BeadArray expression analysis using Bioconductor

Mark Dunning

November 1, 2013

Processing Illumina BeadChips

This practical is an abridged version of the vignette found in the BeadArrayUseCases package, which you should already have installed. In this practical you will go through the main steps in the workflow of analysiing Illumina data from various sources.

Experimental data

The example dataset consists of bead-level data from a series of Human HT-12 version 3 BeadChips hybridized at the Cancer Research UK, Cambridge Research Institute Genomics Core facility. 'Bead-level' refers to the availability of intensity and location information for each bead on each BeadArray in an experiment. In this dataset, BeadArrays were hybridized with either Universal Human Reference RNA (UHRR, Stratagene) or Brain Reference RNA (Ambion) as used in the MAQC project [1]. Bead-level data for all 12 arrays are included in the BeadArrayUseCases package. These data are in the compressed `.bab` format [2], which can be analysed using the beadarray package.
We also use data from the beadarrayExampleData package to avoid running some time-consuming operations in the practical.

1 Analysis of bead-level data using beadarray

Quality assessment using scanner metrics

The first view of array quality can be assessed using the metrics calculated by the scanner. These include the 95th (P95) and 5th (P05) quantiles of all pixel intensities on the image. A signal-to-noise ratio (SNR) can be calculated as the ratio of these two quantities. These metrics can be viewed in real-time as the arrays themselves are being scanned. By tracking these metrics over time, one can potentially halt problematic experiments before they even reach the analysis stage.

Use Case: Plot the P95:P05 signal-to-noise ratio for the HT-12 arrays in this experiment and assess whether any samples appear to be outliers that may need to be removed or down-weighted in further analyses.

```
> ht12metrics <- read.table(system.file("extdata/Chips/Metrics.txt",
+     package = "BeadArrayUseCases"), sep = "\t", header = TRUE,
+     as.is = TRUE)
> ht12metrics
> ht12snr <- ht12metrics$P95Grn/ht12metrics$P05Grn
> labs <- paste(ht12metrics[, 2], ht12metrics[, 3], sep = "_")
> par(mai = c(1.5, 0.8, 0.3, 0.1))
> plot(1:12, ht12snr, pch = 19, ylab = "P95 / P05", xlab = "",
+     main = "Signal-to-noise ratio for HT12 data", axes = FALSE,
+     frame.plot = TRUE)
> axis(2)
> axis(1, 1:12, labs, las = 2)
```

The above code uses standard `R` functions to obtain the P95 and P05 values from the metrics file stored in the package. The `system.file` function is a base function that will locate the directory where the BeadarrayUseCases package is installed. The plotting commands are just a suggestion of how the data could be presented and could be adapted to individual circumstances.

☞ Scanner metrics information is equally as useful for sample quality assessment of summarized data.

☞ The P95 and P05 values will fluctuate over time and are dependant upon the scanner setup. Including SNR values for arrays other than those currently being analysed will give a better indication of whether any outlier arrays exist.

Data import and storage

The first step in our analysis is to read the data into `R` using the `readIllumina` function. The bead-level data you will need for this example are available in the `extdata/Chips` directory of the BeadArrayUseCases package. These files were produced using the BeadDataPackR package in Bioconductor which provides a more compact representation of bead-level data. Each BeadChip to be analysed should be found in a sub-directory of the current working directory, with the directory name being the same as the chip name. A sample sheet is used to specify what samples are hybridised to each array and the corresponding sample group.

Use Case: Read the example bead-level dataset supplied with the BeadArrayUseCases package into R.

```
> library(beadarray)
> chipPath <- system.file("extdata/Chips", package = "BeadArrayUseCases")
> list.files(chipPath)
> sampleSheetFile <- paste(chipPath, "/sampleSheet.csv",
+     sep = "")
> readLines(sampleSheetFile)
> data <- readIllumina(dir = chipPath, sampleSheet = sampleSheetFile,
+     useImages = FALSE, illuminaAnnotation = "Humanv3")
```

Usually the directory will be in a known location, but for convenience we use the `system.file` function to find the directory where BeadArrayUseCases is installed.

The final line executes the reading of the data. The `extdata/Chips` directory is used as a base directory, and each chip to be read is found in a separate sub-directory. As defined by the sample sheet, we are only reading one section from each chip
By setting `useImages` to `FALSE`, we use the intensity values stored in the text files, rather than recomputing them from the images. The argument `illuminaAnnotation` is a character string to identify the organism and annotation revision number of the chip being analysed, but not the number of samples on the chip. Hence the value `Humanv3` can be used for both Human WG-6 and HumanHT12 v3 data. Setting this value correctly will allow beadarray to identify what bead types are to be used for control purposes and convert the numeric ArrayAddressIDs to a more commonly-used format. If you are unsure of the correct annotation to use, this argument can be left blank at this stage.

☞ If the data to be imported are not in `.bab` format, specifying the same arguments to `readIllumina` will search for files of type `.txt` instead.

