R: A self-learn tutorial

1 Introduction

R is a software language for carrying out complicated (and simple) statistical analyses. It includes routines for data summary and exploration, graphical presentation and data modelling. The aim of this document is to provide you with a basic fluency in the language. It is suggested that you work through this document at the computer, having started an **R** session. Type in all of the commands that are printed, and check that you understand how they operate. Then try the simple exercises at the end of each section.

When you work in **R** you create objects that are stored in the current workspace(sometimes called image). Each object created remains in the image unless you explicitly delete it. At the end of the session the workspace will be lost unless you save it. You can save the workspace at any time by clicking on the disc icon at the top of the control panel.

Commands written in R are saved in memory throughout the session. You can scroll back to previous commands typed by using the 'up' arrow key (and 'down' to scroll back again). You can also 'copy' and 'paste' using standard windows editor techniques (for example, using the 'copy' and 'paste' dialog buttons). If at any point you want to save the transcript of your session, click on 'File' and then 'Save History', which will enable you to save a copy of the commands you have used for later use. As an alternative you might copy and paste commands manually into a notepad editor or something similar.

You finish an **R** session by typing

```
> q()
```

at which point you will also be prompted as to whether or not you want to save the current workspace If you do not, it will be lost.

2 Objects and Arithmetic

R stores information and operates on *objects*. The simplest objects are *scalars*, *vectors* and *matrices*. But there are many others: *lists* and *dataframes* for example. In advanced use of **R** it can also be useful to define new types of object, specific for particular application. We will stick with just the most commonly used objects here.

An important feature of **R** is that it will do different things on different types of objects. For example, type:[1]

```
> 4+6
```

The result should be

```
[1] 10
```

So, **R** does scalar arithmetic returning the scalar value 10. (In actual fact, **R** returns a vector of length 1 - hence the [1] denoting first element of the vector.

We can assign objects values for subsequent use. For example:

```
x<-6
y<-4
z<-x+y
```

[1]We adopt the convention of using typeface font to denote things typed in R. The > sign is not typed however; it denotes the prompt symbol.

would do the same calculation as above, storing the result in an object called `z`. We can look at the contents of the object by simply typing its name:

```
z
[1] 10
```

At any time we can list the objects which we have created:

```
> ls()
[1] "x"              "y"              "z"
```

Notice that `ls` is actually an object itself. Typing `ls` would result in a display of the contents of this object, in this case, the commands of the function. The use of parentheses, `ls()`, ensures that the function is *executed* and its result - in this case, a list of the objects in the directory - displayed.

More commonly a function will operate on an object, for example

```
> sqrt(16)
[1]  4
```

calculates the square root of 16. Objects can be removed from the current workspace with the `rm` function:

```
> rm(x,y)
```

for example.

There are many standard functions available in R, and it is also possible to create new ones.

Vectors can be created in **R** in a number of ways. We can describe all of the elements:

```
> z<-c(5,9,1,0)
```

Note the use of the function `c` to *concatenate* or 'glue together' individual elements. This function can be used much more widely, for example

```
> x<-c(5,9)
> y<-c(1,0)
> z<-c(x,y)
```

would lead to the same result by gluing together two vectors to create a single vector.

Sequences can be generated as follows:

```
> x<-1:10
```

while more general sequences can be generated using the `seq` command. For example:

```
> seq(1,9,by=2)
[1] 1 3 5 7 9
```

and

```
> seq(8,20,length=6)
[1]  8.0 10.4 12.8 15.2 17.6 20.0
```

These examples illustrate that many functions in **R** have optional arguments, in this case, either the step length or the total length of the sequence (it doesn't make sense to use both). If you leave out both of these options, **R** will make its own default choice, in this case assuming a step length of 1. So, for example,

```
> x<-seq(1,10)
```

also generates a vector of integers from 1 to 10.

At this point it's worth mentioning the `help` facility. If you don't know how to use a function, or don't know what the options or default values are, type `help`(**functionname**) where **functionname** is the name of the function you are interested in. This will usually help and will often include examples to make things even clearer.

