

Nordic School of Public Health & University of Tartu  
Computer Software in Epidemiology / Statistical Practice in Epidemiology

# Open Source Solutions - ***R***

Mark Myatt, April 2005

## 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, stratified analysis, 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 (i.e. Mantel-Haenszel) analysis as well as analysing data arising from matched case-control studies.

**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, charts with error bars, and simple mesh-maps.

**Exercise 9** : In this exercise we explore some more data management and analysis functions in the context of managing and analysis survey data.

**Exercise 10** : In this exercise we explore ways of implementing computer-intensive methods, such as the *bootstrap* and *computer based simulation*, 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, dialog 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 32-bit 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. Details of the general public license are available from:

```
http://www.gnu.org/copyleft/gpl.html
```

This document is released under the related free documentation license (FDL):

```
http://www.gnu.org/copyleft/fdl.html
```

**R** is available for download 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 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-Plus (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 or terminal scroll bars.

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 in this manual 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 working with files in different formats including (through the use of extension packages) ODBC (open database connectivity) and SQL data sources, EpiInfo, EpiData, Minitab, SPSS, SAS, S-Plus, and Stata format files.

## Introducing *R* : Retrieving data

All of the exercises in this manual assume that the necessary data files are located in the current working directory.

A command such as:

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

Retrieves the data stored in the file named **fem.dat** which is stored in the current working directory.

To retrieve data that is stored in files outside of the current working directory you need to specify the full path to the file. For example:

```
read.table("~/rex/fem.dat", header = TRUE)
```

Will retrieve the data stored in the file named **fem.dat** stored in the **rex** directory under the users *home* directory on UNIX systems.

*R* follows many UNIX operating and naming conventions including the use of the backslash (\) character to specify special characters in strings (e.g. using "\n" to specify a new line in printed output). Windows uses the backslash (\) character to separate directory and file names in paths. This means that Windows users need to *escape* any backslashes in file paths using an additional backslash character. For example:

```
read.table("c:\\rex\\fem.dat", header = TRUE)
```

Will retrieve the data that is stored in the file named **fem.dat** which is stored in the **rex** directory off the *root* directory of the **C:** drive.

The Windows version of *R* allows you to specify UNIX-style path names (i.e. using the forward slash (/) character as a separator in file paths). For example:

```
read.table("c:/rex/fem.dat", header = TRUE)
```

Path names may include shortcut characters such as:

- .        The current working directory
- ..      Up one level in the directory tree from the current working directory
- ~       The user's *home* directory (on UNIX-based systems)

*R* also allows you to retrieve files from any location that may be represented by a standard *uniform resource locator* (URL) string. For example:

```
read.table("file://~/rex/fem.dat", header = TRUE)
```

Will retrieve the data stored in the file named **fem.dat** stored in the **rex** directory under the users *home* directory on UNIX-based systems.

All of the data files used in this manual are stored in the **/datasets** directory on Brixton Health's website. This means, for example, that you can use the command:

```
read.table("http://www.brixtonhealth.com/datasets/fem.dat",  
          header = TRUE)
```

To retrieve the data that is stored in the file named **fem.dat** which is stored in the **/datasets** directory of Brixton Health's website.

## Exercise 1 : Getting acquainted with *R*

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:

<b>ID</b>	Patient ID
<b>AGE</b>	Age in years
<b>IQ</b>	IQ score
<b>ANXIETY</b>	Anxiety (1=none, 2=mild, 3=moderate, 4=severe)
<b>DEPRESS</b>	Depression (1=none, 2=mild, 3=moderate or severe)
<b>SLEEP</b>	Sleeping normally (1=yes, 2=no)
<b>SEX</b>	Lost interest in sex (1=yes, 2=no)
<b>LIFE</b>	Considered suicide (1=yes, 2=no)
<b>WEIGHT</b>	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*

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)
```

## Exercise 1 : Getting acquainted with *R*

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*

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 loss of interest in sex and considering suicide 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*

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*

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 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)
```

The `sapply()` function is part of a group of functions that apply a specified function to data objects:

Function(s)	Applies a function to ...
<code>apply()</code>	rows and columns of matrices, arrays, and tables
<code>lapply()</code> , <code>sapply()</code> , <code>mapply()</code>	components of lists and data.frames
<code>tapply()</code>	subsets of data

Related functions are `aggregate()` which compute summary statistics for subsets of data, `by()` which applies a function to a data.frame split by factors, and `sweep()` which applies a function to an array.

## Exercise 1 : Getting acquainted with *R*

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 *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 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 (GUI) or enter **n** when prompted to save the workspace image (terminal).

## 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 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:

<b>ILL</b>	Ill or not-ill
<b>HAM</b>	Baked ham
<b>BEEF</b>	Roast beef
<b>EGGS</b>	Eggs
<b>MUSHROOM</b>	Mushroom flan
<b>PEPPER</b>	Pepper flan
<b>PORKPIE</b>	Pork pie
<b>PASTA</b>	Pasta salad
<b>RICE</b>	Rice salad
<b>LETTUCE</b>	Lettuce
<b>TOMATO</b>	Tomato salad
<b>COLESLAW</b>	Coleslaw
<b>CRISPS</b>	Crisps
<b>PEACHCAKE</b>	Peach cake
<b>CHOCOLATE</b>	Chocolate cake
<b>FRUIT</b>	Tropical fruit salad
<b>TRIFLE</b>	Trifle
<b>ALMONDS</b>	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 can manipulate the output directly but it is easier if we 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 variables. We can extract data using the values of the row and column variables by enclosing the index values in double quotes ("). For example:

```
tab["1", "1"]
```

The two methods of extracting data may be used together. For example:

```
tab[1, "1"]
```

We can calculate a risk ratio using the extracted data:

```
(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 start an internal script 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 ratios 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 (GUI) or enter **n** when prompted to save the workspace image (terminal).

## 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 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 :** The **attach()** function works with a copy of the data.frame rather than the original data.frame. 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 can 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.) those created by the `glm()` function. The `coef()` function provides a standard way of extracting this data from all classes 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, ]	-1.970967	0.6145691	-3.207071	0.0013409398
[2, ]	2.639115	0.7333899	3.598515	0.0003200388
[3, ]	1.664581	0.8375970	1.987330	0.0468858898
[4, ]	3.195594	1.1516159	2.774879	0.0055222320

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(or, lci, uci)
lreg.or
```



## Exercise 3 : Logistic regression

We seldom need to report estimates and confidence intervals to eight decimal places. We can use the `round()` function to remove the excess digits:

```
round(lreg.or, digits = 4)
```

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, digits = 4)
{
  lreg.coeffs <- coef(summary(model))
  OR <- exp(lreg.coeffs[,1])
  LCI <- exp(lreg.coeffs[,1] - 1.96 * lreg.coeffs[,2])
  UCI <- exp(lreg.coeffs[,1] + 1.96 * lreg.coeffs[,2])
  lreg.or <- round(cbind(OR, LCI, UCI), digits = digits)
  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:

	OR	LCI	UCI
(Intercept)	0.1393	0.0418	0.4647
EGGS	14.0008	3.3257	58.9423
PASTA	5.2835	1.0232	27.2832
LETTUCE	24.4247	2.5560	233.4018

The `digits` parameter of the `lreg.or()` function, which has `digits = 4` as its default value, allows us to specify the precision with which the estimates and their confidence intervals are reported:

```
lreg.or(salex.lreg, digits = 2)
lreg.or(salex.lreg, digits = 8)
```

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. For example, four potential confounders, each with two levels would produce sixteen tables. In such situations, logistic regression might be a better approach. In order to illustrate Mantel-Haenszel techniques in **R** we will work with a 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:

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

Data is available for all forty-five guests at the luncheon. All of the variables are coded 1=yes, 2=no. Retrieve 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 odds ratios for each stratum. We can define a simple 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 can use **apply()** to apply the **or()** function to the two-by-two table in each stratum:

```
tabC <- table(CAESAR, ILL, TOMATO)
apply(tabC, 3, or)
tabT <- table(TOMATO, ILL, CAESAR)
apply(tabT, 3, or)
```

The '3' instructs **apply()** to apply the **or()** function to the third dimension of the table objects (i.e. levels of the potential confounder in **tabC** and **tabT**).

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 odds ratios for each stratum as we did above.

