limma: Basics

Natalie P. Thorne

January 7, 2009

Overview

ℹ `limma` is a package for the analysis of gene expression microarray data, especially the use of linear models for analysing designed experiments and the assessment of differential expression. In this lab we will focus on the `limma` basics - reading in microarray image output files and associated information. Below is a brief overview of typical steps involved in getting started with a `limma` analysis of two colour microarray data sets.

♠ ***Do not type*** *in the following commands, they are there as a reference for the typical sequence of functions used to read data into limma. A typical preliminary analysis would begin by loading the* `limma` *library, reading in information about the target samples, reading in the image output files, identifying different spot types in the genelist, performing background correction and normalisation then making various plots.*

```
> library(limma)
> targets = readTargets()
> RG = read.maimages(targets$FileName)
> RG$targets = targets
> RG$printer = getLayout(RG$genes)
> types = readSpotTypes()
> RG$genes$Status = controlStatus(types, RG)
> RG = backgroundCorrect(RG)
> MA = normalizeWithinArrays(RG)
> plotMA(MA)
```

ℹ `limma` is designed for analysing two-colour microarray data. However there are functions available for analysing Affymetrix and other single-colour array data. We will be concentrating on two-colour array data. For the analysis of two-colour microarray data, usually one image analysis output file is obtained for each array. Such files contain the various calculations on the foreground and background intensity measurements for each spot as well as other information like the Block, Row, Column, ID and probe names for each spot. In most cases it is also desirable to have a *targets* file which describes which RNA sample was hybridised to each channel of each array. A further optional file is the *spot types* file.

Targets file

☞ Make sure the current directory for your `R` session is in the `limmaBasics` folder. All your analysis will happen in this directory. At any point during this session you may save the `workspace` so that you can come back to the same session later.

♠ *You can set the current directory by typing* `setwd("/path/to/limmaBasics")`*. Alternatively, in a GUI implementation (the point-and-click kind) use the option in the drop down menu to change to a different working directory.*

Exercise 1: Load the `limma` library and read in the targets file in the current directory. The targets file has been given the default name `Targets.txt`.

☞ Notice the use of the various arguments to `readTargets`; `path`, `sep` and `row.names`. Look at the help for this function to understand the default settings for these arguments.

```
> library(limma)
> targets = readTargets()
> targets
> targets = readTargets("Targets.txt")
> targets
```

☞ Although reading in the targets file can seem fairly easy, getting the targets file right can be difficult.

♠ *Hint: for finding errors in a targets file, make sure there are no additional tabs (between columns or at the end of each row) or extra carriage returns at the end of the targets file. The best way to do this is to open the targets file in a text editor. Highlighting the text can be a useful way to find misplaced tabs at the end of rows or any extra carriage returns. Each row should contain the same number of columns, and should end with one carriage return (including the last row of the file). Unusual characters such as the* `#` *symbol can cause problems for R when reading in files. Note that opening the files in Excel will not help you to find hidden extra tabs or carriage returns.*

☞ In many exercises in this and other labs you will be required to read in files from different directories. It is important to **remain in the current directory** during this `R` session (i.e. `limmaBasics` not `swirl`; don't change directory during this lab). To read in files in other directories, specify the *relative or full* directory path in the file name or set the `path` argument.

Reading in Data

🛈 `limma` allows you to easily read in microarray data from multiple arrays within an experiment. It uses a function called `read.maimages` to read in image output files one at a time. It extracts the columns of interest from each file, sequentially building up an `RGList` object containing the necessary information from each array.

ℹ limma supports reading in microarray data files from various image analysis programs such as Agilent, Array Vision, BlueFuse, ImaGene, GenePix, QuantArray, ScanArray Express, SMD and SPOT.

Exercise 2: Read the swirl data into R using the read.maimages function. Look up the help on this function and make sure you are familiar with the required arguments. This function may take a while to run, so be patient. Again, don't change directories, instead specify the path argument.

☞ The swirl data set consists of two sets of dye-swap experiments making a total of four replicate hybridizations. Each of the arrays compares RNA from swirl fish with RNA from normal ("wild type") fish.

