limma: A brief introduction to R

Natalie P. Thorne

September 5, 2006

R basics

ℹ R is a command line driven environment. This means you have to type in commands (line-by-line) for it to compute or calculate something. In this practical we want you to get used to this style of programming. For the analysis of microarray data, we hope you won't need much knowledge of R programming. However a basic understanding will be very beneficial. In the following exercises you will discover how to enter data into R and then make calculations involving this data.

Exercise 1: Create an object called `x` and let it be equal to `2`. Create another object called `y` and let it be equal to `3`. Now perform basic arithmatic operations on `x` and `y`.

☞ Notice how we use the `=` symbol to assign a value to a name.

ℹ Some `R` commands give output and some don't. For example typing `x=2` gives no output because you've only asked `R` to assign a *value* to a *name* (there is no result to see, no output). Whereas typing the name of an object such as `x` asks `R` to show the values of that object and so the output in this case is a display of the contents of the object. Whenever `R` displays the contents or values of an object, it will try to format the output in an intuitive manner. So the `[1]` that you see when you type `x` (below) is part of `R`'s formatting, it indicates that the first element of the output begins there.

```
> x = 2
> x
```

☞ We use brackets to specify the order of execution for a series of operations. We create a third object, `z` which depends on `x` and `y`. You can send two commands to the same command line if you separated the commands by a semi colon ";" symbol to indicate that they should be computed separately. The symbol `>`, indicating the beginning of a command line, is called the prompt.

```
> x = 2
> y = 3
> x + y
> y - x
> x * y
> x^2 + y
> z = 3 * (x + y) + 1
> z
```

♠ *If you type something incorrectly in* `R`*, you will get an error. Type* `z=2*(x+y+1` *and you will see that* `R` *detects that you haven't finished the command, it displays a new line with a* `+` *symbol.* `R` *is* waiting *for* additional *code and in particular here, it is waiting for the final bracket from the previous command. Finish the command by adding the closing bracket. Now type* `x=2+2 x` *to get an idea of the kind of error messages* `R` *gives. In this latter example,* `x=2+2;x` *is the correct* syntax.

You can type the following command

```
> objects()
```

at any time during your `R` session to see the names of all the objects you've created. To see the contents of any object type the name of the object at the prompt or use `show`.

```
> show(y)
> x
```

ℹ With the cursor at the prompt, use the up and down arrows to scroll through previous commands - this is particularly useful if you type something slightly incorrectly the first time and want to bring back the command again to change something minor and re-enter the command.

ℹ If at any stage you find that `R` is not responding, it has *hung* or you want to stop a command midway through its execution, go to the `Stop` icon at the top of your `R GUI` interface. In some cicumstances `ESC` is also useful. `ctrl-C` works for those using a `unix` command line version of `R` or the `emacs` environment.

R functions

ℹ R is a program designed for statistical computing and graphical displays of data. There are hundreds of built in functions available at your disposal. Some functions are available as standard, and others are packaged into libraries which you need to install separately. For microarray analysis in this practical lab we will be focussing on the library called `limma`. Lets do a basic exercise to understand what a function is.

Exercise 2: Create a vector called `v` containing a random sample of size 10 from a uniform distribution. Find the average of this sample using the function `mean`.

☞ The function `runif` generates **r**andom values from a **unif**orm distribution. The function requires certain bits of information for it to operate. The bits of information required are specified by the `arguments` of the function. Functions can have default settings for some or all of their arguments. `runif` requires `n` (the sample size) to be specified but has default settings for its other arguments `min` and `max` (defining the minimum and maximum values to randomly sample between). Notice that the vector `v` has 10 elements in it. These 10 elements correspond to each of the ten random samples of real values between the default limits of 0 and 1. The average of such a random sample, which we calculated using the `mean` function, should tend towards to 0.5.

```
> v = runif(10)
> v
> mean(v)
```

```
> x = mean(v)
> x
```

Exercise 3: Now use the functions `help` and `args` to see how the `runif` and `mean` functions work. Make sure you understand how to use the various arguments of the functions `runif` and `mean`. Try increasing and decreasing the sample size of your random sample and try sampling from a different range of values (i.e. instead of using the default values of `min` and `max`, set your own values).

☞ Notice that there is only one argument listed when typing `args(mean)`. `R` displays `x` and then three dots to indicate that there are further arguments (other than `x`) that can be secified to this function. Use the help *man page* on (mean) to find out what these further arguments are. What does the `trim` argument do? How does `R` understand when there is missing data? (Hint: what is an `NA` value?)

☞ Consult the first three sections of the `limma` manual for more discussion on beginning with `R` for `limma`, especially the various ways to get help on a particular function (described in section three).