There appears to be an interaction between **CAESAR**, **WINE**, and **ILL**:

```
tabW <- table(CAESAR, ILL, WINE)
apply(tabW, 3, or)
```

*Woolf's test for interaction* (also known as *Woolf's test for the homogeneity of odds ratios*) 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 + 0.5
  k <- dim(x)[3]
  or <- apply(x, 3, function(x)
    {(x[1, 1] / x[1, 2]) / (x[2, 1] / x[2, 2])})
  w <- apply(x, 3, function(x) {1 / sum(1 / x)})
  chi.sq <- sum(w * (log(or) - weighted.mean(log(or), w))^2)
  p <- pchisq(chi.sq, df = k - 1, lower.tail = FALSE)
  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 test for interaction between **CAESAR**, **TOMATO**, and **ILL** before accepting the results reported by the `mantelhaen.test()` function:

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

We can repeat this analysis using logistic regression. 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")
```

## Exercise 3 : Matched Data

Matching is another way to control for the effects of potential confounding variables. Matching is usually performed during data-collection as part of the design of a study.

In a matched case-control studies, each case is matched with one or more controls which are chosen to have the same values over a set of potential confounding variables.

In order to illustrate how matched data may be analysed using tabulation and stratification in *R* we will start with the simple case of one-to-one matching (i.e. each case has a single matched control):

```
octe <- read.table("octe.dat", header = TRUE)
octe[1:10, ]
```

This data is from a matched case-control study investigating the association between oral contraceptive use and thromboembolism. The cases are 175 women aged between 15 and 44 years admitted to hospital for thromboembolism and discharged alive. The controls are female patients admitted for conditions believed to be unrelated to oral contraceptive use. Cases and controls were matched on age, ethnic group, marital status, parity, income, place of residence, and date of hospitalisation. The variables in the dataset are:

<b>ID</b>	Identifier for the matched sets of cases and controls
<b>CASE</b>	Case (1) or control (2)
<b>OC</b>	Used oral contraceptives in the previous month (1=yes, 2=no)

The dataset consists of 350 records:

```
nrow(octe)
```

There are 175 matched sets of cases and controls:

```
length(unique(octe$ID))
```

In each matched set of cases and controls there is one case and one control:

```
table(octe$ID, octe$CASE)
```

This data may be analysed using *McNemar's chi-squared test* which use the number of discordant (i.e. relative to exposure) pairs of matched cases and controls.

To find the number of discordant pairs we need to split the dataset into cases and controls:

```
octe.cases <- subset(octe, CASE == 1)
octe.controls <- subset(octe, CASE == 2)
```

Sorting these two datasets (i.e. `octe.cases` and `octe.controls`) by the `ID` variable simplifies the analysis:

```
octe.cases <- octe.cases[order(octe.cases$ID), ]
octe.controls <- octe.controls[order(octe.controls$ID), ]
```

Since the two datasets (i.e. `octe.cases` and `octe.controls`) are now sorted by the `ID` variable we can use the `table()` function to retrieve the number of concordant and discordant pairs and store them in a table object:

```
tab <- table(octe.cases$OC, octe.controls$OC)
tab
```

## Exercise 3 : Matched Data

This table object (i.e. `tab`) can then be passed to the `mcnemar.test()` function:

```
mcnemar.test(tab)
```

The `mcnemar.test()` function does not provide an estimate of the odds ratio. This is the ratio of the discordant pairs:

```
r <- tab[1,2]
s <- tab[2,1]
rdp <- r / s
rdp
```

A confidence interval can also be calculated:

```
ci.p <- binom.test(r, r + s)$conf.int
ci.rdp <- ci.p / (1 - ci.p)
ci.rdp
```

This provides a 95% confidence interval. Other (e.g. 99%) confidence intervals can be produced by specifying appropriate values for the `conf.level` parameter of the `binom.test()` function:

```
ci.p <- binom.test(r, r + s, conf.level = 0.99)$conf.int
ci.rdp <- ci.p / (1 - ci.p)
ci.rdp
```

An alternative way of analysing this data is to use the `mantelhaen.test()` function:

```
mantelhaen.test(table(octe$OC, octe$CASE, octe$ID))
```

The Mantel-Haenszel approach is preferred because it can be used with data from matched case-control studies that match more than one control to each case. Multiple matching is useful when the condition being studied is rare or at the early stages of an outbreak (i.e. when cases are hard to find and controls are easy to find).

We will now work with some data where each case has one or more controls:

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

This data is from a matched case-control study investigating the association between the use of different brands of tampon and toxic shock syndrome. Only a subset of the original dataset is used here. The variables in the dataset are:

<b>ID</b>	Identifier for the matched sets of cases and controls
<b>CASE</b>	Case (1) or control (2)
<b>RBTAMP</b>	Used Rely brand tampons (1=yes, 2=no)

The dataset consists of forty-three (43) records:

```
nrow(tsstamp)
```

There are fourteen (14) matched sets of cases and controls:

```
length(unique(tsstamp$ID))
```

Each matched set of cases and controls consists of one case and one or more controls:

```
table(tsstamp$ID, tsstamp$CASE)
```

### Exercise 3 : Matched Data

The *McNemar's chi-squared test* is not useful for this data as it is limited to the special case of one-to-one matching.

Analysing this data using a simple tabulation such as:

```
fisher.test(table(tsstamp$RBTAMP, tsstamp$CASE))
```

Ignores the matched nature of the data and is, therefore, also not useful for this data.

The matched nature of the data may be accounted by stratifying on the variable that identifies the matched sets of cases and controls (i.e. the **ID** variable) using the **`mantelhaen.test()`** function:

```
mantelhaen.test(table(tsstamp$RBTAMP, tsstamp$CASE, tsstamp$ID))
```

Analysis of several risk factors or adjustment for confounding variables not matched for in the design of a matched case-control study cannot be performed using tabulation-based procedures such as the *McNemar's chi-squared test* and Mantel-Haenszel procedures. In these situations a special form of logistic regression, called *conditional logistic regression*, should be used.

We can now quit **R**:

```
q()
```

For this exercise there is no need to save the workspace image so click the **No** button (GUI) or enter **n** when prompted to save the workspace image (terminal).

## 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, the `mantelhaen.test()` function to perform stratified analyses, and the `mantelhaen.test()` and `mcnemar.test()` functions to analyse data from matched case-control studies.

**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:

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

You can use the **plot()** function to represent this table graphically:

```
plot(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
```

You can also use the **plot()** function to represent this table graphically:

```
plot(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:

PARTNERS	HIV	
	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.

## Exercise 4 : Analysing some data with *R*

We can extract this data from a table object:

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

Another way of creating the **trials** object would be to use the **apply()** function to **sum** the rows of the **tab** object:

```
trials <- apply(tab, 1, sum)
```

Pass this data to the **prop.trend.test()** function:

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

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)
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 (GUI) or enter **n** when prompted to save the workspace image (terminal).

## 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 (**response**) 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))
```

If there is only one survival curve to plot then plotting a `survfit` object will plot the survival curve with 95% confidence limits. You can specify that confidence limits should be plotted when there is more than one survival curve but the results can be disappointing:

```
plot(ca.surv, conf.int = TRUE)
```

Plots can be improved slightly by specifying different line types for each curve:

```
plot(ca.surv, conf.int = TRUE, lty = c(1, 2))
```

Clearer plots can be produced by over-plotting:

```
plot(ca.surv, conf.int = TRUE, col = "white")
par(new = TRUE)
plot(ca.surv, lwd = c(1, 2), xlab = "Months", ylab = "Survival")
par(new = TRUE)
plot(ca.surv, lwd = c(1, 2), lty = c(2, 2), conf.int = TRUE)
legend(125, 1, names(ca.surv$strata), lwd = c(1, 2))
```

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

```
survdifftest(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 (GUI) or enter **n** when prompted to save the workspace image (terminal).

## Exercise 5 : Summary

**R** can be extended by adding additional packages. Some 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 hypothesis testing functions supplied with *R* produce objects of the **htest** class 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)
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 (e.g. 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` class objects:

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

The next line tells *R* that the list object called `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 this 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 (GUI) or enter **n** when prompted to save the workspace image (terminal).

## 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 your 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 **ba.plot()** 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 by 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. Let's 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 (or *method*) 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 (GUI) or enter **n** when prompted to save the workspace image (terminal).

