Tutorial: Introduction to R

Rob Nicholls – nicholls@mrc-lmb.cam.ac.uk

MRC LMB Statistics Course 2014

Introduction

The R software suite is ideally suited to data analysis, representation, manipulation, simulation, and software prototyping. The R environment provides:

- A programming language.
- A large collection of tools for data analysis and simulation.
- Graphical facilities for data representation.

This tutorial is meant as an overview of the basics required for successful use of R. For more detailed explanation, see the R documentation, which is distributed with the R package and available online: http://cran.r-project.org/manuals.html.

Installing Packages

One of the benefits of R is that it is free to use, being a GNU open source project. There is a large community who create openly-accessible 'packages' providing functionalities not available in the core distribution. Consequently, there are now well-established packages available for many applications, especially in statistics and data analysis.

There are a large number of user-created packages available for download, many of which are available from the R website: www.r-project.org. To install a package called 'PACKAGE_NAME', type the commands:

```
install.packages("PACKAGE_NAME")
library("PACKAGE_NAME")
```

and the package will be available for use in the current R session.

Contents

1 Getting Started

R can be executed either using the R GUI, or via the unix command prompt (type `R`). Launching R will create a new R session (sessions may be saved and loaded). You will be presented with a console into which R commands can be typed. You can exit R by using the quit function:

```
q()
```

When using R, it is often useful to open a separate plain text file in which to save your R commands. This allows you to easily repeat, reflect on, and improve upon what you've done. Such a text file is called an 'R script', and often has the conventional file extension '`.r`'.

Note that you can select multiple lines in your R script, copy them, and paste them directly into the R console – this allows a sequence of commands to be performed automatically. R scripts can also be executed automatically from the R console (see `source`) or from the shell.

Task 1:

1. Open R.
2. Create a text file (using your text editor of choice) that can act as your working R script during this tutorial session – you can copy commands between this script and the R console as you wish.

2 How to be Self-Sufficient

This tutorial will cover the basics of R. However, once you've got the basics, it's important to know how to go about learning how to use new functionalities that you've never come across before. Consequently, this tutorial will encourage the use of R's invaluable in-built help files.

Given that you're interested in a particular command or function (e.g. `foo`), you can access the help file by typing the name of the command preceded by a '?':

```
?foo
```

Should you ever need to, you can access the R documentation by typing:

```
help.start()
```

Also, you can search the R documentation for a particular phrase (e.g. "bar") by typing:

```
help.search("bar")
```

It may also be useful to know that you can navigate through previous commands by using the ↑ and ↓ keys – this allows you to easily repeat commands you've previously executed.

Task 1:

1. Open the R help file corresponding to the function: `q`
   Is there another name for this function?

3 Commands, Variables and Functions

Commands are typed into the R console. If you type:

```
1+1
```

into the console then the expression will be evaluated, and the result will be printed to the console. Try this now. Simple mathematical operators are available, such as `+`, `-`, `*`, `/` and `^`. Commands may be separated by a newline or by a semicolon.

The result of a single command is always printed to the console, unless it is assigned to a *variable*. In R, variables are used to store data. Variable names can be alphanumeric, but must start with a letter (or a '.'). For example, the following command would assign the value 2 to a variable called `x`:

```
x <- 1+1
```

Here, the assignment operator `<-` assigns the result of the right hand side to the variable on the left hand side. A similar alternative to `<-` is the equality sign `=`. Note that the `<-` operator is very different to the `<` (less than) and `-` (subtraction) operators.

Typing the name of a variable into the console will print the value of that variable. For instance, in this case typing `x` into the console will result in the value `2` being displayed.

Functions may be used to apply a particular operation to a variable. For example, basic mathematical functions available include: `log`, `exp`, `sqrt`, `abs`, `sin`, `cos` and `tan`. These functions take *arguments* as input, and return a result.

For example, typing:

```
y <- log(x)
```

would result in the creation of a new variable called `y` that contains the value of $\log_e(2)$ (approx. 0.69).

Arguments may be variables storing data, fixed values, or parameters specifying how the function should behave. The full list of arguments that may be provided to a particular function can be found in the function's help file.

Many functions have optional arguments. For example, the help file corresponding to the `log` function indicates that the 'base' can be provided as an additional argument. When providing such arguments, the argument name can be used to clarify which arguments are supplied, e.g.:

```
y <- log(x,base=10)
```

results in `y` taking the value of $\log_{10}(2)$ (approx. 0.30).