☞ Notice that the resulting output from the useage of the `args` function below returns `NULL`. This is because there this no numerical result from these commands, hence null output.

♠ *The order of arguments does not matter if you assign values to the names of arguments in a function. See the example below where the arguments of* ***runif*** *have been given in different ways. When giving values as arguments to a function, the* order *only matters when you have* not *specifically assigned the values to the names of the various possible arguments.*

```
> args(runif)
> args(mean)
> help(mean)
> v = runif(n = 20, min = 0, max = 2)
> v = runif(max = 2, min = 0, n = 20)
> v = runif(20, 0, 2)
> mean(v)
> mean(v, trim = 0.1)
```

R object types

🛈 There are three main types of `R` objects (`R` recognisable formats for storing data) that we will be discussing in this section - `vector`, `matrix` and `list` objects. There are other formats or types of `R` objects, but these are beyond the scope of beginner level `R` programming. A `vector` is a one dimensional object and a `matrix` is a two dimensional object. A `list` is well.....a *list* of objects....it can contain multiple different objects. The objects contained in a `list` can be either vectors or matrices. Vectors and matrix objects are fairly intuitive. An `R` list is less intuitive, but lists are useful for putting together multiple objects (that may be related to each other in some way, but not necessarily of the same type or size) into one encompasing object.

Exercise 4 (vectors)**:** Create a vector `v` containing the numbers 2, 4, 3, 1, 7. Confirm the object is of type *vector* and find the length of `v`. Use the subsetting operator in `R` to access the 2nd element in this vector. Practise subsetting different elements of the vector. Reverse the order of the elements in `v`.

☞ A vector is a one dimensional object, containing elements that are *concatenated* (hence the `"c"` in the notation `c(...,...,...)`). Subsetting a vector requires using the `[ ]` operators.

☞ The elements in a vector (the values it contains) may be numeric or character values. Notice at the end of this exercise that we put characters in a vector. Now functions like `length` still work, but `mean` will not (since it requires numeric values to compute a mean). We use double quotes to specify a character rather than the name of an object for `R` to interpret.

```
> v = c(2, 4, 3, 1, 7)
> v
> is.vector(v)
> length(v)
> x = 2
> is.vector(x)
> length(x)
> v[2]
> v[c(5, 4, 1)]
> rev(v)
> v = c(a, b, c)
> v = c("a", "b", "c")
> length(v)
> mean(v)
```

Exercise 5 (matrices)**:** Create a matrix with three rows and four columns with elements ranging between 1 and 12. Practise subsetting elements of the matrix (i.e. get the element in row 2 and column 4; try getting only the elements from column 1 and then only the elements from row 3).

☞ Notice how we used the matrix function to arrange the elements of a vector into rows and columns. The function `dim` can be used to get the number of rows and columns (dimensions) of any matrix - `dim` will not accept a vector as a valid argument. Subsetting a matrix requires providing both row and column subset indexes.

```
> m = matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12), nrow = 3, ncol = 4)
> is.matrix(m)
> dim(m)
> length(m)
> m
> m[2, 4]
> m[, 1]
> m[3, ]
> m2 = matrix(1:12, 3, 4)
> is.matrix(m2)
> dim(m2)
> m2
> v = 1:12
> dim(v)
```

```
> is.matrix(v)
> is.vector(v)
> length(v)
```

Exercise 6 (list)**:** As above in the previous exercises, create a matrix with three rows and four columns with elements ranging between 1 and 12. Create a vector containing the same elements. Now make a list of the objects, put them in a list arrangement. Try extracting various objects from the list separately. Then try to get an element from the matrix in this list by subsetting a single row and column.

☞ Notice the different ways to extract the objects from within a list. Subsetting a list using the square brackets operators gives a subset of the list but doesn't extract a given object. You must use double (not single) square brackets for extracting an object from a list. The objects in the list are arranged in one dimension (similar to elements in a vector, only now we call it a list) and this is why `is.vector(mylist)` returns `TRUE`. So if you subset a list in the usual way (using single square brackets), you simply get a new list (containing the subset of objects). If the objects in the list are named, then you can extract the list objects using the `$` operator.

```
> myvect = c(1:12)
> mymat = matrix(1:12, 3, 4)
> mylist = list(myvect, mymat)
> is.vector(mylist)
> is.matrix(mylist)
> is.list(mylist)
> mylist
```

♠ *If you use single square brackets to* subset *a list, you can be tricked into thinking this is the correct way to* extract *individual objects from a list in* `R`*. Notice below, that* `mylist[1]` *gives output displaying* `[[1]]` *on the first line indicating that you've now got a new list containing only one object (i.e. a subset of the original list, with one object only, but still recognised by* `R` *as an object of type* `list`*).*