## 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:

<b>Time</b>	Month and year (as text)
<b>Cases</b>	Number of cases of malaria reported
<b>Rain</b>	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 peaks 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 (you may need to resize the plot to display the x-axis labels correctly) 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))
```

## Exercise 8 : More graphical functions

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 since 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 still 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:

<b>day</b>	The day after start of diet that measurements were taken
<b>oedema</b>	Type of undernutrition coded as:  1 = Oedematous 2 = Marasmic
<b>diet</b>	The trial diets coded as:  LP = Low protein HP = High protein
<b>wt</b>	Mean weight change (weight velocity) in g / kg / day since the previous measurement
<b>sd</b>	Standard deviation of weight change in g / kg / day
<b>n</b>	Number of subjects at each observation

Examine the dataset:

```
diets
```

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

```
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
<b>x</b>	Data to plot	None
<b>y</b>	Data to plot	None
<b>y.lci</b>	Data to plot	None
<b>y.uci</b>	Data to plot	None
<b>ylim</b>	Limits for $y$ axis	<code>c(min(y.lci), max(y.uci))</code>
<b>xlab</b>	Label for $x$ axis	<code>deparse(substitute(x))</code>
<b>ylab</b>	Label for $y$ axis	<code>deparse(substitute(y))</code>
<b>main</b>	Chart title	<code>paste(ylab, "by", xlab)</code>
<b>type</b>	Type of plot	"l"
<b>lty</b>	Line type	1
<b>axes</b>	Draw $x$ and $y$ axes	TRUE
<b>pch</b>	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, chart titles, chart limits &c.

We can override these default values if we need 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 of data.

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

## Exercise 8 : More graphical functions

The fact that **R** provides flexible graphical functions means that, with little extra work, you can use these functions to present your data in appropriate and interesting ways rather than having to rely on a limited set of basic chart types.

In this exercise we will use the **plot()** function to produce a simple *mesh-map*.

The file **coverage.dat** contains data from a coverage survey for a therapeutic feeding program (TFP) in central Malawi undertaken in March 2003. Data were collected using the *centric systematic area sampling* method to define sampling locations: A number of communities located closest to the centres of thirty 10 x 10 kilometre grid squares were sampled using active (investigative) case-finding.

The columns in this dataset are as follows:

<b>x</b>	x position of grid square
<b>y</b>	y position of grid square
<b>cases</b>	Number of cases found in sampled communities
<b>in.program</b>	Number of cases (from above) enrolled in the TFP

Retrieve the dataset:

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

Examine the dataset:

```
cs
```

We should calculate to observed coverage for each grid:

```
cs <- transform(cs, cover = in.program / cases)
cs <- transform(cs, cover.pch = ifelse(cover == 0, 0, 15))
```

Note that some grid squares have zero coverage. It might be useful to use specific plotting characters (e.g. open and filled squares) to indicate zero and non-zero coverage. We can use the **ifelse()** function to do this:

```
cs <- transform(cs, cover.pch = ifelse(cover == 0, 0, 15))
```

A quick way of seeing the code associated with each plotting symbol is:

```
plot(0:25, pch = 0:25, cex = 2)
```

The size of the plotting symbol may be used to indicate the level of coverage in each quadrat but we must ensure that the symbol used for zero-coverage is not invisibly small:

```
cs <- transform(cs, cover.cex = ifelse(cover == 0, 1, 10 * cover))
```

We can now plot the data:

```
par(pty="s")
plot(cs$x, cs$y, cex = cs$cover.cex, pch = cs$cover.pch)
```

There are some problems with this plot:

The axes and labels distract from the data.

The distance between grid-square centres is wider in the x than in the y direction.

The colour (black) of the plotting symbols is too strong.

## Exercise 8 : More graphical functions

All of these problems can be fixed by specifying values for **plot()** function parameters:

```
plot(cs$x, cs$y, cex = cs$cover.cex, pch = cs$cover.pch,  
      xlab = "", ylab = "", axes = FALSE, xlim = c(0,10),  
      ylim = c(0,10), col = gray(0.5))
```

An alternative way of plotting this data is to use shades of grey (rather than the size of the plotting symbol) to represent the level of coverage in each grid-square:

```
plot(cs$x, cs$y, cex = 10, xlab = "", ylab = "", axes = FALSE,  
      pch = 15, xlim = c(0, 10), ylim = c(0, 10),  
      col = gray(1 - cs$cover))
```

Additional data can be overlaid onto the map by specifying **par(new = TRUE)** before each subsequent plotting command.

In this context it is useful to show the approximate location of therapeutic feeding centres:

```
plot(cs$x, cs$y, cex = cs$cover.cex, pch = cs$cover.pch,  
      xlab = "", ylab = "", axes = FALSE, xlim = c(0,10),  
      ylim = c(0,10), col = gray(0.5))  
  
par(new = TRUE)  
  
plot(c(2.5, 4.5, 5.5), c(6.5, 8.25, 5), xlab = "", ylab = "",  
      axes = FALSE, xlim = c(0, 10), ylim = c(0, 10),  
      pch = 19, cex = 2)
```

The amount of detail that can be easily added to the map in this way is limited. You might want to export the map using the **dev2bitmap()** function and edit the map in a graphics (image editing) package such as The GIMP or Adobe Photoshop.

## 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 then you can use the `par()` function with the `no.readonly` parameter set to `TRUE`:

```
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 or between 65 and 110 cm in height. The columns in the dataset are:

<b>age</b>	Age of child in months
<b>sex</b>	Sex of child (1 = male, 2 = female)
<b>ht</b>	Height of child (cm)
<b>wt</b>	Weight of child (kg)
<b>muac</b>	Mid-upper arm circumference (cm)
<b>oedema</b>	Bilateral pitting oedema
<b>dia</b>	Diarrhoea in previous 14 days (mother's recall)
<b>fev</b>	Fever in previous 14 days (mother's recall)
<b>bf</b>	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 given 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*:

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

Where:

<i>z</i>	=	<i>z-score</i>
<i>wt</i>	=	weight of child
<i>mean</i>	=	mean weight of child for same height and sex in the <i>reference population</i>
<i>s</i>	=	standard deviation from the <i>reference population</i>

A child is considered to be *moderately undernourished* if they have a weight-for-height *z-score* below -2.00. A child is considered to be *severely undernourished* if they have a weight-for-height *z-score* below -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, ]
```

## Exercise 9 : Managing and analysing survey data

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 cut-points < 11.0 cm (severe) and < 12.5 cm (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"]
```

## Exercise 9 : Managing and analysing survey data

We should specify values labels for the other factors:

```
svy$soedema <- as.factor(svy$soedema)
levels(svy$soedema) <- c("Yes", "No")
svy$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)
```

## Exercise 9 : Managing and analysing survey data

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 *median* 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:

```
median(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)  
median(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 <- 5000  
n <- length(d)  
x1 <- matrix(sample(d, n * replicates, replace = TRUE),  
              n, replicates)  
x2 <- apply(x1, 2, h.mean)  
median(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 of the bootstrap method we will use the method to obtain an estimate 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 replicates from the data.frame:

```
boot <- NULL
for(i in 1:500)
{
  sampled.rows <- sample(1:nrow(salex), replace = TRUE)
  x <- salex[sampled.rows, 'EGGS']
  y <- salex[sampled.rows, 'ILL']
  boot[i] <- or(table(x, y))
}
```

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.

Another way around this problem is to use an *adjusted odds ratio* calculated by adding 0.5 to each cell of the two-by-two table:

```
boot <- NULL
for(i in 1:500)
{
  sampled.rows <- sample(1:nrow(salex), replace = TRUE)
  x <- salex[sampled.rows, 'EGGS']
  y <- salex[sampled.rows, 'ILL']
  boot[i] <- or(table(x, y) + 0.5)
}
median(boot)
quantile(boot, c(0.025, 0.975))
```

This procedure is preferred when working with sparse tables.

## Exercise 10 : Computer intensive methods

One problem with the approach used above is that the ratio of cases and controls selected in each sample may differ from the ratio of cases and controls in the original study. This can be fixed by splitting the **salex** data.frame into two separate data.frames (i.e. one for cases and one for controls):

```
salex.cases <- subset(salex, ILL == 1)
salex.controls <- subset(salex, ILL == 2)
```

And sampling cases and controls separately:

```
boot <- NULL
for(i in 1:500)
{
  sampled.rows <- sample(1:nrow(salex.cases), replace = TRUE)
  x.cases <- salex.cases[sampled.rows, 'EGGS']
  y.cases <- salex.cases[sampled.rows, 'ILL']
  sampled.rows <- sample(1:nrow(salex.controls), replace = TRUE)
  x.controls <- salex.controls[sampled.rows, 'EGGS']
  y.controls <- salex.controls[sampled.rows, 'ILL']
  x <- c(x.cases, x.controls)
  y <- c(y.cases, y.controls)
  boot[i] <- or(table(x, y) + 0.5)
}
median(boot)
quantile(boot, c(0.025, 0.975))
```

This approach may be preferred since it retains the ratio of cases to control that were used in the original study.

