Nordic School of Public Health
Computer Software in Epidemiology / Statistical Methods in Epidemiology

Open Source Solutions - *R*

Mark Myatt, December 2001

Introduction

These notes are intended as a practical introduction to using the ***R*** environment for data analysis and graphics to work with epidemiological data. Topics covered include univariate statistics, simple statistical inference, charting data, two-by-two tables, chi-square for trend, logistic regression, survival analysis, computer-intensive methods, and extending ***R*** using user-provided functions. You should follow this material if you are reasonably familiar with the mechanics of statistical estimation (e.g. calculation of odds-ratios and confidence intervals) and require a system that can perform simple or complex analyses to your exact specifications. A more thorough and general introduction can be found in the file **`R-intro.pdf`** which is installed with the system. The file **`refman.pdf`** contains a complete reference to the ***R*** system and language.

These notes are split into ten sections:

> **Introduction** : You are reading this section now!
>
> **Introducing *R*** : Some information about the ***R*** system, the way the ***R*** system works, how to get a copy of ***R***, and how to start ***R***.
>
> **Exercise 1** : Read a dataset, producing descriptive statistics, charts, and perform simple statistical inference. The aim of the exercise is for you to become familiar with ***R*** and some basic ***R*** functions and objects.
>
> **Exercise 2** : In this exercise we explore how to manipulate ***R*** objects and how to write functions that can manipulate and extract data and information from ***R*** objects and produce useful analyses.
>
> **Exercise 3** : In this exercise we explore how ***R*** handles generalised linear models using the example of logistic regression as well as seeing how R can perform stratified (Mantel-Haenszel) analysis.
>
> **Exercise 4** : In this exercise we use ***R*** to analyse a small dataset using the methods introduced in the previous exercises.
>
> **Exercise 5** : In this exercise we explore how ***R*** can be extended using add-in packages. Specifically, we will use an add-in package to perform some basic survival analysis.
>
> **Exercise 6** : In this exercise we explore how to make your own ***R*** functions behave like ***R*** objects so that they return a data-structure that can be manipulated or interrogated by other ***R*** functions.
>
> **Exercise 7** : In this exercise we explore how you can use ***R*** to produce custom graphical functions.
>
> **Exercise 8** : In this exercise we explore some more graphical functions and create custom graphical functions that produce two variable plots, pyramid charts, *Pareto* charts, and charts with error bars.
>
> **Exercise 9** : In this exercise we explore some more data management and analysis functions.
>
> **Exercise 10** : In this exercise we explore ways of implementing computer-intensive methods, such as the *bootstrap*, using standard ***R*** functions.

If you are interested in a system that is flexible, can be tailored to produce exactly the analysis you want, provides modern analytical facilities, and have a basic understanding of the mechanics of hypothesis testing and estimation then you should consider following this module.

Introducing *R*

R is a system for data manipulation, calculation, and graphics. It provides:

Facilities for data handling and storage

A large collection of tools for data analysis

Graphical facilities for data analysis and display

A simple but powerful programming language

R is often described as an *environment* for working with data. This is in contrast to a *package* which is a collection of very specific tools. ***R*** is not strictly a statistics system but a system that provides many classical and modern statistical procedures as part of a broader data-analysis tool. This is an important difference between ***R*** and other 'statistical' systems. In ***R*** a statistical analysis is usually performed as a series of steps with intermediate results being stored in objects. Systems such as SPSS and SAS provide copious output from (e.g.) a regression analysis whereas ***R*** will give minimal output and store the results of a fit for subsequent interrogation or use with other ***R*** functions. This means that ***R*** can be tailored to produce exactly the analysis and results that you want rather than produce an analysis designed to fit all situations.

R is a language based product. This means that you interact with ***R*** by typing commands such as:

```
table(SEX, LIFE)
```

rather than by using menus, dialogue boxes, selection lists, and buttons. This may seem to be a drawback but means that the system is considerably more flexible than one that relies on menus, buttons, and boxes. It also means that every stage of your data management and analysis can be recorded and edited and re-run at a later date. It also provides an audit trail for quality control purposes.

R is available under UNIX (including LINUX), most versions of the Macintosh operating system, and on the 95, 98, Millennium, NT, 2000 versions of Microsoft Windows. The method used for starting ***R*** will vary from system to system. On UNIX systems you may need to issue the `R` command in a terminal session or click on an icon or menu option if your system has a windowing system. On Microsoft Windows systems there will usually be an icon on the 'Start' menu or desktop.

R is an *open source* system and is available under the general public license (GPL) which means that it is available for free but that there are some restrictions on how you are allowed to distribute the system and how you may charge for bespoke data analysis solutions written using the ***R*** system.

R is available for download via the Internet from:

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

This is also the best place to get extension packages and documentation. You may also subscribe to the ***R*** mailing lists from this site. ***R*** is supported through mailing lists. The level of support is at least as good as for commercial packages. It is typical to have any queries answered in a matter of a few hours.

Even though ***R*** is a free package it is more powerful than most commercial packages. Many of the modern procedures found in commercial packages were first developed and tested using ***R*** or ***S*** (the commercial equivalent of ***R***).

Introducing *R*

When you start ***R*** it will issue a prompt when it expects user input. The default prompt is:

```
>
```

This is where you type commands that instruct ***R*** to (e.g.) read a data file, recode data, produce a table, or fit a regression. For example:

```
> table(SEX, LIFE)
```

If a command you type is not complete then the prompt will change to:

```
+
```

on subsequent lines until the command is complete:

```
> table(
+ SEX, LIFE
+ )
```

Previous commands can be recalled and edited using the ↑ and ↓ keys.

Output that has scrolled off the top of the output / command window can be recalled using the window scroll bars or by using the PG UP and PG DN keys (one page at a time) or the CTRL + ↑ and CTRL + ↓ keys (one line at a time). The keys used to scroll through output may differ between operating systems.

Output can be saved using the **`sink()`** function with a file name:

```
sink("results.out")
```

To start recording output, and without a file name to stop recording output:

```
sink()
```

You can also use 'clipboard' functions such as copy and paste to (e.g.) copy then paste selected chunks of output into an editor or word processor running alongside ***R***.

All the sample data files used in the exercises are space delimited text files using the general format:

```
"ID" "AGE" "IQ"
1 39 94
2 41 89
3 42 83
4 30 99
5 35 94
6 44 90
7 31 94
8 39 87
```

R has facilities for dealing with files in different formats including (through the use of extension packages) ODBC (open database connectivity) data sources and STATA files.

Exercise 1 : Getting acquainted with *R* for data analysis

In this exercise we will use ***R*** to read a dataset and produce some descriptive statistics, produce some charts, and perform some simple statistical inference. The aim of the exercise is for you to become familiar with ***R*** and some basic ***R*** functions and objects.

The first thing we will do, after starting ***R***, is issue a command to retrieve an example dataset:

```
fem <- read.table("fem.dat", header = TRUE)
```

This command illustrates some key things about the way ***R*** works. We are instructing ***R*** to assign (using the **<-** operator) the output of the **read.table()** function to an object called **fem**. The **fem** object will contain the data held in the file **fem.dat** as an ***R*** data.frame object:

```
class(fem)
```

You can inspect the contents of the **fem** data.frame (or any other ***R*** object) just by typing its name:

```
fem
```

Note that the **fem** object is built from other objects. These are the named vectors (columns) in the dataset:

```
names(fem)
```

The data consist of 118 records of eight variables for female psychiatric patients. The columns in the dataset are as follows:

```
ID        Patient ID
AGE       Age in years
IQ        IQ score
ANXIETY   Anxiety (1=none, 2=mild, 3=moderate, 4=severe)
DEPRESS   Depression (1=none, 2=mild, 3=moderate, 4=severe)
SLEEP     Sleeping normally (1=yes, 2=no)
SEX       Lost interest in sex (1=yes, 2=no)
LIFE      Considered suicide (1=yes, 2=no)
WEIGHT    Weight change (kg) in previous 6 months
```

The first ten records of the **fem** data.frame are:

ID	AGE	IQ	ANXIETY	DEPRESS	SLEEP	SEX	LIFE	WEIGHT
1	39	94	2	2	2	2	2	2.23
2	41	89	2	2	2	2	2	1.00
3	42	83	3	3	3	2	2	1.82
4	30	99	2	2	2	2	2	-1.18
5	35	94	2	1	1	2	1	-0.14
6	44	90	NA	1	2	1	1	0.41
7	31	94	2	2	NA	2	2	-0.68
8	39	87	3	2	2	2	1	1.59
9	35	-99	3	2	2	2	2	-0.55
10	33	92	2	2	2	2	2	0.36

You may check this by asking ***R*** to display all columns of the first ten records in the **fem** data.frame:

```
fem[1:10, ]
```

NA is a special value meaning *not available* or *missing*.

Exercise 1 : Getting acquainted with *R* for data analysis

You can access the contents of a single columns by name:

```
fem$IQ
fem$IQ[1:10]
```

The **$** sign is used to separate the name of the data.frame and the name of the column of interest. Note that ***R*** is case-sensitive so that **IQ** and **iq** are **not** the same.

You can also access rows, columns, and individual cells by specifying row and column positions. For example, the **IQ** column is the third column in the **fem** data.frame:

```
fem[ ,3]
fem[9, ]
fem[9,3]
```

There are missing values in the **IQ** column which are all coded as **-99**. Before proceeding we must set these to the special **NA** value:

```
fem$IQ[fem$IQ == -99] <- NA
```

The term inside the square brackets is called an *index* and is used to refer to subsets of data held in an object. In this situation we are instructing ***R*** to set the contents of the **IQ** variable to **NA** if the contents of the **IQ** variable is **-99**. Check that this has worked:

```
fem$IQ
```

We can now compare the groups who have and have not considered suicide by tabulating summary statistics of the other columns for the two groups. For example, for **IQ**:

```
by(fem$IQ, fem$LIFE, summary)
```

Look at the help for the **by()** function:

```
help(by)
```

Note that you may use **?by** as a shortcut for **help(by)** .

The **by()** function applies another function (in this case the **summary()** function) to a column in a data.frame (in this case **fem$IQ**) split by the value of another variable (in this case **fem$LIFE**).

It can be tedious to always have to specify a data.frame each time we want to use a particular variable. We can fix this problem by attaching the data.frame:

```
attach(fem)
```

We can now refer to the columns in the **fem** data.frame without having to specify the name of the data.frame. This time we will produce summary statistics for **WEIGHT** by **LIFE**:

```
by(WEIGHT, LIFE, summary)
```

We can view the same data as a box and whisker plot:

```
boxplot(WEIGHT ~ LIFE)
```

We can add axis labels and a title to the graph:

```
boxplot(WEIGHT ~ LIFE, xlab = "Life", ylab = "Weight",
        main = "Weight Change BY Considered Suicide")
```

The groups do not seem to differ much in their medians and the distributions appear to be reasonably symmetrical about their medians with a similar spread of values.

We can look at the distribution as histograms:

```
hist(WEIGHT[LIFE == 1])
hist(WEIGHT[LIFE == 2])
```

and check the assumption of normality using quantile-quantile plots:

```
qqnorm(WEIGHT[LIFE == 1]); qqline(WEIGHT[LIFE == 1])
qqnorm(WEIGHT[LIFE == 2]); qqline(WEIGHT[LIFE == 2])
```

or by using a formal test:

```
shapiro.test(WEIGHT[LIFE == 1])
shapiro.test(WEIGHT[LIFE == 2])
```

Remember that we can use the **`by()`** function to apply a function to a data.frame, including statistical functions such as **`shapiro.test()`**:

```
by(WEIGHT, LIFE, shapiro.test)
```

We can also test whether the variances differ significantly using Bartlett's test for the homogeneity of variances:

```
bartlett.test(WEIGHT, LIFE)
```

There is no significant differences between the two variances.

Having checked normality and homogeneity of variance assumptions we can proceed to carry out a t-test:

```
t.test(WEIGHT[LIFE == 1], WEIGHT[LIFE == 2], var.equal = TRUE)
```

There is no evidence that the two groups differ in their weight change in the previous six months.

Note that we could still have performed a t-test if the variances were not homogenous by setting the **`var.equal`** parameter to the **`t.test()`** function to **`FALSE`**:

```
t.test(WEIGHT[LIFE == 1], WEIGHT[LIFE == 2], var.equal = FALSE)
```

or performed a non-parametric test:

```
wilcox.test(WEIGHT[LIFE == 1], WEIGHT[LIFE == 2])
```

An alternative, and more general, non-parametric test is:

```
kruskal.test(WEIGHT, LIFE)
```

Exercise 1 : Getting acquainted with *R* for data analysis

We can use the **table()** function to examine the differences in depression between the two groups:

```
table(DEPRESS, LIFE)
```

The two distributions look very different from each other. We can test this using a chi-square test on the table:

```
chisq.test(table(DEPRESS, LIFE))
```

Note that we passed the output of the **table()** function directly to the **chisq.test()** function. We could have saved the table as an object first and then passed the object to the **chisq.test()** function:

```
tab <- table(DEPRESS, LIFE)
chisq.test(tab)
```

The **tab** object contains the output of the **table()** function:

```
class(tab)
tab
```

We can pass this table to another function. For example:

```
fisher.test(tab)
```

When we are finished with the tab object we can delete it using the **rm()** function:

```
rm(tab)
```

You can see a list of available objects using the **ls()** function:

```
ls()
```

This should just show the **fem** object.

We can examine the association between considering suicide and loss of interest in sex in the same way:

```
tab <- table(SEX, LIFE)
tab
fisher.test(tab)
```

Note that with a two-by-two table the **fisher.test()** function produces an estimate of and confidence intervals for the odds ratio. Again, we will delete the **tab** object:

```
rm(tab)
```

Note that we could have performed the Fisher exact test without creating the **tab** object by passing the output of the **table()** function directly to the **fisher.test()** function:

```
fisher.test(table(SEX, LIFE))
```

Choose whichever method you find easiest but remember that it is easy to save the results of any function for later use.

Exercise 1 : Getting acquainted with *R* for data analysis

We can explore the correlation between two variables using the **cor()** function:

```
cor(IQ, WEIGHT, use = "pairwise.complete.obs")
```

or by using a scatter plot:

```
plot(IQ, WEIGHT)
```

and by a formal test:

```
cor.test(IQ, WEIGHT)
```

With some functions you can pass a whole data.frame rather than a list of variables:

```
cor(fem, use = "pairwise.complete.obs")
pairs(fem)
```

The output can be a little confusing particularly if it includes categorical or record identifying variables. To get round this we can create a new object that contains only the columns we are interested in using the column binding **cbind()** function:

```
newfem <- cbind(AGE, IQ, WEIGHT)
cor(newfem, use = "pairwise.complete.obs")
pairs(newfem)
```

When we have finished with the **newfem** object we can delete it:

```
rm(newfem)
```

Note that there was no real need to create the **newfem** object as we could have fed the output of the **cbind()** function directly to the **cor()** or **pairs()** function

```
cor(cbind(AGE, IQ, WEIGHT), use = "pairwise.complete.obs")
pairs(cbind(AGE, IQ, WEIGHT))
```

It is probably easier to work with the **newfem** object rather than having to retype the **cbind()** function. This is particularly true if you wanted to continue with an analysis of just the three variables.

The relationship between **AGE** and **WEIGHT** can be plotted using the **plot()** function:

```
plot(AGE, WEIGHT)
```

And tested using the **cor()** and **cor.test()** functions:

```
cor(AGE, WEIGHT, use = "pairwise.complete.obs")
cor.test(AGE, WEIGHT)
```

Or by using the linear modelling **lm()** function:

```
summary(lm(WEIGHT ~ AGE))
```

We use the **summary()** function here to extract summary information from the output of the **lm()** function.

Exercise 1 : Getting acquainted with *R* for data analysis

It can be useful to use **lm()** to create an object:

```
fem.lm <- lm(WEIGHT ~ AGE)
```

And use the output in other functions:

```
summary(fem.lm)
plot(AGE, WEIGHT)
abline(fem.lm)
```

In this case we are passing the intercept and slope information held in the **fem.lm** object to the **abline()** function which draws a regression line.

