

The analysis of aCGH data: Overview

JC Marioni, ML Smith, NP Thorne

January 13, 2006

Overview

■ **snapCGH** (Segmentation, Normalisation and Processing of arrayCGH data) is a package for the analysis of array comparative genomic hybridisation (aCGH) data. It is designed to allow the user to analyse aCGH data from preprocessing and normalisation through to segmenting the data into states with the same underlying copy number and assigning genomic meaning to these states. In this practical, we run through the analysis of a simple dataset using **snapCGH**. In future practicals, we will examine each of the components of the analysis in more depth.

■ *snapCGH* is designed to be used in conjunction with *limma* and so it will automatically load that library before proceeding.

Reading Data

■ Make sure the current directory for your R session is the **snapCGHIntro** folder. All your analysis will take place in this directory. At any point during your 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/snapCGHIntro")`. 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 *snapCGH* library (which automatically loads *limma* and other libraries) and read in the target file in the current directory. The targets file has been given the default name “targets.txt”.

```
> library(snapCGH)
> targets = readTargets("targets.txt")
> targets
```

■ Although reading in the targets file can seem fairly easy, getting the targets file right can be difficult. For examples of some of the problems that might occur, see the *limmaBasics* practical.

Reading in Data

Exercise 2: Read the 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.

❗ `limma` allows the user 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.

❗ The `read.maimages` function supports the reading in of microarray data files from various image analysis programs such as Agilent, Array Vision, ImaGene, GenePix, QuantArray and SPOT.

🔗 `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 clones, and columns correspond to arrays.

🔗 The sample arrays were image analysed using `genepix`. We specify that this is the case by setting the `source` argument in the `read.maimages` function.

```
> RG = read.maimages(targets$FileName, source = "genepix")
> show(RG)
```

Exercise 3: Read in the clone information file and add this information to the `RG` list. Subsequently, add information about the type of clones on the array to the `RG` list. Finally, create a design vector to indicate which channel (Red or Green) is the test channel.

❗ Information about the location of the clones on the genome and how this corresponds to their position on the array can be read in using the `read.clonesinfo` function. In order to do this it is necessary to create a clone information file. Such a file (which can be created using Excel and saved as a “txt” file) must contain columns called `Chr` and `Position` which give the chromosome on which a clone is located and its position (in basepairs) on that chromosome. Subsequently, we use the `getLayout` command to add information about the way the array is structured (i.e. how many blocks, rows and columns there are). The information is added to the `RG$genes` object.

🔗 The data in the clone information file must be ordered in the same way as the information in the output file obtained using the image analysis package.

❗ A spot types file contains information about the control status of particular spots on the array and allows specific spots to be highlighted in many of the plotting functions (i.e. we could highlight all positive control spots). The content of a spot types file is covered extensively within the *limma* manual and within the *limma* pre-processing practical.

❗ We create a design vector and add this element to the `RG` list to indicate which channel (Red or Green) contains DNA from the test sample for each array. Note that the length of the design vector should be equal to the number of arrays. If the test sample has been dyed

with Cy5 (Red), the appropriate entry in the design vector is 1. Alternatively, if the test sample has been dyed with Cy3 (Green), the appropriate entry in the design vector is -1 . This is the case for both arrays in this example.

```
> RG = read.clonesinfo("cloneinfo.txt", RG)
> RG$printer = getLayout(RG$genes)
> types = readSpotTypes("SpotTypes.txt")
> RG$genes$Status = controlStatus(types, RG)
> RG$design = c(-1, -1)
```

Background correction

Exercise 4: Perform background correction on the raw data. Look up the help for `backgroundCorrect` and find out the options for possible background correction methods. 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 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")
```

Exercise 5: Use MA-plots to compare data which has been background corrected (using the “minimum” method) with non background corrected data. Save the MA-plots to a file in the current directory.

♣ *Note that the command `pdf("file.pdf")` opens a file to which all subsequent plots will be written. Make sure you use `dev.off()` to close the file after making the plots you want to write to it.*

```
> pdf("MAcompBG.pdf")
> par(mfrow = c(2, 1))
> plotMA(RG, array = 2, xlim = c(0, 16), ylim = c(-5, 5), main = "bkgcorr: none")
> 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 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 used by limma. However, this method was designed to normalise gene expression data - it should be applied to array CGH data with great caution.*

Exercise 6: Apply the normalisation method to the background corrected data. Check the MA-plots after this normalisation. Compare them to the MA-plots saved in “MAcompBG.pdf”. Make pin-group loess plots to assess the similarity in the loess fit for spots in different grids on the array. Do you think pin-group loess normaliation is necessary?

```
> MA.p = normalizeWithinArrays(RG.mb)
> par(mfrow = c(1, 2))
> for (i in 1:2) {
+   plotMA(MA.p, array = i)
+ }
> par(mfrow = c(1, 1))
> MA = MA.RG(RG.mb)
> plotPrintTipLoess(MA, array = 1, layout = MA$printer)
> plotPrintTipLoess(MA, array = 2, layout = MA$printer)
```

Exercise 7: Perform global loess normalisation. Look at the file containing the MA-plots for these arrays, do you think global loess normalisation is required or does a global median normalisation suffice? Compare the MA-plots after global loess and median normalisation have been carried out.