## Exercise 10 : Computer intensive methods

Computer-intensive methods also offer a general approach to statistical hypothesis testing. To illustrate this we will use *computer based simulation* to investigate spatial clustering around a point.

Before continuing, we will retrieve a dataset:

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

The file **waste.dat** contains the location of twenty-three recent cases of childhood cancer in 5 by 5 km square surrounding an industrial waste disposal site. The columns in the dataset are:

<b>x</b>	The x location of cases
<b>y</b>	The y location of cases

The **x** and **y** variables have been transformed to lie between 0 and 1 with the industrial waste disposal site centrally located (i.e. at **x** = 0.5, **y** = 0.5).

Plot the data and the location of the industrial waste disposal site on the same chart:

```
plot(waste$x, waste$y, xlim = c(0, 1), ylim = c(0, 1))  
points(0.5, 0.5, pch = 3)
```

We can calculate the distance of each point from the industrial waste disposal site using Pythagoras' Theorem:

```
distance.obs <- sqrt((waste$x - 0.5)^2 + (waste$y - 0.5)^2)
```

The observed mean distance of each case from the industrial waste disposal site is:

```
mean(distance.obs)
```

To test whether this distance is unlikely to have arisen by chance (i.e. evidence of spatial clustering) we need to simulate the distribution of distances when no spatial pattern exists:

```
replicates <- 10000  
x.sim <- matrix(runif(replicates * 23), 23, replicates)  
y.sim <- matrix(runif(replicates * 23), 23, replicates)  
distance.run <- sqrt((x.sim - 0.5)^2 + (y.sim - 0.5)^2)  
distance.sim <- apply(distance.run, 2, mean)  
hist(distance.sim, breaks = 20)
```

The probability (i.e. the *p-value*) of observing a mean distance smaller than the *observed mean distance* under the null hypothesis can be estimated as the number of estimates of the mean distance under the null hypothesis falling below the observed mean divided by the total number of estimates of the mean distance under the null hypothesis:

```
sum(ifelse(distance.sim < mean(distance.obs), 1, 0)) / replicates
```

You might like to repeat this exercise using the harmonic mean and median distance.

## Exercise 10 : Computer intensive methods

We can check if this method is capable of detecting a simple cluster using simulated data:

```
x <- rnorm(23, mean = 0.5, sd = 0.2)
y <- rnorm(23, mean = 0.5, sd = 0.2)
plot(x, y, xlim = c(0, 1), ylim = c(0, 1))
points(0.5, 0.5, pch = 3)
```

We need to recalculate the distance of each simulated case from the industrial waste disposal site:

```
distance.obs <- sqrt((x - 0.5)^2 + (y - 0.5)^2)
```

The observed mean distance of each case from the industrial waste disposal site is:

```
mean(distance.obs)
```

We can use the the simulated null hypothesis data to test for spatial clustering:

```
sum(ifelse(distance.sim < mean(distance.obs), 1, 0)) / replicates
```

We should also check if the procedure can detect a *plume* of cases, such as might be created by a prevailing wind at a waste incineration site, in a similar way:

```
x <- rnorm(23, 0.25, 0.1) + 0.5
y <- rnorm(23, 0.25, 0.1) + 0.5
plot(x, y, xlim = c(0, 1), ylim = c(0, 1))
points(0.5, 0.5, pch = 3)
distance.obs <- sqrt((x - 0.5)^2 + (y - 0.5)^2)
mean(distance.obs)
sum(ifelse(distance.sim < mean(distance.obs), 1, 0)) / replicates
```

The method is not capable of detecting plumes. You might like to adapt the simulation code presented here to provide a method capable of detecting plumes of cases.



## Exercise 10 : Computer intensive methods

In the previous example we simulated the expected distribution of data under the *null hypothesis*. Computer based simulations are not limited to simulating data. They can also be used to simulate processes.

In this example we will simulate the behaviour of the *lot quality assurance sampling* (LQAS) survey method when sampling constraints lead to a loss of sampling independence. In this example the sampling process is simulated and applied to real-world data.

LQAS is a small-sample classification technique that is widely used in manufacturing industry to judge the quality of a batch of manufactured items. In this context, LQAS is used to identify batches that are likely to contain an unacceptably large number of defective items. In the public health context, LQAS may be used to identify communities with unacceptably low levels of service (e.g. vaccine) coverage or worrying levels of disease prevalence.

The LQAS method produces data that is easy to analyse. Data analysis is performed as data is collected and consists solely of counting the number of *defects* (e.g. children with a specific disease) in the sample and checking whether a predetermined threshold value has been exceeded. This combination of data collection and data analysis is called a *sampling plan*.

LQAS sampling plans are developed by specifying:

**A TRIAGE SYSTEM** : A classification system that defines *high*, *moderate*, and *low* categories of the prevalence of the phenomenon of interest.

**ACCEPTABLE PROBABILITIES OF ERROR** : There are two probabilities of error. These are termed *provider probability of error* (PPE) and *consumer probability of error* (CPE):

**PROVIDER PROBABILITY OF ERROR (PPE)** : The risk that the survey will indicate that prevalence is *high* when it is, in fact, *low*. PPE is analogous to *type I ( $\alpha$ ) error* in statistical hypothesis testing.

**CONSUMER PROBABILITY OF ERROR (CPE)** : The risk that the survey will indicate that prevalence is *low* when it is, in fact, *high*. CPE is analogous to *type II ( $\beta$ ) error* in statistical hypothesis testing.

Once the upper and lower levels of the triage system and acceptable levels of error have been decided, a set of probability tables are constructed that are used to select a maximum sample size ( $n$ ) and the number of defects or cases ( $d$ ) that are allowed in the sample of  $n$  subjects before deciding that a population is a high prevalence population. The combination of maximum sample size ( $n$ ) and number of defects ( $d$ ) form the *stopping rules* of the sampling plan. Sampling stops when either the maximum sample size ( $n$ ) is met or the allowable number of defects ( $d$ ) is exceeded:

If  $d$  is exceeded then the population is classified as high prevalence.

If  $n$  is met without  $d$  being exceeded then the population is classified as low prevalence.

The values of  $n$  and  $d$  used in a sampling plan depend upon the threshold values used in the triage system and the acceptable levels of error.

The values of  $n$  and  $d$  used in a sampling plan are calculated using binomial probabilities. For example, the probabilities of finding 14 or fewer cases ( $d = 14$ ) in a sample of 50 individuals ( $n = 50$ ) from populations with prevalences of either 20% or 40% are:

`pbinom(q = 14, size = 50, prob = 0.2)`

`pbinom(q = 14, size = 50, prob = 0.4)`

The sampling plan with  $n = 50$  and  $d = 14$  is, therefore, a reasonable candidate for a sampling plan intended to distinguish between populations with prevalences of less than or equal to 20% and populations with prevalences greater than or equal to 40%.

## Exercise 10 : Computer intensive methods

There is no middle ground with LQAS sampling plans. Population are always classified as either *high* or *low* prevalence. Populations with prevalences between the upper and lower standards of the triage system are classified as high or low prevalence populations. The probability of a moderate prevalence population being classified as high or low prevalence is proportional to the proximity of the prevalence in that population to the triage standards. Moderate prevalence populations close to the upper standard will tend to be classified as high prevalence populations. Moderate prevalence populations close to the lower standard will tend to be classified as low prevalence populations. This behaviour is summarised by the operating characteristic (OC) curve for the sampling plan. For example:

```
plot(seq(0, 0.6, 0.01),
     pbinom(14, 50, seq(0, 0.6, 0.01), lower.tail = FALSE),
     main = "OC Curve for LQAS sampling plan n = 50, d = 14",
     xlab = "Proportion diseased",
     ylab = "Probability of classification as high prevalence",
     type = "l", lty = 2)
```

The data we will use for the simulation is stored in forty-eight separate files. These files contain the returns from whole community screens for active trachoma (TF/TI) in children undertaken as part of trachoma control activities in five African countries. Each file has the file suffix **.sim**. The name of the file reflects the country in which the data were collected. The data files are:

File name	Files	Origin
-----	-----	-----
<b>egyptXX.sim</b>	10	Egypt
<b>gambiaXX.sim</b>	10	Gambia
<b>ghanaXX.sim</b>	3	Ghana
<b>tanzaniaXX.sim</b>	14	Tanzania
<b>togoXX.sim</b>	11	Togo
-----	-----	-----

All of these data files have the same structure. The variables in the data files are:

<b>hh</b>	Household identifier
<b>sex</b>	Sex of child (1=male, 2=female)
<b>age</b>	Age of child in years
<b>tfti</b>	Child has active (TF/TI) trachoma (0=no, 1=yes)

Each row in these files represents a single child. For example:

```
x <- read.table("gambia09.sim", header = TRUE)
x[1:10, ]
```

Any rapid survey method that is appropriate for general use in developing countries is restricted to sampling *households* rather than *individuals*. Sampling households in order to sample individuals violates a principal requirement for a sample to be representative of the population from which it is drawn (i.e. that individuals are selected *independently* of each other). This lack of statistical independence amongst sampled individuals may invalidate standard approaches to selecting sampling plans leading to increased probabilities of error. This is likely to be a particular problem if cases tend to be clustered within households. Trachoma is an infectious disease that confers no lasting immunity in the host. Cases are, therefore, very likely to be clustered within households. One solution to this problem would be to sample (i.e. at random) a single child from each of the sampled households. This is not appropriate for active trachoma as the examination procedure often causes distress to younger children. This may influence survey staff to select older children, who tend to have a lower risk of infection, for examination. Sampling is, therefore, constrained to sampling all children in selected households.

## Exercise 10 : Computer intensive methods

The purpose of the simulations presented here is to determine whether the LQAS method is robust to:

1. The loss of sampling independence introduced by sampling households at random and examining all children in selected households for active trachoma.

And:

2. The slight increase in the maximum sample size ( $n$ ) introduced by examining all children in selected households for active trachoma.

Each row in the datasets we will be using represents an individual child. Since we will simulate sampling households rather than individual children we need to be able convert the datasets from one row per child to one row per household. We will write a function to do this.

Create a new function called `ind2hh()`:

```
ind2hh <- function() {}
```

This creates an empty function called `ind2hh()`. Use the `fix()` function to edit the `ind2hh()` function:

```
fix(ind2hh)
```

Edit the function to read:

```
function(data)
{
  n.kids <- n.cases <- NULL
  id <- unique(data$hh)
  for(household in id)
  {
    temp <- subset(data, data$hh == household)
    n.kids <- c(n.kids, nrow(temp))
    n.cases <- c(n.cases, sum(temp$tfti))
  }
  result <- as.data.frame(cbind(id, n.kids, n.cases))
  return(result)
}
```

Once you have created the `ind2hh()` function you should test it for correct operation. We will create a simple test data.frame (`test.df`) for this purpose:

```
test.df <- as.data.frame(cbind(c(1, 1, 2, 2, 2), c(1, 1, 1, 0, 0)))
names(test.df) <- c("hh", "tfti")
test.df
```

The expected operation of the `ind2hh()` function given `test.df` as input is:

hh	tfti	->	id	n.kids	n.cases
1	1		1	2	2
1	1		2	3	1
2	1				
2	0				
2	0				

## Exercise 10 : Computer intensive methods

Confirm this behaviour:

```
test.df
ind2hh(test.df)
```

We can apply this function to the datasets as required. For example:

```
x <- read.table("gambia09.sim", header = TRUE)
x
x.hh <- ind2hh(x)
x.hh
```

We will now write a function that will simulate a single LQAS survey.

Create a new function called `lqas.run()`:

```
lqas.run <- function() {}
```

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

```
fix(lqas.run)
```

Edit the function to read:

```
function(data, n, d)
{
  kids <- cases <- 0
  households <- data[sample(x = 1:nrow(data), replace = TRUE), ]
  for (i in 1:nrow(households))
  {
    kids <- kids + households[i, ]$n.kids
    cases <- cases + households[i, ]$n.cases
    if (cases > d)
    {
      outcome = 1; break
    }
    if (kids >= n & cases <= d)
    {
      outcome = 0; break
    }
  }
  result <- list(kids = kids, cases = cases, outcome = outcome)
  return(result)
}
```

We should try this function on a low, a moderate, and a high prevalence dataset. To select suitable test datasets we need to know the prevalence in each dataset:

```
for(i in dir(pattern = "*.sim"))
{
  cat(i, "\t:\t", mean(read.table(i, header = TRUE)$tfti), "\n")
}
```

The coding scheme used for the `tfti` variable (0=no, 1=yes) allows us to use the `mean()` function to calculate prevalence in these datasets.

## Exercise 10 : Computer intensive methods

If you want to use the `dir()` function to list files stored outside of the current working directory you will need to specify an appropriate value for the `path` parameter. On some systems you may also need to set the value of the `full.names` parameter to `TRUE`. For example:

```
for(i in dir(path = "~/rex", pattern = "*.sim", full.names = TRUE))
{
  cat(i, "\t:\t", mean(read.table(i, header = TRUE)$tfti), "\n")
}
```

Cycles through all files ending in `.sim` (specified by giving the value `"*.sim"` to the `pattern` parameter) that are stored the `rex` directory under the users *home* directory (specified by giving the value `"~/rex"` to the `path` parameter) on UNIX systems. You cannot usefully specify a URL for the `path` parameter of the `dir()` function.

We will use `tanzania04.sim` as an example of a low prevalence dataset:

```
x <- read.table("tanzania04.sim", header = TRUE)
x.hh <- ind2hh(x)
lqas.run(data = x.hh, n = 50, d = 14)
```

Repeat the last function call several times. The function should, for most calls, return:

```
$outcome
[1] 0
```

We will use `tanzania08.sim` as an example of a high prevalence dataset:

```
x <- read.table("tanzania08.sim", header = TRUE)
x.hh <- ind2hh(x)
lqas.run(data = x.hh, n = 50, d = 14)
```

Repeat the last function call several times. The function should, for most calls, return:

```
$outcome
[1] 1
```

We will use `tanzania06.sim` as an example of a moderate prevalence dataset:

```
x <- read.table("tanzania06.sim", header = TRUE)
x.hh <- ind2hh(x)
lqas.run(data = x.hh, n = 50, d = 14)
```

Repeat the last function call several times. The function should return:

```
$outcome
[1] 0
```

And:

```
$outcome
[1] 1
```

In roughly equal proportion.

## Exercise 10 : Computer intensive methods

The simulation will require repeated sampling from the same dataset. We need to write a function to do this.

Create a new function called `lqas.simul()`:

```
lqas.simul <- function() {}
```

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

```
fix(lqas.simul)
```

Edit the function to read:

```
function(data, n, d, runs)
{
  all.runs <- data.frame()
  for (i in 1:runs)
  {
    one.run <- data.frame(lqas.run(data = data, n = n, d = d))
    all.runs <- rbind(all.runs, one.run)
  }
  p <- sum(data$n.cases) / sum(data$n.kids)
  asn <- mean(all.runs$kids)
  p.high <- mean(all.runs$outcome)
  result <- list(p = p, asn = asn, p.high = p.high)
  return(result)
}
```

We can test this function with the same three test datasets:

```
x <- read.table("tanzania04.sim", header = TRUE)
x.hh <- ind2hh(x)
lqas.simul(data = x.hh, n = 50, d = 14, runs = 250)

x <- read.table("tanzania08.sim", header = TRUE)
x.hh <- ind2hh(x)
lqas.simul(data = x.hh, n = 50, d = 14, runs = 250)

x <- read.table("tanzania06.sim", header = TRUE)
x.hh <- ind2hh(x)
lqas.simul(data = x.hh, n = 50, d = 14, runs = 250)
```

The simulation consists of applying this function to each of the datasets in turn and collating the results. We will create a function to do this.