A useful function to apply to the **fem.lm** object is **plot()** which produces diagnostic plots of the linear model:

```
plot(fem.lm)
```

Objects created by the **lm()** function (or any of the modelling functions) can use up a lot of memory so we should remove them when we no longer need them:

```
rm(fem.lm)
```

It might be interesting to see whether a similar relationship exists between **AGE** and **WEIGHT** for those who have and have not considered suicide. This can be done using the **coplot()** function:

```
coplot(WEIGHT ~ AGE | as.factor(LIFE))
```

The two plots looks pretty similar. We could extend the **coplot()** function call to investigate the relationship between **AGE** and **WEIGHT** for both **LIFE** and **SEX**:

```
coplot(WEIGHT ~ AGE | as.factor(LIFE) * as.factor(SEX))
```

Although the numbers are too small for this to be useful here.

Note that we used the **as.factor()** function with the **coplot()** function to ensure that ***R*** was aware that the **LIFE** and **SEX** columns hold categorical data. We can check the way variables are stored using the **data.class()** function:

```
data.class(fem$SEX)
```

We can 'apply' this function to all columns in a data.frame using the **sapply()** function:

```
sapply(fem, data.class)
```

Exercise 1 : Getting acquainted with *R* for data analysis

The parameters of most ***R*** functions have default values. These are usually the most used and most useful parameter values for each function. The **`cor.test()`** function, for example, calculates a Pearson's product moment correlation coefficient by default. This is an appropriate measure for data from a bivariate normal distribution. The **`DEPRESS`** and **`ANXIETY`** variables contain ordered data. An appropriate measure of correlation between **`DEPRESS`** and **`ANXIETY`** is Kendall's *tau*. This can be obtained using:

```
cor.test(DEPRESS, ANXIETY, method = "kendall")
```

Before we finish we should save the **`fem`** data.frame so that next time we want to use it we will not have to bother with recoding the missing values to the special **`NA`** value. This is done with the **`write.table()`** function:

```
write.table(fem, file = "newfem.dat", row.names = FALSE)
```

Everything in ***R*** is either a function or an object. Even the command to quit ***R*** is a function:

```
q()
```

When you call the **`q()`** function you will be asked if you want to save the workspace image. If you save the workspace image then all the objects and functions currently available to you (in this case the **`fem`** data.frame and the **`fem.lm`** linear model) will be saved. These will then be automatically restored the next time you start ***R*** in the current working directory. For this exercise there is no need to save the workspace image so click the **`No`** button.

Exercise 1 : Summary

R is a functional system. Everything is done by calling functions.

R provides a large set of functions for descriptive statistics, charting, and simple statistical inference.

Functions can be chained together so that the output of one function is the input of another function.

R is an object oriented system. We can use functions to create objects that can then be manipulated or passed to other functions for subsequent analysis.

Exercise 2 : Manipulating objects and creating new functions

In this exercise we will explore how to manipulate ***R*** objects and how to write functions that can manipulate and extract data and information from ***R*** objects and produce useful analyses.

Before we go any further we should start ***R*** and retrieve a dataset:

```
salex <- read.table("salex.dat", header = TRUE, na.strings = "9")
```

Missing values are coded as 9 throughout this dataset so we can use the **`na.strings`** parameter of the **`read.table()`** function to replace all 9's with the special **`NA`** code when we retrieve the dataset. Check that this works by examining the **`salex`** data.frame:

```
salex
names(salex)
```

This data comes from a food-borne outbreak. On Saturday 17th October 1992, eighty-two people attended a buffet meal at a sports club. Within fourteen to twenty-four hours fifty-one of the participants developed diarrhoea, with nausea, vomiting, abdominal pain and fever.

The columns in the dataset are as follows:

ILL	Ill or not-ill
HAM	Baked ham
BEEF	Roast beef
EGGS	Eggs
MUSHROOM	Mushroom flan
PEPPER	Pepper flan
PORKPIE	Pork pie
PASTA	Pasta salad
RICE	Rice salad
LETTUCE	Lettuce
TOMATO	Tomato salad
COLESLAW	Coleslaw
CRISPS	Crisps
PEACHCAKE	Peach cake
CHOCOLATE	Chocolate cake
FRUIT	Tropical fruit salad
TRIFLE	Trifle
ALMONDS	Almonds

Data is available for seventy-seven of the eighty-two people who attended the sports club buffet. All of the variables are coded 1=yes, 2=no.

We can use the **`attach()`** function to make it easier to access our data:

```
attach(salex)
```

The two-by-two table is a basic epidemiological tool. In analysing data from a food-borne outbreak collected as a retrospective cohort study, for example, we would tabulate each exposure (suspect foodstuffs) against the outcome (illness) and calculate risk ratios and confidence intervals. ***R*** has no explicit function to calculate risk ratios from two-by-two tables but we can easily write one ourselves.

Exercise 2 : Manipulating objects and creating new functions

The first step in writing such a function would be to create the two-by-two table. This can be done with the **`table()`** function. We will use a table of **`HAM`** by **`ILL`** as an illustration:

```
table(HAM, ILL)
```

This command produces the following output:

```
   ILL
HAM  1  2
  1 46 17
  2  5  9
```

We cannot manipulate the output directly so we need to instruct ***R*** to save the output of the **`table()`** function in an object:

```
tab <- table(HAM, ILL)
```

The **`tab`** object contains the output of the **`table()`** function:

```
tab
```

As it is stored in an object we can examine its contents on an item by item basis. The **`tab`** object is an object of class **`table`**:

```
class(tab)
```

We can extract data from a table object by using indices or row and column co-ordinates:

```
tab[1,1]
tab[1,2]
tab[2,1]
tab[2,2]
```

Note that the numbers in the square brackets refer to the position (as row and column co-ordinates) of the data item in the table **not** the values of the variable.

With this data we can calculate a risk ratio:

```
(tab[1,1]/(tab[1,1]+tab[1,2]))/(tab[2,1]/(tab[2,1]+tab[2,2]))
```

Which returns a risk ratio of **`2.044444`**.

This is a tedious calculation to have to type in every time you need to calculate a risk ratio from a two-by-two table. It would be far better to have a function that calculates and displays the risk ratio automatically. Fortunately, ***R*** allows us to do just that.

Exercise 2 : Manipulating objects and creating new functions

The **function()** function allows us to create new functions in ***R***:

```
tab2by2 <- function(exposure, outcome) {}
```

This creates an empty function called **tab2by2** that expects two parameters called **exposure** and **outcome**. We could type the whole function in at the ***R*** command prompt but it is easier to use a text editor:

```
fix(tab2by2)
```

This will start an editor (under Windows it will probably be the Notepad editor, under UNIX it will probably be the default system editor) with the empty **tab2by2** function already loaded. We can now edit this function to make it do something useful:

```
function(exposure, outcome)
  {
  tab <- table(exposure, outcome)
  a <- tab[1,1]; b <- tab[1,2]; c <- tab[2,1]; d <- tab[2,2]
  rr <- (a / (a + b)) / (c / (c + d))
  print(tab)
  print(rr)
  }
```

Once you have made the changes shown above, save the file and quit the editor.

Before proceeding we should examine the **tab2by2()** function to make sure we understand what is happening.

The first line defines **tab2by2** as a function that expects to be given two parameters. These parameters are called **exposure** and **outcome**.

The body of the function is enclosed within curly brackets (**{}**).

The first line of the body of the function creates a table object (**tab**) using the variables specified when the **tab2by2()** function is called.

The next line creates four new objects (called **a**, **b**, **c**, and **d**) which contain the values of the four cells in the two-by-two table.

The following line calculates the risk ratio using the objects **a**, **b**, **c**, and **d** and stores it in an object called **rr**.

The final two lines print the contents of the **tab** and **rr** objects.

Lets us try the **tab2by2()** function with our test data:

```
tab2by2(HAM, ILL)
```

The **tab2by2()** function displays a table of **HAM** by **ILL** followed by a risk ratio calculated from the data in the table. Try producing another table:

```
tab2by2(PASTA, ILL)
```

Exercise 2 : Manipulating objects and creating new functions

Have a look at the ***R*** objects available to you:

```
ls()
```

Note that there are no **a**, **b**, **c**, **d**, or **rr** objects. Examine the **tab** object:

```
tab
```

This is the table of **HAM** by **ILL** that you created earlier not the table of **PASTA** by **ILL** that was created by the **tab2by2()** function. The **tab**, **a**, **b**, **c**, **d**, and **rr** objects in the **tab2by2()** function are *local* to that function and do not change anything outside of that function. This means that the **tab** object inside the function is independent of any object of the same name outside of the function. When a function completes its work all of the objects that are local to that function are automatically removed. This is useful as it means that you can use object names inside functions that will not interfere with objects of the same name that are stored elsewhere. It also means that you do not clutter up the ***R*** workspace with temporary objects. Just to prove that **tab** in the **tab2by2()** function exists only in the **tab2by2()** function we can delete the tab object from the ***R*** workspace:

```
rm(tab)
```

Now try another call to the **tab2by2()** function:

```
tab2by2(FRUIT, ILL)
```

Now list the ***R*** objects available to you:

```
ls()
```

Note that there are no **tab**, **a**, **b**, **c**, **d**, or **rr** objects.

The **tab2by2()** function is very limited. It only displays a table and calculates and displays a simple ratio. A more useful function would also calculate and display a confidence interval for the risk ratio. This is what we will do now. Use the **fix()** function to edit the **tab2by2()** function:

```
fix(tab2by2)
```

We can now edit this function to add a calculation of the 95% confidence interval of the risk ratio following the method of Katz:

```
function(exposure, outcome)
  {
  tab <- table(exposure, outcome)
  a <- tab[1,1]; b <- tab[1,2]; c <- tab[2,1]; d <- tab[2,2]
  rr <- (a / (a + b)) / (c / (c + d))
  se.log.rr <- sqrt((b / a) / (a + b) + (d / c) / (c + d))
  lci.rr <- exp(log(rr) - 1.96 * se.log.rr)
  uci.rr <- exp(log(rr) + 1.96 * se.log.rr)
  print(tab)
  print(rr)
  print(lci.rr)
  print(uci.rr)
  }
```

Exercise 2 : Manipulating objects and creating new functions

Once you have made the changes shown above, save the file and quit the editor. Now we can test our function:

```
tab2by2(EGGS, ILL)
```

Which produces the following output:

```
         outcome
exposure  1  2
       1 40  6
       2 10 20
[1] 2.608696
[1] 1.553564
[1] 4.38044
```

The function works but the output could be improved. Use the **`fix()`** function to edit the **`tab2by2()`** function:

```
function(exposure, outcome)
  {
  tab <- table(exposure, outcome)
  a <- tab[1,1]; b <- tab[1,2]; c <- tab[2,1]; d <- tab[2,2]
  rr <- (a / (a + b)) / (c / (c + d))
  se.log.rr <- sqrt((b / a) / (a + b) + (d / c) / (c + d))
  lci.rr <- exp(log(rr) - 1.96 * se.log.rr)
  uci.rr <- exp(log(rr) + 1.96 * se.log.rr)
  print(tab)
  cat("\nRR     :", rr, "\n95% CI :", lci.rr, uci.rr, "\n")
  }
```

Once you have made the changes shown above, save the file and quit the editor. Now we can test our function again:

```
tab2by2(EGGS, ILL)
```

Which produces the following output:

```
         outcome
exposure  1  2
       1 40  6
       2 10 20

RR     : 2.608696
95% CI : 1.553564 4.38044
```

The **`tab2by2()`** function displays output but does not behave like a standard ***R*** function in the sense that you cannot save the results of the **`tab2by2()`** function into an object:

```
test2by2 <- tab2by2(EGGS, ILL)
```

displays output but does not save anything in the **`test2by2`** object:

```
test2by2
```

The returned value (**`NULL`**) means that **`test2by2`** is an empty object. We will not worry about this at the moment as the **`tab2by2()`** function is 'good-enough' for our current purposes. In Exercise 6 we will explore how to make our own functions behave like standard ***R*** functions.

Exercise 2 : Manipulating objects and creating new functions

We will now add the calculation of the odds ratio and its confidence interval to the **tab2by2()** function using the **fix()** function.

There are two ways of doing this. We could either calculate the odds ratio from the table and use (e.g.) the method of Woolf to calculate the confidence interval:

```
or <- (a / b) / (c / d)
se.log.or <- sqrt(1 / a + 1 / b + 1 / c + 1 / d)
lci.or <- exp(log(or) - 1.96 * se.log.or)
uci.or <- exp(log(or) + 1.96 * se.log.or)
cat("\nOR     :", or, "\n95% CI :", lci.or, uci.or, "\n")
```

Or use the output of the **fisher.test()** function:

```
ft <- fisher.test(tab)
cat("\nOR     :", ft$estimate, "\n95% CI :", ft$conf.int, "\n")
```

Note that we can refer to components of a function's output using the same syntax as when we refer to columns in a data.frame (e.g. **ft$estimate** to examine the estimate of the odds ratio from the **fisher.test()** function stored in the object **ft**. The names of elements in the output of a standard function such as **fisher.test()** can be found in the documentation or the help system. For example:

```
help(fisher.test)
```

Output elements are listed under the **Value** heading.

Test the **tab2by2()** function when you have added the calculation of the odds-ratio and confidence interval.

Now that we have a function that will calculate risk and odds ratios with confidence intervals from a two-by-two table we can use it to analyse the **salex** data:

```
tab2by2(HAM, ILL)
tab2by2(BEEF, ILL)
tab2by2(EGGS, ILL)
tab2by2(MUSHROOM, ILL)
tab2by2(PEPPER, ILL)
tab2by2(PORKPIE, ILL)
tab2by2(PASTA, ILL)
tab2by2(RICE, ILL)
tab2by2(LETTUCE, ILL)
tab2by2(TOMATO, ILL)
tab2by2(COLESLAW, ILL)
tab2by2(CRISPS, ILL)
tab2by2(PEACHCAKE, ILL)
tab2by2(CHOCOLATE, ILL)
tab2by2(FRUIT, ILL)
tab2by2(TRIFLE, ILL)
tab2by2(ALMONDS, ILL)
```

Make a note of any positive associations (i.e. risk ratio > 1) with confidence intervals that do not include one. We will need these for the next exercise when we will use logistic regression to analyse the data.

Exercise 2 : Manipulating objects and creating new functions

When you create a function you can save it in the workspace when you quit ***R*** and it will be available the next time you start ***R***. It is better, however, to save the function in a separate file:

```
save(tab2by2, file = "tab2by2.r")
```

That can be loaded whenever it is needed:

```
load("tab2by2.r")
```

The **`save()`** function can be used to save any ***R*** object such as data, functions, output from functions.

We can now quit ***R***:

```
q()
```

For this exercise there is no need to save the workspace image so click the **`No`** button.

Exercise 2 : Summary

R objects contain information that can be examined and manipulated.

R can be extended by writing new functions.

New functions can perform simple or complex data analysis.

New functions can be composed of parts of existing function.

New functions can be saved and used in subsequent ***R*** sessions.

Objects defined within functions are *local* to that function and only exist while that function is being used. This means that you can re-use meaningful names within functions without them interfering with each other.

Exercise 3 : Logistic regression

In this exercise we will explore how ***R*** handles generalised linear models using the example of logistic regression. We will continue using the **`salex`** dataset. Start ***R*** and retrieve the **`salex`** dataset:

```
salex <- read.table("salex.dat", header = TRUE, na.strings = "9")
```

When we analysed this data using two-by-two tables and examining the risk ratio and confidence interval associated with each exposure we found many significant positive associations:

Variable	RR	95% CI
EGGS	2.61	1.55, 4.38
MUSHROOM	1.41	1.03, 1.93
PEPPER	1.74	1.27, 2.38
PASTA	1.68	1.26, 2.26
RICE	1.72	1.25, 2.34
LETTUCE	2.01	1.49, 2.73
COLESLAW	1.89	1.37, 2.64
CHOCOLATE	1.39	1.05, 1.87

Some of these associations may be due to confounding in the data. We can use logistic regression to help us identify independent associations.

Logistic regression requires the dependent (*y*) variable to be either 0 or 1. In order to perform a logistic regression we must first recode the **`ILL`** variable so that 0=no and 1=yes:

```
table(salex$ILL)
salex$ILL[salex$ILL == 2] <- 0
table(salex$ILL)
```

We could work with our data as it is but if we wanted to calculate odds-ratios and confidence intervals we would calculate with their reciprocals (i.e. odds-ratios for non-exposure rather than for exposure). This is because of the way the data has been coded (1=yes, 2=no). In order to calculate meaningful odds-ratios the exposure variables should also be coded 0=no, 1=yes. The actual codes used are not important as long as the value used for 'yes' is one greater than the value used for 'no'. We could issue a series of commands similar to the one we have just used to recode the **`ILL`** variable. This is both tedious and unnecessary as the structure of the dataset (i.e. all variables are coded identically) allows us to recode all variables with a single command:

```
salex <- read.table("salex.dat", header = TRUE, na.strings = "9")
salex[1:5, ]
salex <- 2 - salex
salex[1:5, ]
```

WARNING : Commands that manipulate variables in a data.frame may not work as expected if the data.frame has been attached using the **`attach()`** function:

```
salex <- read.table("salex.dat", header = TRUE, na.strings = "9")
attach(salex)
salex <- 2 - salex
table(LETTUCE, ILL)
```

It is better to manipulate data before attaching a data.frame. The **`detach()`** function may be used to remove an attachment prior to any data manipulation.

