# Analysis of Bead-level Data using beadarray

Mark Dunning

October 5, 2010

## Introduction

beadarray is a package for the pre-processing and analysis of Illumina BeadArray. The main advantage is being able to read raw data output by Illumina's scanning software. Data presented in this form are in the same format regardless of the assay (i.e expression, genotyping, methylation) being performed. Thus, beadarray is able to handle all these types of data. Many functions within beadarray have been written to take cope with this flexibility. The BeadArray technology involves randomly-arranged arrays of beads, with beads having the same probe sequence attached being known as a bead-type. BeadArrays are combined in parallel on either a rectangular chip (BeadChip) or array of 8 by 12 hexagonal arrays (Sentrix Array Matrix or SAM). The BeadChip is further divided into strips on the surface known as sections, with each section giving rise to a different image when scanned by BeadScan. These images, and associated text files, comprise the raw data for a beadarray analysis.

This vignette demonstrates the processing of bead-level data using beadarray. The example dataset is taken from an early expression study using a BeadArray platform that is no longer commercially available.

## Citing beadarray

If you use beadarray for the analysis or pre-processing of BeadArray data please cite:

Dunning MJ, Smith ML, Ritchie ME, Tavaré S, **beadarray: R classes and methods for Illumina bead-based data**, *Bioinformatics*, **23**(16):2183-2184

# 1 Asking for help on beadarray

Wherever possible, questions about beadarray should be sent to the Bioconductor mailing list (bioconductor@stat.math.ethz.ch). Therefore all problems and solutions will be kept in a searchable archive. When posting to this mailing list, please state the version of beadarray and R to help to diagnose the problem. This can be done by pasting the output of running the command `sessionInfo()`.

```
> options(width = 50)
```

# 2 Reading bead-level data into beadarray

## 2.1 File formats

The raw images and text files required to perform a bead-level analysis are produced by Illumina's BeadScan software. To modify BeadScan's default settings to obtain bead-level data, see `http://www.compbio.group.cam.ac.uk`
The command to read bead-level data from the current working directory is as follows. The `dir` argument may be used to specifiy an alternative location.

```
> BLData = readIllumina(useImages = FALSE, illuminaAnnotation = "Humanv3")
```

The `useImages` argument specifies whether beadarray will read foreground and background intensities from the TIFF images present in the directory, allowing users to experiment with strategies for image processing. In this example we set `useImages=FALSE` (often a convenient choice), and locally-background corrected intensities will simply be extracted from the txt files. The background values that have been subtracted are taken from the "background" pixel intensities surrounding each bead, and are not be confused with background correction futher along the analysis pipeline which may involve negative control beads to account for non-specific binding.

Note that is not compulsory to specify which type of Illumina assay was used to generate the data when using `readIllumina`. However, for expression data it is convenient to specify the annotation of the platform using one of the strings Humanv4, Humanv3, Humanv2, Humanv1, Mousev2, Mousev1p1, Mousev1 or Ratv1.

beadarray is able to utilize some of Illumina's proprietary files during analysis. These include `.locs` and `.sdf` files that contain the locations of *all* beads on an array (not just those that were decoded) and data about the physical layout of the chip. In combination, using these files can result in significant time improvements to the detection of spatial artefacts and add additional information to some QA plots. These files are not read automatically, but if present, the path to these files is stored by beadarray for future use. If the metrics file generated by BeadScan is present in the directory, it will be read unaltered and stored.

# 3 The beadLevelData class

Once imported, the bead-level data is stored in an object of class `beadLevelData`. This class can handle raw data from both single channel and two-colour BeadArrays. Due to the random nature of the technology, each array generally has a variable number of rows of intensity data, and we use an R environment variable to store this information in a memory efficient way.

The beadLevelData class contains a number of slots useful for describing Illumina data. The data that have been extracted from the text files are found in the beadData slot. This can be thought of as a list, which can be indexed by name or a numeric value representing a particular array-section. A data frame holds the data for that array-section with the number of rows being the number of beads on the section. For convenience, the `getBeadData` is used to access data held in the beadData slot. The function `insertBeadData` function can be used to assign new data to this slot.

Data with one value per array-section are stored in the sectionData slot. For instance, any metrics information present in the directory used by `readIllumina` will be stored here. This is also a convenient place to store any QC information derived during the pre-processing of the data, as we will see.