## Exercise 10 : Computer intensive methods

Create a new function called `main.simul()`:

```
main.simul <- function() {}
```

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

```
fix(main.simul)
```

Edit the function to read:

```
function(n, d, runs)
{
  result <- data.frame()
  for(file.name in dir(pattern = "*.sim"))
  {
    cat(".", sep = "")
    village.data <- ind2hh(read.table(file.name, header = TRUE))
    village.runs <- lqas.simul(village.data, n, d, runs)
    village.runs$file.name <- file.name
    result <- rbind(result, as.data.frame(village.runs))
  }
  return(result)
}
```

We are now ready to run the simulation:

```
z1 <- main.simul(n = 50, d = 14, runs = 250)
```

The returned data.frame object (`z1`) contains the results of the simulation:

```
z1
```

We can examine the prevalences in the test datasets:

```
x11()
hist(z1$p,
     main = "Prevalence in test datasets",
     xlab = "Proportion TF/TI")
```

We examine the sample size required to make classifications at different levels of prevalence as an *average sample number* (ASN) curve:

```
x11()
plot(z1$p,
     z1$asn,
     main = "ASN Curve",
     xlab = "Proportion TF/TI",
     ylab = "Sample size required to make a classification")
```

## Exercise 10 : Computer intensive methods

We can examine the performance of the sampling plan by plotting its *operating characteristic* (OC) curve:

```
x11()
plot(z1$p,
     z1$p.high,
     main = "OC Curve",
     xlab = "Proportion TF/TI",
     ylab = "Probability of classification as high prevalence")
```

We can compare the simulation results with the expected *operating characteristic* (OC) curve under ideal sampling conditions:

```
lines(seq(0, max(z1$p), 0.01),
      pbinom(14, 50, seq(0, max(z1$p), 0.01), lower.tail = FALSE),
      lty = 3)
```

The LQAS method appears to be reasonably robust to the loss of sampling variation introduced by the proposed sampling constraints. There is, however, some deviation from the expected *operating characteristic* (OC) curve at lower prevalences:

```
x11()
plot(z1$p,
     z1$p.high,
     xlim = c(0.10, 0.35),
     main = "OC Curve",
     xlab = "Proportion TF/TI",
     ylab = "Probability of classification as high prevalence")

lines(seq(0.10, 0.35, 0.01),
      pbinom(14, 50, seq(0.10, 0.35, 0.01), lower.tail = FALSE),
      lty = 3)
```

This deviation from the expected *operating characteristic* (OC) curve is likely to be due to a few very large households in which many of the children have active trachoma. You can check this by examining the **p.high** (i.e. the probability of a classification as high prevalence) column in **z1**, and household size and trachoma status in the individual data files that return higher than expected values for **p.high**. The **ind2hh()** function is likely to prove useful in this context.

The observed deviation from the expected *operating characteristic* (OC) curve is small but it is important, in the resource-constrained context of trachoma-endemic countries, to minimise the false positive rate in order to ensure that resources are not devoted to communities that do not need them most. We might be able to improve the performance of the survey method in this regard by restricting the sample so that only younger children are examined. This will have the effect of reducing the number of children examined in each household. It also has the benefit of making the surveys simpler and quicker since younger children will tend to be closer to home than older children.

The range of ages in the datasets can be found using:

```
for(i in dir(pattern = "*.sim"))
{
  cat(i, "\t:\t", range(read.table(i, header = TRUE)$age), "\n")
}
```

We will investigate the effect of restricting the sample to children aged between two and five years (inclusive).



## Exercise 10 : Computer intensive methods

Use the **fix()** function to edit the **main.simul()** function:

```
fix(main.simul)
```

Edit the function to read:

```
function(n, d, runs)
{
  result <- data.frame()
  for(file.name in dir(pattern = "*.sim"))
  {
    cat(".", sep = "")
    data.all <- read.table(file.name, header = TRUE)
    data.restricted <- subset(data.all, age >= 2 & age <= 5)
    village.data <- ind2hh(data.restricted)
    village.runs <- lqas.simul(village.data, n, d, runs)
    village.runs$file.name <- file.name
    result <- rbind(result, as.data.frame(village.runs))
  }
  return(result)
}
```

We are now ready to run the simulation again:

```
z2 <- main.simul(n = 50, d = 14, runs = 250)
```

The data.frame object **z2** contains the results of the simulation:

```
z2
```

We can examine the performance of the sampling plan on the age-restricted datasets by plotting its *operating characteristic* (OC) curve and comparing the simulation results with the expected *operating characteristic* (OC) curve under ideal sampling conditions:

```
x11()
plot(z2$p,
      z2$p.high,
      main = "OC Curve",
      xlab = "Proportion TF/TI",
      ylab = "Probability of classification as high prevalence")

lines(seq(0, max(z2$p), 0.01),
       pbinom(14, 50, seq(0, max(z2$p), 0.01), lower.tail = FALSE),
       lty = 3)
```

## Exercise 10 : Computer intensive methods

We should also take a closer look at the range of prevalences where the deviation from the expected *operating characteristic* (OC) curve was largest and most problematic in the previous simulation:

```
x11()
plot(z2$p,
     z2$p.high,
     xlim = c(0.10, 0.35),
     main = "OC Curve",
     xlab = "Proportion TF/TI",
     ylab = "Probability of classification as high prevalence")

lines(seq(0.10, 0.35, 0.01),
      pbinom(14, 50, seq(0.10, 0.35, 0.01), lower.tail = FALSE),
      lty = 3)
```

We can compare the behaviour of the sampling plan in the unrestricted and age-restricted datasets:

```
x11()
plot(z1$p,
     z1$p.high,
     main = "",
     xlab = "",
     ylab = "",
     axes = FALSE,
     xlim = c(0.10, 0.35))

par(new = TRUE)

plot(z2$p,
     z2$p.high,
     main = "OC Curve",
     xlab = "Proportion TF/TI",
     ylab = "Probability of classification as high prevalence",
     xlim = c(0.10, 0.35),
     pch = 3)

lines(seq(0.10, 0.35, 0.01),
      pbinom(14, 50, seq(0.10, 0.35, 0.01), lower.tail = FALSE),
      lty = 3)
```

Restricting the sample to children aged between two and five years (inclusive) appears to have improved the behaviour of the survey method by lowering the false positive rate to close to expected behaviour under ideal sampling condition. Process simulation has allowed us to improve the performance of the survey method without expensive and lengthy field-work. In practice, the method would now be validated in the field probably by repeated sampling of communities in which prevalence was known from house-to-house screening.

## Exercise 10 : Computer intensive methods

*Cellular automata machines* are simple computing devices that are commonly used to simulate social, biological, and physical processes. Despite their simplicity, cellular automata machines are general purpose computing devices. This means that they may be used for any *computable* problem. The way that problems are specified to cellular automata machines make them simple to program for some types of problem and difficult to program for other types of problem. In this exercise we will explore the use of cellular automata machines to create a simple model of epidemic spread.

Cellular automata machines model a universe in which space is represented by a uniform grid, time advances in steps, and the laws of the universe are represented by a set of rules which compute the future state of each cell of the grid from its current state and from the current state of its neighbouring cells.

Typically, a cellular automata machine has the following features:

1. It consists of a large number of identical cells arranged in a regular grid. The grid is a two-dimensional projection of a torus (a ring-doughnut shaped surface) and has no edges.
2. Each cell can be in one of a limited number of states.
3. Time advances through the simulation in steps. At each time-step, the state of a cell may change.
4. The state of a cell after each time-step is determined by a set of rules that define how the future state of a cell depends on the current state of the cell and the current state of its immediate neighbours. This set of rules is used to update the state of every cell in the grid at each time-step. Since the rules refer only to the state of an individual cells and its immediate neighbours, cellular automata machines are best suited to modelling situations where local interactions give rise to global phenomena.

In this exercise we will simulate a cellular automata machine using **R** and then use the simulated machine to simulate epidemic spread. We will use three functions to simulate the cellular automata machine (CAM):

**cam.run()**

This function will display the initial state of the CAM, examine each cell in the CAM grid, apply the CAM rule-set to each cell, update the CAM grid, and display the state of the CAM at each time-step.

**cam.state.display()**

This function will display the state of the CAM at each time-step. It will be implemented using the **image()** function to plot the contents of matrices held in the **cam.state** list object (see below). This function will be developed as we refine the epidemic model.

**cam.rule()**

This function will contain the rule-set. It will be implemented using **ifelse()** functions to codify rules. This function will also be developed as we refine the epidemic model.

The current and future states of the CAM will be held in two *global* list objects:

**cam.state**

This object will contain the current state of the CAM and must contain a matrix object called **grid**.

**cam.state.new**

This object will contain the the state of the CAM at the next time-step as defined by the the rule-set.

Other *global* objects will be defined as required.

## Exercise 10 : Computer intensive methods

Create a new function called `cam.run()`:

```
cam.run <- function() {}
```

This creates an empty function called `cam.run()`.