Exercise 3 : Logistic regression

Before continuing we will retrieve the **salex** dataset and recode the variables to 1/0:

```
salex <- read.table("salex.dat", header = TRUE, na.strings = "9")
salex <- 2 - salex
```

We can now use the generalised linear model **glm()** function to specify the logistic regression model:

```
salex.lreg <- glm(formula = ILL ~ EGGS + MUSHROOM + PEPPER +
                  PASTA + RICE + LETTUCE + COLESLAW + CHOCOLATE,
                  family = binomial(logit), data = salex)
```

Note that we have saved the output of the **glm()** function in the **salex.lreg** object. The method used by the **glm()** function is defined by the **family** parameter. Here we specify **binomial** errors and a **logit** (logistic) linking function.

We can examine some basic information about the specified model using the **summary()** function:

```
summary(salex.lreg)
```

We will use backwards elimination to remove non-significant variables from the logistic regression model. **CHOCOLATE** is the least significant variable in the model so we will remove this variable from the model. Storing the output of the **glm()** function is useful as it allows us to use the **update()** function to add, remove, or modify variables without having to describe the model in full:

```
salex.lreg <- update(salex.lreg, . ~ . - CHOCOLATE)
summary(salex.lreg)
```

RICE is now the least significant variable in the model so we will remove this variable from the model:

```
salex.lreg <- update(salex.lreg, . ~ . - RICE)
summary(salex.lreg)
```

COLESLAW is now the least significant variable in the model so we will remove this variable from the model:

```
salex.lreg <- update(salex.lreg, . ~ . - COLESLAW)
summary(salex.lreg)
```

PEPPER is now the least significant variable in the model so we will remove this variable from the model:

```
salex.lreg <- update(salex.lreg, . ~ . - PEPPER)
summary(salex.lreg)
```

MUSHROOM is now the least significant variable in the model so we will remove this variable from the model:

```
salex.lreg <- update(salex.lreg, . ~ . - MUSHROOM)
summary(salex.lreg)
```

There are now no non-significant variables in the model.

Unfortunately ***R*** does not present information on the model coefficients in terms of odds-ratios and confidence intervals but we could write a function to calculate them for us.

Exercise 3 : Logistic regression

The first step in doing this is to realise that the **salex.lreg** object contains essential information about the fitted model. To calculate odds ratios and confidence intervals we need the regression coefficients and their standard errors. Both:

```
summary(salex.lreg)$coefficients
```

and:

```
coef(summary(salex.lreg))
```

extract the data we require. The preferred method is to use the **coef()** function. This is because some fitted models may return coefficients in a more complicated manner than (e.g.) the **glm()** function. The **coef()** function provides a standard way of extracting this data from all types of fitted objects.

We can store this data in a separate object to make it easier to work with:

```
salex.lreg.coeffs <- coef(summary(salex.lreg))
salex.lreg.coeffs
```

We can extract information from this object by addressing each piece of information by its row and column position in the object. For example:

```
salex.lreg.coeffs[2,1]
```

Is the regression coefficient for **EGGS**, and:

```
salex.lreg.coeffs[3,2]
```

Is the standard error of the regression coefficient for **PASTA**. Similarly:

```
salex.lreg.coeffs[ ,1]
```

Returns the regression coefficients for all of the variables in the model, and:

```
salex.lreg.coeffs[ ,2]
```

Returns the standard errors of the regression coefficients. The table below shows the indices that address each cell in the table of regression coefficients:

```
          [ ,1]      [ ,2]     [ ,3]        [ ,4]
[1, ] -1.970966  0.6143611 -3.208155 0.0013358946
[2, ]  2.639114  0.7330623  3.600122 0.0003180678
[3, ]  1.664580  0.8371252  1.988448 0.0467621588
[4, ]  3.195573  1.1487054  2.781891 0.0054043245
```

We can use this information to calculate odds-ratios and 95% confidence intervals:

```
or <- exp(salex.lreg.coeffs[ ,1])
lci <- exp(salex.lreg.coeffs[ ,1] - 1.96 * salex.lreg.coeffs[ ,2])
uci <- exp(salex.lreg.coeffs[ ,1] + 1.96 * salex.lreg.coeffs[ ,2])
```

And then make a single object that contains all of the required information:

```
lreg.or <- cbind(lci, or, uci)
lreg.or
```

Exercise 3 : Logistic regression

We have now gone through all the necessary calculations step-by-step but it would be nice to have a function that did it all for us that we could use whenever we needed to. First we will create a template for the function:

```
lreg.or <- function(model) {}
```

And then use the **`fix()`** function to edit the **`lreg.or()`** function:

```
fix(lreg.or)
```

We can now edit this function to add a calculation of odds-ratios and 95% confidence intervals:

```
function(model)
  {
  lreg.coeffs <- coef(summary(model))
  lci <- exp(lreg.coeffs[ ,1] - 1.96 * lreg.coeffs[ ,2])
  or <- exp(lreg.coeffs[ ,1])
  uci <- exp(lreg.coeffs[ ,1] + 1.96 * lreg.coeffs[ ,2])
  lreg.or <- cbind(lci, or, uci)
  lreg.or
  }
```

Once you have made the changes shown above, save the file and quit the editor. Now we can test our function:

```
lreg.or(salex.lreg)
```

Which produces the following output:

```
                   lci         or         uci
(Intercept) 0.04178937  0.1393223   0.4644888
EGGS        3.32780062 14.0007882  58.9043912
PASTA       1.02410041  5.2834538  27.2579560
LETTUCE     2.57052522 24.4241609 232.0691633
```

Before we continue, it is probably a good idea to save this function for later use:

```
save(lreg.or, file = "lregor.r")
```

Which can be reloaded whenever it is needed:

```
load("lregor.r")
```

Exercise 3 : Logistic regression and stratified analysis

An alternative to using logistic regression with data that contains associations that may be due to confounding is to use stratified analysis (i.e. *Mantel-Haenszel* techniques). With several potential confounders, a stratified analysis results in the analysis of many tables which can be difficult to interpret and may yield unreliable results. For example, four potential cofounders, each with two levels would produce sixteen tables. Some of these tables are likely to contain small numbers which may produce unstable or unreliable results. In such situations, logistic regression would be a better approach. In order to illustrate Mantel-Haenszel techniques we will work with simpler dataset.

On Saturday, 21st April 1990, a luncheon was held in the home of Jean Bateman. There was a total of forty-five guests which included thirty-five members of the Department of Epidemiology and Population Sciences at the London School of Hygiene and Tropical Medicine. On Sunday morning, 22nd April 1990, Jean awoke with symptoms of gastrointestinal illness; her husband awoke with similar symptoms. The possibility of an outbreak related to the luncheon was strengthened when several of the guests telephoned Jean on Sunday and reported illness. On Monday, 23rd April 1990, there was an unusually large number of department members absent from work and reporting illness. Data from this outbreak is stored in the file **`bateman.dat`**. The variables in the file are:

ILL	Ill?
CHEESE	Cheddar cheese
CRABDIP	Crab dip
CRISPS	Crisps
BREAD	French bread
CHICKEN	Chicken (roasted, served warm)
RICE	Rice (boiled, served warm)
CAESAR	Caesar salad
TOMATO	Tomato salad
ICECREAM	Vanilla ice-cream
CAKE	Chocolate cake
JUICE	Orange juice
WINE	White wine
COFFEE	Coffee

Data is available for all forty-five guests at the luncheon. All of the variables are coded 1=yes, 2=no. Retrieve this and attach this dataset:

```
bateman <- read.table("bateman.dat", header = TRUE)
bateman
attach(bateman)
```

We can use our **`tab2by2()`** function to analyse this data:

```
tab2by2(CHEESE, ILL)
tab2by2(CRABDIP, ILL)
tab2by2(CRISPS, ILL)
tab2by2(BREAD, ILL)
tab2by2(CHICKEN, ILL)
tab2by2(RICE, ILL)
tab2by2(CAESAR, ILL)
tab2by2(TOMATO, ILL)
tab2by2(ICECREAM, ILL)
tab2by2(TOMATO, ILL)
tab2by2(CAKE, ILL)
tab2by2(JUICE, ILL)
tab2by2(WINE, ILL)
tab2by2(COFFEE, ILL)
```

Exercise 3 : Logistic regression and stratified analysis

Two variables (**CAESAR** and **TOMATO**) are associated with **ILL**. These two variables are also associated with each other:

```
tab2by2(CAESAR, TOMATO)
chisq.test(table(CAESAR, TOMATO))
fisher.test(table(CAESAR, TOMATO))
```

Suggesting the potential for one of these associations to be due to confounding.

We can perform a simple stratified analysis using the **table()** function:

```
table(CAESAR, ILL, TOMATO)
table(TOMATO, ILL, CAESAR)
```

It would be useful to calculate risk-ratios for each stratum. We can use **apply()** to apply a risk-ratio calculation to the two-by-two table in each stratum:

```
tabC <- table(CAESAR, ILL, TOMATO)
apply(tabC, 3, function(x)
     (x[1,1]/(x[1,1]+x[1,2]))/(x[2,1]/(x[2,1]+x[2,2])))
tabT <- table(TOMATO, ILL, CAESAR)
apply(tabT, 3, function(x)
     (x[1,1]/(x[1,1]+x[1,2]))/(x[2,1]/(x[2,1]+x[2,2])))
```

The **mantelhaen.test()** function performs the stratified analysis:

```
mantelhaen.test(tabC)
mantelhaen.test(tabT)
```

It is likely that **CAESAR** salad was a vehicle of food-poisoning, and that **TOMATO** salad was not a vehicle of food-poisoning. Many of those at the luncheon ate **CAESAR** salad **and TOMATO** salad. **CAESAR** confounded the relationship between **TOMATO** and **ILL**. This resulted in a spurious association between **TOMATO** and **ILL**.

It only makes sense to calculate a common odds ratio in the absence of interaction. We can check for interaction 'by eye' by examining and comparing the risk ratios for each stratum as we did above. For example, there appears to be an interaction between **CAESAR**, **WINE**, and **ILL**:

```
tabW <- table(CAESAR, ILL, WINE)
apply(tabW, 3, function(x)
     (x[1,1]/(x[1,1]+x[1,2]))/(x[2,1]/(x[2,1]+x[2,2])))
```

Woolf's test for interaction provides a formal test for interaction. ***R*** does not provide this test but it is possible to write a function to perform the test. First we will create a template for the function:

```
woolf.test <- function(x) {}
```

And then use the **fix()** function to edit the **woolf.test()** function:

```
fix(woolf.test)
```

Exercise 3 : Logistic regression and stratified analysis

We can now edit this function to make it do something useful:

```
function(x)
  {
  x <- x + 1 / 2
  k <- dim(x)[3]
  or <- apply(x, 3, function(x)
            (x[1, 1]* x[2, 2])/(x[1, 2] * x[2, 1]))
  w <-  apply(x, 3, function(x) 1 / sum(1 / x))
  chi.sq <- sum(w * (log(or) - weighted.mean(log(or), w))^2)
  p <- 1 - pchisq(chi.sq, df = k - 1)
  cat("\nWoolf's X2 :", chi.sq, "\np-value    :", p, "\n")
  }
```

Once you have made the changes shown above, save the file and quit the editor. We can use this function to test for a three-way interaction between **CAESAR**, **WINE**, and **ILL**:

```
woolf.test(tabW)
```

Which returns:

```
Woolf's X2 : 3.319492
p-value    : 0.06846297
```

Which is evidence of an interaction. We should also test for interaction between **CAESAR**, **TOMATO**, and **ILL**:

```
woolf.test(tabC)
```

We can repeat this analysis using logistic regression to see how to specify interaction terms with the **glm()** function. We need to change the coding of the variables to 0 and 1 before specifying the model:

```
detach(bateman)
bateman <- 2 - bateman
bateman
bateman.lreg <- glm(formula = ILL ~ CAESAR + TOMATO,
                    family = binomial(logit), data = bateman)
summary(bateman.lreg)
bateman.lreg <- update(bateman.lreg, . ~ . - TOMATO)
summary(bateman.lreg)
```

Interactions are specified using the multiply (*****) symbol in the model formula:

```
bateman.lreg <- glm(ILL ~ CAESAR + WINE + CAESAR * WINE,
                    family = binomial(logit), data = bateman)
summary(bateman.lreg)
```

Before we continue, it is probably a good idea to save the **woolf.test()** function for later use:

```
save(woolf.test, file = "woolf.r")
```

We can now quit ***R***:

```
q()
```

For this exercise there is no need to save the workspace image so click the **No** button.

Exercise 3 : Summary

R provides functions for many kinds of complex statistical analysis. We have looked at using the generalised linear model **`glm()`** function to perform logistic regression as well as the **`mantelhaens()`** function to perform a stratified analysis.

R can be extended by writing new functions. New functions can perform simple or complex data analysis. New functions can be composed of parts of existing function. New functions can be saved and used in subsequent ***R*** sessions. By building your own functions you can use ***R*** to build your own statistical analysis system.

Exercise 4 : Analysing some data with *R*

In this exercise we will use the *R* functions we have already used and the functions we have added to *R* to analyse a small dataset. First we will start *R* and retrieve our new functions:

```
load("tab2by2.r")
load("lregor.r")
```

And then retrieve and attach the sample dataset:

```
gudhiv <- read.table("gudhiv.dat", header = TRUE, na.strings = "X")
attach(gudhiv)
```

This data is from a cross-sectional study of 435 male patients who presented with sexually transmitted infections at an outpatient clinic in The Gambia between August 1988 and June 1990. Several studies have documented an association between genital ulcer disease (GUD) and HIV infection. A study of Gambian prostitutes documented an association between seropositivity for HIV-2 and antibodies against Treponema pallidum (a serological test for syphilis). Prostitutes are not the ideal population for such studies as they may have experienced multiple sexually transmitted infections and it is difficult to quantify the number of times they may have had sex with HIV-2 seropositive customers. A sample of males with sexually transmitted infections is easier to study as they have probably had fewer sexual partners than prostitutes and much less contact with sexually transmitted infection pathogens. In such a sample it is also easier to find subjects and collect data. The variables in the dataset are:

```
MARRIED   Married (1=yes, 0=no)
GAMBIAN   Gambian Citizen  (1=yes, 0=no)
GUD       History of GUD or syphilis (1=yes, 0=no)
UTIGC     History of urethral discharge (1=yes, 0=no)
CIR       Circumcised (1=yes, 0=no)
TRAVOUT   Travelled outside of Gambia and Senegal (1=yes, 0=no)
SEXPRO    Ever had sex with a prostitute (1=yes, 0=no)
INJ12M    Injection in previous 12 months (1=yes, 0=no)
PARTNERS  Sexual partners in previous 12 months (number)
HIV       HIV-2 positive serology (1=yes, 0=no)
```

Data is available for all 435 patients enrolled in the study.

We will start our analysis by examining pairwise associations between the binary exposure variables and the HIV variable using the **tab2by2()** function that we wrote earlier:

```
tab2by2(MARRIED, HIV)
tab2by2(GAMBIAN, HIV)
tab2by2(GUD, HIV)
tab2by2(UTIGC, HIV)
tab2by2(CIR, HIV)
tab2by2(TRAVOUT, HIV)
tab2by2(SEXPRO, HIV)
tab2by2(INJ12M, HIV)
```

Note that our **tab2by2()** function returns misleading risk ratio estimates and confidence intervals for this dataset. This is because the function expects the **exposure** and **outcome** variables to be ordered with exposure-present and outcome-present as the first category (e.g. 1 = present, 2 = absent). This coding is reversed (i.e. 0 = absent, 1 = present) in the **gudhiv** dataset.

