 95% CI looks a bit complicated - it's explained in the “Visualising your data” section of chapter 12.

```
output[i, 3] <- (sd(X1)/sqrt(i)) * qt(0.975, i - 1)
```

This is the last instruction in the loop, so R will add 1 to *i* and will run the loop again, until it gets to the maximum value given in the `for()` command, at which point it will stop. If you run this you'll get a warning message:

```
Warning message: In qt(0.975, i - 1) : NaNs produced
```

“NaN” stand for “Not a Number” and means that you've asked R to do something that can't be done. In this case when *i=1* then *i-1* is equal to zero which causes a division by zero, so we can't calculate a confidence interval for our mean from a sample size of 1. Not really a problem. We can quickly check that all has gone well by looking at the top six rows of the output matrix using the head() function. As we expected we don't have a value for the CI when N is 1 but that makes sense and everything else looks OK.

```
head(output)
```

```
     N      Mean      CI
[1,] 1 -2.15407      NA
[2,] 2 -0.24004 3.0622
[3,] 3  1.51943 0.6633
[4,] 4  0.45943 1.1540
[5,] 5  0.01451 1.2496
[6,] 6  0.77510 0.7642
```

How about visualising the relationship between sample size, mean and 95% CI? Here's some code that will plot the means with error bars.

```
plot(output[, 1], output[, 2], pch = 16, cex = 0.3,
ylim = c(-2, 2), xlab = "Sample size",
    ylab = "Mean and 95% CI", font.lab = 2)
```

```
arrows(output[, 1], output[, 2], output[, 1], output[,
2] + output[, 3], angle = 90,
    length = 0.01, lwd = 0.5)

arrows(output[, 1], output[, 2], output[, 1], output[,
2] - output[, 3], angle = 90,
    length = 0.01, lwd = 0.5)

abline(0, 0, lwd = 0.5)
```

That's a bit impenetrable so let's go through it bit-by-bit.

```
plot(output[, 1], output[, 2], pch = 16, cex = 0.3,
ylim = c(-2, 2), xlab = "Sample size",
    ylab = "Mean and 95% CI")
```

`plot(output[,1],output[,2]` plots the means against the sample sizes, `pch=16` with filled circles as the plot symbols `cex=0.3` plot symbols drawn 3/10ths of the normal size `ylim=c(-2,2)` y axis scaled from –2 to 2 `xlab="Sample size",ylab="Mean and 95% CI"` x and y axis labels specified.

```
arrows(output[, 1], output[, 2], output[, 1], output[,
2] + output[, 3], angle = 90,
    length = 0.01, lwd = 0.5)
```

Uses the `arrows()` function (see Chapter 5) to draw in the positive error bars. `output[,1],output[,2]` gives the X and Y coordinates for each arrow (or error bar) to start at and `output[,1],output[,2]+output[,3]` gives the X and Y coordinates for them to finish at. Because they're drawn vertically the X coordinates (`output[,1]`, the sample sizes) are the same for both start and finish points. The Y coordinates for the start points are the means for each sample (output[,2]), and the Y coordinates for the finish point for each error bar is the mean plus the 95% CI (`output[,2]+output[,3])`. `angle=90` means

that the lines for the arrowhead are drawn at 90 degrees to the main line, length=0.01 makes them quite short (together these last two arguments give us the flat bar at the end of each error bar) and finally `lwd=0.5` makes the line half the normal thickness.

```
arrows(output[, 1], output[, 2], output[, 1], output[,
2] - output[, 3], angle = 90,
    length = 0.01, lwd = 0.5)
```

This draws the negative error bars (the ones dropping down from the points representing the mean) and is the same as the previous expression except that the Y coordinates for the finish points are calculated by subtracting the 95% CI from the mean rather than adding it (`output[,2]-output[,3]` instead of `output[,2]+output[,3]`)

```
abline(0, 0, lwd = 0.5)
```

Draws a horizontal line to show the population mean from which these samples were drawn. Putting all of these together gets us this graph:

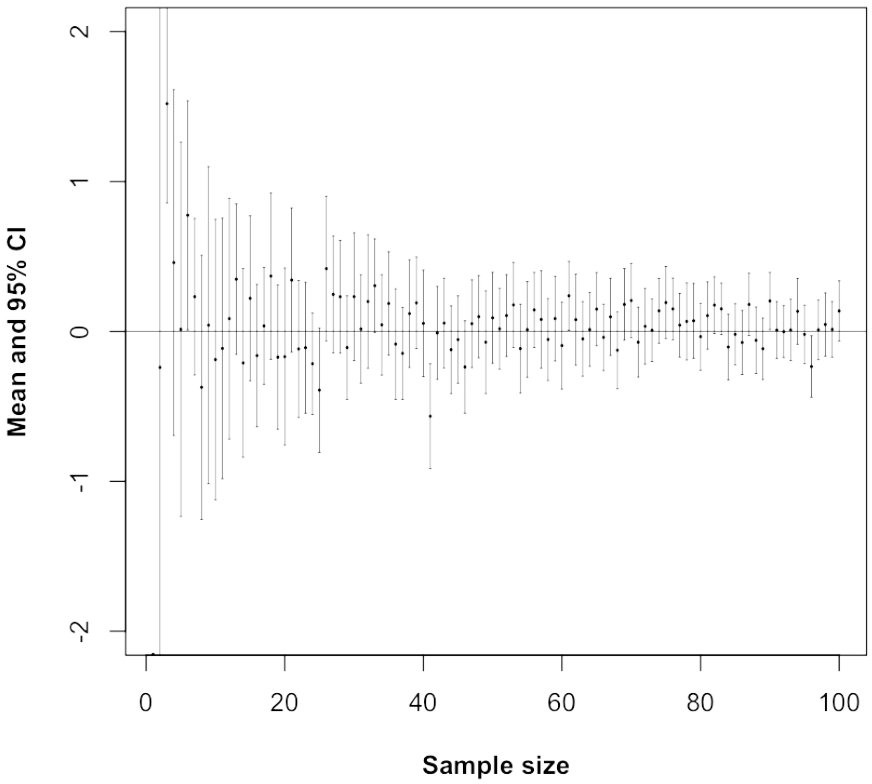


Figure 2: Means and 95% confidence intervals for 100 samples with sample sizes from 1 to 100

Writing a mathematical model using a for() loop

`for()` loops are quite easy to understand and easy to programme, but they can be slow, especially when you need to do a lot of calculations. A lot of the time in R you're better advised to use functions like `apply()` and its relatives when you want to do a lot of calculations (see chapter 15), but one part of programming that needs something like a `for()` loop is when you are doing something *iteratively*: in other words, when the result of one repeat of your operation depends on the output of the previous repeat. One classic example of these is mathematical modelling of populations of animals or other organisms. Here I'm going to show you how you can easily programme a famous model of an insect population and

a parasitic wasp which feeds on it. The model is called the Nicholson-Bailey model, and when it's written down as a set of equations it looks like this.

$$H_{t+1} = \lambda H_t e^{-aP_t}$$

$$P_{t+1} = gH_t\left\{1 - (e^{-aP_t})\right\}$$

This is a mathematical model which predicts the changes in population number over time for an insect host and a parasitic wasp (a “parasitoid”) which attacks the host and develops on it. The model is one which runs as a set of discrete time steps which we assume are equal to the generation time of the host and its parasitoid. If we know the size of the host population (*H*) and the parasitoid population (*P*) at one time step these two equations will tell us what the populations of the two insects will be at the next time step. If we iterate this model - run it repeatedly, giving it the numbers from the previous run each time - it should tell us how the host and parasitoid numbers change over time. If you want to know more there is lots of information on the internet about this model and any decent ecology textbook will tell you about it. If you want lots of detail Mills & Getz (1996), Ecological Modelling 92: 121-143 has a good description of this model and many others.

The various parts of the model are:

H_t is the size of the host population at time *t*, H_{t+1} is the population at the next time step.

P_t is the size of the host population at time *t*, P_{t+1} is the population at the next time step.

λ is the per capita rate of increase of the host population: the number of new hosts that one host will produce next generation

a is the “area of attack” of the parasitoid - in other words, the proportion of the host population that it is able to parasitise

g is the number of new parasitoids produced from each parasitised host. This is often called “c” but I'm using “g” because “c” is the name of a function in R.