Use the `fix()` function to edit the `cam.run()` function:

```
fix(cam.run)
```

Edit the function to read:

```
function(steps)
{
  cam.state.new <<- cam.state
  cam.state.display(time.step = 0)
  max.x <- nrow(cam.state$grid)
  max.y <- ncol(cam.state$grid)
  for(time.step in 1:steps)
  {
    for(y in 1:max.y)
    {
      for(x in 1:max.x)
      {
        cell <- cam.state$grid[x, y]
        north <- cam.state$grid[x, ifelse(y == 1, max.y, y - 1)]
        south <- cam.state$grid[x, ifelse(y == max.y, 1, y + 1)]
        east <- cam.state$grid[ifelse(x == max.x, 1, x + 1), y]
        west <- cam.state$grid[ifelse(x == 1, max.x, x - 1), y]
        cam.rule(cell, north, south, east, west, x, y, time.step)
      }
    }
    cam.state <<- cam.state.new
    cam.state.display(time.step = time.step)
  }
}
```

Note that when we assign anything to the state of the CAM (held in `cam.state` and `cam.state.new`) we use the `<<-` (instead of the usual `<-`) assignment operator. This operator allows assignment to objects outside of the function in the *global* environment. Objects that are stored in the *global* environment are available to all functions.

## Exercise 10 : Computer intensive methods

Create a new function called `cam.state.display()`:

```
cam.state.display <- function() {}
```

This creates an empty function called `cam.state.display()`.

Use the `fix()` function to edit the `cam.state.display()` function:

```
fix(cam.state.display)
```

Edit the function to read:

```
cam.state.display <- function(time.step)
{
  if(time.step == 0)
  {
    x11()
    par(pty = "s")
  }
  image(cam.state$grid, main = paste("Infected at time :", time.step),
        col = c("wheat", "navy"), axes = FALSE)
}
```

The `cam.rule()` function contains the rules of the CAM universe.

We will start with a very simple *infection* rule:

Each cell can be either “infected” or “not-infected”.

If a cell is already “infected” it will remain “infected”.

If a cell is “not-infected” then it will change its state to “infected” based on the state of its neighbours: If a neighbouring cell is “infected” it will infect the cell with a fixed probability or *transmission pressure*.

Create a new function called `cam.rule()`:

```
cam.rule <- function() {}
```

This creates an empty function called `cam.run()`.

Use the `fix()` function to edit the `cam.run()` function:

```
fix(cam.rule)
```

Edit the function to read:

```
cam.rule <- function(cell, north, south, east, west, x, y, time.step)
{
  pressure <- c(north, south, east, west) * rbinom(4, 1, TP)
  cam.state.new$grid[x, y] <-<- ifelse(cell == 1 | sum(pressure) > 0, 1, 0)
}
```

The basic CAM machine is now complete.

## Exercise 10 : Computer intensive methods

We need to specify a value for the transmission pressure (**TP**):

```
TP <- 0.2
```

And define the initial state of the CAM:

```
cases <- matrix(0, nrow = 19, ncol = 19)
cases[10, 10] <- 1
cam.state <- list(grid = cases)
```

We can now run the simulation:

```
cam.run(steps = 20)
```

The number of infected cells is:

```
sum(cam.state$grid)
```

We can use this model to investigate the effect of different transmission pressures by systematically altering the transmission pressure specified in **TP**:

```
cases <- matrix(0, nrow = 19, ncol = 19)
cases[10, 10] <- 1
cam.state.initial <- list(grid = cases)

pressure <- vector(mode = "numeric")
infected <- vector(mode = "numeric")

for(TP in seq(from = 0.05, to = 0.25, by = 0.05))
{
  cam.state <- cam.state.initial
  cam.run(steps = 20)
  graphics.off()
  pressure <- c(pressure, TP)
  infected <- c(infected, sum(cam.state$grid))
}
plot(pressure, infected)
```

In practice we would run the simulation many times for each transmission pressure and plot (e.g.) the median number of infected cells found at the end of each run of the model.

We can extend the model to include host immunity by specifying a new layer of cells (i.e. for host immunity) and modifying the CAM rule-set appropriately.

Use the **fix()** function to edit the **cam.rule()** function:

```
fix(cam.rule)
```

Edit the function to read:

```
cam.rule <- function(cell, north, south, east, west, x, y, time.step)
{
  pressure <- c(north, south, east, west) * rbinom(4, 1, TP)
  cam.state.new$grid[x, y] <<- ifelse(cell == 1
    | (sum(pressure) > 0 & cam.state$immune[x, y] != 1), 1, 0)
}
```

## Exercise 10 : Computer intensive methods

We need to specify values for the transmission pressure (**TP**) and the proportion of the population that is immune (**IM**):

```
TP <- 0.2
IM <- 0.4
```

And define the initial state of the CAM:

```
cases <- matrix(0, nrow = 19, ncol = 19)
cases[10, 10] <- 1
immune <- matrix(rbinom(361, 1, IM), nrow = 19, ncol = 19)
immune[10,10] <- 0
cam.state <- list(grid = cases, immune = immune)
```

We can display the distribution of immune cells using the `image()` function:

```
image(cam.state$immune, main = "Immune",
      col = c("wheat", "navy"), axes = FALSE)
```

We can now run the simulation:

```
cam.run(steps = 30)
```

The number of infected cells is:

```
sum(cam.state$grid)
```

A better summary is the proportion of susceptible (i.e. non-immune) cells that become infected during a run:

```
sum(cam.state$grid) / (361 - sum(cam.state$immune))
```

We can use this model to investigate the effect of different proportions of the population that are immune by systematically altering the value assigned to **IM**:

```
immune.p <- vector(mode = "numeric")
infected.p <- vector(mode = "numeric")

for(IM in seq(0, 0.5, 0.05))
{
  cases <- matrix(0, nrow = 19, ncol = 19)
  cases[10, 10] <- 1
  immune <- matrix(rbinom(361, 1, IM), nrow = 19, ncol = 19)
  immune[10,10] <- 0
  cam.state <- list(grid = cases, immune = immune)
  cam.run(steps = 30)
  graphics.off()
  immune.p <- c(immune.p, sum(cam.state$immune) / 361)
  infected.p <- c(infected.p, sum(cam.state$grid) /
    (361 - sum(cam.state$immune)))
}
plot(immune.p, infected.p)
```

In practice we would run the simulation many times for each value of **IM** and plot (e.g.) the median proportion of susceptible (i.e. non-immune) cells that become infected.

## Exercise 10 : Computer intensive methods

A simple modification to this model would be to record the time-step at which individual cells become infected. We can do this by adding a new layer of cells (i.e. to record the time-step at which a cell becomes infected) and modifying the CAM rule-set appropriately.

Use the **fix()** function to edit the **cam.rule()** function:

```
fix(cam.rule)
```

Edit the function to read:

```
function(cell, north, south, east, west, x, y, time.step)
{
  pressure <- c(north, south, east, west) * rbinom(4, 1, TP)
  cam.state.new$grid[x, y] <<- ifelse(cell == 1
    | (sum(pressure) > 0 & cam.state$immune[x, y] != 1), 1, 0)
  if(cell != 1 & cam.state.new$grid[x, y] == 1)
  {
    cam.state.new$time.infected[x, y] <<- time.step
  }
}
```

We need to specify values for the transmission pressure (**TP**) and the proportion of the population that is immune (**IM**):

```
TP <- 0.2
IM <- 0.4
```

And define the initial state of the CAM:

```
cases <- matrix(0, nrow = 19, ncol = 19)
cases[10, 10] <- 1
immune <-matrix(rbinom(361, 1, IM), nrow = 19, ncol = 19)
immune[10,10] <- 0
time.infected <- matrix(NA, nrow = 19, ncol = 19)
time.infected[10,10] <- 1
cam.state <-list(grid = cases,
  immune = immune,
  time.infected = time.infected)
```

We can now run the simulation:

```
cam.run(steps = 120)
```

Recording the time-step at which individual cells become infected allows us to plot an epidemic curve from the model:

```
x11()
hist(cam.state$time.infected)
```

The **image()** function can provide an alternative view of the same data:

```
image(cam.state$time.infected, axes = FALSE)
```

The colour of each cell reflects the time-step of infection (i.e. the darker cells were infected before the lighter cells).



## Exercise 10 : Computer intensive methods

The CAM models that we have developed are general models of an infectious phenomenon. They could, for example, be models of the spread of an item of gossip, a forest fire, or an ink-spot. They are poorly specified models for the epidemic spread of an infectious disease. In particular, they assume that a cell is infectious to other cells immediately after infection, that an infected cell never lose its ability to infect other cells, recovery never takes place, and immunity is never acquired. These deficits in the models may be addressed by appropriate modification of the CAM rule-set.