Exercise 4 : Analysing some data with *R*

We can produce risk ratio estimates for variables in the **gudhiv** data using the **tab2by2()** function and a simple transformation of the **exposure** and **outcome** variables. For example:

```
tab2by2(2 - GUD, 2 - HIV)
```

The odds ratio estimates returned by the **tab2by2()** function, with or without this transformation, are correct. The **GUD** and **TRAVOUT** variables are associated with **HIV**.

PARTNERS is a continuous variable and we should examine its distribution before doing anything with it:

```
table(PARTNERS)
hist(PARTNERS)
```

The distribution of **PARTNERS** is severely non-normal. Instead of attempting to transform the variable we will produce summary statistics for each level of the **HIV** variable and perform a non-parametric test:

```
by(PARTNERS, HIV, summary)
kruskal.test(PARTNERS, HIV)
```

An alternative way of looking at the data is as a tabulation:

```
table(PARTNERS, HIV)
```

There appears to be an association between the number of sexual **PARTNERS** in the previous twelve months and positive **HIV** serology. The proportion with positive **HIV** serology increases as the number of sexual partners increases:

```
prop.table(table(PARTNERS, HIV), 1) * 100
```

The Chi-square test for trend is an appropriate test to perform on this data. The **prop.trend.test()** function that performs the chi-square test for trend requires you to specify the 'number of events' and the 'number of trials'. In this table:

```
       HIV
PARTNERS   0 1
       1  60 1
       2 128 1
       3 131 2
       4  68 3
       5  21 4
       6   3 3
       7   2 4
       8   1 1
       9   0 2
```

The 'number of events' in each row is in the second column (labelled **1**) and the 'number of trials' is the total number of cases in each row of the table. We can extract this data from a table object:

```
tab <- table(PARTNERS, HIV)
events <- tab[ ,2]
trials <- tab[ ,1] + tab[ ,2]
```

And pass this to the **prop.trend.test()** function:

```
prop.trend.test(events, trials)
```

Exercise 4 : Analysing some data with *R*

With a linear trend such as this we can use **PARTNERS** in a logistic model without recoding or creating indicator variables. We can now specify and fit the logistic regression model:

```
gudhiv.lreg <- glm(formula = HIV ~ GUD + TRAVOUT + PARTNERS,
                   family = binomial(logit))
summary(gudhiv.lreg)
```

We can use the **lreg.or()** function that we wrote earlier to calculate and display odds-ratios and confidence intervals:

```
lreg.or(gudhiv.lreg)
```

PARTNERS is incorporated into the logistic model as a continuous variable. The odds ratio reported for **PARTNERS** is the odds ratio associated with a unit increase in the number of sexual **PARTNERS**. A man reporting five sexual partners, for example, was over three times as likely (odds ratio = 3.1911) to have a positive HIV-2 serology than a man reporting four sexual partners.

An alternative approach would be to have created an indicator variables:

```
part.gt.5 <- ifelse(PARTNERS > 5, 1, 0)
```

This creates a new variable (**part.gt.5**) that indicates whether or not an individual subject reported having more than five sexual partners in the previous twelve months:

```
table(PARTNERS, part.gt.5)
```

You can also inspect this on a case-by-case basis:

```
cbind(PARTNERS, part.gt.5)
```

We can now specify and fit the logistic regression model using our indicator variable:

```
gudhiv.lreg <- glm(formula = HIV ~ GUD + TRAVOUT + part.gt.5,
                   family = binomial(logit))
summary(gudhiv.lreg)
```

We can use the **lreg.or()** function that we wrote earlier to calculate and display odds-ratios and confidence intervals:

```
lreg.or(gudhiv.lreg)
```

We can now quit ***R***:

```
q()
```

For this exercise there is no need to save the workspace image so click the **No** button.

Exercise 4 : Summary

Using built-in functions and our own functions we can use ***R*** to analyse epidemiological data.

The power of ***R*** is that it can be easily extended. Many user-contributed functions are available for download over the Internet. We will use one of these packages in the next exercise.

Exercise 5 : Extending *R* with packages

R has no built-in functions for survival analysis but, because it is an extensible system, survival analysis is available as an add-in package. You can find a list of add-in packages at the ***R*** website.

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

Add-in packages are installed from the Internet. There are a series of ***R*** functions that enable you to download and install add-in packages. The **survival** package adds functions to ***R*** that enable it to analyse survival data. This package may be downloaded and installed using **install.packages("survival")** or from the **Packages** menu if you are using a GUI version of ***R***. Do **not** do this if you are following this tutorial at the NHV as this package is already installed on the NHV computers.

You may also download compressed versions of packages from the ***R*** website. This may be more convenient than using the **install.packages()** function if you access the Internet through a dial-up connection, would like to be able to install packages on machines that are not connected to the Internet, or would like to create a distribution CD for a colleague. Downloaded packages may be installed using the ***R*** installer program.

Packages are loaded into ***R*** as they are needed using the **library()** function. Start ***R*** and load the **survival** package:

```
library(survival)
```

Before we go any further we should retrieve a dataset:

```
ca <- read.table("ca.dat", header = TRUE)
attach(ca)
```

The columns in this dataset on the survival of cancer patients in two different treatment groups are as follows:

```
time     Survival or censoring time (months)
status   Censoring status (1=dead, 0=censored)
group    Treatment group (1 / 2)
```

We next need to create a **survival** object from the **time** and **status** variables using the **Surv()** function:

```
response <- Surv(time, status)
```

We can then specify the model for the survival analysis. In this case we state that survival is dependent upon the treatment group:

```
ca.surv <- survfit(response ~ group)
```

The **summary()** function applied to a **survfit** object lists the survival probabilities at each time point with 95% confidence intervals:

```
summary(ca.surv)
```

Printing the **ca.surv** object provides another view of the results:

```
ca.surv
```

Exercise 5 : Extending *R* with packages

The **plot()** function with a **survfit** object displays the survival curves:

```
plot(ca.surv, xlab = "Months", ylab = "Survival")
```

We can make it easier to distinguish between the two lines by specifying a width for each line using the **lwd** parameter of the **plot()** function:

```
plot(ca.surv, xlab = "Months", ylab = "Survival", lwd = c(1, 2))
```

It would also be useful to add a legend:

```
legend(125, 1, names(ca.surv$strata), lwd = c(1, 2))
```

We can perform a formal test of the two survival times using the **survdiff()** function:

```
survdiff(response ~ group)
```

We can now quit ***R***:

```
q()
```

For this exercise there is no need to save the workspace image so click the **No** button.

Exercise 5 : Summary

R can be extended by adding additional packages. Many packages (e.g. **`eda`** for exploratory data analysis) are included with the standard ***R*** installation but many others are available and may be downloaded from the Internet.

You can find a list of add-in packages at the ***R*** website.

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

Packages may also be downloaded and installed from this site using the **`install.packages()`** function or downloaded and installed using the ***R*** installer program.

Packages are loaded into ***R*** as they are needed using the **`library()`** function. You can use the **`search()`** function to display a list of loaded packages and attached data.frames.

Exercise 6 : Making your own objects behave like *R* objects

In the previous exercises we concentrated on writing functions that take some input data, analyse it, and display the results of the analysis. The standard *R* functions we have used all do this. The **fisher.test()** function, for example, takes a **table** object (or the names of two variables) as input and calculates and displays the p-value for Fisher's exact test and the odds ratio and associated confidence interval for two-by-two tables:

```
fem <- read.table("fem.dat", header = TRUE)
attach(fem)
fisher.test(SEX, LIFE)
```

The results of the **fisher.test()** function may also be saved for later use:

```
ft <- fisher.test(SEX, LIFE)
ft
```

The **fisher.test()** function returns an object of the class **htest**:

```
class(ft)
```

Which is a list containing the output of the **fisher.test()** function. Each item of output is stored as a different named item in the list:

```
names(ft)
```

Each of these items can be referred to by name:

```
ft$estimate
ft$conf.int
```

When you display the output of the **fisher.test()** function either by calling the function directly:

```
fisher.test(SEX, LIFE)
```

Or by typing the name of an object created using the **fisher.test()** function:

```
ft
```

The **print()** function takes over and formatted output is produced. The **print()** function knows about **htest** class objects and produces output of the correct format for that class of object. This means that any function that produces an **htest** object (or any other standard *R* object) does not need to include *R* commands to produce formatted output.

All of the hypothesis testing functions supplied with *R* produce objects of the **htest** type and use the **print()** function to produce formatted output. For example:

```
tt <- t.test(WEIGHT[LIFE == 1], WEIGHT[LIFE == 2], var.equal = TRUE)
class(tt)
```

You can use this feature of *R* in your own functions. We will explore this by writing a function to test the null hypothesis that the variance to mean ratio of a vector of numbers is equal to one. Such a test might be used to investigate the spatial distribution (over natural sampling units such as households) of cases of a disease.

Exercise 6 : Making your own objects behave like *R* objects

Create a new function using the **`function()`** function:

```
v2m.test <- function(data) {}
```

And start the function editor:

```
fix(v2m.test)
```

Now edit this function to make it do something useful:

```
function(data)
  {
  nsu <- length(data); obs <- sum(data)
  obs.nsu.mean <- obs / nsu; obs.nsu.var <- var(data)
  var.mean.ratio <- obs.nsu.var / obs.nsu.mean
  chi2 <- sum((data - obs.nsu.mean)^2) / obs.nsu.mean
  df <- nsu - 1; p <- 1 - pchisq(chi2, df)
  names(chi2) <- "Chi-square"
  names(df) <- "df"
  names(var.mean.ratio) <- "Variance : mean ratio"
  v2m <- list(method = "Variance to mean test",
              data.name = deparse(substitute(data)),
              statistic = chi2,
              parameter = df,
              p.value = p,
              estimate = var.mean.ratio
             )
  class(v2m) <- "htest"
  return(v2m)
  }
```

Once you have made the changes shown above, save the file and quit the editor. Before proceeding we should examine the **`v2m.test()`** function to make sure we understand what is happening.

The first five lines after the opening curly bracket (**`{`**) contain the required calculations. The next three lines use the **`names()`** function to give our variables names that will make sense in formatted output. The next line creates a list of items that the function returns using some of the names used by **`htest`** type objects:

Name of item	Usage
`method`	Text description of the test used to title output
`data.name`	Name(s) of data or variables used for the test
`null.value`	The null value
`statistic`	Value of test statistic
`parameter`	A test parameter such as the degrees of freedom of the test statistic
`p.value`	The p-value of the test
`estimate`	An estimate (e.g. the mean)
`conf.int`	Confidence interval of estimate
`alternative`	Text describing the alternative hypothesis
`note`	Text note

The next line tells *R* that object **`v2m`** is of the class **`htest`**. The final line causes the function to return the **`v2m`** object (i.e. a list of class **`htest`** containing the named items **`method`**, **`data.name`**, **`statistic`**, **`parameter`**, **`p.value`**, and **`estimate`**).

Exercise 6 : Making your own objects behave like *R* objects

We are now ready to test the **v2m.test()** function. This table:

```
Number of cases :       0  1  2  3  4  6
Number of households : 24 29 26 14  5  2
```

shows the number of cases of chronic (stunting) undernutrition found in a random sample of 100 households. We can reproduce the data behind this table using a combination of the **c()** and **rep()** functions:

```
stunt <- c(rep(0,24), rep(1,29), rep(2,26), rep(3,14), rep(4,5),
           rep(5,0), rep(6,2))
table(stunt)
```

And use it to test our new **v2m.test()** function:

```
v2m.test(stunt)
```

Which should produce the following output:

```
Variance to mean test

data:  stunt
Chi-square = 110.1613, df = 99, p-value = 0.2083
sample estimates:
Variance : mean ratio
             1.112740
```

If your **vm2.test()** function does not produce this output then use the **fix()** function:

```
fix(v2m.test)
```

To check and edit the **vm2.test()** function and try again.

The important thing to note from this exercise is that *R* allows us to specify a class for the output of our functions. This means that we can use standard *R* classes and functions to (e.g.) produce formatted output without us having to write commands to format the output ourselves.

More importantly, it also means that we can write functions that return values when we need them to return values but can also produce formatted output when we need them to produce formatted output. Our **v2m.test()** function can produce values for later use:

```
vm <- v2m.test(stunt)
vm$p.value
```

Or produce formatted output:

```
v2m.test(stunt)
```

This way of working is not limited to using standard *R* classes and functions. *R* also allows us to define our own classes. We will explore by defining functions and a new class to deal with two-by-two tables.

Exercise 6 : Making your own objects behave like *R* objects

We need to create two functions. One function will handle the calculations and another function will produce formatted output when required.

Create a new function using the **`function()`** function:

```
rr22 <- function(exposure, outcome) {}
```

And start the function editor:

```
fix(rr22)
```

Now edit this function to make it do something useful:

```
function(exposure, outcome)
  {
  tab <- table(exposure, outcome)
  a <- tab[1,1]; b <- tab[1,2]; c <- tab[2,1]; d <- tab[2,2]
  rr <- (a / (a + b)) / (c / (c + d))
  se.log.rr <- sqrt((b/a) / (a+b) + (d / c) / (c + d))
  lci.rr <- exp(log(rr) - 1.96 * se.log.rr)
  uci.rr <- exp(log(rr) + 1.96 * se.log.rr)
  rr22.output <- list(estimate = rr, conf.int = c(lci.rr, uci.rr))
  class(rr22.output) <- "rr22"
  return(rr22.output)
  }
```

Once you have made the changes shown above, save the file and quit the editor.

The **`rr22()`** function is similar to the **`tab2by2()`** function that you created in the second exercise of this tutorial except that the function now returns a list of values instead of formatted output:

```
fem <- read.table("fem.dat", header = TRUE)
attach(fem)
rr22.test <- rr22(SEX, LIFE)
names(rr22.test)
rr22.test$estimate
rr22.test$conf.int
rr22.test$conf.int[1]
rr22.test$conf.int[2]
```

The function returns a list of class **`rr22`**:

```
class(rr22.test)
```

The displayed output from the **`rr22()`** function is, however, not pretty:

```
print(rr22.test)
rr22(SEX, LIFE)
```

Exercise 6 : Making your own objects behave like *R* objects

This can be fixed by creating a new function:

```
print.rr22 <- function(x) {}
```

And start the function editor:

```
fix(print.rr22)
```

Now edit this function to make it do something useful:

```
function(x)
  {
  cat("RR     : ", x$estimate, "\n",
      "95% CI : ", x$conf.int[1], "; ", x$conf.int[2], "\n",
      sep = "")
  }
```

The function name **`print.rr22()`** indicates that this function contains the **`print()`** method for objects of class **`rr22`**. All objects of class **`rr22`** will use the function **`print.rr22()`** instead of the standard ***R*** **`print()`** function to produce formatted output:

```
rr22(SEX, LIFE)
rr22.test <- rr22(SEX, LIFE)
rr22.test
print(rr22.test)
```

Note that we can still extract returned values from an **`rr22`** class object:

```
rr22.test$estimate
```

The **`print.rr22()`** function only controls the way an entire **`rr22`** object is displayed.

You might like to use the **`save()`** function to save the **`v2m.test()`**, **`rr22()`**, and **`print.rr22()`** functions before quitting ***R***.

We can now quit ***R***:

```
q()
```

For this exercise there is no need to save the workspace image so click the **`No`** button.

Exercise 6 : Summary

R objects can be assigned a class or type.

Objects of a specific class or type may share functions that extract and manipulate data common to members of that class. This allows you to write functions that handle data that is common to all members of that class (e.g. to produce formatted output for hypothesis testing functions).

R provides a set of ready-made classes (e.g. **`htest`**) which can be used by standard ***R*** functions such as the **`print()`** and **`summary()`** functions.

R allows you to create new classes and class-specific functions that can extract and manipulate data common to the new classes.

Classes allows you to create versatile functions that return values when we need them to return values but can also produce formatted output when we need them to produce formatted output.

Classes allow you to write functions that can be chained together so that the output of one function is the input of another function.

Exercise 7 : Writing your own graphical functions

R provides a pretty full set of graphical functions for plotting data as well as **`plot()`** methods for a wide variety of statistical functions. There will be times, however, when you will need to write you own graphical functions to present and analyse data in a specific way. In this exercise we will create a function that produces a plot that may be used for assessing agreement between two methods of clinical measurement as described in:

> Bland MG., Altman DG., *'Statistical methods for assessing agreement between two methods of clinical measurement'*, Lancet, 08/02/1996

Which involves plotting the difference of two measurements against the mean of the two measurements and calculating and displaying limits of agreement.