This might sound a bit technical and complicated but it is remarkably easy to programme in R using a `for()` loop. Firstly we have to give our various parameters and variables values.

```
t <- 30
lambda <- 2
g <- 1
a <- 0.05
```

We're going to run the model for 30 generations (t<-30). Each host can make 2 new hosts per generation (lambda<-3), but if it's parasitised it will be made into a single new parasitoid instead (g<-1). The area of attack is set at an arbitrary 0.05. Now we need to tell the model our starting conditions - the initial populations of hosts and parasitoids.

```
Hzero <- 20
Pzero <- 5
```

In this case we're starting with a population of 20 hosts and two parasitoids.

```
H <- c(Hzero, numeric(t))

P <- c(Pzero, numeric(t))

Time <- 0:t
```

This sets up three vectors. H has the initial population size of the hosts and then

30 empty slots for the model output to go in, and P is the same except it has the initial size of the parasitoid population. Time records the time step for each pair of values. Now for the actual model.

```
for (i in 1:t) {

    H[i + 1] <- lambda * H[i] * exp(-a * P[i])

    P[i + 1] <- g * H[i] * (1 - exp(-a * P[i]))
}

output <- cbind(Time, H, P)
```

Once we've specified our parameters and starting values for the variables, it only takes two lines within a `for()` loop to programme the actual model. Let's go through them in detail.

```
for (i in 1:t)
```

This will run the loop *t* times. The counter *i* will have 1 added to it each time the loop runs.

```
H[i + 1] <- lambda * H[i] * exp(-a * P[i])
```

This calculates the host population at time *t+1*, using the value of *i* as the time. The expression on the right of the allocation symbol does the arithmetic, and you can see that it uses the values of H and P at the previous time step to do so. The calculated value for the host population is then stored in the H vector in the slot indexed by the value of *i* plus one. In other words, for each loop it calculates

the host population at the next timestep and stores it in the H vector.

```
P[i + 1] <- g * H[i] * (1 - exp(-a * P[i]))
```

This does exactly the same, but for the parasitoid population.

```
output <- cbind(Time, H, P)
```

Finally we produce an output matrix by using cbind() to put our three vectors of number of generations, host population and parasitoid population together. We can use our outut matrix plot a graph to show how the populations change over time.

```
max <- max(c(H, P))

plot(output[, 1], output[, 2], type = "l", xlab =
"Generations", ylab = "Population",
    ylim = c(0, max), font.lab = 2, lwd = 1.5, col =
"steelblue", main = "Time series")

points(output[, 1], output[, 3], type = "l", col =
"brown3", lwd = 1.5)

legend("topleft", legend = c("Host", "Parasitoid"),
col = c("steelblue", "brown3"),
    lwd = 1.5)
```

Here's the finished script with annotation.

```
#Starting conditions

t <- 30

lambda <- 2

g <- 1

a <- 0.05

Hzero <- 20

Pzero <- 5



#Set up host and parasitoid population vectors and
vector for time
H <- c(Hzero, numeric(t))

P <- c(Pzero, numeric(t))

Time <- 0:t



#Loop to calculate populations at each generation
for (i in 1:t) {

    H[i + 1] <- lambda * H[i] * exp(-a * P[i])

    P[i + 1] <- g * H[i] * (1 - exp(-a * P[i]))
}



#Output matrix
output <- cbind(Time, H, P)
```

```
#Find maximum population value
max <- max(c(H, P))



#PLot graph
plot(output[, 1], output[, 2], type = "l", xlab =
"Generations", ylab = "Population",
    ylim = c(0, max), font.lab = 2, lwd = 1.5, col =
"steelblue", main = "Time series")

points(output[, 1], output[, 3], type = "l", col =
"brown3", lwd = 1.5)

legend("topleft", legend = c("Host", "Parasitoid"),
col = c("steelblue", "brown3"),
    lwd = 1.5)
```