Another useful function for building vectors is the `rep` command for repeating things. For example

```
> rep(0,100)
  [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 [38] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 [75] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

or

```
> rep(1:3,6)
 [1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
```

Notice also a variation on the use of this function

```
> rep(1:3,c(6,6,6))
 [1] 1 1 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3 3
```

which we could also simplify cleverly as

```
> rep(1:3,rep(6,3))
 [1] 1 1 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3 3
```

As explained above, **R** will often adapt to the objects it is asked to work on. For example:

```
> x<-c(6,8,9)
> y<-c(1,2,4)
> x+y
[1]  7 10 13
```

and

```
> x*y
[1]  6 16 36
```

showing that **R** uses componentwise arithmetic on vectors. **R** will also try to make sense if objects are mixed. For example,

```
> x<-c(6,8,9)
> x+2
[1]  8 10 11
```

though care should be taken to make sure that **R** is doing what you would like it to in these circumstances.

Two particularly useful functions worth remembering are `length` which returns the length of a vector (i.e. the number of elements it contains) and `sum` which calculates the sum of the elements of a vector.

Exercises

1. Define

```
> x<-c(4,2,6)
> y<-c(1,0,-1)
```

   Decide what the result will be of the following:

   (a) `length(x)`
   (b) `sum(x)`
   (c) `sum(x^2)`
   (d) `x+y`
   (e) `x*y`
   (f) `x-2`
   (g) `x^2`

   Use **R** to check your answers.

2. Decide what the following sequences are and use **R** to check your answers:

   (a) `7:11`
   (b) `seq(2,9)`
   (c) `seq(4,10,by=2)`
   (d) `seq(3,30,length=10)`
   (e) `seq(6,-4,by=-2)`

3. Determine what the result will be of the following **R** expressions, and then use **R** to check you are right:

   (a) `rep(2,4)`
   (b) `rep(c(1,2),4)`
   (c) `rep(c(1,2),c(4,4))`
   (d) `rep(1:4,4)`
   (e) `rep(1:4,rep(3,4))`

4. Use the `rep` function to define simply the following vectors in R.

   (a) 6,6,6,6,6,6
   (b) 5,8,5,8,5,8,5,8
   (c) 5,5,5,5,8,8,8,8

3 Summaries and Subscripting

Let's suppose we've collected some data from an experiment and stored them in an object `x`:

```
> x<-c(7.5,8.2,3.1,5.6,8.2,9.3,6.5,7.0,9.3,1.2,14.5,6.2)
```

Some simple summary statistics of these data can be produced:

```
> mean(x)
[1] 7.216667
> var(x)
[1] 11.00879
> summary(x)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.200   6.050   7.250   7.217   8.475  14.500
```

which should all be self explanatory. It may be, however, that we subsequently learn that the first 6 data correspond to measurements made on one machine, and the second six on another machine. This might suggest summarizing the two sets of data separately, so we would need to extract from `x` the two relevant subvectors. This is achieved by subscripting:

```
> x[1:6]
```

and

```
> x[7:12]
```

give the relevant subvectors. Hence,

```
> summary(x[1:6])
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  3.100   6.075   7.850   6.983   8.200   9.300
> summary(x[7:12])
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.200   6.275   6.750   7.450   8.725  14.500
```

Other subsets can be created in the obvious way. For example:

```
> x[c(2,4,9)]
[1] 8.2 5.6 9.3
```

Negative integers can be used to *exclude* particular elements. For example

```
x[-(1:6)]
```

has the same effect as `x[7:12]`.

Exercises

1. If `x<- c(5,9,2,3,4,6,7,0,8,12,2,9)` decide what each of the following is and use **R** to check your answers:

   (a) `x[2]`

   (b) `x[2:4]`

   (c) `x[c(2,3,6)]`

(d) `x[c(1:5,10:12)]`

(e) `x[-(10:12)]`

2. The data `y<-c(33,44,29,16,25,45,33,19,54,22,21,49,11,24,56)` contain sales of milk in litres for 5 days in three different shops (the first 3 values are for shops 1,2 and 3 on Monday, etc.) Produce a statistical summary of the sales for each day of the week and also for each shop.

4 Matrices

Matrices can be created in **R** in a variety of ways. Perhaps the simplest is to create the columns and then glue them together with the command `cbind`. For example,