Retrieve and attach the sample dataset:

```
ba <- read.table("ba.dat", header = TRUE)
attach(ba)
```

The **`ba`** data.frame contains measurements (in litres per minute) taken with a *Wright peak flow meter* and a *Mini-Wright peak flow meter.* This is the same data that is presented in the referenced Lancet article:

```
   Wright Mini
1     494  512
2     395  430
3     516  520
4     434  428
5     476  500
6     557  600
7     413  364
8     442  380
9     650  658
10    433  445
11    417  432
12    656  626
13    267  260
14    478  477
15    178  259
16    423  350
17    427  451
```

You can examine the **`ba`** data.frame using the **`print()`** and **`summary()`** functions:

```
print(ba)
ba
summary(ba)
```

The **`function()`** function allows us to create new functions in ***R***:

```
ba.plot <- function(a, b) {}
```

This creates an empty function called **`baplot`** that expects two parameters called **`a`** and **`b`**. We could type the whole function in at the ***R*** command prompt but it is easier to use a text editor:

```
fix(ba.plot)
```

This will start an editor (under Windows it will probably be the Notepad editor, under UNIX it will probably be the default system editor) with the empty **`ba.plot()`** function already loaded. We can now edit this function to make it do something useful.

Exercise 7 : Writing your own graphical functions

We will start be writing a basic function which we will gradually improve throughout this exercise. Edit the **ba.plot()** function to read:

```
function(a, b)
  {
  mean.two <- (a + b) / 2
  diff.two <- a - b
  plot(mean.two, diff.two)
  }
```

Once you have made the changes shown above, save the file and quit the editor.

The function calculates the mean and the difference of the two measures and then plots the results. Lets us try the **ba.plot()** function with the test data:

```
ba.plot(Wright, Mini)
```

The resulting plot is rather plain and lacks meaningful titles and axis labels. Use the **fix()** function to edit the **ba.plot()** function:

```
fix(ba.plot)
```

Edit the function to read:

```
function(a, b, title = "Bland and Altman Plot")
  {
  a.txt <- deparse(substitute(a))
  b.txt <- deparse(substitute(b))
  x.lab <- paste("Mean of", a.txt, "and", b.txt)
  y.lab <- paste(a.txt, "-", b.txt)
  mean.two <- (a + b) / 2
  diff.two <- a - b
  plot(mean.two, diff.two, xlab = x.lab, ylab = y.lab, main = title)
  }
```

Once you have made the changes shown above, save the file and quit the editor.

We have added a new parameter (**title**) to the function and given this a default value of **Bland and Altman Plot**. Adding **title** as a parameter means that we will be able to specify a title for the plot when we call the function. We have also used the function combination **deparse(substitute())** to retrieve the names of the vectors passed to parameters **a** and **b**. The **paste()** function pastes pieces of text together. It is used here to create the text for the axis labels used with the **plot()** function.

Let us try the **ba.plot()** function with the test data:

```
ba.plot(Wright, Mini)
```

We may also specify a title for the plot using the title parameter:

```
ba.plot(Wright, Mini, title = "Difference vs. mean for PEFR data")
```

We can now edit the function to calculate and plot mean difference and the limits of agreement.

Exercise 7 : Writing your own graphical functions

Use the **`fix()`** function to edit the **`ba.plot()`** function:

```
fix(ba.plot)
```

Edit the function to read:

```
function(a, b, title = "Bland and Altman Plot")
  {
  a.txt <- deparse(substitute(a))
  b.txt <- deparse(substitute(b))
  x.lab <- paste("Mean of", a.txt, "and", b.txt)
  y.lab <- paste(a.txt, "-", b.txt)
  mean.two <- (a + b) / 2
  diff.two <- a - b
  plot(mean.two, diff.two, xlab = x.lab, ylab = y.lab, main = title)
  mean.diff <- mean(diff.two)
  sd.diff <- sd(diff.two)
  upper <- mean.diff + 1.96 * sd.diff
  lower <- mean.diff - 1.96 * sd.diff
  lines(x = range(mean.two), y = c(mean.diff, mean.diff), lty = 3)
  lines(x = range(mean.two), y = c(upper, upper), lty = 3)
  lines(x = range(mean.two), y = c(lower, lower), lty = 3)
}
```

Once you have made the changes shown above, save the file and quit the editor.

We have used the **`mean()`** and **`sd()`** functions to calculate the mean and standard deviation of the difference between the two measures and calculated the limits of agreement (**`upper`** and **`lower`**) assuming that the differences are *Normally* distributed. The **`lines()`** function is used to plot the mean and the limits of agreement on top of the existing scatter plot.

The parameter **`lty = 3`** used with the **`lines()`** function specifies dotted lines. ***R*** provided a great number of graphical parameters that can be used to customise plots. You can see a list of these parameters using:

```
help(par)
```

These parameters can be specified for almost all graphical functions.

Lets us try the **`ba.plot()`** function with the test data:

```
ba.plot(Wright, Mini, title = "Difference vs. mean for PEFR data")
```

The function is almost complete. All that remains to do is to label the lines with the values of the mean difference and the limits of agreement.

Exercise 7 : Writing your own graphical functions

Use the **fix()** function to edit the **ba.plot()** function:

```
fix(ba.plot)
```

Edit the function to read:

```
function(a, b, title = "Bland and Altman Plot")
  {
  a.txt <- deparse(substitute(a))
  b.txt <- deparse(substitute(b))
  x.lab <- paste("Mean of", a.txt, "and", b.txt)
  y.lab <- paste(a.txt, "-", b.txt)
  mean.two <- (a + b) / 2
  diff.two <- a - b
  plot(mean.two, diff.two, xlab = x.lab, ylab = y.lab, main = title)
  mean.diff <- mean(diff.two)
  sd.diff <- sd(diff.two)
  upper <- mean.diff + 1.96 * sd.diff
  lower <- mean.diff - 1.96 * sd.diff
  lines(x = range(mean.two), y = c(mean.diff, mean.diff), lty = 3)
  lines(x = range(mean.two), y = c(upper, upper), lty = 3)
  lines(x = range(mean.two), y = c(lower, lower), lty = 3)
  mean.text  <- round(mean.diff, digits = 1)
  upper.text <- round(upper , digits = 1)
  lower.text <- round(lower, digits = 1)
  text(max(mean.two), mean.diff, mean.text, adj = c(1,1))
  text(max(mean.two), upper, upper.text, adj = c(1,1))
  text(max(mean.two), lower, lower.text, adj = c(1,1))
  }
```

Once you have made the changes shown above, save the file and quit the editor.

We have used the **round()** function to limit the display of the mean difference and the limits of agreement to one decimal place and used the **text()** function to display these (rounded) values. The **adj** parameter to the **text()** function controls the position and justification of text.

Lets us try the **ba.plot()** function with the test data:

```
ba.plot(Wright, Mini, title = "Difference vs. mean for PEFR data")
```

The graphical function is now complete.

One improvement that we could make is for the function to produce a chart and return the values of the mean difference and the limits of agreement. We would do this in exactly the same way as we would with a non-graphical function. We would return the mean difference and the limits of agreement as members of a list. We could also specify a class for the returned list and create a class specific **print()** function to produce nicely formatted output.

Exercise 7 : Returning values from graphical functions

Use the **`fix()`** function to edit the **`ba.plot()`** function:

```
fix(ba.plot)
```

Edit the function to read:

```
function(a, b, title = "Bland and Altman Plot")
  {
  a.txt <- deparse(substitute(a))
  b.txt <- deparse(substitute(b))
  x.lab <- paste("Mean of", a.txt, "and", b.txt)
  y.lab <- paste(a.txt, "-", b.txt)
  mean.two <- (a + b) / 2
  diff.two <- a - b
  plot(mean.two, diff.two, xlab = x.lab, ylab = y.lab, main = title)
  mean.diff <- mean(diff.two)
  sd.diff <- sd(diff.two)
  upper <- mean.diff + 1.96 * sd.diff
  lower <- mean.diff - 1.96 * sd.diff
  lines(x = range(mean.two), y = c(mean.diff, mean.diff), lty = 3)
  lines(x = range(mean.two), y = c(upper, upper), lty = 3)
  lines(x = range(mean.two), y = c(lower, lower), lty = 3)
  mean.text  <- round(mean.diff, digits = 1)
  upper.text <- round(upper , digits = 1)
  lower.text <- round(lower, digits = 1)
  text(max(mean.two), mean.diff, mean.text, adj = c(1,1))
  text(max(mean.two), upper, upper.text, adj = c(1,1))
  text(max(mean.two), lower, lower.text, adj = c(1,1))
  ba <- list(mean = mean.diff, limits = c(lower, upper))
  class(ba) <- "ba"
  return(ba)
  }
```

Once you have made the changes shown above, save the file and quit the editor.

Create a **`print()`** function for objects of the **`ba`** class:

```
print.ba <- function(x) {}
```

Use the **`fix()`** function to edit the new function:

```
fix(print.ba)
```

Edit the function to read:

```
function(x)
  {
  cat("Mean difference     : ", x$mean, "\n",
      "Limits of agreement : ", x$limits[1], "; ", x$limits[2], "\n",
      sep = "")
  }
```

Once you have made the changes shown above, save the file and quit the editor.

Exercise 7 : Returning values from graphical functions

Lets us try the **ba.plot()** function with the test data:

```
ba.plot(Wright, Mini, title = "Difference vs. mean for PEFR data")
```

The function produces the plot and returns the mean difference and limits of agreement as a list of class **ba** which is formatted and printed by the **print.ba()** function

We can manipulate the returned values just as we would with any other function:

```
ba.test <- ba.plot(Wright, Mini)
print(ba.test)
ba.test
ba.test$mean
ba.test$limits
ba.test$limits[1]
ba.test$limits[2]
```

You might like to use the **save()** function to save the **ba.plot()** and **print.ba()** functions before quitting ***R***.

We can now quit ***R***:

```
q()
```

For this exercise there is no need to save the workspace image so click the **No** button.

Exercise 7 : Summary

R allows you to create functions that produce graphical output.

R allows you to create functions that produce graphical output and return values.

R objects can be assigned a class or type.

R allows you to create new classes and class-specific functions that can extract and manipulate data common to the new classes.

Classes allows you to create versatile functions that return values when we need them to return values but can also produce formatted output when we need them to produce formatted output.

Classes allow you to write functions that can be chained together so that the output of one function is the input of another function.

Exercise 8 : More graphical functions

Graphical functions in ***R*** are just like any other function in ***R*** in the sense that ***R*** provides you with a set of functions which can be altered or added to. In this exercise we will experiment with some of the graphical functions provided by ***R*** to demonstrate the flexibility of graphical functions in ***R***. We will then use the graphical functions that we experiment with to create some useful graphical functions of our own.

The first function that we will develop will be a function that is capable of plotting two data series on a single graph. We will take this exercise slowly in order to introduce some further graphical functions.

Before we go any further we should retrieve a dataset:

```
mal <- read.table("malaria.dat", header = TRUE)
attach(mal)
```

The file **malaria.dat** contains data on rainfall (in mm) and the number of cases reported from health centres from an administrative district of Ethiopia between July 1997 and July 1999. The columns in this dataset are as follows:

```
Time    Month and year (as text)
Cases   Number of cases of malaria reported
Rain    Rainfall in mm
```

Examine the dataset:

```
mal
```

First we will plot the number of cases of malaria seen over time using the **plot()** function:

```
plot(Cases, type = "l")
```

The problem with this plot is that it does not treat the data as a time series. Adding the **Time** variable to the plot does not solve the problem:

```
plot(Time, Cases, type = "l")
```

Because **Time** is a factor variable. If you convert **Time** to a character variable using **as.character()** or prevent ***R*** from converting **Time** to a factor using the **as.is** parameter to the **read.table()** function the **plot()** function will return an error because it expects a numeric x-axis variable. We should, instead, specify a time series (**ts**) class object. Rather than change the original data, we will create a new object using the **ts()** function:

```
cases.ts <- ts(Cases, start = c(1997, 7), frequency = 12)
```

Examine the cases.ts object:

```
cases.ts
```

We can now plot **cases.ts** as a time series:

```
plot(cases.ts)
```

We might want to explore the association between the **Rain** and **Cases** variables. A simple scatter plot is not particularly informative:

```
plot(Rain, Cases)
```

Exercise 8 : More graphical functions

It is better to treat both variables as time series (which they are) and use the built-in **plot()** methods for objects of class **ts**:

```
rain.cases.ts <- ts(cbind(Rain, Cases), start = c(1997,7),
                    frequency = 12)
plot(rain.cases.ts)
```

The association between the **Rain** and **Cases** variables is now clearer with the number of malaria cases peaking shortly after peak in rainfall.

The **plot()** function when used with objects of class **ts** produces useful output but it is not particularly flexible and the output is, sometimes, not particularly pretty. We can however use basic graphical functions to produce multiple plots. First we will set the **mfrow** graphical parameter using the **par()** function:

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

The **par()** function sets a graphical parameter. The **mfrow** parameter is used to set the number of charts that will appear on a page in rows and columns. We have specified two rows with one chart per row. Test this by plotting two charts:

```
plot(Rain, type = "l")
plot(Cases, type = "l")
```

We will want to have tick-marks on the x-axis of each for each record. We can set the number of tick-marks on axes by setting the **lab** graphical parameter using the **par()** function:

```
par(lab = c(length(Time), 10, 7))
```

The **par()** function sets a graphical parameter. The **lab** parameter is used to set the number tick-marks on the x and y axes and the label size. We have specified a tick-mark on the x-axis for each record (i.e. using **length(Time)**), ten tick-marks on the y-axis, and a label length of seven. Test this by plotting two charts:

```
plot(Rain, type = "l")
plot(Cases, type = "l")
```

The problem with these charts is that the month and year are not displayed on the x-axis. We can get round this by plotting a chart without axes and then specifying the axes and labels directly:

```
plot(Rain, type = "l", axes = FALSE, xlab = "Time", ylab = "mm",
     main = "Rainfall")
axis(side = 1, labels = as.character(Time))
axis(side = 2)
plot(Cases, type = "l", axes = FALSE, xlab = "Time", ylab = "n",
     main = "Cases")
axis(side = 1, labels = as.character(Time))
axis(side = 2)
```

The resulting charts now look much better but it would be nice to be able draw the two lines on a single chart. Before proceeding we will use the **par()** function to specify one plot per window (using the **mfrow** parameter) and set the default number of tick-marks on the axes (using the **lab** parameter):

```
par(mfrow = c(1, 1))
par(lab = c(5, 5, 7))
```

And then use the **plot()** and **lines()** function to draw the two lines on the same graph:

```
plot(Cases, type = "l")
lines(Rain, lty = 2)
```

The problem with this is that the ranges of the two variables are different and the **plot()** function automatically sets the y-axis to the range of the specified variable. To fix this problem we need to set the limits of the y-axis to the minimum and maximum value of both of variables using the **ylim** parameter of the **plot()** function:

```
plot(Cases, type = "l", ylim = c(min(Cases, Rain), max(Cases, Rain)))
lines(Rain, lty = 2)
```

We can improve the chart by adding a legend:

```
legend(18, 1000, legend = c("Cases", "Rainfall (mm)"), lty = c(1,2))
```

We could continue to improve the chart (e.g. by adding labels for the x-axis tick-marks taken from the Time variable, specifying more meaningful axis labels, and specifying a title) but the chart would be more useful if each variable made full use of the plotting area. We can do this by plotting one chart on top of another by using the **new** graphical parameter:

```
par(lab = c(length(Time), 5, 7))
plot(Cases, type = "l", lty = 1, axes = FALSE)
axis(side = 2)
par(new = TRUE)
plot(Rain, type = "l", lty = 2, axes = FALSE)
axis(side = 4)
axis(side = 1)
```

This chart is much clearer but there are still some improvements that could be made. The chart should have a title. We can do this using the **main** parameter of either of the **plot()** functions. The y-axis labels are displayed on top of each other beside the left-hand y-axis. We can solve this problem by preventing the second **plot()** function from displaying a y-axis label (i.e. by specifying an empty character string for the **ylab** parameter). We will need to make room on the right-hand side of the chart for an axis label (i.e. by setting the **mar** (margin) graphical parameter) and place the label there ourselves (using the **mtext()** function). The x-axis should display the month and year which are held as character strings in the **Time** variable. We can do this using the **labels** parameter of the **axis()** function after setting the appropriate number of tick-marks using the **lab** graphical parameter. The x-axis should be properly labelled. We can do this using the **xlab** parameters of the **plot()** functions. An empty string must be specified for one of the **plot()** functions in order to override the default label being displayed. Try this now:

```
par(mar = c(5, 4, 4, 4))
par(lab = c(length(Time), 5, 7))
plot(Cases, type = "l", lty = 1, axes = FALSE,
     xlab = "", ylab = "", main = "Malaria cases and rainfall")
axis(side = 2)
mtext(text = "Malaria cases", side = 2, line = 2)
par(new = TRUE)
plot(Rain, type = "l", lty = 2, axes = FALSE, xlab = "Month & Year",
     ylab = "")
axis(side = 4)
mtext(text = "Rainfall (mm)", side = 4, line = 2)
axis(side = 1, labels = as.character(Time))
```

Exercise 8 : More graphical functions

Now that we know how to create a two-axis chart, we can write a function that we will be able to use whenever we need to plot two variables on the same chart. Create a new function called **`plot2var()`**:

```
plot2var <- function() {}
```

This creates an empty function called **`plot2var()`**. Use the **`fix()`** function to edit the **`plot2var()`** function:

```
fix(plot2var)
```

Edit the function to read:

```
function(y1,
         y2,
         x.ticks,
         x.lab = deparse(substitute(x.ticks)),
         y1.lab = deparse(substitute(y1)),
         y2.lab = deparse(substitute(y2)),
         main = paste(y1.lab, "&", y2.lab)
        )
  {
  old.par.mar <- par("mar")
  old.par.lab <- par("lab")
  par(mar = c(5, 4, 4, 4))
  if(!missing(x.ticks))
    {
    par(lab = c(length(x.ticks), 5, 7))
    }
  plot(y1, type = "l", lty = 1, axes = FALSE, xlab = "",
       ylab = "", main = main)
  axis(side = 2)
  mtext(text = y1.lab, side = 2, line = 2)
  par(new = TRUE)
  plot(y2, type = "l", lty = 2, axes = FALSE, ylab = "",
       xlab = x.lab)
  axis(side = 4)
  mtext(text = y2.lab, side = 4, line = 2)
  if(!missing(x.ticks))
    {
    axis(side = 1, labels = as.character(x.ticks))
    } else {axis(side = 1)}
  par(mar = old.par.mar)
  par(lab = old.par.lab)
  }
```

Once you have made the changes shown above, save the file and quit the editor.