Selecting or checking the annotation for a dataset

If you are unsure of the correct annotation to use and thus left the `illuminaAnnotation` argument to `readIllumina` as the default, `suggestAnnotation` can be employed to identify the platform, based on the ArrayAddressIDs that are present in the data. The `setAnnotation` function can then be used to assign this annotation to the dataset.

Use Case: Verify the version number of the dataset that has been read in and set the annotation of the bead-level data object accordingly.

```
1 > suggestAnnotation(data, verbose = TRUE)
2 > annotation(data) <- "Humanv3"
```

☞ The result of `suggestAnnotation` only gives guidance about which annotation to use. Hence, the results may be unpredictable on custom arrays, or arrays that are not listed in the `suggestAnnotation` output.

The *beadLevelData* class

Once imported, bead-level data are stored in an object of class *beadLevelData*. This class can handle raw data from both single-channel and two-color BeadArray platforms.

```
1 > slotNames(data)
```

The command above gives us an overview of the structure of the *beadLevelData* class, which is composed of several slots. The `experimentData`, `sectionData` and `beadData` slots can be viewed as a hierarchy, with each containing data at a different level. Each can be accessed using the `@` operator, although we will see an easier way of accessing the data shortly.

The `experimentData` slot holds information that is consistent across the entire dataset. Quantities with one value per array-section are stored in the `sectionData` slot. For instance, any metrics information read by `readIllumina`, along with section names and the total number

of beads, will be stored there. This is also a convenient place to store any QC information derived during the preprocessing of the data. The data extracted from the individual text files are stored in the `beadData` slot.

Accessing data in a *beadLevelData* object

Use Case: Output the data stored in the `sectionData` slot, and determine the section names and number of bead intensities available from each section. Access the intensities, x-coordinates and probe IDs for the first 5 beads on the first array section.

```
1 > data@sectionData
2 > sectionNames(data)
3 > numBeads(data)
4 > head(data[[1]])
5 > getBeadData(data, array = 1, what = "Grn")[1:5]
6 > getBeadData(data, array = 1, what = "GrnX")[1:5]
7 > getBeadData(data, array = 1, what = "ProbeID")[1:5]
```

Using the `@` operator to access the data in particular slots is not particularly convenient or intuitive. The functions `sectionNames`, `numBeads` and `getBeadData` provide more convenient interfaces to the *beadLevelData* object to retrieve specific information.
The first line above uses the `@` operator to access all the data in the `sectionData` slot, which can be quite large and unwieldy. The commands below it (lines 2-3) are accessor functions for retrieving a specific subset of data from the same slot.

Line 4 shows that if a *beadLevelData* object is accessed in the same fashion as a list, a data frame containing the bead-level data for the specified array is returned. To access a specific entry in this data frame, we can use a further subset, or the data can be accessed using the `getBeadData` function. In addition to the *beadLevelData* object, you need to specify the section (`array=...`) of interest and the column heading you want.

Extracting transformed data

In this example, the data stored in the *beadLevelData* object by `readIllumina` are extracted directly from the Illumina text files. The values in the `Grn` column are intensity values inferred from a known location in the scanned image and there are a number of steps involved before these can be translated into quantities that relate to the expression levels. The scanner generally produces values in the range 0 to $2^{16}-1$, although the image manipulation and background subtraction steps can lead to values outside this range. This is not a convenient scale for visualization and analysis and it is common to convert intensities onto the approximate range 0 to 16 using a $\log_2$ transformation (possibly after an additional step to adjust non-positive intensities).

Although this is simple to do in isolation using `R`'s built in functions, it becomes more complicated within a function, leading to a large number of arguments being required in order to specify whether the function should process the green or red channel, use foreground or background intensities, convert to the $\log_2$ scale etc. Even in this situation the user is restricted to

the options that are provided by the arguments.

A more flexible way to obtain transformed per-bead data from a *beadLevelData* object is to define a transformation function that takes as arguments the *beadLevelData* object and an array index. The function then manipulates the data in the desired manner and returns a vector the same length as the number of beads on the array. Many of the plotting and quality assessment functions within beadarray take such a function as one of their arguments. By using such a system, beadarray provides a great deal of flexibility over exactly how the data is analysed.

Use Case: Extract the green intensities on the $\log_2$ scale for the first 10 probes from the first array section.

```
1 > log2(data[[1]][1:10, "Grn"])
2 > log2(getBeadData(data, array = 1, what = "Grn")[1:10])
3 > logGreenChannelTransform(data, array = 1)[1:10]
```

The above example shows three different ways of obtaining the log green channel intensity data. Lines 1 and 2 use R's `log2` function on data extracted using the methods we've already seen. The third entry uses one of beadarray's built in transformation functions. To view an example of how a transformation function is defined you can enter the name of one of beadarray's pre-defined functions without any parentheses or arguments.

```
1 > logGreenChannelTransform
```

☞ In addition to the `logGreenChannelTransform` function shown above, beadarray provides predefined functions for extracting the green intensities on the unlogged scale (`greenChannelTransform`), analogous functions for two-channel data (`logRedChannelTransform`, `redChannelTransform`), and functions for computing the log-ratio between channels (`logRatioTransform`).