🛈 It is important to distinguish between *extracting* an object from a list and *subsetting* a list. Remember, single brackets are for subsetting, double brackets are for extracting an object from a list. If the objects in the list are named, then one can use the `$` operator to extract them.

```
> a = mylist[[1]]
> mean(a)
> b = mylist[1]
> mean(b)
> is.list(b)
> mean(b[[1]])
> is.list(a)
> is.vector(a)
```

```
> (mylist[[2]])[1, 4]
> namedlist = list(V = myvect, M = mymat)
> namedlist$V
> namedlist$M
> namedlist$M[1, 4]
```

ℹ The `list` object is used extensively in `limma`. For example in order to store the foreground and background information for two colours, multiple arrays and thousands of genes; a list of matrices are used (each matrix contains information on every array (columns) and every gene (rows)).

R graphics basics

ℹ Probably, the most important graphing function in `R` is `plot`. Most of the `limma` plotting functions are based on this function. In this section we will look briefly at the `plot` function paying particular attention to understanding some of the key plotting parameters that can be altered to control the plot display features.

Exercise 7 (plot)**:** Reproduce the code below to learn how to control plot features such as the axis labels and plotting character size, shape and colour. Use the help on `par` to help clarify your understanding of making plots in `R`.

☞ The basic format for a plot is based on specifying a vector of x values and a vector of y values.

```
> plot(x = 1:20, y = 1:20)
```

☞ Below, we plot twenty random observations between 0 and 1.

```
> plot(x = 1:20, y = runif(20))
```

☞ The code below produces the same plot as above.

```
> plot(runif(20))
```

☞ Below, we create a matrix and plot the first column against the second.

```
> m = matrix(1:12, 3, 4)
> plot(m[, 1], m[, 2])
```

☞ In the next plot, we introduce various plotting parameters including `mfrow` for putting multiple figures into a plot, `pch` for specifying the plotting character, `cex` for specifying the character size used in the plot and `col` for specifying the colour of the plotting characters.

♠ *You might be beginning to feel like avoiding typing all these lines of code (wanting to copy and paste commands instead of typing by hand). Indeed, learning* `R` *initially takes lots of concentration. You may be finding that it takes a long time to type these more complex lines of code by hand. Also, with lots of code on a single line you are more likely to make errors and it certainly becomes harder to* debug *or locate errors in your commands. A misplaced or missed comma, a forgotten or extra bracket or an accidental full stop will produce a* syntax *error. However, resisting the temptation to simply copy and paste will bring it's rewards. It is at this stage, by typing all commands out fully, that you will actually grasp the basic logic behind* `R` *programming. Putting in the extra effort early on will benefit you greatly later. In some of the more advanced practicals we will shift our focus from the details of* `R/limma` *programming to learning more conceptual ideas about the analysis of microarray data. By then you should be practised enough in typing commands efficiently or have enough knowledge of the code that simply copying and pasting commands from electronic lab notes will not significantly detriment your learning.*

```
> par(mfrow = c(2, 2))
> plot(1:20, pch = 16)
> plot(1:20, pch = 1:20)
> plot(1:20, pch = c(14, 16), cex = 2, col = "blue")
> plot(1:20, pch = c(1, 1, 1, 1, 15, 15, 15, 15, 17, 17, 17, 17, 19, 19, 19, 19))
```

ℹ `mfrow` stands for **m**ultiple **f**igures arranged by **row**. Therefore `mfrow=c(2,2)` means that you set up the graphics device so as to arrange your figures in two rows and two columns, filling one row at a time. `mfcol` works in the same way as `mfrow` except the multiple figures are filled one column at a time.

☞ In the plots below, we introduce a new plotting capability, adding points to the current plot using `points`. `points` works in much the same way as `plot` but instead of making a new plot device, it simply adds points to the current graphics device. You can also add text to the current plot using `text`

```
> par(mfcol = c(2, 3))
> plot(1:20, pch = c(rep(1, 4), rep(15, 4), rep(17, 4), rep(19, 4)))
> plot(1:20, pch = 16, cex = 2, col = c("blue", "red", "green", "yellow"))
> plot(1:20, pch = c(1, 1, 1, 1, 15, 15, 15, 15, 17, 17, 17, 17, 19, 19, 19, 19))
> plot(1:20, main = "myplot", xlab = "my x label", ylab = "my y label")
> text(7, 15, "text here")
> text(10, 5, "and here")
> plot(1:10, 1:10)
> points(1:10, runif(10, 1, 10), col = "magenta", pch = 18, cex = 2)
> plot(1:10, 1:10, pch = ".")
> text(1:10, 1:10, labels = c("a", "b", "c", "d", "e", "f", "g", "h", "i", "j"))
```