Here's the output

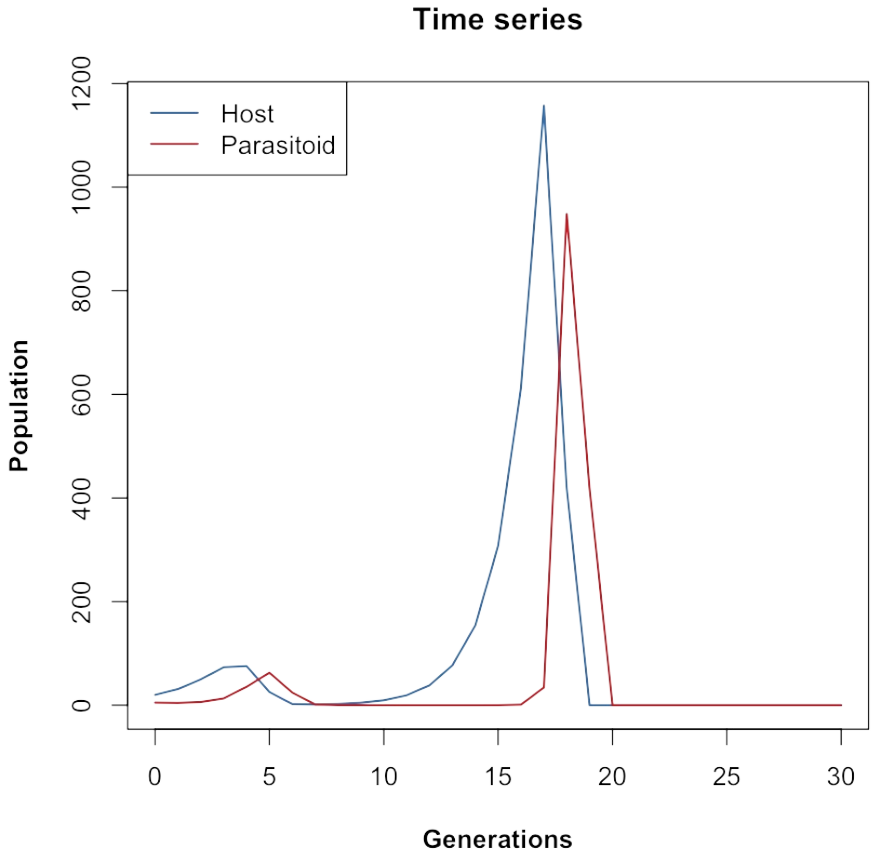


Figure 3: Population sizes for a host and parasitoid population as predicted by a Nicholson-Bailey model

The host and parasitoid population undergo a couple of unstable oscillations until both go extinct. This is an important early model of parasitoid-host population dynamics which demonstrated that such systems, and by extension predator-prey systems, tend to be fundamentally unstable. An awful lot of ecological research since then has focussed on finding out what it is that stabilises real predator prey systems which obviously don't all go extinct in a few generations.

Other useful loops

while() and repeat()

There are a couple of other ways of using loops: these are the while() and

repeat() functions. while() will work in much the same way as a for() loop but will carry on repeating so long as a logical expression in the brackets remains true. Let's say that you have a set of measurements of weight for an animal and you want to extract all of the measurements that were made before the weight became greater than 80g, which is a cut-off point above which the animal is considered to be obese. You can do this with a while() loop with the following code.

```
animal1.wt <- c(56, 55, 61, 68, 67, 75, 77, 69, 75,
79, 81, 79, 82, 77, 82, 81)

i <- 1

wt.before80 <- numeric()



while (animal1.wt[i] < 80) {

    wt.before80 <- c(wt.before80, animal1.wt[i])

    i <- i + 1
}



wt.before80
```

```

```

```
 [1] 56 55 61 68 67 75 77 69 75 79
```

repeat() will carry on executing a set of instructions forever unless it's told to stop, which can be done by telling it to break - this is something that can be used in all the different loop functions to stop a looping operation. We can use the same example as before.

```
animal1.wt <- c(56, 55, 61, 68, 67, 75, 77, 69, 75,
79, 81, 79, 82, 77, 82, 81)

i <- 1

wt.before80 <- numeric()


repeat {

    if (animal1.wt[i] >= 80) {

        break
    }

    wt.before80 <- c(wt.before80, animal1.wt[i])

    i <- i + 1

}


wt.before80
```

```
 [1] 56 55 61 68 67 75 77 69 75 79
```

This time we have an `if()` statement (see chapter 17) contained in the repeat loop, and R will keep running the loop until either it comes across a number in animal1.wt which is greater than or equal to 80, in which case the `if()` statement becomes true and break is executed, or if there are no numbers in animal1.wt that meet the inequality the loop will repeat until i becomes greater than the length of animal1.wt, at which point you'll get an error message.