☞ If you are interested in some of advance image processing features of beadarray, please see the BeadArrayUseCases vignette at a later date.

Quality assessment for raw and bead-level data

Boxplots of intensity values

Boxplots are routinely used to assess the dynamic range of each sample and look for unusual signal distributions.

Use Case: Create boxplots of the green channel intensities for all arrays. Do you notice any arrays with unusual distributions?

```
1 > boxplot(data, transFun = logGreenChannelTransform, col = "green",
2 +     ylab = expression(log[2](intensity)), las = 2, outline = FALSE,
3 +     main = "HT-12 MAQC data")
```

The `boxplot` function is a standard function in R that we has been extended to work for the *beadLevelData* class. Consequently, the standard parameters to `boxplot`, such as changing the title of the plot, scale and axis labels are possible, some of which are shown in the final five arguments above. The help page for the `par` function provides information on these and other arguments that can be supplied to `boxplot`. The only beadarray specific argument is the second, `transFun`, which takes a transformation function of the format shown previously. In this case we have selected to use the $\log_2$ of the green channel, which is also the default.

Plotting control probes

Illumina have designed a number of control probes for each BeadArray platform. Two particular controls on expression arrays are housekeeping and biotin controls, which are expected to be highly expressed in any sample. With the `poscontPlot` function, we can plot the intensities of any ArrayAddressIDs that are annotated as belonging to the Housekeeping or Biotin control groups.

Use Case: Generate plots of the housekeeping and biotin controls for the 6th, 7th, 8th and 12th arrays from this dataset.

```
> par(mfrow = c(2, 2))
> for (i in c(6, 7, 8, 12)) {
+     poscontPlot(data, array = i, main = paste(sectionNames(data)[i],
+         "Positive Controls"), ylim = c(4, 15))
+ }
```

Provided the annotation of the *beadLevelData* object has been correctly set, the `poscontPlot` function should be able to identify the revelant probes and intensities. This code generates positive controls plots for four arrays in the dataset; one good quality array and three that we have noted problems with. A different view of the control probes is now provided by the `combinedControlPlot` function in beadarray, which allows better comparison of control types.

Use Case: Make combined control plots for the same arrays as above

```
> combinedControlPlot(data, array = 6)
> combinedControlPlot(data, array = 7)
> combinedControlPlot(data, array = 8)
> combinedControlPlot(data, array = 12)
```

Visualizing intensities across an array surface

The combination of both an intensity and a location for each bead on the array allows us to visualize how the intensities change across the array surface. This kind of visualization is not possible when using the summarized output, as the summary values are averaged over spatial positions. The `imageplot` function can be used to create false color representations of the array surface.

Use Case: Produce imageplots for the same set of arrays and look for any evidence of spatial artefacts.

```
> imageplot(data, array = 6, high = "darkgreen", low = "lightgreen",
+     zlim = c(4, 10))
> imageplot(data, array = 7, high = "darkgreen", low = "lightgreen",
+     zlim = c(4, 10))
> imageplot(data, array = 8, high = "darkgreen", low = "lightgreen",
+     zlim = c(4, 10))
> imageplot(data, array = 12, high = "darkgreen", low = "lightgreen",
+     zlim = c(4, 10))
```

Plotting the location of outliers

Recall that the BeadArray technology includes many replicates (typically ~ 20 of each probe type in each sample on an HT-12 array). BeadStudio/GenomeStudio removes outliers greater than 3 median absolute deviations (MADs) from the median prior to calculating summary values.

Use Case: Plot the location of outliers on the arrays with the most obvious spatial artefacts and plot their location.

```
> par(mfrow = c(2, 1))
> for (i in c(8, 12)) {
+     outlierplot(data, array = i, main = paste(sectionNames(data)[i],
+         "outliers"))
+ }
```

By default, the `outlierplot` function uses Illumina's 'three MADs from the median' rule, but applied to log-transformed data. It then plots the identified outliers by their location on the surface of the array section. Of course, the user can specify alternative rules and transformations and the function is additionally able to accept arguments to `plot` such as `main`.

Excluding beads affected by spatial artefacts

It should be apparent that some arrays in this dataset have significant artefacts and, although it appears that most beads in these regions are classed as outliers, it would be desirable to mask all beads from these areas from further analysis. Our preferred method for doing this is to use BASH [3], which takes local spatial information into account when determining outliers, and uses replicates within an array to calculate residuals.

BASH performs three types of artefact detection in the style of the affymetrix-oriented Harshlight [4] package: *Compact analysis* identifies large clusters of outliers, where each outlying bead must be an immediate neighbour of another outlier; *Diffuse analysis* finds regions that contain more outliers than would be anticipated by chance, and *Extended analysis* looks for chip-wide variation, such as a consistent gradient effect.

The output of BASH is a list containing a variety of data, including a list of weights indicating the beads that BASH has identified as problematic. These weights may be saved to the *beadLevelData* object for future reference by using the `setWeights` function. The locations of the masked beads can be visualized using the `showArrayMask` function.