☞ RG is a list object. It contains named objects, R, G, Rb and Gb, matrices for the red and green foreground and background spot intensities respectively. Rows in these matrices correspond to genes, and columns correspond to arrays.

☞ The swirl arrays were image analysed using Spot. This is the default for the source argument in read.maimages. For completness in the exercise below we specify the source argument anyway. What objects are read in by read.maimages from SPOT output files? What are the array names (i.e. column names of data matrices in RG)?

```
> RG = read.maimages(targets$FileName, path = "swirl", source = "spot")
> show(RG)
> names(RG)
> colnames(RG$R)
```

ℹ The list returned by read.maimages is a special kind of list. It is a kind of list object that is recognised in a special way by the limma library functions. It is called an RGList not just a list. There are also MAList objects in limma too. Many of limma's functions expect an RGList or MAList object. It makes writing functions easier, because you can give one of these objects as an argument to a limma function and it will automatically know what and where the information is stored in these objects. In this way, fewer arguments need to be specified in a function because the limma function knows exactly what to extract from the list. The function plotMA is a good example of a limma function which recognises the RGList or MAList object types. You will become very familiar with this function later on.

☞ In plotMA you don't need to specify the x and y axis data, you can give it an RGList object, it will calculate M and A values and by default plot M versus A for the first array. There are more fancy capabilities to plotMA that you will see later.

```
> is.list(RG)
> is(RG, "RGList")
> class(RG)
> plotMA(RG)
```

Exercise 3: Practise subsetting the RGList object which we named RG. Make sure you understand what sets of data you are getting from each command below. Calculate the signal to noise ratio for the red channel for the second array.

```
> RG$R[1:5, ]
> RG$Gb[1001, ]
> s2n = RG$R[, 2]/RG$Rb[, 2]
> plot(s2n)
> boxplot(s2n)
```

Saving and recovering the workspace

Exercise 4: Save the current `workspace` and `Rhistory` and close down `R`. Now restart `R` in the same directory and load this workspace and command history again. Use `objects()` to see that you have recovered all the analysis you performed.

☞ In the GUI version of `R` you can use the pull down menus to save the current workspace and then load it again. It will save to the current directory - so be careful to note what directory you are in (should be `limmaBasics`). After you restart `R`, make sure you are in the correct directory before loading the workspace. The workspace gets saved as a hidden file. By default this hidden file is called `.RData` - in Windows this will appear as a large semi-transparent blue `R` icon; in Mac's you won't see any icon when viewing files in a directory - but it will be there! Alternatively, you can name the workspace, for example `myRanalysis.RData`.

☞ To save the history of your commands, use the drop down menu item `save history`. This history will be saved to the current directory also, with a default name that `R` will recognise and restore when the workspace is started again in the same directory.

```
> save.image()
> save.image("myRanalysis.RData")
> savehistory()
> savehistory("myHistory.Rhistory")
> q()
> load("myRanalysis.RData")
> loadhistory("myHistory.Rhistory")
> objects()
```

Exercise 5: Make sure you can access the objects you had created before you quit `R`. Can you see the recent history of commands from the `R` session before you quit? Check that the current working directory is again `limmaBasics`.

♠ *Each time you restart R you will also have to reload the* `limma` *library.*

Gene annotation

🛈 For most image analysis programs, the basic gene annotation information is contained within each image output file. Some formats do not contain the gene annotation information and some formats have column headings for the annotation information that are different from the default settings in `limma`.

Exercise 6: Get the gene annotation information for the swirl `Spot` arrays and store it in the `genes` component of `RG`.

♠ *Spot image analysis program is unusual because it does not contain gene annotation information in the output files. Therefore, for Spot files, one must read in the gene annotation information from a separate file called the GAL file.*

```
> library(limma)
> RG$genes = readGAL(path = "swirl")
> names(RG)
> dim(RG$genes)
> colnames(RG$genes)
> RG$genes[1:10, ]
```

Exercise 7: Get the print grid layout for the arrays in this data set. Confirm for yourself that the printer layout obtained is correct and consistent with the number of genes (rows of `RG`). Take a look at the objects stored in `RG` now. Make a new `RGList` object for the first two arrays only.