Note that with this function we have given some of the parameters default values in the function definition and we have also used the **`if()`** function to check whether the user specified a value for the **`x.ticks`** parameter. We also save and restore the graphical parameters **`mar`** and **`lab`** so as to prevent changes to these parameters in the **`plot2var()`** function affecting other graphical functions.

Exercise 8 : More graphical functions

Lets us try the **`plot2var()`** function with the test data:

```
plot2var(Rain, Cases)
plot2var(Rain, Cases, Time)
```

Note how the function has used default values for the axis labels and chart title. We can override these default values if we want to:

```
plot2var(Rain, Cases, Time, x.lab = "Month and Year",
         y1.lab = "Rainfall (mm)", y2.lab = "Cases of malaria")
```

You might like to use the **`save()`** function to save the **`plot2var()`** function.

As an exercise you might want to edit the **`plot2var()`** function to automatically add a legend to the two-axis chart using the **`legend()`** function with **`y1.lab`** and **`y2.lab`**.

Exercise 8 : More graphical functions

Another common chart type that is not available in many statistical applications is the population pyramid. Before we go any further we should retrieve a dataset:

```
pop <- read.table("pop.dat", header = TRUE)
attach(pop)
```

The file **pop.dat** contains data on the age (in months) and sex of 438 children aged between six and sixty months collected as part of a nutritional anthropometry survey of the Khosh Valley in Northeast Afghanistan. The columns in this dataset are as follows:

```
AGE     Age of the child in months
SEX     Sex of the child (M/F)
```

Examine the first twenty records of the dataset:

```
pop[1:20, ]
```

The first step is to make groups from the **AGE** variable. Many ages are biased towards full years:

```
table(AGE)
barplot(table(AGE), col = "white")
```

So we will centre the age-groups around the months representing full years:

```
age.group <- cut(AGE, c(0, 17, 29, 41, 53, 99))
```

We can check that the grouping operation has worked as expected by tabulating **AGE** and **age.group**:

```
table(AGE, age.group)
```

We now use the **table()** function to produce the summary data for the population pyramid:

```
table(age.group, SEX)
```

We will construct our population pyramid using the **barplot()** function:

```
barplot(table(age.group, SEX))
```

The default behaviour of the **barplot()** function is to produce stacked bars. We can set the **beside** parameter to display the bars side-by-side:

```
barplot(table(age.group, SEX), beside = TRUE)
```

We can also use the **horiz** parameter to present the data as horizontal bars:

```
barplot(table(age.group, SEX), beside = TRUE, horiz = TRUE)
```

Exercise 8 : More graphical functions

In order to centre the bars around zero we need to make one column of the summary data table contain negative numbers:

```
tab <- table(age.group, SEX)
tab
tab[ ,1] <- -tab[ ,1]
tab
barplot(tab, beside = TRUE, horiz = TRUE)
```

This is looking better but we need to shift the second set of bars down beside the first set of bars using the **space** parameter:

```
barplot(tab, beside = TRUE, horiz = TRUE, space = c(0, -nrow(tab)))
```

The axis labels are wrong but we can fix that using the **names.arg** parameter:

```
bar.names <- c(dimnames(tab)$age.group, dimnames(tab)$age.group)
barplot(tab, beside = TRUE, horiz = TRUE, space = c(0, -nrow(tab)),
        names.arg = bar.names)
```

The chart can still be improved upon by making the fill-colour of each bar white and by expanding the x-axis slightly so that the bars do not touch the y-axis:

```
barplot(tab, beside = TRUE, horiz = TRUE, space = c(0, -nrow(tab)),
        col = "white", xlim = c(min(tab) * 1.1, max(tab) * 1.1),
        names.arg = bar.names)
```

Now we know how to create a population pyramid, we can write a function that we will be able to use whenever we need to plot a population pyramid. Create a new function called **pyramid.plot()**:

```
pyramid.plot <- function() {}
```

This creates an empty function called **pyramid.plot()**. Use the **fix()** function to edit the **pyramid.plot()** function:

```
fix(pyramid.plot)
```

Edit the function to read:

```
function(x,
         g,
         main = paste("Pyramid plot of", deparse(substitute(x)),
                      "by", deparse(substitute(g))),
         xlab = paste(deparse(substitute(g)), "(", levels(g)[1], "/",
                      levels(g)[2],")"),
         ylab = deparse(substitute(x))
        )
  {
  tab <- table(x, g); tab[ ,1] <- -tab[ ,1]
  barplot(tab, horiz = TRUE, beside = TRUE, space = c(0, -nrow(tab)),
          names.arg = c(dimnames(tab)$x, dimnames(tab)$x),
          xlim = c(min(tab) * 1.1, max(tab) * 1.1), col = "white",
          main = main, xlab = xlab, ylab = ylab)
  }
```

Exercise 8 : More graphical functions

Note that with this function we have given some of the parameters default values in the function definition. Giving default values to parameters is useful because it means that you do not need to specify parameters such as titles and axis labels unless you want to. Many ***R*** functions use default parameters which are usually set to the most frequently used values.

Lets us try the **`pyramid.plot()`** function with the test data:

```
pyramid.plot(age.group, SEX)
```

Note how the function has used default values for the axis labels and chart titles. We can override these default values if we want to:

```
pyramid.plot(age.group, SEX, ylab = "Months", xlab = "Sex F / M",
             main = "Number of children by age and sex")
```

You might like to use the **`save()`** function to save the **`pyramid.plot()`** function.

Exercise 8 : More graphical functions

Another type of chart that is missing from many statistical applications is the *Pareto* chart which is a bar chart where the bars are sorted by the bar value with the largest bar drawn first. Such a chart is easier to interpret than a pie chart particularly when there are more than a few categories being plotted.

Before we go any further we should retrieve a dataset:

```
sssw <- read.table("sssw.dat", header = TRUE)
attach(sssw)
```

The file **sssw.dat** contains data on the marital status, home circumstances, and ethnic group of 152 persons recruited into a study into the levels of stress experienced by student social workers in the United Kingdom. The columns in this dataset are as follows:

```
marital     Marital status coded as:

                   1 = Married
                   2 = Single
                   3 = Divorced
                   4 = Separated
                   5 = Cohabiting
                   6 = Widowed

living      Living with ... coded as:

                   1 = Alone
                   2 = Parents or siblings
                   3 = Partner
                   4 = Partner and children
                   5 = Children
                   6 = Friends or colleagues

ethnic      Ethnic group coded as:

                   1 = African
                   2 = West-Indian
                   3 = Indian
                   4 = Pakistani
                   5 = Bangladeshi
                   6 = East African Asian
                   7 = Chinese
                   8 = Cypriot
                   9 = Black European
                  10 = White European
                  11 = Other
```

Examine the dataset:

```
sssw[1:20, ]
```

Producing a bar chart from this data is simple as long as we remember to pass summary data (i.e. created using the **table()** function) to the **barplot()** function instead of the variable name:

```
barplot(table(marital))
barplot(table(living))
barplot(table(ethnic))
```

Exercise 8 : More graphical functions

Creating a *Pareto* chart only requires us to sort the summary data. We do this using the **rev()** and **sort()** functions:

```
barplot(rev(sort(table(marital))))
```

Having to specify **rev(sort(table(***variable***)))** each time we want to produce a *Pareto* plot is rather tedious but now that we know how to create a *Pareto* chart, we can write a function that we will be able to use whenever we need to plot a *Pareto* chart. Create a new function called **pareto()**:

```
pareto <- function() {}
```

This creates an empty function called **pareto()**. Use the **fix()** function to edit the **pareto()** function:

```
fix(pareto)
```

Edit the function to read:

```
function(x,
         xlab = deparse(substitute(x)),
         ylab = "Count",
         main = paste("Pareto Chart of", deparse(substitute(x)))
        )
  {
  barplot(rev(sort(table(x))),
          xlab = xlab,
          ylab = ylab,
          main = main,
          col = "white")
  }
```

Lets us try the **pareto()** function with the test data:

```
pareto(marital)
```

Note how the function has used default values for the axis labels and chart titles. We can override these default values if we want to:

```
pareto(marital, ylab = "n", xlab = "Marital Status",
       main = "Marital Status")
```

Note that we can use value labels if the variable we plot is a **factor** with value labels as **levels** rather than a simple **numeric** vector:

```
ms <- as.factor(marital)
levels(ms) <- c("Married", "Single", "Divorced", "Separated",
                "Cohabiting", "Widowed")
table(ms)
pareto(ms, xlab = "Marital Status", main = "Marital Status")
```

You might like to use the **save()** function to save the **pareto()** function.

Exercise 8 : More graphical functions

You may want to plot your data with confidence intervals or error bars. ***R*** does not have a function to do this but it is a relatively simple matter to write a function to do so. On the way we will use some of ***R***'s data management functions as well.

Before we go any further we should retrieve a dataset:

```
diets <- read.table("diets.dat", header = TRUE)
```

The file **diets.dat** contains data from a trial of two different diets undertaken at an adult therapeutic feeding centre in Somalia. The columns in this dataset are as follows:

day The day after start of diet that measurements were taken

oedema Type of undernutrition coded as:

1 = Oedematous
2 = Marasmic

diet The trial diets coded as:

LP = Low protein
HP = High protein

wt Mean weight change (weight velocity) in g / kg / day since the previous measurement

sd Standard deviation of weight change in g / kg / day

n Number of subjects at each observation

Examine the dataset:

```
diets
```

Note that the dataset contains a summary of the results from the four trial arms:

```
oedema == 1, diet == "HP"
oedema == 2, diet == "HP"
oedema == 1, diet == "LP"
oedema == 2, diet == "LP"
```

With observations at 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, and 33 days after admission.

We can calculate a confidence interval for the mean weight velocity (**wt**) using the data in the **sd** and **n** variables. We will use the **transform()** function to do this:

```
diets <- transform(diets, lci = wt - sd / sqrt(n),
                          uci = wt + sd / sqrt(n))
```

In this case we are calculating confidence intervals as plus or minus one standard error of the mean. The **transform()** function is very useful as it can add columns directly to a data.frame or transform data already stored in a data.frame.

Exercise 8 : More graphical functions

Examine the **diets** data.frame:

```
diets
```

Two new columns (**lci** and **uci**) have been added.

Now that we have calculated the confidence intervals we should, for convenience, split the **diets** data.frame into four separate data.frames (one for each arm of the trial):

```
oed.hp <- subset(diets, oedema == 1 & diet == "HP")
oed.lp <- subset(diets, oedema == 1 & diet == "LP")
mar.hp <- subset(diets, oedema == 2 & diet == "HP")
mar.lp <- subset(diets, oedema == 2 & diet == "LP")
```

Check that each data.frame contains the data that you expect it to:

```
oed.hp
oed.lp
mar.hp
mar.lp
```

We can now plot the data for one arm of the trial:

```
plot(oed.hp$day, oed.hp$wt, type = "l")
```

We can add error bars using the **arrows()** function:

```
arrows(oed.hp$day, oed.hp$lci, oed.hp$day, oed.hp$uci,
       code=3, angle=90, length=0.1)
```

The scale of the *y* axis is wrong because the **plot()** function automatically scales axes to the ranges of the *x* and *y* data it is given. We can fix this by specifying a different set of limits (from **lci** and **uci**) for the *y* axis using the **ylim** parameter:

```
plot(oed.hp$day, oed.hp$wt, type = "l",
     ylim = c(min(oed.hp$lci), max(oed.hp$uci)))

arrows(oed.hp$day, oed.hp$lci, oed.hp$day, oed.hp$uci,
       code=3, angle=90, length=0.1)
```

The plot might also be improved by adding plotting symbols:

```
points(oed.hp$day, oed.hp$wt)
```

Now that we know how to plot error bars, we can write a function that we will be able to use whenever we need to plot data with error bars.

Exercise 8 : More graphical functions

Before continuing we will consider what the new function should be able to do. This will help us when it comes to writing the function. Our new function should:

1. Take four numeric vectors (x, y, lower CI for y, and upper CI for y) and plot them.
2. Be able to plot the data points as unconnected points or as points joined by lines.
3. Calculate appropriate limits for the *y* axis.
4. Produce a plot without axes so that more than one data series may be plotted on the same chart.
5. Provide sensible default values for axis limits and labels.

From this list we know that we need the function to take several parameters:

Name	Purpose	Default value
`x`	Data to plot	None
`y`	Data to plot	None
`y.lci`	Data to plot	None
`y.uci`	Data to plot	None
`ylim`	Limits for *y* axis	**`c(min(y.lci), max(y.uci))`**
`xlab`	Label for *x* axis	**`deparse(substitute(x))`**
`ylab`	Label for *y* axis	**`deparse(substitute(y))`**
`main`	Chart title	**`paste(ylab, "by", xlab)`**
`type`	Type of plot	**`"l"`**
`lty`	Line type	**`1`**
`axes`	Draw x and y axes	**`TRUE`**
`pch`	Type of points to plot	**`1`**

The parameter names have been chosen to be the same as the parameter names to **`plot()`** and **`points()`**. This makes the function easier to use. It also makes the function easier to write.

Create a new function called **`plot.ci()`**:

```
plot.ci <- function() {}
```

This creates an empty function called **`plot.ci()`**. Use the **`fix()`** function to edit the **`plot.ci()`** function:

```
fix(plot.ci)
```

Exercise 8 : More graphical functions

Edit the function to read:

```
function(x,
         y,
         y.lci,
         y.uci,
         ylim = c(min(y.lci), max(y.uci)),
         xlab = deparse(substitute(x)),
         ylab = deparse(substitute(y)),
         main = paste(ylab, "by", xlab),
         type = "l",
         lty = 1,
         axes = TRUE,
         pch = 1
        )
  {
  plot(x, y, type = type, ylim = ylim, xlab = xlab, ylab = ylab,
       main = main, lty = lty, axes = axes)
  points(x, y, pch = pch)
  arrows(x, y.lci, x, y.uci, code=3, angle=90, length=0.1, lty = lty)
  }
```

Lets us try the **plot.ci()** function with the test data:

```
plot.ci(oed.hp$day, oed.hp$wt, oed.hp$lci, oed.hp$uci)
```

Note how the function has used default values for the axis labels and chart titles. We can override these default values if we want to:

```
plot.ci(oed.hp$day, oed.hp$wt, oed.hp$lci, oed.hp$uci,
        ylim = c(-6, 10), xlab = "Day",
        ylab = "Weight gain (g/kg/day)",
        main = "Oedematous")
```

We should also check that we can plot another data series on this chart:

```
par(new = TRUE)
plot.ci(oed.lp$day, oed.lp$wt, oed.lp$lci, oed.lp$uci, lty = 2,
        axes = FALSE, pch = 2, xlab = "", ylab = "", main = "")
```

We can also add a legend:

```
legend(5, 8, legend = c("High protein", "Low protein"),
       lty = c(1,2), pch = c(1, 2))
```

We should also check that we can produce plots of unconnected points:

```
plot.ci(oed.hp$day, oed.hp$wt, oed.hp$lci, oed.hp$uci, type = "p")
```

Try plotting the data for the marasmic patients using the **plot.ci()** function.