The numeric identifiers for the bead-types in the *beadLevelData* are known as ArrayAddress IDs in Illumina's annotation files. For downstream analysis it is convenient to convert these into the form ILMN_... used in most annotation packages. Mapping objects to convert these IDs are supplied with beadarray in the `extdata` directory, but this conversion may be performed automatically if the annotation of the *beadLevelData* object is known.

```
> data(BLData)
> is(BLData)

[1] "beadLevelData"

> class(BLData)

[1] "beadLevelData"
attr(,"package")
[1] "beadarray"
```

```
> slotNames(BLData)

[1] "beadData"       "sectionData"    "phenoData"      "experimentData"
[5] "history"

> BLData[[1]][1:10, ]

      ProbeID       Grn GrnB    GrnX    GrnY Weights
 [1,]       0 2585.5922  711 790.367 684.381       1
 [2,]       0  953.4268  711 737.497 672.404       1
 [3,]       0 1004.5892  711 784.063 729.965       1
 [4,]       0 1018.3674  712 743.029 687.744       1
 [5,]       0  988.5614  705 711.104 663.610       1
 [6,]       0  991.9884  719 847.430 790.153       1
 [7,]       0 1008.3654  718 763.267 796.704       1
 [8,]       0 1141.9531  721 605.274 671.232       1
 [9,]       0 1046.1734  717 686.294 521.007       1
[10,]       0  979.5010  725 733.478 536.014       1

> getBeadData(BLData, array = 1, what = "Grn")[1:10]

 [1] 2585.5922  953.4268 1004.5892 1018.3674  988.5614  991.9884 1008.3654
 [8] 1141.9531 1046.1734  979.5010

> uIDs = unique(getBeadData(BLData, array = 1, what = "ProbeID"))
> uIDs[1:10]

 [1]  0  2  3 10 21 23 27 28 30 31
```

# 4 Transformation Functions

A more flexible way to obtain 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. The `logGreenChannelTransform` is the default transformation in many plotting / QA functions within beadarray. Users with two-channel data may also wish to experiment with the similarly defined `logRedChannelTransform` or `logRatioTransform` when plotting.

```
> log2(BLData[[1]][1:10, 2])

 [1] 11.336279  9.896978  9.972390  9.992042  9.949187  9.954179  9.977803
 [8] 10.157288 10.030906  9.935903

> logGreenChannelTransform

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

> logGreenChannelTransform(BLData, array = 1)[1:10]
```

```
 [1] 11.336279  9.896978  9.972390  9.992042  9.949187  9.954179  9.977803
 [8] 10.157288 10.030906  9.935903
```

```
> logRedChannelTransform
```

```
function (BLData, array)
{
    x = getBeadData(BLData, array = array, what = "Red")
    log2.na(x)
}
<environment: namespace:beadarray>
```

In this example dataset, the local background-corrected intensities were not read from text files and separate foreground and background intensities were calculated for each bead (option `useImages = FALSE`). The simple background correction that subtracts background from foreground is implemented in the `backgroundCorrectSingleSection` function. More elegant solutions, such as using the `normexp` functionality in limma would be easy implement.

```
> for (i in 1:10) {
+     BLData = backgroundCorrectSingleSection(BLData, array = i)
+ }
> head(BLData[[1]])
```

```
     ProbeID       Grn GrnB    GrnX    GrnY Weights    Grn.bc
[1,]       0 2585.5922  711 790.367 684.381       1 1874.5922
[2,]       0  953.4268  711 737.497 672.404       1  242.4268
[3,]       0 1004.5892  711 784.063 729.965       1  293.5892
[4,]       0 1018.3674  712 743.029 687.744       1  306.3674
[5,]       0  988.5614  705 711.104 663.610       1  283.5614
[6,]       0  991.9884  719 847.430 790.153       1  272.9884
```

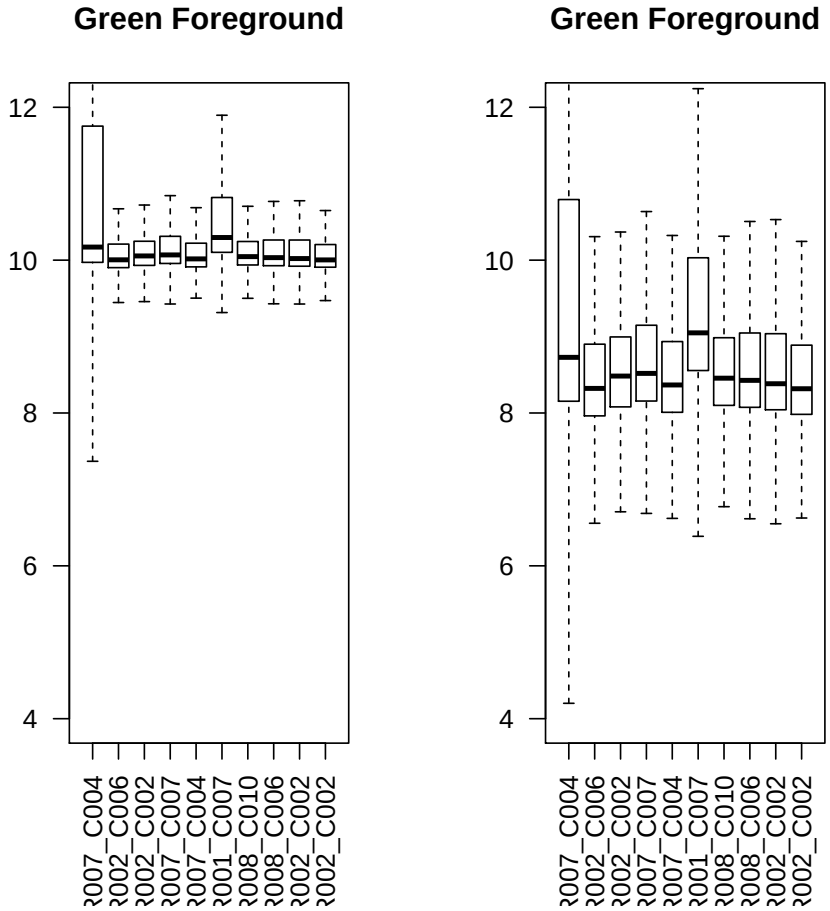
```
> normExpPars = matrix(nrow = 12, ncol = 3)
> for (i in 1:12) {
+     fooMat = cbind(getBeadData(BLData, array = i, what = "Grn"),
+         getBeadData(BLData, array = i, what = "Grn"))
+     res = limma:::normexp.fit.control(x = fooMat, status = beadStatusVector(BLData,
+         array = i), negctrl = "permuted_negative")
+     normExpPars[i, ] = res[1, ]
+ }
> for (i in 1:12) {
+     corSignal = limma:::normexp.signal(x = BLData[[i]][, 4],
+         par = normExpPars[i, ])
+     BLData = insertBeadData(BLData, array = i, data = corSignal,
+         what = "Grn.normexp")
+ }
> head(BLData[[1]])
```