Arguments are always specified in parentheses `()` after the function name. Where functions take multiple arguments, the arguments are separated by commas within the parentheses. Even if a function is called without any arguments (e.g. the `q()` function), the parentheses are always specified – this distinguishes between functions and variables.

Commands may be combined and nested. For example, the following is valid:

```
y <- sqrt(exp(x))-(x+1)/2
```

Note that parentheses may be used to force the order in which subexpressions are evaluated.

Task 1:

1. Create a variable `x` and assign it a numerical value.
2. Evaluate the expression: $\sin^2(x) + \cos^2(x)$, assigning the result to another variable `y` (you will need to use either the `*` or `^` operators). Is there anything interesting about the result?

4 Vectors

In R, *vectors* are a type of *data structure* used to store an ordered sequence of values. Vectors may be created using the `c` function, which can be used to combine values to form a vector. For example, the command:

```
x <- c(1,2,3)
```

results in the variable `x` being assigned as a vector containing the values 1, 2 and 3. The `c` function can take any number of arguments, which may include existing vectors. This allows the concatenation of vectors. For example, the sequence of commands:

```
x <- c(1,2,3)
y <- c(5,6)
z <- c(x,4,y)
```

results in the variable `z` containing the values 1, 2, 3, 4, 5 and 6.

Also, the colon symbol can be used to generate a vector of consecutive numbers between two values. Note that the following five expressions are equivalent – each produces a vector containing the numbers from 1 to 5:

```
c(1,2,3,4,5)
1:5
c(1:5)
c(1,2:4,5)
c(1:3,c(4,5))
```

More complicated sequences of numbers can be generated using the `seq` function, and values/vectors can be repeated/replicated using the `rep` function. Other functions of interest when working with vectors include `length`, `sum`, `sort` and `order`.

Task 1:

1. Create a vector `x` containing the values 1, 2 and 3.
2. Use `x`, along with mathematical operators, to create a new vector `y` equivalent to: `c(1,4,9)`.
3. Use `y`, along with vector concatenation, to create a vector equivalent to: `c(1,4,9,1,4,9,1,4,9)`.

Task 2:

1. Look at the help files for `seq` and `rep`.
2. Use `seq` and `rep` to create a vector equivalent to: `c(1,4,7,1,4,7,1,4,7)` in a single command, without using the `c` function.

Task 3:

1. Use `seq` and `sum` to evaluate the sum of all odd numbers between 0 and 1000.

2. Re-evaluate the sum of all odd numbers between 0 and 1000, using a vector but without using `seq` (hint – use mathematical operators).

5 Vector Indexing

Individual elements of a vector can be accessed by specifying the *index* of the element. The index is specified in square brackets after the variable name. For example, typing:

```
x <- c(7,4,10)
y <- x[2]
```

will result in the variable `y` adopting the value of the 2nd element in `x`, which is 4. Similarly, in this example `x[1]` is equal to 7 and `x[3]` is equal to 10. Note that it is not required to assign the vector to a variable before accessing elements – typing `c(7,4,10)[2]` is valid, and would return the value 4.

Importantly, note the difference between parentheses ( ) and brackets [ ] in R. Parentheses are used to specify arguments when calling functions, whilst square brackets are used to extract elements of an indexable object (e.g. a vector).

An index should always be a positive integer; specifying a negative integer would result in the inverse selection being returned, i.e. all elements except those with the specified indices would be returned.

Multiple elements can be accessed by providing a vector as an index, allowing the extraction of sub-vectors. For example, typing:

```
(10:20)[3:5]
```

would return the vector `(12,13,14)`.

Note that it is not necessary for index vectors to contain consecutive numbers – the following sequence of commands is valid:

```
x <- c(7,4,10,3,15,8,1)
y <- c(6,2:4)
z <- x[y]
```

resulting in the variable `z` being equal to the vector `c(8,4,10,3)`.

Indexing the result of an expression is allowed, providing the expression is surrounded by parentheses. For example,

```
x <- 1:3
(x*x+1)[2]
```

would return the value 5.

Note that vector arithmetic is allowed, and that vector multiplication is performed element-by-element – multiplying a vector of length n by another vector of length n results in a vector of length n. Vectors of different lengths can only be multiplied if the length of one of the vectors is a multiple of the other.