You might like to use the **save()** function to save the **plot.ci()** function.

Exercise 8 : More graphical functions

You can use a similar technique to add error bars to different types of plot. If we plot the weight velocities for oedematous patients receiving the high protein diet as a bar chart:

```
barplot(oed.hp$wt, names.arg = oed.hp$day, col = "white",
        ylim = c(min(oed.hp$lci), max(oed.hp$uci)))
```

We could add error bars using the **arrows()** function as we did with a line plot:

```
arrows(oed.hp$day, oed.hp$lci, oed.hp$day, oed.hp$uci,
       code=3, angle=90, length=0.1)
```

But this does not produce the expected results because the centres of the bars are not placed on the chart at the positions held in **oed.hp$day**. This is easily fixed as **barplot()** returns a numeric vector (or matrix, when **beside = TRUE**) containing the co-ordinates of the bar midpoints:

```
bar.positions <- barplot(oed.hp$wt, names.arg = oed.hp$day,
                         ylim = c(min(oed.hp$lci), max(oed.hp$uci)),
                         col = "white")
bar.positions
```

We can now use the information stored in **bar.positions** to specify the positions of the error bars:

```
arrows(bar.positions, oed.hp$lci, bar.positions, oed.hp$uci,
       code=3, angle=90, length=0.1)
```

Armed with this information, we can write a function that we will be able to use whenever we need to plot a bar chart with error bars. Create a new function called **barplot.ci()**:

```
barplot.ci <- function() {}
```

This creates an empty function called **barplot.ci()**. Use the **fix()** function to edit the **barplot.ci()** function:

```
fix(barplot.ci)
```

Edit the function to read:

```
function(y, bar.names, lci, uci, ylim = c(min(lci), max(uci)),
         xlab = deparse(substitute(bar.names)),
         ylab = deparse(substitute(y)),
         main = paste(ylab, "by", xlab)
         )
  {
  bp <- barplot(y, names.arg = bar.names, ylim = ylim, xlab = xlab,
                ylab = ylab, main = main, col = "white")
  arrows(bp, lci, bp, uci, code=3, angle=90, length=0.1)
  }
```

Lets us try the **barplot.ci()** function with the test data:

```
barplot.ci(oed.hp$wt, oed.hp$day, oed.hp$lci, oed.hp$uci)
```

Exercise 8 : More graphical functions

Try plotting the weight velocities for the marasmic patients receiving the high protein diet using the **barplot.ci()** function:

```
barplot.ci(mar.hp$wt, mar.hp$day, mar.hp$lci, mar.hp$uci)
```

The chart looks wrong. This is because we have set the wrong limits for the *y* axis. The bars are drawn from zero to the data point but we have specified a limit for the *y* axis that is not constrained to include zero. This is easy to fix. Edit the **barplot.ci()** function to read:

```
function(y, bar.names, lci, uci, ylim = c(min(0, lci), max(0, uci)),
         xlab = deparse(substitute(bar.names)),
         ylab = deparse(substitute(y)),
         main = paste(ylab, "by", xlab)
         )
  {
  bp <- barplot(y, names.arg = bar.names, ylim = ylim, xlab = xlab,
              ylab = ylab, main = main, col = "white")
  arrows(bp, lci, bp, uci, code=3, angle=90, length=0.1)
  }
```

Try plotting the data for the marasmic patients using the **barplot.ci()** function:

```
barplot.ci(mar.hp$wt, mar.hp$day, mar.hp$lci, mar.hp$uci)
```

Check that the function still operates as expected with the data for oedematous patients:

```
barplot.ci(oed.hp$wt, oed.hp$day, oed.hp$lci, oed.hp$uci)
```

The end of one of the error bars touches the *x* axis. This can also be fixed by slightly widening the limits for the *y* axis. It might also be useful to plot the centre position of each error bar. We can use the **points()** function to do this. Edit the **barplot.ci()** function to read:

```
function(y, bar.names, lci, uci, ylim = c(min(0, lci), max(0, uci)),
         xlab = deparse(substitute(bar.names)),
         ylab = deparse(substitute(y)),
         main = paste(ylab, "by", xlab)
         )
  {
  ylim <- ylim * 1.1
  bp <- barplot(y, names.arg = bar.names, ylim = ylim, xlab = xlab,
              ylab = ylab, main = main, col = "white")
  arrows(bp, lci, bp, uci, code=3, angle=90, length=0.1)
  points(bp, y)
  }
```

And check that the function works as expected:

```
barplot.ci(oed.hp$wt, oed.hp$day, oed.hp$lci, oed.hp$uci)
barplot.ci(mar.hp$wt, mar.hp$day, mar.hp$lci, mar.hp$uci)
```

The **barplot.ci()** function now works as expected with both sets of data. It is important, when developing your own functions, to test them with different data so as to ensure that they work correctly with a wide range if data.

You might like to use the **save()** function to save the **barplot.ci()** function.

Exercise 8 : More graphical functions

The techniques introduced in this section allow you to write custom graphical functions but they can also be used to change the default behaviour of standard graphical functions.

In Exercise 1 (Getting acquainted with ***R*** for data analysis), we saw how the **`plot()`** function could be applied to a fitted object:

```
fem <- read.table("fem.dat", header = TRUE)
attach(fem)
fem.lm <- lm(WEIGHT ~ AGE)
plot(fem.lm)
```

Each of the diagnostic plots are presented as a separate chart. We could use the **`mfrow`** parameter of the **`par()`** function to present all four diagnostic plots on a single chart:

```
par(mfrow = c(2, 2))
plot(fem.lm)
```

It might improve the appearance of the chart if each of the diagnostic plots were square rather than rectangular:

```
par(pty = "s")
plot(fem.lm)
```

Graphical parameters set using the **`par()`** function affect all subsequent plot commands and must be reset explicitly:

```
par(mfrow = c(1, 1), pty = "m")
plot(fem.lm)
```

It is possible to save graphical parameters into an ***R*** object and use this object to restore original graphical parameters:

```
old.par <- par()
par(mfrow = c(2, 2), pty = "s")
plot(fem.lm)
par(old.par)
plot(fem.lm)
```

The ability to save and apply graphical parameters means that you can create a library of graphical parameter sets that can be applied with the **`par()`** function as required:

```
default.par <- par()
par(mfrow = c(2, 2), pty = "s")
plot.lm.par <- par()
par(default.par)
plot(fem.lm)
par(plot.lm.par)
plot(fem.lm)
par(default.par)
plot(fem.lm)
```

R produces warning messages when you save and restore graphical parameters in this way. This is because some graphical parameters are read only and cannot be changed using the **`par()`** function. This has no effect other than to cause ***R*** to issue warning messages.

Exercise 8 : More graphical functions

If you do not like the warning messages than you can use the **par()** function with **no.readonly** parameter:

```
default.par <- par(no.readonly = TRUE)
par(mfrow = c(2, 2), pty = "s")
plot.lm.par <- par(no.readonly = TRUE)
par(default.par)
plot(fem.lm)
par(plot.lm.par)
plot(fem.lm)
par(default.par)
plot(fem.lm)
```

Graphical parameter sets, like any other ***R*** object, may be saved and loaded using the **`save()`** and **`load()`** functions.

Exercise 8 : Summary

R allows you to create functions that produce graphical output.

R graphical functions are flexible so that you can create functions that can produce chart types that are not available in ***R*** or many other statistical applications. Standard plots may also be customised using the **`par()`** function.

R allows you to specify default values for function parameters making functions calls easier by removing the requirement to specify values for every function parameter.

Exercise 9 : Managing and analysing survey data

In this exercise we will use ***R*** to analyse some survey data. Along the way we will use some more of ***R***'s data management functions as well as making use of some of the functions we wrote earlier in this tutorial.

The functions we have written so far are stored in a file called **nhvfunctions.r**. This is a plain text file containing the function definitions for the **`tab2by2()`**, **`lreg.or()`**, **`v2m.test()`**, **`rr22()`**, **`print.rr22()`**, **`ba.plot()`**, **`print.ba()`**, **`plot2var()`**, **`pyramid.plot()`**, **`pareto()`**, **`plot.ci()`**, and **`barplot.ci()`** functions. You can retrieve these functions with the **`source()`** function:

```
source("nhvfunctions.r")
```

Before continuing, we will retrieve a dataset:

```
svy <- read.table("nut.dat", header = TRUE)
```

The file **`nut.dat`** contains a subset of data from a survey of nutritional status of children aged between 6 and 60 months of between 65 and 110 cm in height. The columns in the dataset are:

```
age         Age of child in months
sex         Sex of child (1 = male, 2 = female)
ht          Height of child (cm)
wt          Weight of child (kg)
muac        Mid-upper arm circumference (cm)
oedema      Bilateral pitting oedema
dia         Diarrhoea in previous 14 days (mother's recall)
fev         Fever in previous 14 days (mother's recall)
bf          Breast feeding (1 = no, 2 = exclusive, 3 = mixed)
```

Examine the first ten records in the **`svy`** data.frame:

```
svy[1:10, ]
```

One way of measuring nutritional status is to use *z-scores*. To do this we express the difference between a weight and the average weight for any give height and sex in a *reference population* as the number of standard deviations it is away from the average weight for a given height and sex in the reference population. If we know the weight, height, and sex of a child and also know the average weight and the standard deviation for children of the same height and sex in the reference population we can calculate a *weight-for-height z-score*:

$$z = \frac{wt - mean}{s}$$

Where:

z	=	*z-score*
wt	=	weight of child
mean	=	mean weight of child for same height and sex in the reference population
s	=	standard deviation of mean

A child is considered to be *moderately undernourished* if they have a weight-for-height z-score of less than -2.00. A child is considered to be *severely undernourished* if they have a weight-for-height z-score of less than -3.00 or have bilateral pitting oedema.

Exercise 9 : Managing and analysing survey data

The file **whz.dat** contains the reference data. We will use this data to calculate weight-for-height z-scores for the surveyed children. Retrieve the reference data:

```
whz <- read.table("whz.dat", header = TRUE)
```

Examine the first ten records in the **whz** data.frame:

```
whz[1:10, ]
```

The first ten records of the **whz** data.frame are:

```
htsex median  sd
65.01    7.1 0.7
65.02    7.0 0.7
65.51    7.3 0.8
65.52    7.1 0.7
66.01    7.4 0.7
66.02    7.3 0.8
66.51    7.6 0.8
66.52    7.4 0.7
67.01    7.7 0.7
67.02    7.5 0.7
```

The **htsex** column is a *composite identifier* composed from height (cm) and sex (coded 1 = male, 2 = female). For example, 65.52 is used to identify the row containing the reference data for 65.5 cm tall girls.

The **median** column contains median weights for given heights and sexes in the reference population. The median rather than the mean is used because nutritional status is sometimes measured using percentages of the median weight in the reference population. This makes little practical difference as the reference data is very nearly normally distributed and the mean and median are virtually identical.

The **sd** column contains the standard deviation of weights for given heights and sexes in the reference population.

In order to calculate weight-for-height z-scores for the surveyed children in the **svy** data.frame we need to retrieve the appropriate reference data from the **whz** data.frame. To do this, we need to create a composite identifier for each record in the **svy** data.frame. The composite identifier should be composed of the value of **ht** rounded to the nearest 0.5 cm plus **sex / 100.**

First we will use the **trunc()** function to return the decimal part of **ht**:

```
dec.ht <- svy$ht - trunc(svy$ht)
```

And check the result:

```
cbind(svy$ht, dec.ht)[1:10, ]
```

And then use the **cut()** function to group **dec.ht** and checking the result:

```
grp.dec.ht <- as.numeric(cut(dec.ht,
                             breaks = c(0, 0.25, 0.75, 1),
                             include.lowest = TRUE))
cbind(svy$ht, dec.ht, grp.dec.ht)[1:10, ]
```

These groups can then be used to create a new variable corresponding to the value of **ht** rounded to the nearest 0.5 cm:

```
tmp.ht <- vector(mode = "numeric")
tmp.ht[grp.dec.ht == 1] <- trunc(svy$ht[grp.dec.ht == 1])
tmp.ht[grp.dec.ht == 2] <- trunc(svy$ht[grp.dec.ht == 2]) + 0.5
tmp.ht[grp.dec.ht == 3] <- trunc(svy$ht[grp.dec.ht == 3]) + 1
cbind(svy$ht, dec.ht, grp.dec.ht, tmp.ht)[1:10, ]
```

The new variable can then be combined with **sex** (as **sex / 100**) and added to the **svy** data.frame:

```
svy <- transform(svy, htsex = tmp.ht + sex / 100)
svy[1:10, ]
```

Now that both data.frames have a composite identifier (**htsex**), the two data.frames can be joined using the **merge()** function:

```
svy <- merge(svy, whz, by = "htsex")
svy[1:10, ]
```

It is now an easy matter to calculate the weight-for-height z-score and add these to the **svy** data.frame:

```
svy <- transform(svy, z = (wt - median) / sd)
svy[1:10, ]
```

Now that we have calculated the z-scores we can drop the reference data from the **svy** data.frame. We can do this by selecting only the columns that want from the **svy** data.frame (i.e. 2 = **age**, 3 = **sex**, 4 = **ht**, 5 = **wt**, 6 = **muac**, 7 = **oedema**, 8 = **dia**, 9 = **fev**, 10 = **bf**, 13 = **z**):

```
svy <- svy[ ,c(2:10, 13)]
svy[1:10, ]
```

We can now apply our case definitions for undernutrition:

```
un <- as.numeric(cut(svy$z, breaks = c(-99, -3.01, -2.01, 99)))
un[svy$oedema == 1] <- 1
cbind(svy$z, svy$oedema, un)[1:20, ]
table(svy$oedema, un)
```

The vector **un** is coded:

1 = severe undernutrition	(**z <= -3.01 | oedema == 1**)
2 = moderate undernutrition	(**z <= -2.01 & z >= -3.01**)
3 = adequately nourished	(**z >= -2.00**)

We might also want to create a vector that indicates whether a child is undernourished or not:

```
global <- vector(mode = "numeric")
global[un == 1 | un ==2] <- 1
global[un == 3] <- 2
cbind(un, global)[1:20, ]
table(un, global)
```

Exercise 9 : Managing and analysing survey data

Another definition of undernutrition is based on mid-upper-arm-circumference (**muac**) using the cut-points < 11.0 (severe) and < 12.5 (moderate). The presence of bilateral pitting oedema is also indicative of severe undernutrition. This definition can, however, only be applied to children aged one year or older:

```
un.muac <- as.numeric(cut(svy$muac,
                          breaks = c(0, 10.9, 12.4, 99)))
un.muac[svy$oedema == 1] <- 1
un.muac[svy$age < 12] <- NA
cbind(svy$age, svy$muac, svy$oedema, un.muac)
table(svy$oedema, un.muac)
global.muac <- vector(mode = "numeric")
global.muac[un.muac == 1 | un.muac == 2] <- 1
global.muac[un.muac == 3] <- 2
cbind(un.muac, global.muac)
table(un.muac, global.muac)
```

We might also like to make groups from the **svy$age** variable. Many ages are biased towards full years:

```
table(svy$age)
barplot(table(svy$age), col = "white")
```

So we will centre the age-groups around the months representing full years:

```
age.group <- cut(svy$age, c(0, 17, 29, 41, 53, 99))
```

We can check that the grouping operation has worked as expected by tabulating **svy$age** and **age.group**:

```
table(svy$age, age.group)
```

We can add these new columns to the **svy** data.frame:

```
svy <- cbind(svy, un, global, un.muac, global.muac, age.group)
svy[1:10, ]
```

Analysis and interpretation of data might be simplified if we specify which variables are factors and specify value labels. We will try this first for **svy$sex**:

```
table(svy$sex)
svy$sex <- as.factor(svy$sex)
levels(svy$sex) <- c("Male", "Female")
table(svy$sex)
levels(svy$sex)
svy[1:10, ]
```

Note that the output now shows the value labels rather the numeric codes. We can also use value labels in index expressions:

```
table(svy$sex)["Male"]
table(svy$sex)["Female"]
```

We should specify values labels for the other factors:

```
svy$oedema <- as.factor(svy$oedema)
levels(svy$oedema) <- c("Yes", "No")
svy$dia <- as.factor(svy$dia)
levels(svy$dia) <- c("Yes", "No")
svy$fev <- as.factor(svy$fev)
levels(svy$fev) <- c("Yes", "No")
svy$bf <- as.factor(svy$bf)
levels(svy$bf) <- c("No", "Exclusive", "Mixed")
svy$un <- as.factor(svy$un)
levels(svy$un) <- c("Severe", "Moderate", "Adequate")
svy$global <- as.factor(svy$global)
levels(svy$global) <- c("Undernourished", "Adequate")
svy$un.muac <- as.factor(svy$un.muac)
levels(svy$un.muac) <- c("Severe", "Moderate", "Adequate")
svy$global.muac <- as.factor(svy$global.muac)
levels(svy$global.muac) <- c("Undernourished", "Adequate")
```

Check that the levels have been correctly specified:

```
svy[1:20, ]
```

Having done so much work on the **svy** data.frame, we should save it:

```
save(svy, file = "svydat.r")
```

We can now start describing and analysing the survey dataset.