☞ The layout information which gets stored in the `printer` component will be used later. For example, the layout information is required for making spatial plots and for pin group (block or grid) normalisation.

☞ You might notice that, unlike ordinary `list` objects the `RGList` and `MAList` class of objects in `limma` have a special subset ability. For example, `dim` works on an `RGList` but not on ordinary lists in `R`. Below we subset the first two arrays only and create a new `RGList` object called `RG12`.

```
> RG$printer = getLayout(RG$genes)
> RG$printer
> is(RG, "RGList")
> class(RG)
> names(RG)
> dim(RG)
> RG12 = RG[, 1:2]
> is(RG12, "RGList")
> class(RG12)
> show(RG12)
```

♠ *Note that `read.maimages` might not work correctly for generic output files if you are working with an R console on a Mac.*

Spot types file

🛈 Spot types files contain information about the various types of spots that are on your array. Like the targets file, this is another tab-delimited text file. It allows you to identify different types of spots from your genelist, control their plotting character, colour and other plotting parameters in `plotMA`.

Exercise 8: Read in the spot types file that has been created for the swirl data. Then use the function `controlStatus` to set the control status for each spot on the array.

☞ In the spot types file for swirl, you are asking it to find two types of spots. The first (and most general) is a spot which has been defined as SpotType `gene`. The SpotType name is the user defined name. The `ID` and `Name` columns are determined by the entries in the `genes` matrix. The second type of spot defined in this SpotType file is the `control` type. These are also a type called a `gene`, but are more specific than a `gene` (which can be any spot on your genelist). SpotTypes called `control` have "control" as their `ID` for these arrays.

```
> types = readSpotTypes(path = "swirl")
> types
> Status = controlStatus(types, RG)
> plotMA(RG, status = Status)
```

Exercise 9: Read in the second SpotTypes file in the `swirl` directory called `SpotTypes2.txt`. Make a new control status and make a new `plotMA`. Then try changing entries in the file yourself to adjust the appearance of the control spots in the $MA-$plot.

```
> types2 = readSpotTypes("SpotTypes2.txt", path = "swirl")
> Status2 = controlStatus(types2, RG)
> plotMA(RG, status = Status2)
```

🛈 You may wish to read in extra columns of data about each spot for each array for the purposes of quality assessment. Subsequent investigation may reveal this information to be useful in the analysis or interpretation of your data or in particular for assessing the quality of each spot on the array (perhaps leading to creating spot quality weights). In `limma` you can create weights for each spot on each array, depending on the quality or informativeness of each spot. The weights are used to make sure that poorer quality spots don't contribute as much to statistical calculations as good quality, informative spots. Weights should be values between 0 and 1. The default weights for each spot are 1, meaning that all spots are considered equally in any statistical analyses. A simple weights vector for an array might consist of binary values (0's or 1's) for *bad* and *good* spots respectively. This is effectively the same as filtering out spots. Whereas a more sophisticated vector of weights for an array might have a range of values between 0 and 1, meaning that the degree of quality for different spots varies.

♠ *Note that spot weights are relative, i.e. if all spots have weight of 0.3 then, because their relative weights are equal, none are downweighted. To downweight whole arrays see* `arrayWeights`.

🛈 `limma` has three default methods for calculating spot weights. One function `wtarea` is designed for `SPOT` data, `wtflags` is designed for `GenePix` data and `wtIgnore.Filter` is designed for QuantArray data.