Similarly to how sub-vectors may be extracted from vectors, sub-vectors can also be assigned new values. This allows the contents of vectors to be edited. For example, the sequence of commands:

```
x <- 1:7
x[2:4] <- 10:12
x[c(5,7)] <- 3
```

results in the vector `x` containing the values 1, 10, 11, 12, 3, 6 and 3. Sub-vector assignment must be performed either using a vector of length equal to the sub-vector being replaced (e.g. `10:12` is equal in length to `x[2:4]` in the above example), or using a single value, in which case all elements of the specified sub-vector adopt that value.

Task 1:

1. Recall the vectors `x <- c(7,4,10,3,15,8,1)` and `y <- c(6,2:4)` from the example above. Create these vectors.
2. Use `x`, `y` and vector indexing to return a vector containing the numbers 7, 15 and 1.

Task 2:

1. Create a vector containing the numbers from -5 to 5.
2. Use sub-vector assignment to set all negative values in the vector (i.e. the first five elements) to 0.
3. Use sub-vector assignment to set all odd values in the vector to 0.

Task 3:

1. Create a vector containing all numbers from 0 to 1000.
2. Use sub-vector assignment to set all even numbers in this vector to 0.
3. Use this vector to evaluate the sum of all odd numbers between 0 and 1000.

6 Logical Operators and Statements

The logical values `TRUE` and `FALSE`, along with `NA` (not applicable) are special reserved objects in R. These values are returned by *logical statements.* The *logical operators* in R include:

`==`	equal
`!=`	not equal
`<`	less than
`<=`	less than or equal to
`>`	greater than
`>=`	greater than or equal to
`&`	and
`|`	or

Note the important difference between the equality operator '`==`' and the assignment operator '`=`'.

For illustration, the results of various logical statements are tabulated below:

Statement	Result
`1 == 2`	`FALSE`
`1 == 2-1`	`TRUE`
`1 != 2`	`TRUE`
`1 < 2`	`TRUE`
`2 < 2`	`FALSE`
`2 <= 2`	`TRUE`
`0/0 == 0`	`NA`

Note that invalid statements incur the result `NA` – in the above example, this is due to `0/0` being equal to the special `NaN` value ('not a number'). The same result would have arisen for `sqrt(-1)`. Such operations incur warning messages in R.

Whilst '`!=`' is a logical operator, '`!`' alone is a special symbol representing logical negation, and is used to invert the result of a logical statement. As such, `!TRUE` is equal to `FALSE`, and `!FALSE` is equal to `TRUE`. For example:

Statement	Result
`1 == 1`	`TRUE`
`!(1 == 1)`	`FALSE`
`2 < 1`	`TRUE`
`!(2 < 1)`	`FALSE`

In addition, the `&` (and) and `|` (or) operators can be used to combine logical statements, allowing for more complex selection criteria. Non-vector versions of these of these operators also exist (`&&` and `||`).

Logic can also be applied to vectors, resulting in the creation of *logical vectors.* For example, the sequence of commands:

```
x <- 1:5
x < 3
```

would return a vector containing the values `TRUE`, `TRUE`, `FALSE`, `FALSE` and `FALSE`.

Logical vectors can be used to index another vector of equal length. This results in a vector being returned that contains only the values for which the corresponding value in the logical vector is `TRUE` (or `NA`). Continuing our example, the command:

```
x[x < 3]
```

would return a vector of length 2, containing the values 1 and 2.

Complications arise when the logical vector contains `NA` values, which may be invalid due to computational reasons or correspond to missing data. For example, the sequence of commands:

```
x <- 0:4
y <- x^2/x
y < 3
```

would return a vector containing the values `NA`, `TRUE`, `TRUE`, `FALSE` and `FALSE` due to the attempted calculation of 0/0. Consequently, the command:

```
y[y < 3]
```

would return a vector of length 3, containing the values `NA`, 1 and 2. The function `is.na` can be used to identify, thus aid removal of, `NA` values. This function returns a logical vector where `NA` values are identified as `TRUE`, and all others as `FALSE`. Returning to our example, the sequence of commands:

```
z <- y[!is.na(y)]
z[z < 3]
```

would return a vector of length 2, containing the values 1 and 2, as required. Note that this could have been achieved using a single command by using the `&` operator – the command:

```
y[y < 3 & !is.na(y)]
```

yields the same result.

