# BeadArray expression analysis using Bioconductor

Mark Dunning, Wei Shi, Andy Lynch, Mike Smith and Matt Ritchie

May 3, 2018

## Introduction to this vignette

This vignette describes how to process Illumina BeadArray gene expression data in its various formats; raw files produced by the scanning software as well as summarized BeadStudio/GenomeStudio output, and data deposited in a public microarray database. For each file format we introduce a series of 'use cases' in the order in which they might be encountered by an analyst, and demonstrate how to perform specific tasks on the example datasets using R code from Bioconductor or base packages. The R code is explained immediately afterwards and where appropriate we given an interpretation of the resultant output (indicated by the ☑ symbol), and pointers (☞) to how the R code might be adapted to other datasets or use cases. Where attention must be paid to avoid alarming results, we give warnings indicated by the 🔔 symbol.
It is assumed that the reader has familiarity with basic R data structures and operations such as plotting and reading text files, therefore explanations of these functions will not be given.

The version of R used to compile this vignette is 3.5.0. The Bioconductor packages required can be installed by typing the following commands at the R command prompt:

```
1 > source("http://www.bioconductor.org/biocLite.R")
2 > biocLite(c("beadarray", "limma", "GEOquery", "illuminaHumanv1.db",
3 +     "illuminaHumanv2.db", "illuminaHumanv3.db", "BeadArrayUseCases"))
```

There are also a few packages that are used to illustrate particular use case, but are not crucial to the vignette. They can be installed in advance, or as required.

```
1 > biocLite(c("GOstats", "GenomicRanges", "Biostrings"))
```

## Introduction to the BeadArray technology

Illumina Inc. (San Diego, CA) are manufacturers of the BeadArray microarray technology, which can be used in genomic studies to profile transcript expression or methylation status, or genotype SNPs. The BeadArray platform makes use of 50mer oligonucleotide probes attached to beads that are randomly assembled and replicated many times on each array. Different probes are designed to target specific positions (transcripts/SNPs) in the genome. A series of decoding hybridizations performed in-house by Illumina determines the identity of each bead

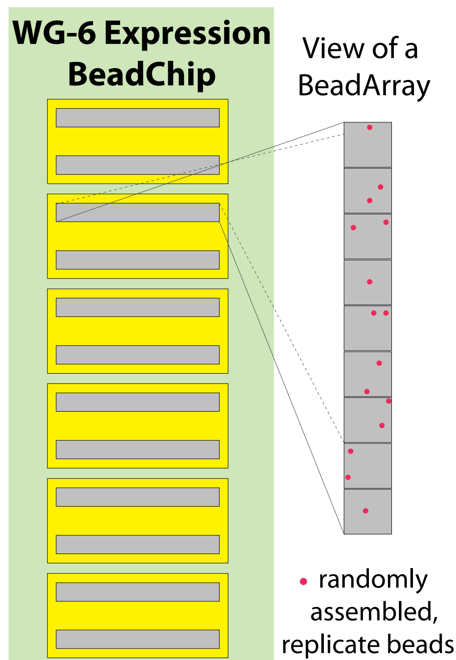



Figure 1: Schematic of a WG-6 BeadChip and BeadArray section. This type of BeadChip is made up of 12 sections (also referred to as 'strips') which are paired to allow 6 samples to be processed in parallel per chip. Sections are densely packed with beads which are randomly assembled on the array surface. Another feature of the technology is the high degree of replication of beads assigned a particular probe sequence (from a possible set of $\sim$ 48,000 unique sequences).

on each array.

A BeadChip comprises of a series of rectangular BeadArray sections on a slide, with each section (sometimes referred to as a strip) containing many thousands of different probes (see Figure 1). The naming of a chip is decided by the number of unique hybridizations possible and the version of the probe annotation. For example, there are six pairs of sections on each Human WG-6 version 2 BeadChip, and 12 sections, one per sample on a Human HT-12 version 3 BeadChip. The high degree of replication makes robust measurements for each probe possible and provides the opportunity to detect and correct for spatial artefacts that occur on the array surface. Illumina's gene expression arrays also contain many control probes, and one particular class of these, known as 'negative' controls can be used in the analysis to improve inference.

## Experimental data

We make use of three datasets in this vignette (summarized in Table 1) to illustrate how data in different formats that have undergone varying degrees of preprocessing can be imported and analyzed using Bioconductor software.

The first example 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 file `beadlevelbabfiles.zip` which is available as part of the BeadArrayUseCases package. These data are in the compressed `.bab` format [2], which can be analysed using the beadarray package. For one array (4613710052_B) we also provide the uncompressed data including the raw TIFF image. This allows us to demonstrate the processing options available when pixel level information is at hand.

The second dataset is the `AsuragenMAQC_BeadStudioOutput.zip` file from Illumina's website `http://www.switchtoi.com/datasets.ilmn`
This experiment uses a Human WG-6 version 2 BeadChip, and consists of 3 replicates of each of the UHRR and Brain Reference samples. These data are available at the summary level as generated by Illumina's BeadStudio software. The Bioconductor packages lumi, limma and beadarray can all handle data in this format. For this dataset, we focus on tools available in the limma package.

The final dataset, which is also at the summary level, is publicly available from the GEO database (accession GSE5350) and was deposited by the MAQC project [1]. This experiment included pure UHRR and Brain Reference RNA samples as well as mixtures of these two samples (the mixture proportions were 75:25 and 25:75) hybridized to Human WG-6 version 1 BeadChips. We retrieve this experiment from the GEO database using the GEOquery package.

The goal of each analysis is to find differentially expressed probes between the two distinct RNA samples included in each experiment (UHRR and Brain Reference). For the differential expression analysis, we make use of the linear modelling approach available in the limma package.

Table 1: Summary of the datasets analyzed in the vignette.

| | | | Number of arrays | | Number of array sections | |
|---|---|---|---|---|---|---|
| Dataset | Type of data | Array Generation | UHRR | Brain | UHRR | Brain |
| 1 | raw/bead-level | HT-12 version 3 | 6 | 6 | 6 | 6 |
| 2 | summary (BeadStudio) | WG-6 version 2* | 3 | 3 | 6 | 6 |
| 3 | summary (GEO) | WG-6 version 1 | 15 | 15 | 30 | 30 |

* Note that these data can not be analyzed at the section-level (raw or bead-level data would be required for this).

# 1 Analysis of bead-level and raw data using beadarray

## Raw and bead-level data types

Reading raw or bead-level data into `R` using the beadarray package requires several files produced by Illumina's scanning software. We briefly describe these files below.

- text files (*required - unless bab files present*) - a text file (`.txt` or `.csv`) for each section which stores the position, identity and intensity of each bead. These files are usually named `chipID_section.txt` for arrays from BeadChips (e.g. `4613710017_B.txt`) and are required because of the random arrangement of probes on the array surface that is unique for each BeadArray.

- locs files (*optional*) - 1 (single channel) or 2 (two-color) for each section on a BeadChip. These are usually named using the convention `chipID_section_channel.locs`. The locs file stores the locations of all beads on the array, including all those that could not be decoded (beads present on the array, but not in the text files). The locs files are particularly useful for investigating spatial phenomena on the arrays.

- bab files (*optional*) - one for each section of a BeadChip. These files contain all of the information from the text and locs files, stored in a substantially more efficient manner [2].

- tiff files (*optional*) - 1 (single channel) or 2 (two-color) for each section on a BeadChip. These are usually named using the convention `chipID_section_channel.tif`. For example, `4613710017_B_Grn.tif` is the Cy3 (green) image for the sample in position B on BeadChip 4613710017. The Cy5 (red) files end in `_Red.tif`. We refer to these as the raw data, as access to these images allows the user to carry out their own image processing, and provides access to the background intensities (the values stored in the text files have already been background corrected).

- sdf file (*optional*) - Illumina's sample descriptor file (one per BeadChip, e.g. `4613710017.sdf`). This file is used to determine the physical properties of a section and which sections to combine for each sample.

- targets file (*recommended*) - contains sample information for each array. See the file `targetsHT12.txt` for an example.

- metrics file (*optional*) - one for each BeadChip (usually named `Metrics.txt`) which contains summary information about intensity, the amount of saturation, focus and registration on the image(s) from each section.