```
> x<-c(5,7,9)
> y<-c(6,3,4)
> z<-cbind(x,y)
> z
     x y
[1,] 5 6
[2,] 7 3
[3,] 9 4
```

The dimension of a matrix can be checked with the `dim` command:

```
> dim(z)
[1] 3 2
```

i.e., three rows and two columns. There is a similar command, `rbind`, for building matrices by gluing rows together.

The functions `cbind` and `rbind` can also be applied to matrices themselves (provided the dimensions match) to form larger matrices. For example,

```
> rbind(z,z)
     [,1] [,2]
[1,]    5    7
[2,]    9    6
[3,]    3    4
[4,]    5    7
[5,]    9    6
[6,]    3    4
```

Matrices can also be built by explicit construction via the function `matrix`. For example,

```
z<-matrix(c(5,7,9,6,3,4),nrow=3)
```

results in a matrix `z` identical to `z` above. Notice that the dimension of the matrix is determined by the size of the vector and the requirement that the number of rows is 3, as specified by the argument `nrow=3`. As an alternative we could have specified the number of columns with the argument `ncol=2` (obviously, it is unnecessary to give both). Notice that the matrix is 'filled up' column-wise. If instead you wish to fill up row-wise, add the option `byrow=T`. For example,

```
> z<-matrix(c(5,7,9,6,3,4),nr=3,byrow=T)
> z
     [,1] [,2]
[1,]    5    7
[2,]    9    6
[3,]    3    4
```

Notice that the argument `nrow` has been abbreviated to `nr`. Such abbreviations are always possible for function arguments provided it induces no ambiguity - if in doubt always use the full argument name.

As usual, **R** will try to interpret operations on matrices in a natural way. For example, with z as above, and

```
> y<-matrix(c(1,3,0,9,5,-1),nrow=3,byrow=T)
> y
     [,1] [,2]
[1,]    1    3
[2,]    0    9
[3,]    5   -1
```

we obtain

```
> y+z
     [,1] [,2]
[1,]    6   10
[2,]    9   15
[3,]    8    3
```

and

```
> y*z
     [,1] [,2]
[1,]    5   21
[2,]    0   54
[3,]   15   -4
```

Notice, multiplication here is componentwise rather than conventional matrix multiplication. Indeed, conventional matrix multiplication is undefined for `y` and `z` as the dimensions fail to match. Let's now define

```
> x<-matrix(c(3,4,-2,6),nrow=2,byrow=T)
> x
     [,1] [,2]
[1,]    3    4
[2,]   -2    6
```

Matrix multiplication is expressed using notation `%*%`:

```
> y%*%x
     [,1] [,2]
[1,]   -3   22
[2,]  -18   54
[3,]   17   14
```

Other useful functions on matrices are `t` to calculate a matrix transpose and `solve` to calculate inverses:

```
> t(z)
     [,1] [,2] [,3]
[1,]    5    9    3
[2,]    7    6    4
```

and

```
> solve(x)
           [,1]       [,2]
[1,] 0.23076923 -0.1538462
[2,] 0.07692308  0.1153846
```

As with vectors it is useful to be able to extract sub-components of matrices. In this case, we may wish to pick out individual elements, rows or columns. As before, the `[ ]` notation is used to subscript. The following examples should make things clear:

```
> z[1,1]
[1] 5

> z[c(2,3),2]
[1] 6 4

> z[,2]
[1] 7 6 4

> z[1:2,]
     [,1] [,2]
[1,]    5    7
[2,]    9    6
```

So, in particular, it is necessary to specify which rows and columns are required, whilst omitting the integer for either dimension implies that every element in that dimension is selected.

Exercises

1. Create in **R** the matrices

$$x = \begin{bmatrix} 3 & 2 \\ -1 & 1 \end{bmatrix}$$

and

$$y = \begin{bmatrix} 1 & 4 & 0 \\ 0 & 1 & -1 \end{bmatrix}$$

Calculate the following and check your answers in R:

(a) `2*x`

(b) `x*x`

(c) `x%*%x`

(d) `x%*%y`

(e) `t(y)`

(f) `solve(x)`

2. With `x` and `y` as above, calculate the effect of the following subscript operations and check your answers in R.

   (a) `x[1,]`

   (b) `x[2,]`

   (c) `x[,2]`

   (d) `y[1,2]`

   (e) `y[,2:3]`

5 Attaching to objects

R includes a number of datasets that it is convenient to use for examples. You can get a description of what's available by typing

```
> data()
```

To access any of these datasets, you then type `data(`**dataset**`)` where **dataset** is the name of the dataset you wish to access. For example,

```
> data(trees)
```

Typing

```
> trees[1:5,]
  Girth Height Volume
1   8.3     70   10.3
2   8.6     65   10.3
3   8.8     63   10.2
4  10.5     72   16.4
5  10.7     81   18.8
```

gives us the first 5 rows of these data, and we can now see that the columns represent measurements of girth, height and volume of trees (actually cherry trees: see `help(trees)`) respectively.

Now, if we want to work on the columns of these data, we can use the subscripting technique explained above: for example, `trees[,2]` gives all of the heights. This is a bit tedious however, and it would be easier if we could refer to the heights more explicitly. We can achieve this by attaching to the `trees` dataset:

```
> attach(trees)
```

Effectively, this makes the contents of `trees` a directory, and if we type the name of an object, **R** will look inside this directory to find it. Since `Height` is the name of one of the columns of `trees`, **R** now recognises this object when we type the name. Hence, for example,

```
> mean(Height)
[1] 76
```

and

```
> mean(trees[,2])
[1] 76
```

are synonymous, while it is easier to remember exactly what calculation is being performed by the first of these expressions. In actual fact, `trees` is an object called a dataframe, essentially a matrix with named columns (though a dataframe, unlike a matrix, may also include non-numerical variables, such as character names). Because of this, there is another equivalent syntax to extract, for example, the vector of heights:

```
> trees$Height
```

which can also be used without having first attached to the dataset.

Exercises

1. Attach to the dataset `quakes` and produce a statistical summary of the variables `depth` and `mag`.
2. Attach to the dataset `mtcars` and find the mean weight and mean fuel consumption for vehicles in the dataset (type `help(mtcars)` for a description of the variables available).

6 The `apply` function

It is possible to write loops in R, but they are best avoided whenever possible. A common situation is where we want to apply the same function to every row or column of a matrix. For example, we may want to find the mean value of each variable in the `trees` dataset. Obviously, we could operate on each column separately but this can be tedious, especially if there are many columns. The function `apply` simplifies things. It is easiest understood by example:

```
> apply(trees,2,mean)
   Girth   Height   Volume
13.24839 76.00000 30.17097
```

has the effect of calculating the mean of each column (dimension 2) of `trees`. We'd have used a 1 instead of a 2 if we wanted the mean of every row.

Any function can be applied in this way, though if optional arguments to the function are required these need to be specified as well - see `help(apply)` for further details.

Exercise

1. Repeat the analyses of the datasets `quakes` and `mtcars` using the function `apply` to simplify the calculations.
2. If

$$y = \begin{bmatrix} 1 & 4 & 1 \\ 0 & 2 & -1 \end{bmatrix}$$

what is the result of `apply(y[,2:3],1,mean)`? Check your answer in R.

7 Statistical Computation and Simulation

Many of the tedious statistical computations that would once have had to have been done from statistical tables can be easily carried out in R. This can be useful for finding confidence intervals etc. Let's take as an example the Normal distribution. There are functions in **R** to evaluate the density function, the distribution function and the quantile function (the inverse distribution function). These functions are, respectively, `dnorm`, `pnorm` and `qnorm`. Unlike with tables, there is no need to standardize the variables first. For example, suppose $X \sim N(3, 2^2)$, then