Task 1:

1. Create a vector `x` containing all integers from 1 to 10.
2. Use logical statements and vector indexing to extract a vector containing the numbers 1–3, 5, and 7–10 from `x`, using exactly two `|` operators (and no `&` operators).
3. Use logical statements and vector indexing to extract a vector containing the numbers 1–3, 5, and 7–10 from `x`, using exactly one `&` operator (and no `|` operators).
4. Use logical statements and vector indexing to extract a vector containing the numbers 1–3, 5, and 7–10 from `x`, using exactly one `|` operator (and no `&` operators).

Task 2:

1. Create a vector `x` containing all integers from -10 to 10.
2. Create a vector `y` such that $\texttt{y} = \sqrt{\texttt{x}}$.
3. Using vector indexing, logical statements and the `is.na` function, use `x` and `y` to generate a vector of positive integers `z` for which $\sqrt{\texttt{z}} > 2$ (note that `z` must not contain any `NA` values).

7 Matrices and Arrays

In addition to vectors, other data structures in R include *matrices* and *arrays*.

Unlike vectors, matrices are 2D objects. The number of rows and columns in a matrix can be specified upon initialisation, and retrieved, using the `nrow` and `ncol` functions. For example, a 3×4 matrix of 1's can be initialised as follows:

```
x <- matrix(1,nrow=3,ncol=4)
```

Elements of a matrix can be accessed (and reassigned) in a way similar to vectors. However, due to the increased dimensionality, elements of a matrix are indexed by separating indices by a comma. For example, the command:

```
x[2,3]
```

specifies the element that occupies the 2^{nd} row, 3^{rd} column. Similarly to as with vectors, subsets of a matrix can be specified by providing vectors of indices.

All rows/columns in a matrix can be specified by leaving the index blank (although the commas are still required). For example, `x` is equivalent to `x[,]`. The command:

```
x[2,]
```

would display the 2^{nd} row from the matrix, whereas the command:

```
x[,2]
```

would display the 2^{nd} column from the matrix.

A warning – be aware that standard matrix multiplication (e.g. `x*x`) is element-by-element, like with vectors. For true matrix multiplication, see the `crossprod` and `tcrossprod` functions, and the `%*%` operator (see also the outer product function and `%o%` operator). Other simple matrix-related functions include `t`, `diag`, `sign` and `det`.

Arrays are relatively similar to vectors and matrices, except that they allow extension to higher dimensions – arrays can be 1D, 2D, 3D, or higher. The dimensionality of an array can be set and retrieved using the `dim` function. For example, the `dim` function can be used to coerce a vector into an nD array. Arrays can also be explicitly constructed, e.g. the command:

```
array(NA,dim=c(4,5,6))
```

returns an empty $4 \times 5 \times 6$ array.

Similarly to matrices, elements of an array are indexed by separating indices by commas. For example, if `x` is a 3D array, then:

```
x[2,3,4]
```

specifies the element with index [2,3,4]. As with vectors and matrices, subsets of an array can be specified by providing vectors of indices.

Task 1:

1. Create a matrix `x` of zeros with 5 rows and 8 columns.
2. Using a single command, for all elements that lie on any of rows 2, 3 and 4 *and* any of columns 3, 5 and 7, set their values to 1.
3. Using the `crossprod` function, generate the 8×8 matrix $\mathtt{x}^T\mathtt{x}$, and then calculate the sum of the elements of that matrix using the `sum` function.

Task 2:

1. Create a vector `x` containing the numbers from 1 to 24.
2. Use the `dim` function to coerce `x` into a 4×6 array.
3. Use the `dim` function to coerce `x` into a $4 \times 3 \times 2$ array.
4. Use the `dim` function to coerce `x` into a $2 \times 2 \times 3 \times 2$ array.
5. Use the `dim` function to coerce `x` into a 1D array.
6. Use the `as.vector` function to coerce `x` back into a vector.

8 Types, Objects, Classes and Character Strings

So far in this tutorial we have only considered *objects* with numeric and logical *data types*. However, R can deal with various primitive data types, most notably: numeric, complex, logical, character and raw. These primitive (or 'atomic') data types are referred to as *modes*. The `mode` function can be used to extract an object's mode. Here, an object is an entity, such as a variable or function (the set of objects in the current environment can be extracted using the `object` function).

In addition to a mode, all objects have a *class*, which is important for determining how other objects interact with it. This is powerful, as it allows context-dependent behaviour. An object's class can be extracted using the `class` function.