♠ *To obtain the tiff and text files from Illumina's BeadScan software version 3.1 you will need to modify the* `settings.xml` *file used by the software. For further details see the Scanning Settings section of* `http://www.compbio.group.cam.ac.uk/Resources/illumina/`*. For iScan, the steps are similar, although there is a separate settings file for each platform (expression, Infinium genotyping, etc).*

### 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.

```
1  > ht12metrics <- read.table(system.file("extdata/Chips/Metrics.txt",
2  +     package = "BeadArrayUseCases"), sep = "\t", header = TRUE,
3  +     as.is = TRUE)
4  > ht12snr <- ht12metrics$P95Grn/ht12metrics$P05Grn
5  > labs <- paste(ht12metrics[, 2], ht12metrics[, 3], sep = "_")
6  > par(mai = c(1.5, 0.8, 0.3, 0.1))
7  > plot(1:12, ht12snr, pch = 19, ylab = "P95 / P05", xlab = "",
8  +     main = "Signal-to-noise ratio for HT12 data", axes = FALSE,
9  +     frame.plot = TRUE)
10 > axis(2)
11 > 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.

☑ The SNR for these arrays seems to be over 15 in most cases, although there is one exception that is below 2. Illumina recommend that this ratio be above 10 and in our experience a value below 2 would be grounds for discarding a sample from further analysis. The P95 value for this sample is typical of what we would expect from a blank array with no RNA hybridized!

☞ 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 next 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 file `beadlevelbabfiles.zip` (133 MB) which is located in the `extdata/BeadLevelBabFiles` 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.

**Use Case:** Read the sample information and bead-level data (stored in compressed bab format) into `R`.

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

Usually the directory will be in a known location, but for convenience we use the `system.file` function this directory and the sample sheet. The section names to be read are then constructed using the contents of the sample sheet.
The final line executes the reading of the data. By default `readIllumina` will look in the current working directory and try to find all BeadArray-associated files. The function will read any text (or `.bab`) and TIFF files (if instructed to do so) and save the names of any locs and sdf files for future reference.
Here we have used the `dir` and `sectionNames` arguments to specify what directory to look in and the names of the sections that should be imported. 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.

♠ *Use of the `sectionNames` argument is recommended when the directory containing bead-level data contains files other than those produced by the Illumina scanner. Extraneous files in such directories may confuse the `readIllumina` function.*

♣ *Storing and reading bead-level data requires a considerable amount of disk space and RAM. For this example, around 1.1 GB of RAM is required to read in and store the data. One could process bead-level data in smaller batches if memory is limited and then combine at the summary-level.*

### Selecting 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"
```

☑ You should see that the percentage of overlapping IDs is greatest for the `Humanv3` platform.

☞ 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.

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` vector 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
```

```
function (BLData, array)
{
    x = getBeadData(BLData, array = array, what = "Grn")
    return(log2.na(x))
}
<bytecode: 0x12208b78>
<environment: namespace:beadarray>
```

☞ 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`).

## Analysis of raw data when the images are available

The bead-level expression intensity values that Illumina's software provides (i.e. those stored in the `.txt` or `.bab` files) are the result of a certain amount of preprocessing and so are not strictly the raw data. In most situations, these values are sufficient for our use, but we may on occasion wish to begin from the image file, either to reassure ourselves that there are no concerns or to address a problem that has become manifest.

It is important then that we understand what preprocessing steps Illumina apply by default. These are well-documented elsewhere, but to summarize: a local background value is calculated by taking the mean of the five lowest intensity pixels from a square around the bead. A filter is then applied to the image (to concentrate the intensities in the centre of beads) and foreground values are calculated as a weighted sum of the intensities in a $4 \times 4$ square around the bead centre. It is worth noting that the filter applied to the image, a sharpening filter, contrasts the value of a pixel with the pixels surrounding it, and as such itself could be viewed as a background correction step. The final intensity for a bead is then calculated as the foreground value minus the local background value.

By starting from the image we can adjust any of these steps (e.g. for the background we can adjust the size of the area around the bead or the function applied to it, for the foreground we can change the filtering step or adjust the pixel weighting scheme, or finally we can use a more

sophisticated measure of intensity than 'foreground minus local background' to avoid negative values. beadarray includes three functions that closely mirror the processing performed by Illumina: `illuminaBackground`, `illuminaSharpen` and `illuminaForeground`.
We will illustrate a change to the Illumina process that we recommend if beginning from the image.

**Use Case:** Identify abnormally low intensity pixels and then plot a section of the image that illustrates the benefit of adjusting the standard analysis.

```
1  > TIFF <- readTIFF(system.file("extdata/FullData/4613710052_B_Grn.tif",
2  +     package = "BeadArrayUseCases"))
3  > cbind(col(TIFF)[which(TIFF == 0)], row(TIFF)[which(TIFF ==
4  +     0)])
5  > xcoords <- getBeadData(data, array = 2, what = "GrnX")
6  > ycoords <- getBeadData(data, array = 2, what = "GrnY")
7  > par(mfrow = c(1, 3))
8  > offset <- 1
9  > plotTIFF(TIFF + offset, c(1517, 1527), c(5507, 5517),
10 +     values = T, textCol = "yellow", main = expression(log[2](intensity +
11 +         1)))
12 > points(xcoords[503155], ycoords[503155], pch = 16, col = "red")
13 > plotTIFF(TIFF + offset, c(1202, 1212), c(13576, 13586),
14 +     values = T, textCol = "yellow", main = expression(log[2](intensity +
15 +         1)))
16 > points(xcoords[625712], ycoords[625712], pch = 16, col = "red")
17 > plotTIFF(TIFF + offset, c(1613, 1623), c(9219, 9229),
18 +     values = T, textCol = "yellow", main = expression(log[2](intensity +
19 +         1)))
20 > points(xcoords[767154], ycoords[767154], pch = 16, col = "red")
```

☑ There are three pixels of value zero in the image that must be an imaging artefact rather than a true measure of intensity for that location (note that we add an offset of 1 to avoid taking the log of zero). These pixels will bias the estimate of background for all nearby beads and so we will try a more robust estimate of the background. Rather than using the mean of the five lowest pixel values, we will use the median (equivalently, the third lowest pixel value). This is done using the `medianBackground` function.

**Use Case:** Calculate a robust measure of background for this array and store it in a new slot "GrnRB".

```
1  > Brob <- medianBackground(TIFF, cbind(xcoords, ycoords))
2  > data <- insertBeadData(data, array = 2, what = "GrnRB",
3  +     Brob)
```

The `medianBackground` function takes two arguments, the first of which is the image itself and the second is a two-column data frame specifying the bead-centre coordinates. We then use `insertBeadData` to add the new values to the existing *beadLevelData* object.

Because the presence of extremely low intensity beads is a known issue, the authors have provided the `medianBackground` function within beadarray to perform an alternative local

background correction. However the same methodology can be used to perform the image processing in any way the user sees fit.

**Use Case:** Calculate foreground values in the normal way, and subtract the median background values to get locally background corrected intensities.

```
1  > TIFF2 <- illuminaSharpen(TIFF)
2  > IllF <- illuminaForeground(TIFF2, cbind(xcoords, ycoords))
3  > data <- insertBeadData(data, array = 2, what = "GrnF",
4  +     IllF)
5  > data <- backgroundCorrectSingleSection(data, array = 2,
6  +     fg = "GrnF", bg = "GrnRB", newName = "GrnR")
```

We could have chosen a more sophisticated background correction method (approximately a hundred beads end up with a negative intensity using a crude subtraction) but this suffices to illustrate the point. The majority of the beads have an intensity that barely changes, and those that do we can attribute to the effect of these problematic pixels.

**Use Case:** Compare the Illumina intensity with the robust intensity we have just calculated, plot the locations of beads whose expressions change substantially, and overlay the locations of the implausibly low-intensity pixels in red.

```
1  > oldG <- getBeadData(data, array = 2, "Grn")
2  > newG <- getBeadData(data, array = 2, "GrnR")
3  > summary(oldG - newG)
4  > par(mfrow = c(1, 2))
5  > plot(xcoords[(abs(oldG - newG) > 50)], ycoords[(abs(oldG -
6  +     newG) > 50)], pch = 16, xlab = "X", ylab = "Y", main = "entire array")
7  > points(col(TIFF)[TIFF < 400], row(TIFF)[TIFF < 400],
8  +     col = "red", pch = 16)
9  > plot(xcoords[(abs(oldG - newG) > 50)], ycoords[(abs(oldG -
10 +     newG) > 50)], pch = 16, xlim = c(1145, 1180), ylim = c(15500,
11 +     15580), xlab = "X", ylab = "Y", main = "zoomed in")
12 > points(col(TIFF)[TIFF < 400], row(TIFF)[TIFF < 400],
13 +     col = "red", pch = 16)
```

Of course, in practice we would probably save the new intensities in the `Grn` column of a section in the `beadData` slot so as not to use more memory than necessary, nor to cause confusion later on.

The image file may also be of value for the purposes of quality control and assessment. In particular, by plotting the recorded co-ordinates of the bead centres over the TIFF image, we can visually check that the image has been correctly registered, however there are other approaches to checking this as will be seen in the next section.

## 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.