♠ *The following commands can be used to run BASH on array 8 in the dataset. However, due to time-constraints we suggest that you do not run these in the practical.*

```
> BASHoutput <- BASH(data, array = 8)
> data <- setWeights(data, wts = BASHoutput$wts, array = 8)
```

Two arrays in this dataset have already been analysed by BASH and can be found in the beadarrayExampleData package

Use Case: Load the beadarrayExampleData package, which contains the BASH output from two arrays in this dataset. Find out how many beads have been masked for removal and visualise their locations.

```
> library(beadarrayExampleData)
> data(exampleBLData)
> sectionNames(exampleBLData)
> head(exampleBLData[[1]])
> table(getBeadData(exampleBLData, what = "wts", array = 1))
> table(getBeadData(exampleBLData, what = "wts", array = 2))
> par(mfrow = c(1, 2))
> for (i in c(1, 2)) {
+     showArrayMask(exampleBLData, array = i, override = TRUE)
+ }
```

You should see that after having run BASH and `setArrayWeights`, a `wts` column appears in the object. We call `showArrayMask`, which creates a plot similar to `outlierplot` mentioned earlier. In addition to displaying the beads classed as outliers, `showArrayMask` shows in red the beads that are currently masked from further analysis. By default the function refuses to create the plot if over 200,000 beads have been masked, as this can cause considerable slowdown on older computers, so the `override` argument has been used in this example to force the plot creation.

Producing control tables

With knowledge of what ArrayAddressIDs match different control types, we can easily provide summaries of these control types on each array. In `quickSummary` the mean and standard deviation of all control types is calculated for a specified array, using intensities of all beads that correspond to the different control types. Note that these summaries may not correspond to similar quantities reported in Illumina's BeadStudio/GenomeStudio software, as the summaries are produced after removing outliers. The `makeQCTable` function extends this functionality to produce a table of summaries for all sections in the *beadLevelData* object. These data can be stored in the `sectionData` slot for future reference. It is also informative to compare the expression level of various control types to the background level of the array. This is done by

the `controlProbeDetection` function that returns the percentage of each control type that are significantly expressed above background level. For positive controls we would prefer this to be near 100% on a good quality array.

Use Case: Summarize the control intensities for the first array, then tabulate the mean and standard deviation of all control probes on every array.

```
1 > quickSummary(data, array = 1)
2 > qcReport <- makeQCTable(data)
3 > head(qcReport)[, 1:5]
```

The above code generates a quality control summary for a single array (Line 1), then for all arrays in the *beadLevelData* object using the `makeQCTable` function.

Saving control tables to a bead-level object

The `insertSectionData` function allows the user to modify the `sectionData` slot of a `beadLevelData` object. We can use this functionality to store any quality control (QC) values that we have computed.

Use Case: Store the computed QC values into the bead-level data object.

```
1 > data <- insertSectionData(data, what = "BeadLevelQC",
2 +     data = qcReport)
3 > names(data@sectionData)
```

The `insertSectionData` function requires a data frame with the same number of rows as the number of sections in the *beadLevelData* object. The `what` parameter is used to assign a name to the data in the `sectionData` slot.

☞ You could also save the QC table to disk using the `write.csv` function (or similar).

Summarizing the data

The summarization procedure takes the *beadLevelData* object, where each probe is replicated a varying number of times on each array, and produces a summarized object which is more amenable for making comparisons between arrays and sample groups. For each array section represented in the *beadLevelData* object, all observations are extracted, transformed, and then grouped together according to their ArrayAddressID. Outliers are removed and the mean and standard deviation of the remaining beads are calculated.

There are many possible choices for the extraction, transformation and choice of summary statistics and beadarray allows users to experiment with different options via the definition of an *illuminaChannel* class. For expression data, the green intensities will be the quantities to be summarised. However, the *illuminaChannel* class is designed to cope with two-channel data where the green or red (or some combination of the two) may be required with minimal changes to the code. The `summarize` function is used to produce summary-level data and has the default setting of performing a $\log_2$ transformation. With the appropriate changes

it is also possible to summarize on the un-logged scale, as if the data had been processed by GenomeStudio.

Use Case: Generate summary-level data for this dataset on the $\log_2$ and unlogged scale.

```
1 > datasumm <- summarize(BLData = data)
2 > grnchannel.unlogged <- new("illuminaChannel", transFun = greenChannelTransform,
3 +     outlierFun = illuminaOutlierMethod, exprFun = function(x) mean(x,
4 +         na.rm = TRUE), varFun = function(x) sd(x, na.rm = TRUE),
5 +     channelName = "G")
6 > datasumm.unlogged <- summarize(BLData = data, useSampleFac = FALSE,
7 +     channelList = list(grnchannel.unlogged))
```

Line 1 uses the default settings of `summarize` to produce summary-level data. Line 2 shows the full gory details of how to modify how the bead-level data are summarized by the creation of an *illuminaChannel* object. We use a transformation function that just returns the Grn intensities, Illumina's default outlier function and modified mean and standard deviation functions that remove `NA` values. This new channel definiton is then passed to `summarize`. In this call we also explicitly set the `useSampleFac` argument to `FALSE`. The `useSampleFac` parameter should be used in cases where multiple sections for the same biological sample are to be combined, which is not applicable in this case.