Exercises

NB some of these exercises can be done equally well, if not better using other code without loops, but if you're learning how to write loops then they are nice examples of how they work.

Comparing t-test results as means and standard deviations differ

- Set up a matrix (call it something like “mat1”) with 10 columns and 500 rows, with the data being “NA” for the moment
- Using a `for()` loop, write a set of 10 randomly drawn numbers into each row. The numbers should be drawn from a normal distribution with standard deviation zero. Start with a mean of 0.01 and for each row of the table, increase the mean of the distribution the numbers were drawn from by 0.01. Two hints - remember you can access a complete row of a matrix by using a subscript like mat1[5,] - this will access the 5th row of a matrix called mat1. Second hint - if you want a value that starts at 0.01 and increases by 0.01 for each iteration of the loop, just divide *i* by 100.
- Set up a second matrix (“mat2”?) with the same number of rows and columns. Fill it with random numbers as before but this time start with a mean of 5, and decrease the mean of the normal distribution the numbers are drawn from by 0.01 each time.
- Now we want to calculate a t-test to compare the mean of each row of mat1 with the mean of the corresponding row of mat2. Ultimately we'd like to draw a graph to show how the p-value varies with the difference in the means the populations were drawn from, so we need the p-value for each t-test. You can do this by making a new vector to store the p-values in, and then using a loop to perform a t-test to compare each row in turn. Remember that you can access the p-value of a t-test if you allocate the t-test output to an object and then use object.name$p.value.
- To make our graph nice we need a vector of the same length as our results vector which has the differences between the means of the

populations our samples were drawn from, to use as an x-axis. You can do this again with a `for()` loop (NB we don't want the actual means of the samples, we want the means of the populations they came from).

- Finally, plot a graph of the differences between the means of the populations on the x-axis and the p-values on the y-axis. It's best to either draw a line graph (type="l") or use very small points because you're plotting quite a bit of data.
- If you're really keen, keep your graph and go back to your script and change the code so that the samples are drawn from populations with standard deviations of 2 rather than one. Run the script again and compare the graphs.

Another population model

In 1973 Maynard-Smith and Slatkin published a mathematical model which describes a single population growing in the absence of any natural enemies, but with varying degrees of *intraspecific competition* - competition between members of the same species for food or other resources. In particular, this model has a parameter *b* which allows you to change the nature of the competition from *undercompensating*, ($b<1$) where competition has quite mild effects, to *compensating*, to *overcompensating*, where competition has strong effects - essentially with overcompensating competition if the population is large then most of the individuals will die, rather than just enough individuals to reduce the population down to some equilibrium level. The model looks like this.

$$N_{t+1} = \frac{RN_t}{1 + (\alpha N_t)^b}$$

N_t is the size of the population at time t, N_{t+1} is the population at the next time step.

R is the amount that the population grows by in the absence of competition

b is the parameter that determines the extent by which density dependence is under- or over-compensating. Values of *b<1* give undercompensating density dependence, values of *b>1* give overcompensating density dependence

a is the strength of intraspecific competition, with small values leading to less competition

You can see that, like the previous example of a population model, this one works in *discrete time*, with the model running as a series of time steps and the size of the population at each time step depending on the population size at the previous step.

- Using a `for()` loop, try to write a script that will run this model for you for 50 time steps. Remember that you'll have to specify your parameter values and the staring size of the population to begin with. I suggest the following values:

b=1, *a*=0.001, *R*=3, and a starting population of 20.

- Use the output from your model to draw a nice graph showing how population size changes over time.
- Now go back to your script and run it again with values of \(b\) ranging from 1 to 8. You should see that the behaviour of the population changes from a steady increase to an equilibrium, to damped oscillations to equilibrium, to cycles and then finally to complex, unpredictable fluctuations - *deterministic chaos*.

Footnotes

1) This is a simple example to introduce the concept of a loop and show how it works. There is actually an R function `cumsum()` which will do the same thing for you with a lot less effort.

Table of Contents