```
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 have 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.

☑ Most arrays have a similar distribution of intensities with a median value of around 5.7. Array 4616443081_B has a lower median and IQR than other arrays and 4616443081_H has a higher median and IQR. Further quality assessment will focus on these arrays and whether they should be excluded from the analysis.

### 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 a graphic with imageplots of all arrays in the dataset, with each image on the same scale.

```
1 > par(mfrow = c(6, 2))
2 > par(mar = c(1, 1, 1, 1))
3 > for (i in 1:12) {
4 +     imageplot(data, array = i, high = "darkgreen", low = "lightgreen",
5 +         zlim = c(4, 10), main = sectionNames(data)[i])
6 + }
```

The `imageplot` function has a number of arguments; the first two shown above are the object containing the data to be visualized and the index of the desired array. The `high` and `low` arguments specify the colors representing the extreme values. The function automatically interpolates the colors for values in between. The use here of `zlim` ensures that the color

range is the same between arrays. Any value that falls outside this range will be shown in the same color as these limits. This is beneficial for making comparisons between arrays, and can prevent minor variations, on otherwise perfectly acceptable arrays, from being exaggerated. By contrast, we need to be cautious that the use of `zlim` may be detrimental if the same limits are not appropriate for each array, which could lead to some spatial artefacts being overlooked. This is especially the case for our example, since these arrays have not been normalized.

☑ White patches on the imageplot are due to beads that could not be (or were not) decoded for the end-user by Illumina. As each array is randomly constructed, decoding takes place at Illumina before the BeadChips are supplied, in order to identify the sequence attached to each bead. Beads that could not be decoded are not present in the bead-level text files (nor do they contribute to Illumina's summary data). Hence their intensities are not available for display on the imageplot. You should notice obvious spatial artefacts on arrays 8 and 12 (4616443081_H and 4616494005_A).

### Plotting the location of outliers

Recall that the BeadArray technology includes many replicates (typically $\sim 20$ 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.

```
1  > par(mfrow = c(2, 1))
2  > for (i in c(8, 12)) {
3  +     outlierplot(data, array = i, main = paste(sectionNames(data)[i],
4  +         "outliers"))
5  + }
```

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`.

☑ In this example, the regions of the arrays that appear to be spatial artefacts are also flagged as outliers, which would be removed before creating summarized values for each probe as reported in BeadStudio/GenomeStudio. Blue points represent outliers which are below average, while pink points are outliers above the average (these colors can be set by the user - refer to the `outlierplot` help page for details).
The dotted red lines running vertically indicate the segment boundaries, with each `Humanv3` section made up of 9 segments that are physically separated on the section surface. The locations of these segments are taken from an sdf file, or can be specified manually if the sdf file is not available.

### 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.

**Use Case:** Run `BASH` on the two arrays identified previously to have the most serious spatial artefacts, mask the affected beads and visualize the regions that have been excluded. How many beads does BASH mask on the two arrays?

```
1  > BASHoutput <- BASH(data, array = c(8, 12))
2  > data <- setWeights(data, wts = BASHoutput$wts, array = c(8,
3  +     12))
4  > head(data[[8]])
5  > par(mfrow = c(1, 2))
6  > for (i in c(8, 12)) {
7  +     showArrayMask(data, array = i, override = TRUE)
8  + }
9  > table(getBeadData(data, what = "wts", array = 8))
10 > table(getBeadData(data, what = "wts", array = 12))
11 > BASHoutput$QC
```

Line 1 calls the `BASH` function, with the `array` argument specifying that it should be run on arrays 8 and 12. If this argument was not specified the default behaviour is to process each array. As mentioned above, the output from `BASH` is a list with several entries. The `$wts` entry is a further list, where each entry is a vector of weights (one value per-bead) with 0 indicating a that a bead should be masked.

We then use the `setWeights` function to assign these weights to our `beadLevelData` object. Line 4 shows how the `setWeights` function has added an additional column to the specified array in the *beadLevelData* object.

Finally 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. Finally, we can use the `getBeadData` function to see how many beads were assigned a weight of 0 (i.e. completely masked) on the arrays. The numbers can also retrieved from the QC information that BASH returns.

☑ You should see that the `setWeights` has added an extra `wts` column into the `beadLevelData` object. The positions of the masked beads are indicated in red in the result of `showArrayMask` and should agree well with the outlier locations, however BASH should have identified more beads than straightforward outlier removal may have missed.

♣ *Running BASH for this example uses around 2.2 GB of RAM and takes 10 minutes per sample on our computer system. If you are running short on time, you may wish to skip this exercise.*

☞ The different components of BASH to find compact or diffuse defects plus the extended score analysis can be run separately; see the `BASHCompact`, `BASHExtended` and `BASHExtended` functions.

☞ Try running the `BASHExtended` function for some of the other arrays in this dataset. e.g.

```
BASHExtended(data, array=1)
```

You should see extended scores of around 0.1.

### Removing intensity gradients

The *Extended* score returned by `BASH` in the previous use case gives an indication of the level of variability across the entire surface of the chip. If this value is large it may indicate a significant gradient effect in the intensities. The `HULK` function can be used to smooth out any gradients that are present.
`HULK` uses information about neighbouring beads, but rather than mask them out as in `BASH`, it adjusts the log-intensities by the weighted average of residual values within a local neighbourhood.

**Use Case:** Run `HULK` on the first array, and replace the original intensities with the gradient adjusted values.

```
1 > HULKoutput <- HULK(data, array = 1, transFun = logGreenChannelTransform)
2 > data <- insertBeadData(data, array = 1, data = HULKoutput,
3 +     what = "GrnHulk")
```

Similar to `BASH`, the primary argument to `HULK` is a *beadLevelData* object. It also takes a transformation function, allowing the intensity adjustment to be performed on any data stored within the object.

♠ *Typically, we would run BASH followed by HULK on a dataset. However, the order in which one does BASH and HULK could be a topic for research. In cases of severe gradients across the array, you might get a lot of beads masked at one edge of the array. However, if HULK were run first these beads might be saved.*

### Checking image registration

Tools such as `BASH` and `HULK` are of no use if the process of generating the image and finding beads as the array is scanned (known as 'registration') fails. We have previously encountered arrays where the position of the beads within the image was not found correctly [5], resulting in the identity of all beads being scrambled. The function `checkRegistration` can be used to identify chips where such mis-registration may have taken place.

♠ *This functionality is only available in* beadarray *version 2.3.0 or newer*

```
1 > registrationScores <- checkRegistration(data, array = c(1,
2 +     7))
3 > boxplot(registrationScores, plotP95 = TRUE)
```

The `checkRegistration` function takes a *beadLevelData* object as it's primary argument, along with the indices of the array sections that should be checked. The output from `checkRegistration` is an object of class *beadRegistrationData*. We have extended the standard `boxplot` function to accept this class, providing an easy way to visualise the registration scores.

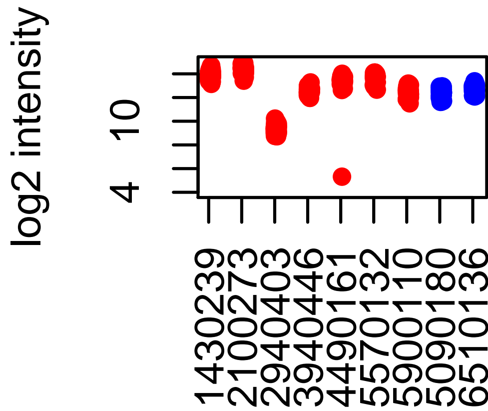
☑ The registration scores are generated by looking at the the difference between the within bead-type variance for the given bead IDs and a randomised set of IDs. If the registration has been sucessful you expect this value to be greater than zero. Any section where the median registration score is close to zero is of concern. There are two reasons why a median value of zero may be seen, either then array was misregistered or there are no beads present in the image. The plotP95 argument shown above allows these cases to be distinguished.