The storage mode and mode of an object can be set using the `storage.mode` and `mode` functions. All values in a vector, matrix or array must always be of the same mode (e.g. numeric), and stored using the same storage mode (e.g. integer, double). For example, the vector returned by the command:

```
1:5
```

has mode ‘numeric’ and class ‘integer’. However, the vector returned by the command:

```
1:5 + 0.5
```

has mode ‘numeric’ and class ‘numeric’. Note that these two vectors have the same mode, but different class.

It is sometimes necessary for objects of one type to be coerced to another type. This can be achieved using the `as.TYPE_NAME` functions, where `TYPE_NAME` is a data type (e.g. numeric, integer, character, vector, etc.). For example, the command:

```
as.integer(1:5 + 0.5)
```

returns a vector that has mode ‘numeric’ and class ‘integer’. In this case, the numbers are trimmed down to integers, and the returned vector contains the integers from 1 to 5.

When attempting to coerce an object from one type to another, `NA` values will be returned if it is not possible to sensibly convert the values.

One situation where this is particularly relevant is when a vector of numbers is actually of mode ‘character’, rendering it impossible to perform any numerical calculations with the object. In this case, the `as.numeric` or `as.integer` functions can be used to coerce the object into a suitable type.

Text, or character strings, can be assigned to variables using quotes. For example, the command:

```
x <- "some text"
```

results in `x` being of mode ‘character’. The number of characters in a character string is returned by the `nchar` function.

Vectors of character strings can be constructed similarly to vectors of numbers. For example,

```
x <- c("abc","123","xyz")
```

constructs a vector of character strings. Elements of this vector can be extracted and reassigned in the same way as numeric vectors. Note that the 2nd element of `x` has mode ‘character’, despite comprising only numeric characters, and that the command:

```
as.numeric(c("abc","123","xyz"))
```

would return a vector of length 3 and mode ‘numeric’ containing the values `NA`, 123 and `NA`.

Whilst all values in vectors, matrices and arrays must have the same primitive data type (mode), there are other data storage types that allow storage of data with different types, namely *lists* and *data frames*.

Task 1:

1. Look at the R help page for the function `paste`, which is useful for concatenating character strings.
2. Use paste to concatenate the individual characters `"a"` and `"b"`, separated by an underscore (`"_"`).
3. Create the vector of character strings: `x <- c("a","b","c")`, and then use `paste` to concatenate the elements of `"x"` in order to achieve the single character string `"abc"`.
4. Create the numeric vector: `x <- 1:9`, and then use `paste` to concatenate the elements of `"x"`. Coerce the result to an object of 'integer' type, resulting in the number 123456789.
5. Repeat the previous step, this time using the numeric vector: `x <- 1:10`. Again, use `paste` to concatenate the elements of `"x"` and coerce the result to an 'integer'. What happens this time? What happens if you instead coerce it to type 'numeric'?

9 Reading and Writing Files, and Data Frames

The ability to read and write files is paramount, enabling data analysis to be performed on real datasets. There are a few ways to read data files, depending on the file contents. Here, we will focus on the `read.table` function, which allows table-like data to be imported from a simple plain text file. The `read.table` help file provides details of other similar functions (including `scan`). The command:

```
x <- read.table("FILE_PATH",header=FALSE)
```

may be used to read a table from a plain text file, storing the results in the variable `x`. Here, `FILE_PATH` specifies the name and location of the data file, and the argument 'header' is used to specify whether or not the table includes column titles. Other arguments are listed in the help file.

The result of the `read.table` function is of a type called a *data frame*. Data frame objects are similar to matrices in that they are 2D objects that have a defined number of rows and columns. However, unlike a matrix, different columns in a data frame can have different atomic data types (modes) – it's possible to have one column that contains alphanumeric data, another column that contains integer numbers, and another column that contains floating point numbers.

If a header is specified (i.e. the columns have titles), then columns can be accessed by specifying the name of the column after a $ symbol. For example,

```
x$COLUMN_NAME
```

would return the contents of the column called `COLUMN_NAME`. Otherwise, if a header is not specified, the column titles will be automatically assigned the names `V1`, `V2`, etc.

In addition to the '$' convention, rows, columns and elements of the data frame can be accessed in the same way as matrices (e.g. `x[2,3]` specifies the element that

occupies the 2nd row, 3rd column).
Similarly to `read.table`, the function `write.table` will write an existing data frame to file. Further details can be found in the R help file.