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

☞ The output of the normalisation function is a new type of object called an *MAList*. This is composed of the \log_2 ratios, intensities and the clone and slide layout information which it obtains from the *RG* object.

Exercise 8: Plot the normalised \log_2 ratios against the position of the clones on the genome. Similarly, plot the normalised \log_2 ratios against the position of the clones on the genome for a particular chromosome. Save the resulting plots to a file in the current directory.

📌 Plotting the \log_2 ratios against the physical location of the clones on the genome enables the user to get a feeling for which regions of the genome might be gained and which might be lost. Additionally, such a plot may give the user the ability to judge whether the normalisation may be biased due to many (non-symmetric) gains and losses. The input for this function can be an *MAList* or *SegList* object.

❏ Additionally, by using the spot types file read in earlier, the user can highlight particular clones of interest. For information on how this can be used and on other options available within this function, read the relevant helpfile.

```
> pdf("WholeGenome.pdf")
> plotGenome(MA.m)
> dev.off()
> pdf("Chromosome8.pdf")
> plotGenome(MA.m, chrom.to.plot = 8)
> dev.off()
```

Processing

❏ After the preprocessing and normalisation steps have been carried out, we are ready to segment the genome into regions with the same underlying copy number.

Exercise 9: Use the `processCGH` function to re-order the *MAList* so that clones are ordered according to their position on the genome and remove clones that are not mapped to chromosomes. This function also averages the \log_2 ratios for replicated clones and allows the option of imputing missing observations. The imputation is necessary since the segmentation functions (as presently written) do not segment the data successfully if there are missing values.

🔗 In order to gain a better understanding of all of the arguments that are available within this function, look at the help file for the `processCGH` function.

```
> MA2 = processCGH(MA.m.method.of.averaging = mean)
```

The segmentation step

Exercise 10: Segment the data using a Hidden Markov Model [1], a circular binary segmentation algorithm (DNAcopy) [3] and an adapted weights smoothing algorithm (GLAD) [2]. Subsequently, merge segments whose mean values are adjudged to be too close to one another.

❏ For more details of the relative merits of the above schemes and a thorough description of the various merging algorithms, see [4]. In order to understand the different arguments that are available when applying these different models, see the relevant help files.

♣ For larger data sets this step can take a fairly long time.

```
> SegInfo.HMM = runHomHMM(MA2, criteria = "AIC")
> SegInfo.DNACopy = runDNACopy(MA2)
> SegInfo.GLAD = runGLAD(MA2)
```

❏ All of the segmentation methods occasionally suffer from the tendency to fit a model with segments whose means are very close to one another. In order to overcome this problem *snapCGH* allows the user the option of merging states using one of two different methods described in [4]. For more information on the two merging options, see the relevant helpfiles.

```
> SegInfo.HMM.merged <- mergeStates(SegInfo.HMM, MergeType = 1)
> SegInfo.DNACopy.merged <- mergeStates(SegInfo.DNACopy, MergeType = 1)
> SegInfo.GLAD.merged <- mergeStates(SegInfo.GLAD, MergeType = 1)
```

Exercise 11: Use the `findGenomicEvents` to assign biological meanings to the different states.

❏ This function, which is based on methods described in [1], assigns biological meaning to the different states and also to clones within these states. These assignments are used in subsequent plotting functions.

✎ The function `findGenomicEvents` stores the output from the above function in a `GEList` (Genomic Events List) class object.

```
> GenList.HMM = findGenomicEvents(SegInfo.HMM.merged)
> GenList.DNACopy = findGenomicEvents(SegInfo.DNACopy.merged)
> GenList.GLAD = findGenomicEvents(SegInfo.GLAD.merged)
```

Summarising the output

Exercise 12: Summarise the segmented output using the `plotSegmentedGenome` function.

❏ The `plotSegmentationStates` function provides a visual representation of the processed \log_2 ratios for a particular chromosome alongside the predicted values produced by the segmentation algorithm. It requires both a `SegList` and a `GEList` object. The `array` argument specifies the array we would like to plot, whilst the desired chromosome is passed via the `chrom.to.plot` parameter.

```
> plotSegmentedGenome(SegInfo.HMM.merged, array = 1, chrom.to.plot = 8,
+   X = FALSE, Y = FALSE)
```

References

- [1] Fridlyand, J., Snijders, A.M., Pinkel, D., Albertson, D.G., Jain, A.N. (2004) Hidden Markov models approach to the analysis of array CGH data, *Journal of Multivariate Analysis*, **90**, 132-153.
- [2] Hupé, P., Stransky, N., Thiery, J-P., Radvanyi, F., Barillot, E. (2004) Analysis of array CGH data: from signal ratio to gain and loss of DNA regions, *Bioinformatics*, **20**, 3413-3422.
- [3] Olshen, A.B., Venkatraman, E.S., Lucito, R., Wigler, M. (2004) Circular binary segmentation for the analysis of array-based DNA copy number data, *Biostatistics*, **5**, 557-572.
- [4] Willenbrock, H., Fridlyand, J. (2005) A comparison study: applying segmentation to array CGH data for downstream analyses, *Bioinformatics*, **21**, 4084-4091.