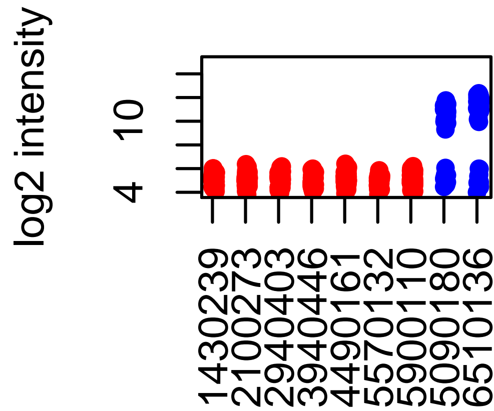
### Plotting control probes

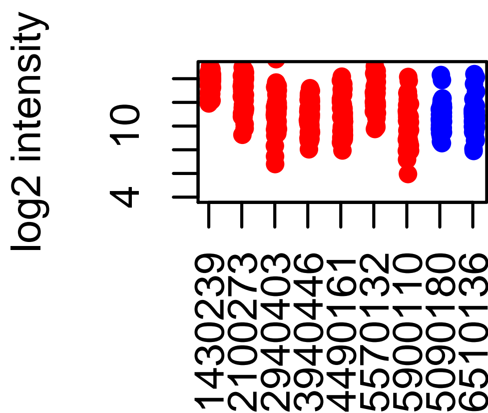
Illumina have designed a number of control probes for each BeadArray platform. Two particular controls on expression arrays are housekeeping and biotin controls, which are expected to be highly expressed in any sample. With the `poscontPlot` function, we can plot the intensities of any ArrayAddressIDs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

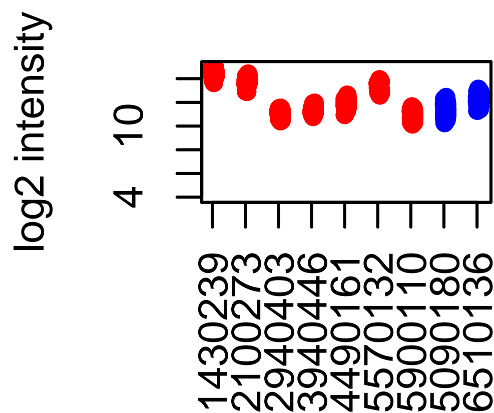
**Use Case:** Generate plots of the housekeeping and biotin controls for the 6th, 7th, 8th and 12th arrays from this dataset.

```
1 > par(mfrow = c(2, 2))
2 > for (i in c(6, 7, 8, 12)) {
3 +     poscontPlot(data, array = i, main = paste(sectionNames(data)[i],
4 +         "Positive Controls"), ylim = c(4, 15))
5 + }
```




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.

☑ For the good quality array, the control probes are highly-expressed as expected. For the array with poor scanner metrics, all the housekeeping beads are lowly expressed but the Biotin controls are highly-expressed, indicating successful staining, but unsuccessful hybridization. For the arrays with severe artefacts, 4616443081_H has a large spread of values for the control probes, indicating that much of the array is affected by the artefact. On the other hand, 4616494005_A shows high expression of the control probes, giving hope that this array can be used in further analysis.

### Producing control tables

With knowledge of which 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.

☑ You should notice that the housekeeping controls are lower for array 7, as we have noted in previous quality assessments.

### 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).

### Defining additional control probes

beadarray allows flexibility in the way that control reports are generated. For instance, users

are able to define their own control reporters. We have observed that certain probes located on the Y chromosome are beneficial in discriminating the gender of samples in a population. Below we provide an example of how these probes can be incorporated into a QC report. This utilizes the fact that internally, beadarray uses a very simple matrix to assign ArrayAddressIDs to control types.

```
1 > cprof <- makeControlProfile("Humanv3")
2 > sexprof <- data.frame(ArrayAddress = c("5270068", "1400139",
3 +     "6860102"), Tag = rep("Gender", 3))
4 > cprof <- rbind(cprof, sexprof)
5 > makeQCTable(data, controlProfile = cprof)
```

Line 1 obtains the control information currently in use for the Humanv3 platform. We then create a new data frame with three rows, each representing a probe from chromosome Y, and row-bind this to the Humanv3 control object. The `makeQCTable` is able to accept the new object as an argument, rather than using the default Humanv3 profile.

### Generating QC reports

The generation of quality assessment plots for all sections in the *beadLevelData* object is provided by the `expressionQCPipeline` function. Results are generated in a directory of the users choosing. This report may be generated at any point of the analysis. If the `overWrite` paramater is set to `FALSE`, then any existing plots in the directory will not be re-generated. Futhermore, QC tables that have been stored in the *beadLevelData* object already can be used.

**Use Case:** Generate a QC report for all arrays.

```
1 > expressionQCPipeline(data)
```

The above code runs the QC pipeline with the default options. However, many aspects of the `expressionQCPipeline` function are configurable, such as the transformation function applied to the data, or the identities of the controls.

♠ *The `expressionQCPipeline` function will attempt to create graphics and HTML files in the specified directory, so it is important that this directory is writable by the user.*

## 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. 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.

**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.

☑ `summarize` produces verbose output which firstly gives details on how many sections are to be combined (none in this case) and how many ArrayAddressIDs are to be summarized. Notice that we summarize all arrays in *beadLevelData* object, even though we may not use the poor quality arrays in the analysis. This is just for convenience as each array is summarized independently (so there is no way of the data from a poor quality array to contaminate the data from other arrays) and it is simpler to remove outlier arrays from the summarized objects.

☞ For WG-6 arrays, which have two sections per biological sample, BeadStudio combines the two sections together prior to calculating means and standard deviations. It is possible to mimic this behaviour by setting the `useSampleFac = TRUE` argument in `summarize`. This either uses information from the sdf file (if present) or the value of the `sampleFac` argument. However, we recommend summarizing each section separately.

☞ If we wanted to have the un-logged and $\log_2$ intensities in the same summary object, we could have used supplied both channels in a list as an argument to `summarize`.

```
datasumm.all <- summarize(data, list(grnchannel, grnchannel.unlogged),
 useSampleFac=FALSE)
```

☞ 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 (for a more complete definition, refer to section 2). 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.

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

The `dim` function has been extended to report the key dimensions of the data, namely the number of probes and samples. The expression matrix and associated probe-specific variability are returned by lines 2 and 3 (However, see the note below about the `se.exprs` function)
The `calculateDetection` function uses the annotation information stored in the *ExpressionSetIllumina* object to identify the negative control and do the detection calculation, giving a detection value for each probe on each array.

☑ The dimensions should be reported as 49,576 Features (probes) and 12 samples and 1 channel. The boxplot of the expression matrix should agree with your observations from the bead-level data. The boxplot showing the distribution of bead numbers used in the calculation of the summary values for each array can reveal arrays with significant spatial artefacts (arrays 8 and 12 in this case).

♠ *Pay attention to the scale on the y-axis. The median level of expression should be somewhere around 5 to 6 with the lowest values around 4. If the median level is around 2 to 3 you may have logged the data twice.*

♣ *The `se.exprs` function has been named to be consistent with existing Bioconductor functions. However, it may not always return standard errors as its name suggests. The data returned will depend on the definition of the illuminaChannel class. In the example above, the line*

```
>se.exprs(datasumm)[1:10,1:2]
```

*returns standard deviations, which must be divided by the sqrt of the `nObservations` values to report standard errors.*

☞ `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`).

## Concluding remarks

So far we have looked at the kinds of low-level analysis that are possible when you have access to the raw and bead-level data. Once quality assessment is complete and the probe intensities have been summarized, one can continue down the usual analysis path of normalizing between samples, and assessing differential expression using limma or other Bioconductor tools. We will leave this task for now and revisit it later.

## 2 Analysis of summary data from BeadStudio/GenomeStudio using **limma**

BeadStudio/GenomeStudio is Illumina's proprietary software for analyzing data output by the scanning system (BeadScan/iScan). It contains different modules for analyzing data from different platforms. For further information on the software and how to export summarized data, refer to the user's manual. In this section we consider how to read in and analyze output from the gene expression module of BeadStudio/GenomeStudio.

The example dataset used in this section consists of an experiment with one Human WG-6 version 2 BeadChip. These arrays were hybridized with the control RNA samples used in the MAQC project (3 replicates of UHRR and 3 replicates of Brain Reference RNA).
The non-normalized data for regular and control probes was output by BeadStudio/GenomeStudio.
The example BeadStudio output used in this section is available in the file `AsuragenMAQC_BeadStudioOutput.zip` which can be downloaded from
tt http://www.switchtoi.com/datasets.ilmn.
You will need to download and unzip the contents of this file to the current `R` working directory. Inside this zip file you will find several files including summarized, non-normalized data and a file containing control information. We give a more detailed description of each of the particular files we will make use of below.