☞ During the summarization process, the numeric ArrayAddressIDs used in the *beadLevelData* object are converted to the more commonly-used Illumina IDs. However, control probes retain their original ArrayAddressIDs and any IDs that cannot be converted (e.g. internal controls used by Illumina for which no annotation exists) are removed unless `removeUnMappedProbes = TRUE` is specified.

The *ExpressionSetIllumina* class

Summarized data are stored in an object of type *ExpressionSetIllumina* which is an extension of the *ExpressionSet* class developed by the Bioconductor team as a container for data from high-throughput assays.

Objects of this type use a series of slots to store the data. For consistency with the definition of other *ExpressionSet* objects, we refer to the expression values as the `exprs` matrix (this stores the probe-specific average intensities) which can be accessed using `exprs` and subset in the usual manner. The `se.exprs` matrix, which stores the probe-specific variability can be accessed using `se.exprs`, and phenotypic data for the experiment can be accessed using `pData`. To accommodate the unique features of Illumina data we have added an `nObservations` slot, which gives the number of beads that we used to create the summary values for each probe on each array after outlier removal. The detection score, or detection p-value is a standard measure for Illumina expression experiments, and can be viewed as an empirical estimate of the p-value for the null hypothesis that a particular probe in not expressed. These can be calculated for summarized data provided that the identity of the negative controls on the array is known using the function `calculateDetection`.

Use Case: Produce boxplots of the summarized data and calculate detection scores. How many probes are present in the summarized data?

```
> dim(datasumm)
> exprs(datasumm)[1:10, 1:2]
> se.exprs(datasumm)[1:10, 1:2]
> par(mai = c(1.5, 1, 0.2, 0.1), mfrow = c(1, 2))
> boxplot(exprs(datasumm), ylab = expression(log[2](intensity)),
+     las = 2, outline = FALSE)
> boxplot(nObservations(datasumm), ylab = "number of beads",
+     las = 2, outline = FALSE)
> det <- calculateDetection(datasumm)
> head(det)
> Detection(datasumm) <- det
```

☞ `calculateDetection` assumes that the information about the negative controls is found in a particular part of the *ExpressionSetIllumina* object, and takes the form of a vector of characters indicating whether each probe in the data is a control or not. This vector can be supplied as the `status` argument along with an identifier for the negative controls (`negativeLabel`).

feature and pheno data

The `fData` and `pData` functions are useful shortcuts to find more information about the features (rows) and samples (columns) in the summary object. These annotations are created automatically whenever a bead-level data is summarized or read from a BeadStudio file. The `fData` will be added to later, but initially contains information on whether each probe is a control or not. In this example the `phenoData` denotes the sample group for each array; either Brain or UHRR (Universal Human Reference RNA).

Use Case: Find out how many control probes were present on the array and what arrays belong to the Brain and UHRR groups.

```
> head(fData(datasumm))
> table(fData(datasumm)[, "Status"])
> pData(datasumm)
```

Subsetting the data

As we have seen, the expression matrix of the `ExpressionSetIllumina` object can be subset by column or row, In fact, the same subset operations can be performed on the `ExpressionSetIllumina` object itself. In the following code, notice how the number of samples and features changes in the output.

Use Case: Create a new summary object for just the UHRR arrays.

```
> uhrrData <- datasumm[, 1:6]
> uhrrData
> pData(uhrrData)
```

The object can also be subset by a vector of characters which must correspond to the names of features (i.e. row names). Currently, no analogous functions is available to subset by sample.

Use Case: Subset the summary data by row; take a subset of the first 1000 rows, and then by 1000 randomly chosen Illumina IDs

```
> datasumm[1:1000, ]
> randIDs <- sample(featureNames(datasumm), 1000)
> datasumm[randIDs, ]
```

Exploratory analysis using boxplots

Boxplots of intensity levels and the number of beads are useful for quality assessment purposes. We have already seen how to create boxplots of the expression and nObservations matrices using standard R graphics. However, beadarray includes a modified version of the `boxplot` function that can take any valid *ExpressionSetIllumina* object and plot the expression matrix by default using the ggplot2 system. This allows sample and probe information to be incorporated easily into the plots. For these examples we plot just a subset of the original `datasumm` object using random row IDs.

Use Case: Use beadarray's modified boxplot function to plot the expression values for each sample.

```
> boxplot(datasumm)
```

The function can also plot other `assayData` items, such as the number of observations.