A simple and useful model of epidemic spread is the *SIR* model. The letters in *SIR* refer to the three states that influence epidemic spread that an individual can exist in. The three states are Susceptible, Infectious, and Recovered.

We will now modify our CAM model to follow the *SIR* model using the following parameters:

### Ssusceptible

A cell may be immune or non-immune. A cell may be immune prior to the epidemic or acquire immunity fourteen time-steps after infection. Once a cell is immune it remains immune.

### Infectious

An infected cell is infectious from eight to fourteen time-steps after being infected.

### Recovered

A cell is clinically sick from ten to twenty time-steps after being infected.

If one time-step is taken to equal one day, these parameters provide a coarse simulation of the course of a measles infection.

Use the `fix()` function to edit the `cam.rule()` function:

```
fix(cam.rule)
```

Edit the function to read:

```
function(cell, north, south, east, west, x, y, time.step)
{
  time.since.infection <- time.step - cam.state$time.infected[x, y]
  cam.state.new$grid[x, y] <<- ifelse(time.since.infection
    %in% INFECTION, 1, 0)
  cam.state.new$clinical.features[x, y] <<- ifelse(time.since.infection
    %in% CLINICAL.FEATURES, 1, 0)
  cam.state.new$immune[x, y] <<- ifelse(!is.na(time.since.infection)
    & time.since.infection > IMMUNITY, 1, cam.state$immune[x,y])
  if(cam.state$infected[x, y] == 1)
  {
    cam.state.new$infected[x, y] <<- 1
    cam.state.new$time.infected[x, y] <<- cam.state$time.infected[x, y]
  }
  else
  {
    pressure <- c(north, south, east, west) * rbinom(4, 1, TP)
    if(sum(pressure) > 0 & cam.state$immune[x, y] != 1)
    {
      cam.state.new$infected[x, y] <<- 1
      cam.state.new$time.infected[x, y] <<- time.step
    }
  }
}
```

## Exercise 10 : Computer intensive methods

It will also be useful to have a more detailed report of the state of the CAM at each time-step.

Use the `fix()` function to edit the `cam.state.display()` function:

```
fix(cam.state.display)
```

Edit the function to read:

```
function(time.step)
{
  if(time.step == 0)
  {
    x11(width = 9, height = 9)
    par(mfrow = c(2, 2))
    par(pty = "s")
  }
  image(cam.state$grid,
        main = paste("Infectious at time :", time.step),
        col = c("wheat", "navy"), axes = FALSE)
  image(cam.state$clinical.features,
        main = paste("Clinical cases at time :", time.step),
        col = c("wheat", "navy"), axes = FALSE)
  image(cam.state$time.infected,
        main = paste("Infected at time :", time.step), axes = FALSE)
  image(cam.state$immune, main = paste("Immune at time :", time.step),
        col = c("wheat", "navy"), axes = FALSE)
}
```

We need to specify values for the transmission pressure (**TP**) and the proportion of the population that is immune (**IM**):

```
TP <- 0.2
IM <- 0.2
```

And the *SIR* parameters:

```
CLINICAL.FEATURES <- 10:20
INFECTIOUS <- 8:14
IMMUNITY <- 14
```

And define the initial state of the CAM:

```
infected <- matrix(0, nrow = 19, ncol = 19)
infected[10, 10] <- 1
time.infected <- matrix(NA, nrow = 19, ncol = 19)
time.infected[10,10] <- 0
infectious <- matrix(0, nrow = 19, ncol = 19)
immune <-matrix(rbinom(361, 1, IM), nrow = 19, ncol = 19)
immune[10,10] <- 0
clinical.features <- matrix(0, nrow = 19, ncol = 19)
cam.state <-list(grid = infectious,
                infected = infected,
                time.infected = time.infected,
                immune = immune,
                clinical.features = clinical.features)
```

## Exercise 10 : Computer intensive methods

We should also record the number of susceptible cells for later use:

```
susceptibles <- 361 - sum(cam.state$immune)
```

We can now run the simulation:

```
cam.run(steps = 200)
```

We can now calculate the proportion of susceptible (i.e. non-immune) cells that become infected:

```
sum(cam.state$infected) / susceptibles
```

We can use this model to test the effect of an intervention such as isolating an infected cell for a short period after clinical features first appear. We can do this, imperfectly because it does not allow us to specify compliance, by shortening the infectious period to include only the non-symptomatic time-steps and one time-step after clinical features have appeared:

```
INFECTIOUS <- 8:11
```

Resetting the initial state of the CAM:

```
infected <- matrix(0, nrow = 19, ncol = 19)  
infected[10, 10] <- 1  
time.infected <- matrix(NA, nrow = 19, ncol = 19)  
time.infected[10,10] <- 0  
infectious <- matrix(0, nrow = 19, ncol = 19)  
immune <-matrix(rbinom(361, 1, IM), nrow = 19, ncol = 19)  
immune[10,10] <- 0  
clinical.features <- matrix(0, nrow = 19, ncol = 19)  
cam.state <-list(grid = infectious,  
                 infected = infected,  
                 time.infected = time.infected,  
                 immune = immune,  
                 clinical.features = clinical.features)
```

Recording the number of susceptible cells:

```
susceptibles <- 361 - sum(cam.state$immune)
```

Running the simulation:

```
cam.run(steps = 200)
```

And recalculating the proportion of susceptible (i.e. non-immune) cells that become infected:

```
sum(cam.state$infected) / susceptibles
```

## Exercise 10 : Computer intensive methods

We have developed a simple but realistic model of epidemic spread using a cellular automata machine. Such a model could be used to investigate the relative effect of model parameters (e.g. initial proportion immune, initial number of infective cells, transmission pressure, &c.) on epidemic spread by changing a single parameter at a time and running the simulation. Since the model is stochastic, the effect of each parameter change would be simulated many times and suitable summaries calculated.

The model could be improved by, for example:

Allowing for an *open population* with births (or immigration) and deaths (or emigration). This could be implemented by allowing immune cells to become susceptible after a specified number of time-steps. The ratio of births to deaths could then be modelled as the ratio of the length of the infectious period to the length of the immune period.

Specifying a non-uniform distribution of transmission pressure during the infectious period.

Allowing for individual variation in susceptibility.

Allowing for individual variation in the duration of the infectious period.

Simulating a non-uniform population density by allowing cells to be empty. Note that an immune cell is the same as an empty cell in the current model.

Simulating a clustered distribution of immunity in the initial state of the CAM.

Allowing a small proportion of infections to be infectious without exhibiting clinical features.

Specifying rules that are applied at different time-points such as introducing isolation only after a certain number of clinical cases have appeared (i.e. after an epidemic has been detected).

Allowing the coverage of interventions (e.g. the coverage of a vaccination campaign or compliance with isolation instructions) that are introduced after an epidemic has been detected to be specified.

Simulating a more complex social structure. This could be implemented by allowing cells to belong to one of a finite set of castes with different initial conditions (e.g. immunity) and having rules that specify the level of interaction (i.e. the transmission pressure) between members of separate castes and the levels of intervention coverage achievable in the separate castes.

Extending the neighbourhood definition by using the corner (i.e. north-east, south-east, south-west, and north-west) cells as neighbours for consideration in the CAM rule-set.

We can now quit **R**:

**q()**

For this exercise there is no need to save the workspace image so click the **No** button (GUI) or enter **n** when prompted to save the workspace image (terminal).

## Exercise 10 : Summary

Computer intensive methods provide an alternative to classical statistical techniques for both estimation and statistical hypothesis testing. They have the advantage of being simple to implement and they remain simple even with complex estimators.

Computer based simulation can simulate both data and processes. Process simulations may be arbitrarily complex. Process simulation is a useful development tool that can save considerable time and expense when developing systems and methods.

Process simulation can also be used to model complex social phenomena such as epidemic spread. Such simulations allow (e.g.) the relative efficacy of interventions to be evaluated.

**R** provides functions that allow you to implement computer intensive methods such as the bootstrap and computer based simulation.

## What now?

Now that you have had a taste of using **R** you will be able to decide whether it 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 document 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 document 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/>

John Verzani has a complete introductory course designed to be used with an introductory statistics textbook to "illustrate the features of **R** that can be learned in a one-semester introductory statistics course" available from:

<http://www.math.csi.cuny.edu/Statistics/R/simpleR/index.html>

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.