Task 1:

1. There are a number of datasets distributed as part of the R package. Type `data(CO2)` to load the 'CO2' dataset, which corresponds to data from an experiment on a species of grass' tolerance to cold. For more info, type `?CO2`. The object 'CO2' will have been added to your current session.
2. Now type `CO2`. You will see that the 'CO2' object corresponds to a table with five columns – 'Plant', 'Type', 'Treatment', 'conc' and 'uptake'. Note that the different columns have different data types.
3. Type `CO2$uptake` to become experienced with using the '$' method of accessing data frame columns.
4. Use your knowledge of vector indexing and logic statements to get the vector of 'uptake' values such that the 'Type' is equal to 'Quebec' and the 'Treatment' is equal to 'chilled'.
5. Use the `summary` command to extract information regarding the distribution of this selection of 'uptake' values.

Task 2:

1. If you have your own dataset available as a text file (or can find one online to experiment with), try using the `read.table` function to load the dataset into R.

10 Plotting and Graphical Facilities

Data representation and visualisation is a key aspect of R. Whilst there are many different types of graphical output available, this tutorial will only cover the basics.

The simplest type of plotting can be achieved using the `plot` function, which is used to display one variable plotted against another. In its simplest form, the `plot` function can be used by executing the command:

```
plot(x,y)
```

where `x` and `y` are equal-lengthed 1D objects containing the data to be plotted. This command is equivalent to the command:

```
plot(y~x)
```

which reads as "plot `y` against `x`".

Further arguments are available in the help files for `plot` and `par`. The function `par` is used to set or query graphical parameters, allowing a high degree of customisation over the presentation of graphs. Any changes applied using `par` are semi-permanent, being applied to all subsequent graphical output for the rest of the current session.

Once a graph has been created, it is possible to overplot onto the existing graph. For example, this can be used to add additional datasets onto an existing plot, e.g. using the `points` or `lines` functions.

When preparing graphics, it is often useful to initialise and configure a graphics device driver. The particular device depends on the operating system – often `quartz` is used on OS X, and `x11` is used on linux systems. See the R help pages for details. For example, the device dimensions can be set using the command:

```
quartz(width=8,height=4)
```

It is then possible to specify for two figures to be drawn side-by-side in the plotting area by using the command:

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

which specifies for a 1×2 array of figures to be drawn in the plotting area instead of the default (1×1). Also, the command:

```
par(cex=0.6)
```

can be used to reduce all subsequent plotting text and symbols to 60% default size. A wealth of other options are available using the `par` command.

In addition to `plot`, other useful plotting tools include: `hist`, `qqplot`, `boxplot`, `image` and `heatmap`, amongst many others.

Task 1:

1. There are a number of datasets distributed as part of the R package. Type `data(Puromycin)` to load the 'Puromycin' dataset. This corresponds to the reaction velocity versus substrate concentration in an enzymatic reaction involving untreated cells or cells treated with Puromycin.

2. Now type `Puromycin`. You will see that the 'Puromycin' object corresponds to a table with three columns – 'conc', 'rate' and 'state'.

3. Enter the command: `plot(Puromycin$rate~Puromycin$conc)` in order to plot the reaction rate against the substrate concentration. Now add a title for the plot, as well as titles for the x and y axes – look in the R help file for `plot` in order to learn how to do this.

4. At present, the dots plotted on the graph are hollow. Adjust the plotting command so that the points are solid circles, rather than hollow circles. To learn how to do this, investigate the 'pch' graphical parameter, as explained in the R help file for `points`.

5. It would be nice to be able to visually distinguish between the two 'states' ('treated' and 'untreated'). Overplot red solid circles corresponding to the 'treated' state onto the current plot. This can be done using the `points` function to overplot onto the existing figure, using vector indexing and logical

statements to filter the data corresponding to the 'treated' state, and using the 'col' parameter described in the R help file for `par` to colour data points.

Task 2:

1. Create a vector containing synthetically-generated Normally-distributed data using the command: `x <- rnorm(10000)`.
2. It is often useful to plot histograms to aid visualisation of data distributions. Create a histogram for the synthetic Normal data using the `hist` function.
3. The 'breaks' argument changes the number of bars in a histogram. Try changing the number of breaks to 100. Also, try colouring the histogram using the 'col' argument, as described in the R help file for `par`.
4. Redraw the histogram with density on the y-axis instead of frequency. This can be achieved using the 'freq' argument, as described in the R help file for `hist`.
5. Add a line to the histogram using the command: `lines(density(x),lwd="2")`. Note that the parameter 'lwd' changes the thickness of the line.