Use Case: Now, use beadarray's modified boxplot function to plot the number of observations for each sample.

```
> boxplot(datasumm, what = "nObservations")
```

The default boxplot plots a separate box for each array, but often it is beneficial for compare expression levels between different sample groups. With standard R graphics, this might involve subsetting up the plot window according and having to subset the dataset appropriately. However, with the beadarray `boxplot` information stored in the *phenoData* slot it can be incorporated into the plot. If the `sampleFactor` argument to `boxplot` is set to a column found in the phenoData

Use Case: Compare the overall expression level between UHRR and Brain samples on a boxplot

```
> boxplot(datasumm, sampleFactor = "Sample_Group")
```

In a similar manner, we may wish to visualize the differences between sample groups for particular probe groups. As a simple example, we might want to look at the difference between negative controls and regular probes for each array. Information about control types is stored in the `Status` column of the `fData` output

Use Case: Plot the intensity difference between negative and regular probes for all arrays. To simplify the plot you can use the 1000 genes subset we created earlier. Then, plot the difference between control types between UHRR and Brain samples.

```
> boxplot(datasumm[randIDs, ], probeFactor = "Status")
> boxplot(datasumm, probeFactor = "Status", sampleFactor = "Sample_Group")
```

☞ These boxplots have been generated using the ggplot2 package, and cannot be modified in the same way as traditional R graphics. For instance, you may be familiar with using the `main` or `ylim` arguments to change the title or range of a plot. Instead, ggplot uses carefully selected default settings to creates a basic plot, and allows other options to be added (using the '+' function) to the plot afterwards. The ggplot2 website, `http://had.co.nz/ggplot2/`, contans excellent documentation about how to customise plots and the code below demonstrates several options that are possible.

```
> boxplot(datasumm[randIDs, ], probeFactor = "Status") +
+     labs(title = "My fancy title", x = "Sample Type",
+         y = "Normalized Intensity", colour = "legend title") +
+     ylim(5, 9) + scale_fill_manual(name = "My fancy legend",
+     values = c("slateblue3", "violetred4"), labels = c("Type1",
+         "Type2")) + coord_flip() + theme(plot.background = theme_rect("khaki"),
+     panel.background = theme_rect("peachpuff"))
```

Preprocessing, quality assessment and filtering

The controls available on the Illumina platform can be used in the analysis to improve inference. As we have already seen in section 1, positive controls can be used to identify suspect arrays. Negative control probes, which measure background signal on each array, can be used to assess the proportion of expressed probes that are present in a given sample [5]. The `propexpr` function estimates the proportion of expressed probes by comparing the empirical intensity distribution of the negative control probes with that of the regular probes. A mixture model is fitted to the data from each array to infer the intensity distribution of expressed probes and estimate the expressed proportion.

Use Case: Estimate the proportion of probes which are expressed above the level of the negative controls on the MAQC samples using the `propexpr` function. Do you notice a difference between the expressed proportions in the UHRR and Brain Reference RNA samples?

```
> library(limma)
> proportion <- propexpr(exprs(datasumm), status = fData(datasumm)$Status)
> proportion
> t.test(proportion[1:6], proportion[7:12])
```

☞ Systematic differences exist between different BeadChip versions, so these proportions should only be compared within a given platform type [5]. This estimator has a variety of applications. It can be used to distinguish heterogeneous or mixed cell samples from pure samples and to provide a measure of transcriptome size.

Checking for batch effects

Multidimensional scaling (MDS), assesses sample similarity based on pair-wise distances between samples. This dimension reduction technique uses the top 500 most variable genes between each pair of samples to calculate a matrix of Euclidean distances which are used to generate a 2 dimensional plot. Ideally, samples should separate based on biological variables (RNA source, sex, treatment, etc.), but often technical effects (such as samples processed together on the same BeadChip) may dominate. Principal component analysis (PCA) is another dimension reduction technique frequently applied to microarray data.

Use Case: Generate a multidimensional scaling (MDS) plot of the data using the `plotMDS` function. Assess whether the samples cluster together by sample group.

```
> plotMDS(exprs(datasumm), col = c(rep("red", 6), rep("blue",
+     6)), labels = pData(datasumm)$Sample_Name)
```

If there are no clear batch effects in the data, you will still need to normalise to make the arrays comparable. The most popular way of doing this is the quantile normalisation that you will be familiar with from other microarray technologies.

Use Case: Remove any poor quality arrays from the dataset, and apply quantile normalisation to the remaining arrays.

```
> normData <- normaliseIllumina(datasumm[, c(1:6, 9:12)])
> dim(normData)
> boxplot(normData)
```

If you have imported data from GenomeStudio, you might want to consider an alternative approach to normalisation, as described in the BeadArrayUseCases vignette, that makes use of the negative controls on the array. The normal-exponential convolution model has proven useful in background correction of both Affymetrix [6] and two-color data [7]. Having a well-behaved set of negative controls simplifies the parameter estimation process for background parameters in this model. Applying this approach to Illumina gene expression data has been shown to offer improved results in terms of bias-variance trade-off and reduced false positives [8]. The `neqc` function [8] in limma fits such a convolution model to the intensities from each sample, before quantile normalizing and $\log_2$ transforming the data to standardize the signal between samples.