We can examine the sex ratio of the sampled children using the **table()** function:

```
table(svy$sex)
```

This might be better expressed as proportions:

```
table(svy$sex) / sum(table(svy$sex))
```

The **prop.tables()** function produces similar output:

```
prop.table(table(svy$sex))
prop.table(table(svy$sex)) * 100
```

We can also use the **table()** function to describe the number of children sampled by age and sex:

```
table(svy$age.group, svy$sex)
```

We can use the **prop.table()** function with **margin = 1** (where **1** = rows and **2** = columns) to calculate and display the proportions of males and females (i.e. the row proportions) in each age group:

```
prop.table(table(svy$age.group, svy$sex), margin = 1)
```

This data may be better displayed using the **pyramid.plot()** function that we developed earlier:

```
pyramid.plot(svy$age.group, svy$sex)
```

Exercise 9 : Managing and analysing survey data

Examine the prevalence of undernutrition using the **table()** and **prop.table()** functions:

```
table(svy$un)
prop.table(table(svy$un))
table(svy$global)
prop.table(table(svy$global))
```

The **prop.test()** and **binom.test()** functions calculate confidence intervals for a single proportion:

```
prop.test(table(svy$global)["Undernourished"],
          sum(table(svy$global)))
binom.test(table(svy$global)["Undernourished"],
           sum(table(svy$global)))
```

Examine the association between sex and undernutrition:

```
table(svy$sex, svy$global)
chisq.test(table(svy$sex, svy$global))
fisher.test(table(svy$sex, svy$global))
```

We could also use the **tab2by2()** or **rr22()** that we developed earlier to examine the association:

```
tab2by2(svy$sex, svy$global)
rr22(svy$sex, svy$global)
```

Examine the association between age and undernutrition:

```
table(svy$age.group, svy$global)
chisq.test(table(svy$age.group, svy$global))
```

There appears to be a reasonably linear decrease in prevalence with increasing age:

```
prop.table(table(svy$age.group, svy$global), margin = 1)
plot(table(svy$age.group, svy$global))
```

We can test this using the **prop.trend.test()** function:

```
tab <- table(svy$age.group, svy$global)
events <- tab[ ,1]
trials <- tab[ ,1] + tab[ ,2]
prop.trend.test(events, trials)
```

Examine the association between diarrhoea (**dia**) and undernutrition (**global**):

```
tab2by2(svy$dia, svy$global)
```

Examine the association between fever (**fev**) and undernutrition (**global**):

```
tab2by2(svy$fev, svy$global)
```

Both variables are associated with undernutrition but they are also associated with age:

```
table(svy$age.group, svy$dia)
tab <- table(svy$age.group, svy$dia)
prop.table(tab, margin = 1)
plot(tab)
events <- tab[ ,1]
trials <- tab[ ,1] + tab[ ,2]
prop.trend.test(events, trials)

table(svy$age.group, svy$fev)
tab <- table(svy$age.group, svy$fev)
prop.table(tab, margin = 1)
plot(tab)
events <- tab[ ,1]
trials <- tab[ ,1] + tab[ ,2]
prop.trend.test(events, trials)
```

As is breast feeding:

```
table(svy$age.group, svy$bf)
chisq.test(table(svy$age.group, svy$bf))
prop.table(table(svy$age.group, svy$bf), margin = 1)
plot(table(svy$age.group, svy$bf), col = c("white", "gray", "gray"))
on.breast <- vector(mode = "numeric")
on.breast[svy$bf == "Mixed" | svy$bf == "Exclusive"] <- 1
on.breast[svy$bf == "No"] <- 2
on.breast <- as.factor(on.breast)
levels(on.breast) <- c("Yes", "No")
table(svy$bf, on.breast)
tab2by2(on.breast, svy$global)
```

Which is also associated with both diarrhoea (**dia**) and fever (**fev**):

```
tab2by2(on.breast, svy$dia)
tab2by2(on.breast, svy$fev)
```

Some of these associations may be due to confounding. We can use logistic regression to help us identify independent associations. We will first create a working data.frame with each variable as numbers rather than factors:

```
lr.df <- data.frame(as.numeric(svy$global),
                    as.numeric(svy$dia), as.numeric(svy$fev),
                    svy$age, as.numeric(on.breast))
lr.df[1:10, ]
names(lr.df) <- c("global", "dia", "fev", "age", "on.breast")
lr.df[1:10, ]
```

We could work with our data as it is but if we wanted to calculate odds-ratios and confidence intervals we would calculate with their reciprocals (i.e. odds-ratios for non-exposure rather than for exposure). This is because of the way the data has been coded (1=yes, 2=no). In order to calculate meaningful odds-ratios the exposure variables should also be coded 0=no, 1=yes. The actual codes used are not important as long as the value used for ‘yes’ is one greater than the value used for ‘no’.

Exercise 9 : Managing and analysing survey data

We need to recode the **global**, **dia**, **fev**, and **on.breast** variables before proceeding:

```
lr.df$global <- 2 - lr.df$global
lr.df$dia <- 2 - lr.df$dia
lr.df$fev <- 2 - lr.df$fev
lr.df$on.breast <- 2 - lr.df$on.breast
lr.df[1:20, ]
```

We can now use the generalised linear model **glm()** function to specify the logistic regression model:

```
lr.svy <- glm(formula = global ~ dia + fev + age + on.breast,
              family = binomial(logit), data = lr.df)
summary(lr.svy)
```

We will use backwards elimination to remove non-significant variables from the model. Diarrhoea (**dia**) is the least significant variable in the model so we will remove this variable from the model. Storing the output of the **glm()** function is useful as it allows us to use the **update()** function to add, remove, or modify variables without having to describe the model in full:

```
lr.svy <- update(lr.svy, . ~ . - dia)
summary(lr.svy)
```

Breast feeding (**on.breast**) is now the least significant variable in the model so we will remove this variable from the model:

```
lr.svy <- update(lr.svy, . ~ . - on.breast)
summary(lr.svy)
```

There are now no non-significant variables in the model. We can use the **lreg.or()** function that we created earlier to calculate and display odds ratios and their confidence intervals:

```
lreg.or(lr.svy)
```

Fever (**fev**) is positively associated with undernutrition (**global**). Increasing age (**age**) is negatively associated with undernutrition (**global**).

As an exercise you might like to perform a similar analysis of the mid-upper arm circumference data.

Exercise 9 : Summary

R provides a full set of data management functions that, amongst other things, allow you to:

Merge files side by side using a common identifying variable.

Create a new variable in a data.frame.

Group data.

Specify value labels for each value of a variable.

Change the names of variables in data.frames.

Add variables to data.frames.

Save data to disk files.

Together with ***R***'s extensive statistical and graphical functions and the ease with which ***R*** can be extended with user-defined functions this makes ***R*** a good candidate for being your principal data management and analysis tool for use with epidemiological data.

Exercise 10 : Computer intensive methods

Estimation involves the calculation of a measure with some sense of precision based upon sampling variation. Only a few estimators (e.g. the sample mean from a normal population) have exact formulae that may be used to estimate sampling variation. Typically, estimates of variability are based upon approximations informed by expected or postulated properties of the sampled population. The development of variance formulae for some measures may require in-depth statistical and mathematical knowledge or may even be impossible to derive.

Bootstrap methods are computer-intensive methods that can provide estimates and measures of precision (e.g. confidence intervals) without resort to theoretical models, higher mathematics, or assumptions about the sampled population. They rely on repeated sampling (sometimes called *resampling*) of the observed data.

As a simple example of how such methods work, we will start by using bootstrap methods to estimate the mean from a normal population. We will work with a very simple dataset which we will enter directly:

```
x <- c(7.3, 10.4, 14.0, 12.2, 8.4)
```

We can summarise this data quite easily:

```
mean(x)
```

The **`sample()`** function can be used to select a bootstrap *replicate*:

```
sample(x, length(x), replace = TRUE)
```

Enter this command several times to see some more bootstrap replicates. The **`length()`** parameter is not required for taking bootstrap replicates and can be omitted.

It is possible to apply a summary measure to a replicate:

```
mean(sample(x, replace = TRUE))
```

Enter this command several times. A bootstrap estimate of the mean of **`x`** can be made by repeating this process many times and taking the mean of the means for each replicate.

One way of doing this is to create a matrix where each column contains a bootstrap replicate and then use the **`apply()`** and **`mean()`** functions to get at the estimate.

First create the matrix of replicates. Here we take ten replicates:

```
x1 <- matrix(sample(x, length(x) * 10, replace = TRUE),
             length(x), 10)
x1
```

Then calculate and stores the means of each replicate. We can do this using the **`apply()`** function to apply the **`mean()`** function to the columns of matrix **`x1`**:

```
x2 <- apply(x1, 2, mean)
x2
```

The bootstrap estimate of the mean is:

```
mean(x2)
```

Exercise 10 : Computer intensive methods

The bootstrap estimate will probably differ somewhat from mean of **x**:

```
mean(x)
```

The situation is improved by increasing the number of replicates. Here we take 5000 replicates:

```
x1 <- matrix(sample(x, length(x) * 5000, replace = TRUE),
             length(x), 5000)
x2 <- apply(x1, 2, mean)
mean(x2)
```

This is a pretty useless example as estimating the mean / standard deviation, or standard error of the mean of a sample from a normal population can be done using standard formulae.

The utility of bootstrap methods is that they can be applied to summary measures that are not as well understood as the arithmetic mean. The bootstrap method also has the advantage of retaining simplicity even with complicated measures.

To illustrate this, we will work through an example of using the bootstrap to estimate the harmonic mean. Again, we will work with a simple dataset which we will enter directly:

```
d <- c(43.64, 50.67, 33.56, 27.75, 43.35, 29.56, 38.83, 35.95, 20.01)
```

The data represents distance (in kilometres) from a point source of environmental pollution for nine female patients with oral / pharyngeal cancer.

This data is presented in Selvin (1998) and the following exercise is based upon Selvin's example of using bootstrap methods with this data.

The harmonic mean is considered to be a sensitive measure of spatial clustering. The first step is to construct a function to calculate the harmonic mean:

```
h.mean <- function(x) {length(x) / sum(1 / x)}
```

Calling this function with the sample data:

```
h.mean(d)
```

Should return an estimated harmonic mean distance of 33.47 kilometres. This is simple. The problem is that calculating the variance of this estimate is complicated using standard methods. This problem is relatively simple to solve using bootstrap methods:

```
replicates <- 1000
n <- length(d)
x1 <- matrix(sample(d, n * replicates, replace = TRUE),
             n, replicates)
x2 <- apply(x1, 2, h.mean)
mean(x2)
```

Confidence intervals can be extracted from **x2** using the **quantile()** function:

```
quantile(x2, c(0.025, 0.975))
```

Exercise 10 : Computer intensive methods

As a final example we will use bootstrap methods to calculate estimates of an odds-ratio from a two-by-two table. We will work with the **salex** dataset which we used in exercises 2 and 3:

```
salex <- read.table("salex.dat", header = TRUE, na.strings = "9")
table(salex$EGGS, salex$ILL)
```

We should set up our estimator function to calculate an odds-ratio from a two-by-two table:

```
or <- function(x) {(x[1, 1] / x[1, 2]) / (x[2, 1] / x[2, 2])}
```

We should test this:

```
or(table(salex$EGGS, salex$ILL))
```

The problem is to take a bootstrap replicate from two vectors in a data.frame. This can be achieved by using **sample()** to create a vector of row indices and then use this sample of indices to select pairs of replicates from the data.frame:

```
boot <- NULL
for(i in 1:500)
{
  sampled.cases <- sample(1:nrow(salex), replace = TRUE)
  x <- salex[sampled.cases, 'EGGS']
  y <- salex[sampled.cases, 'ILL']
  boot[i] <- or(table(x, y))
}
median(boot[boot != Inf])
quantile(boot[boot != Inf], c(0.025, 0.975))
```

The vector **boot** now contains the odds ratio calculated from 500 replicates. Estimates of the odds-ratio and its confidence interval may be obtained using the **median()** and **quantile()** functions

```
median(boot[boot != Inf])
quantile(boot[boot != Inf], c(0.025, 0.975))
```

Note that we select only those values of **boot** that are not (**!=**) infinite (**Inf**). Infinite values are due to division by zero when calculating the odds ratio for some replicates.

Exercise 10 : Summary

Bootstrap methods provide an alternative to classical statistical techniques for estimation. They have the advantage of being simple to implement and they remain simple even with complex estimators.

R provides functions that allow you to implement computer intensive methods such as the bootstrap.

Add-in packages for implementing computer intensive methods in ***R*** are available (**`boot`** and **`bootstrap`**) . These packages make it easy to use these techniques with complex data and complicated estimators.

What now?

Now that you have had a taste of using ***R*** you will be able to decide whether it is meets your requirements for a data-analysis system. The file **`R-intro.pdf`** which is installed with the ***R*** system contains the document 'An Introduction to ***R***'. This documents provides a solid introduction to ***R***. The file **`refman.pdf`** which is also installed with the ***R*** system contains the document 'The ***R*** reference index'. This documents provides a complete function-by-function reference to the ***R*** base system and several standard function libraries (packages). Other documents are available from the ***R*** Website:

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

R and ***S*** (the basis of the commercial ***S-Plus*** system) are very similar to each other. Any books dealing with ***S*** or ***S-Plus*** will prove useful in learning to use ***R***. Some useful titles are:

Introductory texts

Krause A., Olson M., *'The Basics of S and S-Plus (Second Edition)'*, Springer, New York, 2000

Spector, P., *'An Introduction to S and S-Plus'*, Duxbury, Belmont, 1994

Maindonald, J., *'Data Analysis and Graphics Using **R** - An introduction'*, Australian National University, 2000 (available over the Internet - links on the ***R*** website)

Using *R* and *S* for statistics

Everitt, B.S. *'A Handbook of Statistical Analysis Using S-Plus'*, Chapman & Hall, London 1994

Chambers, J.M., Hastie, T.J., *'Statistical Models in S'*, Wadsworth, Pacific Grove, 1992

Selvin, S, *'Modern Applied Biostatistical Methods Using S-Plus'*, Oxford University Press, New York, 1998

Venables, W.N., Ripley, B.D., *'Modern Applied Statistics with S-Plus (Third Edition)'*, Springer, New York, 1999

Writing *R* functions and packages

Venables, W.N., Ripley, B.D., *'S Programming'*, Springer, New York, 2000

Chambers, J.M., *'Programming with Data - A Guide to the S Language'*, Springer, New York, 1998

Further notes on using ***R*** to work with epidemiological data (focusing on vital / routine statistics and surveillance data) are currently being developed and are available from:

`http://www.medepi.org/epitools/`

The Internet mailing list provides an excellent level of support for all levels of users. You can subscribe to the Internet mailing lists from the ***R*** website.