```
> dnorm(x,3,2)
```

will calculate the density function at points contained in the vector `x` (note, `dnorm` will assume mean 0 and standard deviation 1 unless these are specified. Note also that the function assumes you will give the standard deviation rather than the variance. As an example

```
> dnorm(5,3,2)
[1] 0.1209854
```

evaluates the density of the $N(3, 4)$ distribution at $x = 5$. As a further example

```
> x<-seq(-5,10,by=.1)
> dnorm(x,3,2)
```

calculates the density function of the same distribution at intervals of 0.1 over the range $[-5, 10]$. The functions `pnorm` and `qnorm` work in an identical way - use `help` for further information.

Similar functions exist for other distributions. For example, `dt`, `pt` and `qt` for the t-distribution, though in this case it is necessary to give the degrees of freedom rather than the mean and standard deviation. Other distributions available include the binomial, exponential, Poisson and gamma, though care is needed interpreting the functions for discrete variables.

One further important technique for many statistical applications is the simulation of data from specified probability distributions. **R** enables simulation from a wide range of distributions, using a syntax similar to the above. For example, to simulate 100 observations from the $N(3, 4)$ distribution we write

```
> rnorm(100,3,2)
```

Similarly, `rt`, `rpois` for simulation from the t and Poisson distributions, etc.

Exercises

1. Suppose $X \sim N(2, 0.25)$. Denote by f and F the density and distribution functions of X respectively. Use **R** to calculate
   (a) $f(0.5)$
   (b) $F(2.5)$
   (c) $F^{-1}(0.95)$ (recall that F^{-1} is the quantile function)
   (d) $\Pr(1 \leq X \leq 3)$
2. Repeat question 1 in the case that X has a t-distribution with 5 degrees of freedom.
3. Use the function `rpois` to simulate 100 values from a Poisson distribution with a parameter of your own choice. Produce a statistical summary of the result and check that the mean and variance are in reasonable agreement with the true population values.
4. Repeat the previous question replacing `rpois` with `rexp`.

8 Graphics

R has many facilities for producing high quality graphics. A useful facility before beginning is to divide a page into smaller pieces so that more than one figure can be displayed. For example:

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

creates a window of graphics with 2 rows and 2 columns. With this choice the windows are filled up row-wise. Use `mfcol` instead of `mfrow` to fill up column-wise. The function `par` is a general function for setting graphical parameters. There are many options: see `help(par)`.

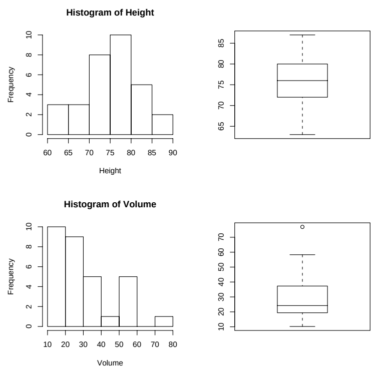

Figure 1: Tree heights and volumes

So, for example

```
> par(mfrow=c(2,2))
> hist(Height)
> boxplot(Height)
> hist(Volume)
> boxplot(Volume)
> par(mfrow=c(1,1))
```

produces Figure 1. Note the final use of `par` to return the graphics window to standard size.

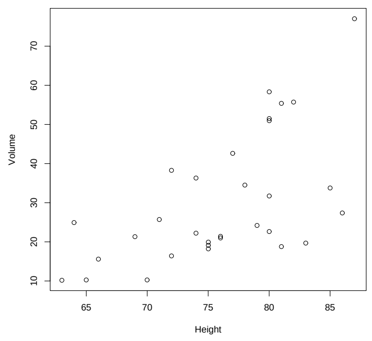

Figure 2: Scatterplot of tree heights and volumes

We can also plot one variable against another using the function `plot`:

```
> plot(Height,Volume)
```

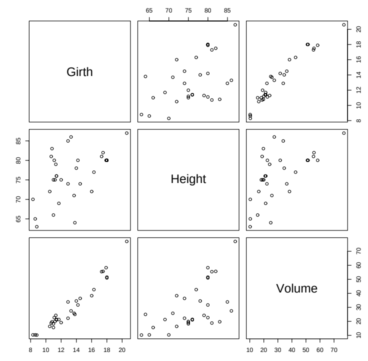

Figure 3: Scatterplot matrix for `tree` data

See Figure 2.

R can also produce a scatterplot matrix (a matrix of scatterplots for each pair of variables) using the function `pairs`:

```
> pairs(trees)
```

See Figure 3. Like many other functions `plot` is object-specific: its behaviour depends on the object to which it is applied. For example, if the object is a matrix, `plot` is identical to `pairs`: try `plot(trees)`. For some other possibilities try:

```
> data(nhtemp)
> plot(nhtemp)

> data(faithful)
> plot(faithful)

> data(HairEyeColor)
> plot(HairEyeColor)
```

There are also many optional arguments in most plotting functions that can be used to control colours, plotting characters, axis labels, titles etc. The functions `points` and `lines` are useful for adding points and lines respectively to a current graph. The function `abline` is useful for adding a line with specified intercept and slope.

To print a graph, point the cursor over the graphics window and press the right button on the mouse. This should open up a menu which includes 'print' as an option. You also have the option to save the figure in various formats, for example as a postscript file, for storage and later use.

Exercises

1. Use

   ```
   > x<-rnorm(100)
   ```

   or something similar, to generate some data. Produce a figure showing a histogram and boxplot of the data. Modify the axis names and title of the plot in an appropriate way.

2. Type the following

```
> x<- (-10):10
> n<-length(x)
> y<-rnorm(n,x,4)
> plot(x,y)
> abline(0,1)
```

Try to understand the effect of each command and the graph that is produced.

3. Type the following:

   ```
   > data(nhtemp)
   > plot(nhtemp)
   ```

   This produces a time series plot of measurements of annual mean temperatures in New Hampshire, U.S.A.

4. The previous example illustrated that `plot` acts differently on objects of different types - the object `nhtemp` has the special class `time series`. More generally, we may have the data of yearly observations in a vector, but need to build the time series plot for ourselves. We can mimic this situation by typing

   ```
   > temp<-as.vector(nhtemp)
   ```

   which creates a vector `temp` that contains only the annual temperatures. We can produce something similar to the time series plot by typing

   ```
   > plot(1912:1971,temp)
   ```

   but this plots points rather than lines. To join the data via lines we would use

   ```
   > plot(1912:1971,temp,type='l')
   ```

   instead. To get points and lines, use `type='b'` instead.

9 Writing functions

An important feature of **R** is the facility to extend the language by writing your own functions. For example, although there is a function `var` which will calculate the variance of set of data in a vector, there is no function for calculating the standard deviation. We can define such a function as follows:

```
> sd <- function(x) sqrt(var(x))
```

Then, for example, if

```
x<-c(9,5,2,3,7)

> sd(x)
[1] 2.863564
```

If we want to change the function, the command `fix(sd)` will open the function in an editor which we can then use to modify it. We could actually have used this form also to *define* the function `sd`. If the function doesn't already exist, the effect is to open an editor containing the template

```
function ()
{
}
```

which you can use to insert the body of your function, the editor making it easy to make insertions and corrections. Note that the braces enable several lines of commands to be included in the function. For example, we might type

```
fix(several.plots)
```

and then use the editor to define the function:

```
several.plots<-function(x){
par(mfrow=c(3,1))
hist(x[,1])
hist(x[,2])
plot(x[,1],x[,2])
par(mfrow=c(1,1))
apply(x,2,summary)
}
```

The result of the function is stated in the final line. Previous lines are regarded as only intermediary, though graphical output will also be sent to screen if it is included. This particular function - which only makes sense if `x` is a matrix with two columns - produces a histogram of each column and a scatterplot of one column against the other. Finally, it returns a summary of each column.

To print the contents of a function, use `fix` to open a window editor containing the body of the function and then click on 'file' and 'print'.

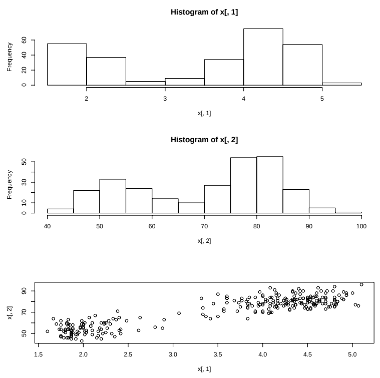

Figure 4: Output of `several.plots`

Hence, for example

```
> several.plots(faithful)
        eruptions waiting
Min.        1.600    43.0
```

```
1st Qu.     2.163    58.0
Median      4.000    76.0
Mean        3.488    70.9
3rd Qu.     4.454    82.0
Max.        5.100    96.0
```

which also produces Figure 4.

Exercises

1. Write a function that takes as its argument two vectors, `x` and `y`, produces a scatterplot, and calculates the correlation coefficient (using `cor(x,y)`).
2. Write a function that takes a vector $(x_1, \ldots, x_n)$ and calculates both $\sum x_i$ and $\sum x_i^2$. (Remember the use of the function `sum`).

10 Other things

There are many other facilities in R. These include:

1. Functions for fitting statistical models such as linear and generalized linear models.
2. Functions for fitting curves to smooth data.
3. Functions for optimisation and equation solving.
4. Facilities to program using loops and conditional statements such as `if` and `while`.
5. Plotting routines to view 3-dimensional data.

There is also the facility to 'bolt-on' additional libraries of functions that have a specific utility. Typing

```
> library()
```

will give a list and short description of the libraries available. Typing

```
> library(libraryname)
```

where `libraryname` is the name of the required library will give you access to the functions in that library.

11 Getting More Help

This tutorial guide is intended to be only introductory. Much more help can be obtained from:

1. The `help` system included in the language;
2. The manuals included in the language: click on 'help', follow the route to 'manuals'. The 'Introduction to R' is especially useful.
3. Books: there are now many which cover the use of **R** (and/or the similar language S-Plus).

12 Downloading R for your own PC

R is a freeware system; you can download it onto your own PC, and you may find it more convenient to do your coursework that way. Don't forget that if your lecturer has provided datasets or programs for you to use in addition to the basic **R** system then you will need to get copies of those too.

Go to the website

```
http://www.stats.bris.ac.uk/R
```

and click on `Windows (95 or later)` under the heading Precompiled Binary Distributions. Then click on `base`. From the list on the next screen choose the Setup program, which will have a name something like `rw1061.exe`. Click on this, to initiate the download. The file is very large (18 megabytes or so), so this is not really practicable over a telephone line. Save the file on your hard disc.

If you need to download onto one PC and then copy the files using floppy disks to install them onto a different PC, there is an option to do so. Click on `mini` instead of the name of the Setup program: you will get access to a directory of files which will fit onto 8 floppy disks.

To install the **R** system, run the Setup program (by clicking on its name in Explorer, or by entering its name in the Run command on the Start menu). You can accept most of the default options - in particular, the location

```
c:\Program Files\R
```

is a good place to install the files. On the 'Select components' menu, it is a good idea to choose Main files, HTML help files, Online (PDF) manuals, and Reference manual.

When the installation is complete, you should have a desktop icon displaying the **R** symbol. It is a good idea to edit the Properties of this icon (right-click on the icon to get the menu where you can choose this), and edit the entry in the 'Start in' box under the 'Shortcut' tab to specify a directory of your choice for your work.