Use Case: Normalise the unlogged summarized data using the neqc method

```
> normData.neqc <- normaliseIllumina(datasumm.unlogged[,
+     c(1:6, 9:12)], method = "neqc")
```

```
> boxplot(normData.neqc)
> dim(normData.neqc)
> table(fData(normData.neqc)[, "Status"])
```

Filtering based on probe annotation

Filtering non-responding probes from further analysis can improve the power to detect differential expression. One way of achieving this is to remove probes whose probe sequence has undesirable properties. The illuminaHumanv3.db annotation package provides access to the re-annotation information provided by Barbosa-Morais *et al.* [9]. In this paper, a scoring system was defined to quantify the reliability of each probe based on its 50 base sequence. These mappings are based on the probe sequence and not the RefSeq ID, as for the standard annotation packages and can give extra criteria for interpreting the results. For instance, probes with multiple genomic matches, or matches to non-transcribed genomic locations are likely to be unreliable. This information can be used as a basis for filtering promiscuous or un-informative probes from further analysis, as shown above.
The illuminaHumanv3.db package is an example of a Bioconductor annotation package built using infrastructure within the AnnotationDBi package. More detailed descriptions of how to access data within annotation packages (and how it is stored) is given with the AnnotationDbi package. Essentially, each annotation package comprises a database of mappings between a defined set of microarray identifiers and genomic properties of interest. However, the user of such packages does not need to know the details of the database scheme as convenient wrapper functions are provided.

Use Case: Add annotation data from the Human v3 annotation package and verify that probes annotated as 'Bad' or 'No match' generally have lower signal. Exclude such probes from further analysis.

```
> ids <- as.character(featureNames(normData))
> qual <- unlist(mget(ids, illuminaHumanv3PROBEQUALITY,
+     ifnotfound = NA))
> qual <- gsub("*", "", qual, fixed = TRUE)
> table(qual)
> fData(normData) <- cbind(fData(normData), qual)
> boxplot(normData, probeFactor = "qual", sampleFactor = "Sample_Group")
> rem <- qual == "No match" | qual == "Bad"
> normData.filt <- normData[which(!rem), ]
> dim(normData)
> dim(normData.filt)
```

Here, the `featureNames` function is a shortcut to find out the probe IDs. The qualities returned by `illuminaHumanv3PROBEQUALITY` contain some entries with a `*` appended. These refer to special cases, such as mismatches between the transcriptome and genome, or lack or coding annotation and we will ignore these for now.

☞ The functions used to annotate Illumina BeadArrays have a very simple convention which is `illumina` followed by an organism name, annotation version number, and the mapping

you want to use. Hence, the above code could be executed for a Humanv4 array by simply replacing `v3` with `v4`.

```
library(illuminaHumanv4.db)

illuminaHumanv4()
###...
qual <- unlist(mget(ids, illuminaHumanv4PROBEQUALITY, ifnotfound=NA))

##etc....
```

It is possible to find a few outliers in the 'Bad' category that have consistently high expression. Some strategies for probe filtering would retain these probes in the analysis, so it is worth considering whether they are of value to an analysis.

Use Case: Investigate any IDs that have high expression despite being classed as 'Bad'.

```
> AveSignal <- rowMeans(exprs(normData))
> queryIDs <- names(which(qual == "Bad" & AveSignal > 13.5))
> unlist(mget(queryIDs, illuminaHumanv3REPEATMASK))
> unlist(mget(queryIDs, illuminaHumanv3SECONDMATCHES))[1:5]
```

☞ If you know how to, you could manually BLAT particular sequences (e.g. using UCSC genome browser) and check their mappings for yourself

Use Case: (OPTIONAL) Retrieve probe sequences for some dubious probes and manually BLAT them against the genome using the UCSC genome browser

```
> mget("ILMN_1692145", illuminaHumanv3PROBESEQUENCE)
```

Differential expression analysis

The differential expression methods available in the limma package can be used to identify differentially expressed genes. The functions `lmFit`, `contrasts.fit` `eBayes` can be applied to the normalized and filtered data.

Use Case: Fit a linear model to summarize the values from replicate arrays and compare UHRR with Brain Reference by setting up a contrast between these samples. Assess array quality using empirical array weights and incorporate these in the final linear model. Is there strong evidence of differential expression between these samples?

```
> rna <- pData(normData)$Sample_Group
> design <- model.matrix(~0 + as.factor(rna))
> colnames(design) <- levels(rna)
> aw <- arrayWeights(exprs(normData.filt), design)
> aw
> fit <- lmFit(exprs(normData.filt), design, weights = aw)
```

```
> contrasts <- makeContrasts(UHRR - Brain, levels = design)
> contr.fit <- eBayes(contrasts.fit(fit, contrasts))
> topTable(contr.fit, coef = 1)
> par(mfrow = c(1, 2))
> volcanoplot(contr.fit, main = "UHRR - Brain")
> smoothScatter(contr.fit$Amean, contr.fit$coef, xlab = "average intensity",
+     ylab = "log-ratio")
> abline(h = 0, col = 2, lty = 2)
```

The code above shows how to set up a design matrix for this experiment to combine the data from the UHRR and Brain Reference replicates to give one value per condition. Empirical array quality weights [10] can be used to measure the relative reliability of each array. A variance is estimated for each array by the `arrayWeights` function which measures how well the expression values from each array follow the linear model. These variances are converted to relative weights which can then be used in the linear model to down-weight observations from less reliable arrays which improves power to detect differential expression.
We then define a contrast comparing UHRR to Brain Reference and calculate moderated t-statistics with empirical Bayes' shrinkage of the sample variances.
For more information about the `lmFit`, `contrasts.fit` and `eBayes` functions, refer to the limma documentation.