Diagnostic plots

Exercise 10: Make MA-plots of the four swirl arrays with and without the control genes highlighted. Try plotting with and without the legend. Use the `for` function to loop through each array number, consecutively making an MA-plot for each array. Also try using `plotMA3by2`; look up the help on this function to find out what it does.

```
> plotMA(RG)
> par(mfrow = c(2, 2))
> plotMA(RG, array = 1)
> plotMA(RG, array = 2)
> plotMA(RG, array = 3)
> plotMA(RG, array = 4)
> par(mfrow = c(2, 2))
> for (i in 1:4) {
+     plotMA(RG, array = i)
+ }
> par(mfrow = c(2, 2))
> for (i in 1:4) {
+     plotMA(RG, array = i, status = Status)
+ }
> par(mfrow = c(2, 2))
> for (i in 1:4) {
+     plotMA(RG, array = i, status = Status, legend = FALSE)
+ }
> plotMA3by2(RG)
```

Exercise 11: Make spatial plots of the red and green background for the first array. Use the `low` and `high` arguments in `imageplot` to adjust the colour range of the spatial plots. Then make multiple figure plots for all arrays. Use the `ask` argument in the `par` function to control the nature of plotting over many pages. Also try using `imageplot3by2`; look up the help on this function to see what it does.

☞ When you use the argument `ask=TRUE` in the `par` function, it is necessary to click on the graphics window or to press ENTER in order to see the next set of plots.

```
> par(mfrow = c(1, 1))
> imageplot(RG$Rb[, 1], layout = RG$printer)
> imageplot(RG$Rb[, 1], layout = RG$printer, low = "white", high = "red")
> imageplot(RG$Rb[, 1], layout = RG$printer, low = "red", high = "darkred")
> par(mfrow = c(1, 2))
> imageplot(RG$Rb[, 1], layout = RG$printer, low = "white", high = "red")
> imageplot(RG$Gb[, 1], layout = RG$printer, low = "white", high = "green")
> par(mfrow = c(2, 2))
> for (i in 1:4) {
+     imageplot(RG$Rb[, i], layout = RG$printer, low = "white", high = "red")
```

```
+     imageplot(RG$Gb[, i], layout = RG$printer, low = "white", high = "green")
+ }
> par(mfrow = c(2, 2), ask = TRUE)
> for (i in 1:4) {
+     imageplot(RG$Rb[, i], layout = RG$printer, low = "white", high = "red")
+     imageplot(RG$Gb[, i], layout = RG$printer, low = "white", high = "green")
+ }
> par(ask = FALSE)
> imageplot3by2(RG, "Gb")
```

Exercise 12: Look at the distribution of foreground log-intensities in the red and green channels for each array using `plotDensities`. Consider whether the shapes of the densities are similar between red and green channels and between arrays.

```
> par(mfrow = c(1, 1))
> plotDensities(RG)
> plotDensities(RG, log = TRUE)
> plotDensities(RG, log = FALSE)
> plotDensities(RG, groups = c(1, 2, 3, 4), col = c("cyan", "orange", "magenta",
+     "blue"))
```

Exercise 13: Make `boxplot` summaries of the red and green foreground and background log-intensities for each slide. Notice whether the arrays with higher background also have higher foreground values. Are the red and green background levels similar? Which arrays have the highest background levels?

```
> par(mfrow = c(1, 1))
> boxplot(log2(RG$R) ~ col(RG$R), col = "red")
> fgbg = cbind(RG$R, RG$Rb, RG$G, RG$Gb)
> boxplot(log2(fgbg) ~ col(fgbg), col = c(rep("red", 8), rep("green", 8)))
```

Exercise 14: Look at the trend in foreground log-intensity values according to the order in which the spots were printed on the array. The function `plotPrintorder` may take a while to calculate the print order before plotting. Is there any trend in the overall intensity of spots according to the print order; is the trend similar between red and green channels and between slides?

☞ The `plotPrintorder` function may take a while to run.

```
> plotPrintorder(RG, layout = RG$printer)
> plotPrintorder(RG, layout = RG$printer, separate.channels = TRUE)
> plotPrintorder(RG, layout = RG$printer, separate.channels = TRUE, slide = 2)
```

Background correction

Exercise 15: Perform background correction on the raw data. Look up the help for `backgroundCorrect` and find out the options for the possible methods for background correction.

Try performing no background correction, the default subtraction method and the minimum method.