- Sample probe profile (`AsuragenMAQC-probe-raw.txt`) (*required*) - text file which contains the non-normalized summary values as output by BeadStudio. Inside the file is a data matrix with some 48,000 rows. In newer versions of the software, these data are preceded by several lines of header information. Each row is a different probe in the experiment and the columns give different measurements for the gene. For each array, we record the summarized expression level (AVG_Signal), standard error of the bead replicates (BEAD_STDERR), number of beads (Avg_NBEADS) and a detection $p$-value (Detection Pval) which estimates the probability of a gene being detected above the background level. When exporting this file from BeadStudio, the user is able to choose which columns to export.

- Control probe profile (`AsuragenMAQC-controls.txt`) (*recommended*) - text file which contains the summarized data for each of the controls on each array, which may be useful for diagnostic and calibration purposes. Refer to the Illumina documentation for information on what each control measures.

- targets file (*optional*) - text file created by the user specifying which sample is hybridized to each array. No such file is provided for this dataset, however we can extract sample annotation information from the column headings in the sample probe profile.

Files with normalized intensities (those with `avg` in the name), as well as files with one intensity value per gene (files with `gene` in the name) instead of separate intensities for different probes targeting the same transcript, are also available in this download. We recommend users work with the non-normalized probe-specific data in their analysis where possible. Illumina's

background correction step, which subtracts the intensities of the negative control probes from the intensities of the regular probes, should also be avoided.

**Use Case:** Read the example files using the `read.ilmn` function. Work out how many different classes of probes are present on these Human arrays.

```
1  > library(limma)
2  > maqc <- read.ilmn(files = "AsuragenMAQC-probe-raw.txt",
3  +     ctrlfiles = "AsuragenMAQC-controls.txt", probeid = "ProbeID",
4  +     annotation = "TargetID", other.columns = c("Detection Pval",
5  +         "Avg_NBEADS"))
6  > dim(maqc)
7  > maqc$targets
8  > maqc$E[1:5, ]
9  > table(maqc$genes$Status)
```

The `files` argument is the only compulsory argument in `read.ilmn`. The directory where the probe profile or control files is located can be specified using the `path` and `ctrlpath` arguments respectively.

☑ The intensities are stored in an object of class `EListRaw` defined in the limma package. The dimensions of this object should be 50,127 rows and 6 columns; in other words, there are 50,127 probes and 6 arrays. Each row in the object has an associated status which determines if the intensities in the row are for a control probe, or a regular probe that we might want to use in an analysis. The `maqc$targets` data frame indicates the biological samples hybridized to each array.

### 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 [6]. 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?

```
1  > proportion <- propexpr(maqc)
2  > proportion
3  > t.test(proportion[1:3], proportion[4:6])
```

☞ Systematic differences exist between different BeadChip versions, so these proportions should only be compared within a given platform type [6]. 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.

☑ The UHRR and Brain Reference samples used in this experiment have a similar proportion of expressed probes.

### Background correction and normalization

A second use for the negative controls is in the background correction and normalization of the data [7]. The normal-exponential convolution model has proven useful in background correction of both Affymetrix [8] and two-color data [9]. 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 [7]. The `neqc` function [7] 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:** Apply the `neqc` function to calibrate the background level, normalize and transform the intensities from each sample. Make boxplots of the regular probes and negative control probes before normalization and the regular probes after normalization and assess whether the `neqc` procedure improves the consistency between different samples.

```
1  > maqc.norm <- neqc(maqc)
2  > dim(maqc.norm)
3  > par(mfrow = c(3, 1))
4  > boxplot(log2(maqc$E[maqc$genes$Status == "regular", ]),
5  +     range = 0, las = 2, xlab = "", ylab = expression(log[2](intensity)),
6  +     main = "Regular probes")
7  > boxplot(log2(maqc$E[maqc$genes$Status == "NEGATIVE",
8  +     ]), range = 0, las = 2, xlab = "", ylab = expression(log[2](intensity)),
9  +     main = "Negative control probes")
10 > boxplot(maqc.norm$E, range = 0, ylab = expression(log[2](intensity)),
11 +     las = 2, xlab = "", main = "Regular probes, NEQC normalized")
```

☑ The `neqc` preprocessed intensities are stored in an `EList` object in which the control probes have been removed, leaving us with 48,701 regular probes. In this dataset, the intensities are fairly consistent to begin with, so calibration, normalization and transformation with `neqc` does not dramatically change the intensities on any array.

☞ Data exported from BeadStudio/GenomeStudio may already be normalized, however we recommend where possible analyzing the non-normalized intensities, which can be normalized in `R`.

☞ Recall that there are 6 samples per WG-6 BeadChip. Boxplots allow within-BeadChip trends, such as intensity gradients from top to bottom of the chip to be assessed. Differences between BeadChips hybridized at different times may also be expected.

☞ The `neqc` function is also able to accept a matrix as an argument, rather than the limma `EListRaw` class. Therefore users who might have read the data using lumi or processed bead-

level data with beadarray (as per section 1) will still be able to use this processing method. However, the `status`, `negctrl` and `regular` arguments will need to be set appropriately. The beadarray package includes `neqc` as an option in its `normaliseIllumina` function.

♠ *When applying quantile normalization, it is assumed that the distribution in signal should be the same from each array. This assumption may be unreasonable in some experiments, and should be carefully checked with diagnostic plots.*

### Dealing with 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 RNA source.

```
1 > plotMDS(maqc.norm$E)
```

☑ In this experiment, the first dimension separates the UHRR samples from the Brain Reference samples. The second dimension separates replicate Brain Reference samples, indicating that these are less consistent than the UHRR samples. The scale for dimension 2 is much reduced compared to dimension 1, indicating that the underlying biological differences between the two RNA sources explains most of the between sample variation.

☞ The `plotMDS` function accepts a *matrix* as an argument so could be used on the expression matrix of an *ExpressionSetIllumina* object, extracted using the `exprs` function.

### 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. Annotation quality information is available from the platform specific annotation packages, which are discussed in more detail later.
The illuminaHumanv2.db annotation package provides access to the reannotation information provided by Barbosa-Morais *et al.* [10]. 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.

Four basic annotation quality categories ('Perfect', 'Good', 'Bad' and 'No match') are defined and have been shown to correlate with expression level and measures of differential expression. We recommend removing probes assigned a 'Bad' or 'No match' quality score after normalization. This approach is similar to the common practice of removing lowly-expressed probes, but with the additional benefit of discarding probes with a high expression level caused by non-specific hybridization.
The illuminaHumanv2.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:** Retrieve quality information from the Human v2 annotation package and verify that probes annotated as 'Bad' or 'No match' generally have lower signal. Exclude such probes from further analysis.

```
1  > library(illuminaHumanv2.db)
2  > illuminaHumanv2()
3  > ids <- as.character(rownames(maqc.norm))
4  > ids2 <- unlist(mget(ids, revmap(illuminaHumanv2ARRAYADDRESS),
5  +     ifnotfound = NA))
6  > qual <- unlist(mget(ids2, illuminaHumanv2PROBEQUALITY,
7  +     ifnotfound = NA))
8  > table(qual)
9  > AveSignal = rowMeans(maqc.norm$E)
10 > boxplot(AveSignal ~ qual)
11 > rem <- qual == "No match" | qual == "Bad"
12 > maqc.norm.filt <- maqc.norm[!rem, ]
13 > dim(maqc.norm)
14 > dim(maqc.norm.filt)
```

All mappings available in the illuminaHumanv2.db can be listed by the `illuminaHumanv2` function. The default keys for each mapping are Illumina IDs that have the prefix `ILMN`. As the row names in the `maqc.norm` object are numeric ArrayAddressIDs we first have to convert them. This is achieved by use of the `revmap` function which reverses the direction of standard mapping from Illumina ID to ArrayAddressID. The `mget` function can then be used to query the probe quality for our new IDs and return the result as a list, which we then convert to a vector. The `rowMeans` is a `base` function to calculate the mean for each row in a matrix. We then make a boxplot of the average signal using probe quality as a factor.

☑ The output of `illuminaHumanv2()` lists all mappings available in the package and these are divided into two sections. Firstly there are mappings that use the RefSeq ID assigned to each probe to map to genomic properties, and secondly there are mappings based on the probe sequence itself using the procedure described by Barbosa-Morais *et al.* [10].
You should see that the expression level of the probes annotated as 'No match' or 'Bad' are generally lower than other categories. After applying this filtering step, we are left with

31,304 probes (out of a possible 48,701 regular probes) for further analysis.