Annotating the results of a differential expression analysis

The `topTable` function displays the results of the empirical Bayes analysis alongside the IDs assigned by Illumina to each probe in the linear model fit. Obviously this will not provide sufficient information to infer biological meaning from the results. Within Bioconductor, annotation packages are available for each of the major Illumina expression array platforms that map the probe sequences designed by Illumina to functional information useful for downstream analysis. As before, the illuminaHumanv3.db package can be used for the arrays in this example dataset. The `addFeatureData` function provides a convenient way to add some commonly used annotations to an existing object.

Use Case: Retrieve annotation information for the normalised data and add this information to the limma results. Write the results of the limma analysis out to disk.

```
> normData.filt <- addFeatureData(normData.filt)
> head(fData(normData.filt))
> anno <- fData(normData.filt)
> contr.fit$genes <- anno
> topTable(contr.fit)
> write.fit(contr.fit, file = "maqcresults.txt")
```

Wrap-up

The version of `R` and the packages used to complete this tutorial are listed below. If you have further questions about using any of the Bioconductor packages used in this tutorial, please email the Bioconductor mailing list (`bioconductor@stat.math.ethz.ch`).

```
> sessionInfo()
```

```
R version 3.0.1 (2013-05-16)
Platform: x86_64-pc-linux-gnu (64-bit)

locale:
 [1] LC_CTYPE=en_GB.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=en_GB.UTF-8        LC_COLLATE=en_GB.UTF-8
 [5] LC_MONETARY=en_GB.UTF-8    LC_MESSAGES=en_GB.UTF-8
 [7] LC_PAPER=C                 LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_GB.UTF-8 LC_IDENTIFICATION=C

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods
[7] base

loaded via a namespace (and not attached):
[1] tools_3.0.1
```

Acknowledgements

We thank James Hadfield and Michelle Osbourne for generating the HT-12 data used in the first section and the attendees of the various courses we have conducted on this topic for their feedback, which has helped improve this document.

References

[1] MAQC Consortium. The MicroArray Quality Control (MAQC) project shows inter- and intraplatform reproducibility of gene expression measurements. *Nat Biotechnol*, 24(9):1151–61, September 2006.

[2] M.L.Smith and A.G.Lynch. BeadDataPackR: A Tool to Facilitate the Sharing of Raw Data from Illumina BeadArray Studies. *Cancer Inform*, 9:217–27, 2010.

[3] J.˜M. Cairns, M.˜J. Dunning, M.˜E. Ritchie, R.˜Russell, and A.˜G. Lynch. BASH: a tool for managing BeadArray spatial artefacts. *Bioinformatics*, 24(24):2921–2, 2008.

[4] Mayte Suarez-Farinas, Asifa Haider, and Knut Wittkowski. "harshlighting" small blemishes on microarrays. *BMC Bioinformatics*, 6(1):65, 2005.

[5] W.˜Shi, C.˜A. de˜Graaf, S.˜A. Kinkel, A.˜H. Achtman, T.˜Baldwin, L.˜Schofield, H.˜S. Scott, D.˜J. Hilton, and G.˜K. Smyth. Estimating the proportion of microarray probes expressed in an RNA sample. *Nucleic Acids Res*, 38(7):2168–76, 2010.

[6] R.A. Irizarry, B.˜Hobbs, F.˜Collin, Y.˜D. Beazer-Barclay, K.J. Antonellis, U.˜Scherf, and T.P. Speed. Exploration, normalization, and summaries of high density oligonucleotide array probe level data. *Biostatistics*, 4(2):249–64, 2003.

[7] M.˜E. Ritchie, J.˜Silver, A.˜Oshlack, M.˜Holmes, D.˜Diyagama, A.˜Holloway, and G.˜K. Smyth. A comparison of background correction methods for two-colour microarrays. *Bioinformatics*, 23(20):2700–7, 2007.

[8] W.˜Shi, A.˜Oshlack, and G.˜K. Smyth. Optimizing the noise versus bias trade-off for Illumina Whole Genome Expression BeadChips. *Nucleic Acids Res*, 38(22):e204, 2010.

[9] N.˜L. Barbosa-Morais, M.˜J. Dunning, S.˜A. Samarajiwa, J.˜F. Darot, M.˜E. Ritchie, A.˜G. Lynch, and S.˜Tavaré. A re-annotation pipeline for Illumina BeadArrays: improving the interpretation of gene expression data. *Nucleic Acids Res*, 38(3):e17, 2010.

[10] M.˜E. Ritchie, D.˜Diyagama, J.˜Neilson, R.˜van Laar, A.˜Dobrovic, A.˜Holloway, and G.˜K. Smyth. Empirical array quality weights in the analysis of microarray data. *BMC Bioinformatics*, 19(7):261, 2006.