☞ Background correction can be performed during the normalisation step. However, to investigate the different background correction methods separately from the normalisation methods, in this example we use the `backgroundCorrect` function in isolation.

```
> RG.nb = backgroundCorrect(RG, method = "none")
> RG.sb = backgroundCorrect(RG, method = "subtract")
> RG.mb = backgroundCorrect(RG, method = "minimum")
> names(RG)
> names(RG.nb)
> names(RG.sb)
> names(RG.mb)
```

Exercise 16: Use MA-plots to compare the data resulting from these different background correction methods for background measurments from `SPOT`. Save the MA-plots to a file in the current directory. Note that the command `pdf("file.pdf")` opens a file such that all subsequent plots to the graphics device will be written to this file rather than to the graphics window in `R`. Make sure you use `dev.off()` to close the file after making the plots you want to store to the file.

```
> pdf("MAswirl.pdf")
> par(mfrow = c(2, 2))
> plotMA(RG.nb, array = 2, xlim = c(0, 16), ylim = c(-5, 5), main = "bkgcorr: none")
> plotMA(RG.sb, array = 2, xlim = c(0, 16), ylim = c(-5, 5), main = "bkgcorr: subract")
> plotMA(RG.mb, array = 2, xlim = c(0, 16), ylim = c(-5, 5), main = "bkgcorr: minimum")
> dev.off()
```

Normalisation

🛈 There are numerous normalisation methods available in `limma`. These include both within and between array methods for normalising log-ratios and also methods for the normalisation of the single-channel log-intensity data from two-colour experiments. In `limma` background correction may be performed during the normalisation step. This is especially useful if you already have a prior idea of which background correction method you want to use.

☞ Print-tip loess normalisation is the default normalisation method.

Exercise 17: Apply the normalisation method to the background corrected data from the `swirl` directory. Check the MA-plots (and spatial plots, if you have time) after this normalisation. Compare them to the MA-plots saved to file for this data. Make pin-group loess plots to assess the similarity in the loess fit for spots in the different grids on the array. Do you think pin-group loess normaliation is necessary or would a global loess fit be sufficient?

```
> MA = MA.RG(RG.mb)
> MAp = normalizeWithinArrays(RG.mb)
```

```
> par(mfrow = c(2, 2))
> for (i in 1:4) {
+     plotMA(MAp, array = i)
+ }
> par(mfrow = c(1, 1))
> plotPrintTipLoess(MA, array = 1, layout = MA$printer)
> plotPrintTipLoess(MA, array = 2, layout = MA$printer)
> plotPrintTipLoess(MA, array = 3, layout = MA$printer)
> plotPrintTipLoess(MA, array = 4, layout = MA$printer)
```

Exercise 18: Perform global loess normalisation. Look at the file containing the MA-plots for these arrays, do you think global loess normalisation is required? Try global median normalisation also. Compare the MA-plots for these data after global loess and median normalisation.

```
> MAl = normalizeWithinArrays(RG.mb, method = "loess")
> MAm = normalizeWithinArrays(RG.mb, method = "median")
> par(mfrow = c(2, 2), ask = T)
> for (i in 1:4) {
+     plotMA(MA, array = i, main = "no norm")
+     plotMA(MAl, array = i, main = "loess norm")
+     plotMA(MAm, array = i, main = "median norm")
+     plotMA(MAp, array = i, main = "pin-gp loess norm")
+ }
> par(ask = F)
```

Exercise 19: Use boxplots to compare the distribution of log-ratios between arrays after different normalisation methods. What does a boxplot summarise about a set of numbers (look up the help for `boxplot`)? Which normalisation methods affect the location (vertical position of the boxplot) of log-ratios and which also affect the overall variability of log-ratios within arrays (height of boxplots)?

```
> boxplot(MA$M ~ col(MA$M), main = "no norm")
> boxplot(MAm$M ~ col(MAm$M), main = "median norm")
> boxplot(MAl$M ~ col(MAl$M), main = "loess norm")
> boxplot(MAp$M ~ col(MAp$M), main = "pin-gp loess norm")
```

☞ `limma` also has functions for performing between array normalisation using `normalizeBetweenArrays`. We will not be using this function in these practicals. Between array normalisation for two-colour array data should be done with great caution so as not to remove real biological differences between arrays.