☞ The packages used to annotate Illumina BeadArrays have a very simple convention which is `illumina` followed by an organism name, followed by annotation version number. Hence, the above code could be executed for a Humanv3 array by simply replacing `v2` with `v3`.

```
library(illuminaHumanv3.db)

illuminaHumanv3()
###...
ids2 <- unlist(mget(ids, revmap(illuminaHumanv3ARRAYADDRESS), ifnotfound=NA))
##etc....
```

You may notice 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'.

```
1 > queryIDs <- names(which(qual == "Bad" & AveSignal > 12))
2 > unlist(mget(queryIDs, illuminaHumanv2REPEATMASK))
3 > unlist(mget(queryIDs, illuminaHumanv2SECONDMATCHES))
4 > mget("ILMN_1692145", illuminaHumanv2PROBESEQUENCE)
```

☑ You should see that probes annotated as 'Bad' hybridize to multiple places in the genome and often contain repetitive sequences; making their inclusion in the analysis questionable. The probe sequences themselves can be retrieved and subjected to manual BLAT search (e.g. using the UCSC genome browser). The above example (ILMN_1692145) will return many matches.

### Other data visualisation

Although we will concentrate on a differential expression analysis, there are many other common analysis tasks that could be performed once a normalized expression matrix is available.

**Use Case:** Pick a set of highly-variable probes and cluster the samples.

```
1 > IQR <- apply(maqc.norm.filt$E, 1, IQR, na.rm = TRUE)
2 > topVar <- order(IQR, decreasing = TRUE)[1:500]
3 > d <- dist(t(maqc.norm.filt$E[topVar, ]))
4 > plot(hclust(d))
```

We calculate the interquartile range (IQR) of each probe across all samples and then order to find the 500 (an arbitrary number) most variable. These probes are then used to cluster the data in a standard way.

**Use Case:** Make a heatmap to show the differences between the groups.

```
1  > heatmap(maqc.norm.filt$E[topVar, ])
```

♠ *Not all datasets will show such large differences between biological groups!*

### 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?

```
1  > rna <- factor(rep(c("UHRR", "Brain"), each = 3))
2  > design <- model.matrix(~0 + rna)
3  > colnames(design) <- levels(rna)
4  > aw <- arrayWeights(maqc.norm.filt, design)
5  > aw
6  > fit <- lmFit(maqc.norm.filt, design, weights = aw)
7  > contrasts <- makeContrasts(UHRR - Brain, levels = design)
8  > contr.fit <- eBayes(contrasts.fit(fit, contrasts))
9  > topTable(contr.fit, coef = 1)
10 > par(mfrow = c(1, 2))
11 > volcanoplot(contr.fit, main = "UHRR - Brain")
12 > smoothScatter(contr.fit$Amean, contr.fit$coef, xlab = "average intensity",
13 +     ylab = "log-ratio")
14 > 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 [11] 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.

☑ The array weights are lowest for the first and second Brain Reference samples, which means the observations from these samples will be down-weighted slightly in the linear model analysis. You will recall that the Brain Reference samples were less consistent than the UHRR samples in the MDS plot (the second dimension separated out different replicate Brain Reference samples). The UHRR and Brain Reference RNA samples are very different and we find many differentially expressed genes between these two conditions in our analysis.

☞ The `lmFit` function is able to accept a *matrix* as well as a limma object. Hence, users with summarised bead-level data (as created in section 1) can also use this function after extracting the expression matrix using the `exprs` function. The code would look something like.

```
fit <- lmFit(exprs(datasumm), design, weights=aw)
```

### Annotation

#### Annotating the results of a differential expression analysis

The `topTable` function displays the results of the empirical Bayes analysis alongside the annotation assigned by Illumina to each probe in the linear model fit. Often 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 illuminaHumanv2.db package can be used for the arrays in this example dataset.

**Use Case:** Use the appropriate annotation package to annotate our differential expression analysis. Include the genome position, RefSeq ID, Entrez Gene ID, Gene symbol and Gene name information. Write the results from this analysis out to file.

```
1  > library(illuminaHumanv2.db)
2  > illuminaHumanv2()
3  > ids <- as.character(contr.fit$genes$ProbeID)
4  > ids2 <- unlist(mget(ids, revmap(illuminaHumanv2ARRAYADDRESS),
5  +     ifnotfound = NA))
6  > chr <- mget(ids2, illuminaHumanv2CHR, ifnotfound = NA)
7  > cytoband <- mget(ids2, illuminaHumanv2MAP, ifnotfound = NA)
8  > refseq <- mget(ids2, illuminaHumanv2REFSEQ, ifnotfound = NA)
9  > entrezid <- mget(ids2, illuminaHumanv2ENTREZID, ifnotfound = NA)
10 > symbol <- mget(ids2, illuminaHumanv2SYMBOL, ifnotfound = NA)
11 > genename <- mget(ids2, illuminaHumanv2GENENAME, ifnotfound = NA)
12 > anno <- data.frame(Ill_ID = ids2, Chr = as.character(chr),
13 +     Cytoband = as.character(cytoband), RefSeq = as.character(refseq),
14 +     EntrezID = as.numeric(entrezid), Symbol = as.character(symbol),
15 +     Name = as.character(genename))
16 > contr.fit$genes <- anno
17 > topTable(contr.fit)
18 > write.fit(contr.fit, file = "maqcresultsv2.txt")
```

#### Other useful applications of annotation packages

We now give a few brief example use cases that we have encountered in our own analyses.

**Use Case:** Retrieve the Illumina Humanv2 IDs that are part of the cell cycle according to GO (GO:0007049) or KEGG (04110)

```
> cellCycleProbesGO <- mget("GO:0007049", illuminaHumanv2GO2PROBE)
> cellCycleProbesKEGG <- mget("04110", illuminaHumanv2PATH2PROBE)
```

**Use Case:** Retrieve the Illumina Humanv2 IDs representing the ERBB2 oncogene. Which probe seems to be most appropriate for analysis?

```
> queryIDs <- mget("ERBB2", revmap(illuminaHumanv2SYMBOL))
> mget("ERBB2", revmap(illuminaHumanv2SYMBOL))
> mget(unlist(queryIDs), illuminaHumanv2PROBEQUALITY)
```

The `illuminaHumanv2SYMBOL` mapping is defined to map Illumina IDs to probe symbol, so if we want to go the opposite way, we have to use the `revmap` function.

☑ There are three probes on the illuminaHumanv2 platform for ERBB2. All three are classed as Perfect, although only one is located at the 3' end of the gene.

**Use Case:** Calculate the GC content of all Humanv2 probes and plot a histogram.

We will illustrate this use case using the Biostrings package. If you do not have this package you can install it in the usual way.

```
> source("http://www.bioconductor.org/biocLite.R")
> biocLite("Biostrings")
```

```
> require("Biostrings")
> probeseqs <- unlist(as.list(illuminaHumanv2PROBESEQUENCE))
> GC = vector(length = length(probeseqs))
> ss <- BStringSet(probeseqs[which(!is.na(probeseqs))])
> GC[which(!is.na(probeseqs))] = letterFrequency(ss, letters = "GC")
> hist(GC/50, main = "GC proportion")
```

This code requires the Bioconductor Biostrings package which implements efficient string operations. The `as.list(illuminaHumanv2PROBESEQUENCE)` command is simply a shortcut to return the probe seqeunce for all mapped keys. We then convert the sequences into a Biostrings class, on which we can use the `letterFrequency` function to give the desired result.

**Use Case:** Convert the Humanv2 probe locations into a RangedData object.

We will illustrate this use case using the GenomicRanges package. If you do not have this package you can install it in the usual way.

```
> source("http://www.bioconductor.org/biocLite.R")
> biocLite("GenomicRanges")
```

```
> require("GenomicRanges")
> allLocs <- unlist(as.list(illuminaHumanv2GENOMICLOCATION))
> chrs <- unlist(lapply(allLocs, function(x) strsplit(as.character(x),
+     ":")[[1]][1]))
> spos <- as.numeric(unlist(lapply(allLocs, function(x) strsplit(as.character(x),
+     ":")[[1]][2])))
```

```
7  > epos <- as.numeric(unlist(lapply(allLocs, function(x) strsplit(as.character(x),
8  +     ":")[[1]][3])))
9  > strand <- substr(unlist(lapply(allLocs, function(x) strsplit(as.character(x),
10 +     ":")[[1]][4])), 1, 1)
11 > validPos <- !is.na(spos)
12 > Humanv2RD <- RangedData(space = chrs[validPos], ranges = IRanges(start = spos[validPos],
13 +     end = epos[validPos]), names = names(allLocs)[validPos],
14 +     strand = strand[validPos])
```

Again we retrieve all genomic locations using the `unlist(as.list(..)   )`trick. The location of each probe is encoded as a string with the chromosome, start and end separated by a `:` character. We can use the `base` `strsplit` function to decompose this string within an `lapply` to process all sequences at once. The *RangedData* object can be created now, although we add a check to make sure that no `NA` values are passed.

**Use Case:** Find all Humanv2 probes located on chromsome 8 between bases 28,800,001 and 36,500,000. What gene symbols do they target?

```
1 > query <- IRanges(start = 28800001, end = 36500000)
2 > olaps <- findOverlaps(Humanv2RD, RangedData(query, space = "chr8"))[["chr8"]]
3 > matchingProbes <- as.matrix(olaps)[, 1]
4 > Humanv2RD[space(Humanv2RD) == "chr8", ][matchingProbes,
5 +     ]
6 > Humanv2RD[space(Humanv2RD) == "chr8", ]$names[matchingProbes]
7 > unlist(mget(Humanv2RD[space(Humanv2RD) == "chr8", ]$names[matchingProbes],
8 +     illuminaHumanv2SYMBOL))
```

This example code uses functions from the IRanges package, so users wanting a deeper understanding of the commands should consult the documentation for that package, and the appropriate help files. Essentially, we define a query region using the desired chromosome, start and end values. The `findOverlaps` function will then find regions on chromosome 8 of the `Humanv2RD` object that overlap with the query. The indices of the matching probes are given in the `matchMatrix` slot, which can be used to subset the Humanv2 Ranges.

### What next?

The results of a differential expression analysis are often not the end-point of an analysis, and there is an increasing desire to relate findings to biological function. Although this is beyond the scope of this vignette, we will give a brief example of how a Gene Ontology analysis could be performed within Bioconductor. There is a wide range of online tools can perform similar analyses, of which DAVID and Genetrail seem to be the most popular.

**Use Case:** Using GOstats find over-represented GO terms amongst the probes that show evidence for differential expression.

We will illustrate this use case using the GOstats package. If you do not have this package you can install it in the usual way.

```
1 > source("http://www.bioconductor.org/biocLite.R")
2 > biocLite("GOstats")
```

```
> require("GOstats")
> universeIds <- anno$EntrezID
> dTests <- decideTests(contr.fit)
> selectedEntrezIds <- anno$EntrezID[dTests == 1]
> params = new("GOHyperGParams", geneIds = selectedEntrezIds,
+     universeGeneIds = universeIds, annotation = "illuminaHumanv2",
+     ontology = "BP", pvalueCutoff = 0.05, conditional = FALSE,
+     testDirection = "over")
> hgOver = hyperGTest(params)
> summary(hgOver)[1:10, ]
```

The `decideTests` function is useful for selecting probes that show evidence for differential expression after correcting for multiple testing. We also have to define a universe of all possible Entrez IDs in the dataset, and Entrez IDs that are significant. The GOstats package will then map the Entrez IDs to GO terms (for both universe and significant probes) and use a hypergeometric test to see if any GO terms appear more often in the significant list than we would expect by chance. See the vignette of the GOstats package for more details.

# 3 Analysis of public data using GEOquery

In this section we show how to retrieve an Illumina MAQC dataset (Human WG-6 version 1) from the Gene Expression Omnibus database (GEO) using the GEOquery package [12]. The GEO database is a public repository supporting MIAME-compliant data submissions of microarray and sequence-based experiments.

**Use Case:** Read in the MAQC submitted data [1] from the Illumina platform using the `getGEO` function. Extract information about the site each sample was prepared at, and the RNA sample hybridized. Make a boxplot of the intensities, color-coded by site to look for systematic differences between labs.

```
1  > library(GEOquery)
2  > library(limma)
3  > library(illuminaHumanv1.db)
4  > gse <- getGEO(GEO = "GSE5350")[["GSE5350-GPL2507_series_matrix.txt.gz"]]
5  > dim(gse)
6  > exprs(gse)[1:5, 1:2]
7  > samples <- as.character(pData(gse)[, "title"])
8  > sites <- as.numeric(substr(samples, 10, 10))
9  > shortlabels <- substr(samples, 12, 13)
10 > rnasource <- pData(gse)[, "source_name_ch1"]
11 > levels(rnasource) <- c("UHRR", "Brain", "UHRR75", "UHRR25")
12 > boxplot(log2(exprs(gse)), col = sites + 1, names = shortlabels,
13 +     las = 2, cex.names = 0.5, ylab = expression(log[2](intensity)),
14 +     outline = FALSE, ylim = c(3, 10), main = "Before batch correction")
```

☞ Data deposited in the GEO database may be either raw or normalized. This preprocessing information is annotated in the database entry, and is accessible by typing `pData(gse)$data_processing` in this example. Boxplots of the data can also be used to see whether normalization has taken place.

☑ This dataset consists of 59 arrays, each of which contains 47,293 probes. Samples were processed and hybridized in 3 different labs (see the `sites` vector), and subject to cubic spline normalization which appears to have been applied separately to the data from each site.

☞ Microarray data deposited in ArrayExpress, the other major public database of high-throughput genomics experiments, can be imported into `R` using the ArrayExpress package [13].

♣ *If the above code does not work due to a network error, the same dataset, albeit with fewer replicates, may be obtained from an existing Bioconductor package named MAQCsubsetILM.*

```
1 > source("http://www.bioconductor.org/biocLite.R")
2 > biocLite("MAQCsubsetILM")
3 > library(MAQCsubsetILM)
4 > data(refA)
5 > data(refB)
6 > data(refC)
```

```
7  > data(refD)
8  > gse = combine(refA, refB, refC, refD)
9  > sites = pData(gse)[, 2]
10 > shortlabels = substr(sampleNames(gse), 7, 8)
11 > rnasource = pData(gse)[, 3]
12 > levels(rnasource) = c("UHRR", "Brain", "UHRR75", "UHRR25")
13 > boxplot(log2(exprs(gse)), col = sites + 1, names = shortlabels,
14 +     las = 2, cex.names = 0.5, ylab = expression(log[2](intensity)),
15 +     outline = FALSE, ylim = c(3, 10), main = "Before batch correction")
```

**Use Case:** Remove any differences between labs using the `removeBatchEffect` function and make a boxplot and MDS plot of the corrected data to assess the effectiveness of this step. Next filter out probes with poor annotation and perform a differential expression analysis between the UHRR and Brain Reference samples. Annotate the results with the same information obtained in section 2 (genome position, RefSeq ID, Entrez Gene ID, Gene symbol and Gene name) with an appropriate annotation package. Write the results out to file.

```
1  > gse.batchcorrect <- removeBatchEffect(log2(exprs(gse)),
2  +     batch = sites)
3  > par(mfrow = c(1, 2), oma = c(1, 0.5, 0.2, 0.1))
4  > boxplot(gse.batchcorrect, col = sites + 1, names = shortlabels,
5  +     las = 2, cex.names = 0.5, ylab = expression(log[2](intensity)),
6  +     outline = FALSE, ylim = c(3, 10), main = "After batch correction")
7  > plotMDS(gse.batchcorrect, labels = shortlabels, col = sites +
8  +     1, main = "MDS plot")
9  > ids3 <- featureNames(gse)
10 > qual2 <- unlist(mget(ids3, illuminaHumanv1PROBEQUALITY,
11 +     ifnotfound = NA))
12 > table(qual2)
13 > rem2 <- qual2 == "No match" | qual2 == "Bad" | is.na(qual2)
14 > gse.batchcorrect.filt <- gse.batchcorrect[!rem2, ]
15 > dim(gse.batchcorrect)
16 > dim(gse.batchcorrect.filt)
17 > design2 <- model.matrix(~0 + rnasource)
18 > colnames(design2) <- levels(rnasource)
19 > aw2 <- arrayWeights(gse.batchcorrect.filt, design2)
20 > fit2 <- lmFit(gse.batchcorrect.filt, design2, weights = aw2)
21 > contrasts2 <- makeContrasts(UHRR - Brain, levels = design2)
22 > contr.fit2 <- eBayes(contrasts.fit(fit2, contrasts2))
23 > topTable(contr.fit2, coef = 1)
24 > volcanoplot(contr.fit2, main = "UHRR - Brain")
25 > ids4 <- rownames(gse.batchcorrect.filt)
26 > chr2 <- mget(ids4, illuminaHumanv1CHR, ifnotfound = NA)
27 > chrloc2 <- mget(ids4, illuminaHumanv1CHRLOC, ifnotfound = NA)
28 > refseq2 <- mget(ids4, illuminaHumanv1REFSEQ, ifnotfound = NA)
29 > entrezid2 <- mget(ids4, illuminaHumanv1ENTREZID, ifnotfound = NA)
30 > symbols2 <- mget(ids4, illuminaHumanv1SYMBOL, ifnotfound = NA)
31 > genename2 <- mget(ids4, illuminaHumanv1GENENAME, ifnotfound = NA)
32 > anno2 <- data.frame(Ill_ID = ids4, Chr = as.character(chr2),
33 +     Loc = as.character(chrloc2), RefSeq = as.character(refseq2),
34 +     Symbol = as.character(symbols2), Name = as.character(genename2),
35 +     EntrezID = as.numeric(entrezid2))
36 > contr.fit2$genes <- anno2
```

```
37 > write.fit(contr.fit2, file = "maqcresultsv1.txt")
```

☑ The batch correction procedure equalizes the mean expression level from each lab, which removes the differences that were evident in the earlier boxplot. The MDS plot shows that samples separate by RNA source, with the first dimension separating the pure samples (UHRR (A) and Brain Reference (B)) from each other, and the second dimension separating the pure samples (A and B) from the mixture samples (75% UHRR:25% Brain Reference (C) and 25% UHRR:75% Brain Reference (D)). After filtering out the probes with poor annotation, we are left with 25,797 probes. As we saw in the Asuragen MAQC dataset, there is a great deal of differential expression between the UHRR and Brain Reference samples. Having a greater number of replicate samples (15 each for UHRR and Brain Reference) leads to greater statistical significance for this comparison relative to the Asuragen dataset, which had just 3 replicates for each RNA source.

### Combining data from different BeadChip versions

In this section we illustrate how to combine data from different BeadChip versions from the same species, using the GEO MAQC dataset (Human version 1) and the Asuragen MAQC dataset (Human version 2). The approach taken in this example can be used to combine data from different microarray platforms as well.

**Use Case:** Use Entrez Gene identifiers to match genes from different BeadChip versions. Make a data frame which includes fold-changes between UHRR and Brain Reference samples for both datasets for the genes which could be matched between platforms and plot the log-fold-changes. Does expression agree between platforms?

```
1  > z <- contr.fit[!is.na(contr.fit$genes$EntrezID), ]
2  > z <- z[order(z$genes$EntrezID), ]
3  > f <- factor(z$genes$EntrezID)
4  > sel.unique <- tapply(z$Amean, f, function(x) x == max(x))
5  > sel.unique <- unlist(sel.unique)
6  > contr.fit.unique <- z[sel.unique, ]
7  > z <- contr.fit2[!is.na(contr.fit2$genes$EntrezID), ]
8  > z <- z[order(z$genes$EntrezID), ]
9  > f <- factor(z$genes$EntrezID)
10 > sel.unique <- tapply(z$Amean, f, function(x) x == max(x))
11 > sel.unique <- unlist(sel.unique)
12 > contr.fit2.unique <- z[sel.unique, ]
13 > m <- match(contr.fit.unique$genes$EntrezID, contr.fit2.unique$genes$EntrezID)
14 > contr.fit.common <- contr.fit.unique[!is.na(m), ]
15 > contr.fit2.common <- contr.fit2.unique[m[!is.na(m)],
16 +     ]
17 > lfc <- data.frame(lfc_version1 = contr.fit2.common$coef[,
18 +     1], lfc_version2 = contr.fit.common$coef[, 1])
19 > dim(lfc)
20 > options(digits = 2)
21 > lfc[1:10, ]
22 > plot(lfc[, 1], lfc[, 2], xlab = "version 1", ylab = "version 2")
23 > abline(0, 1, col = 2)
```

☑ Even though the probes were redesigned between versions 1 and 2 of the Human gene expression BeadChip, we see good concordance between the log-ratios from each dataset.

☞ A representative probe was selected for each gene on each platform before the two datasets could be combined. In this example, the probe with the largest average intensity among all the probes belonging to the same gene was selected from each version.

**Use Case:** (*Advanced*) Compare the version 1 and 2 results with those obtained from the Human HT-12 version 3 dataset analyzed earlier in this tutorial. You will first need to between-array normalize the HT-12 data, and may find it useful to filter poorly annotated probes and down-weight observations for less reliable arrays identified during the quality assessment process. Differential expression can be assessed using linear modelling and contrasts as previously shown. You will then need to annotate the probes using Entrez Gene IDs. How many probes can be matched between all 3 platforms? How well does the version 3 data agree with data from earlier BeadChip versions.

☞ Example code for this final use case has not been provided. The reader should be able to complete this exercise independently using knowledge gained from the previous examples given in the vignette.

There are many other analysis tools available from R or Bioconductor that could be used for downstream analysis on these data. For further details, see the relevant vignettes or help pages.

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).

```
1 > sessionInfo()
```

```
R version 3.5.0 (2018-04-23)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Ubuntu 16.04.4 LTS

Matrix products: default
BLAS: /home/biocbuild/bbs-3.7-bioc/R/lib/libRblas.so
LAPACK: /home/biocbuild/bbs-3.7-bioc/R/lib/libRlapack.so

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

attached base packages:
[1] stats4    parallel  stats     graphics  grDevices utils
[7] datasets  methods   base

other attached packages:
 [1] bindrcpp_0.2.2            limma_3.36.0
```

```
 [3] GenomicRanges_1.32.0      GenomeInfoDb_1.16.0
 [5] Biostrings_2.48.0         XVector_0.20.0
 [7] illuminaHumanv3.db_1.26.0 illuminaHumanv1.db_1.26.0
 [9] illuminaHumanv2.db_1.26.0 org.Hs.eg.db_3.6.0
[11] AnnotationDbi_1.42.0      IRanges_2.14.1
[13] S4Vectors_0.18.1          GEOquery_2.48.0
[15] beadarray_2.30.0          ggplot2_2.2.1
[17] Biobase_2.40.0            BiocGenerics_0.26.0

loaded via a namespace (and not attached):
 [1] tidyselect_0.2.4       reshape2_1.4.3
 [3] purrr_0.2.4            colorspace_1.3-2
 [5] utf8_1.1.3             blob_1.1.1
 [7] BeadDataPackR_1.32.0   rlang_0.2.0
 [9] pillar_1.2.2           glue_1.2.0
[11] DBI_1.0.0              bit64_0.9-7
[13] GenomeInfoDbData_1.1.0 plyr_1.8.4
[15] bindr_0.1.1            stringr_1.3.0
[17] zlibbioc_1.26.0        munsell_0.4.3
[19] gtable_0.2.0           memoise_1.1.0
[21] curl_3.2               illuminaio_0.22.0
[23] Rcpp_0.12.16           readr_1.1.1
[25] openssl_1.0.1          scales_0.5.0
[27] base64_2.0             bit_1.1-12
[29] hms_0.4.2              digest_0.6.15
[31] stringi_1.2.2          dplyr_0.7.4
[33] grid_3.5.0             cli_1.0.0
[35] tools_3.5.0            bitops_1.0-6
[37] magrittr_1.5           lazyeval_0.2.1
[39] RCurl_1.95-4.10        tibble_1.4.2
[41] RSQLite_2.1.0          crayon_1.3.4
[43] tidyr_0.8.0            pkgconfig_2.0.1
[45] xml2_1.2.0             assertthat_0.2.0
[47] R6_2.2.2               compiler_3.5.0
```

## 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] M. Suarez-Farinas, A. Haider, and K. Wittkowski. 'harshlighting' small blemishes on microarrays. *BMC Bioinformatics*, 6(1):65, 2005.

[5] M. L. Smith, M. J. Dunning, S. Tavaré, and A. G. Lynch. Identification and correction of previously unreported spatial phenomena using raw Illumina BeadArray data. *BMC Bioinformatics*, 11:208, 2010.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] S. Davis and P. S. Meltzer. GEOquery: a bridge between the Gene Expression Omnibus (GEO) and Bioconductor. *Bioinformatics*, 23(14):1846–7, 2007.

[13] A. Kauffmann, T. F. Rayner, H. Parkinson, M. Kapushesky, M. Lukk, A. Brazma, and W. Huber. Importing ArrayExpress datasets into R/Bioconductor. *Bioinformatics*, 25(16):2092–4, 2009.