# 5 Boxplots and imageplots

Two standard quality assessment plot supported by `beadarray` are the imageplot and boxplot. Boxplots can be used to compare foreground and background intensities between arrays. Image plots can be used to identify spatial artefacts on the array surface that can occur from mis-handling or scanning problems.

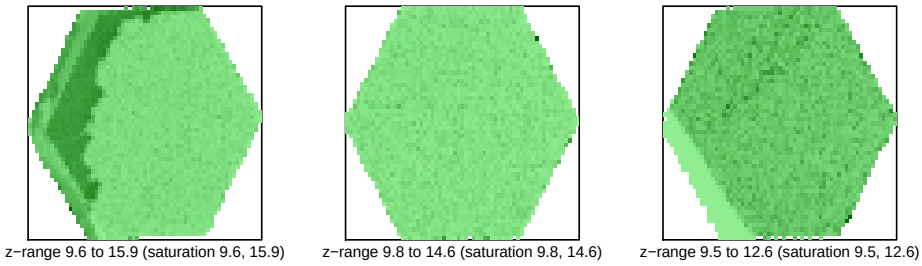
With the raw bead-level data, we can plot false images of each array. This kind of visualisation is not possible when using the summarised BeadStudio output, as the summary values are averaged over spatial positions. Image plots in R are also more convenient than scrutinising the original tiffs, as multiple arrays can be visualised on the one page. By default, the array surface is plotting with the longest edge going horizontally. Both the `boxplot` and `imageplot` functions take a transformation function as an argument, with the default to do a log2 transformation. In the code we show how to extract the background-corrected intensities that we have just calculated and display them on the boxplot.

```
> getBackgroundCorrectionIntensities = function(BLData, array) {
+     log2(getBeadData(BLData, array = array, what = "Grn.bc"))
+ }
> par(mfrow = c(1, 2))
> boxplot(BLData, las = 2, outline = FALSE, ylim = c(4, 12), main = "Green Foreground")
> boxplot(BLData, las = 2, transFun = getBackgroundCorrectionIntensities,
+     outline = FALSE, ylim = c(4, 12), main = "Green Foreground")
```




The imageplot can be configured in many ways (see manual page for more details). Sections from a BeadChip often have one edge that is much longer than the other, and it is important to recognise this when producing the plots. By default, `beadarray` makes imageplots with the longest edge on the x-axis (suitable for widescreen monitors). However, with `horizontal = FALSE`, the imageplot will be displayed in the same orientation as the original TIFF image from the directory. With the `squareSize` we can control how many pixels from the original image make up the pixels in the resulting imageplot.

```
> par(mfrow = c(1, 3), mar = c(2, 2, 2, 2))
> imageplot(BLData, array = 1, low = "lightgreen", high = "darkgreen",
+     horizontal = FALSE, squareSize = 25)
> imageplot(BLData, array = 2, low = "lightgreen", high = "darkgreen",
+     horizontal = FALSE, squareSize = 25)
> imageplot(BLData, array = 3, low = "lightgreen", high = "darkgreen",
+     horizontal = FALSE, squareSize = 25)
```




# 6 BASH

BASH is a method for managing the spatial artefacts that may be found on an array. Three types of defect...

Extended is only a quality score, may use extended = FALSE to make it quicker.

Does not automatically remove beads.

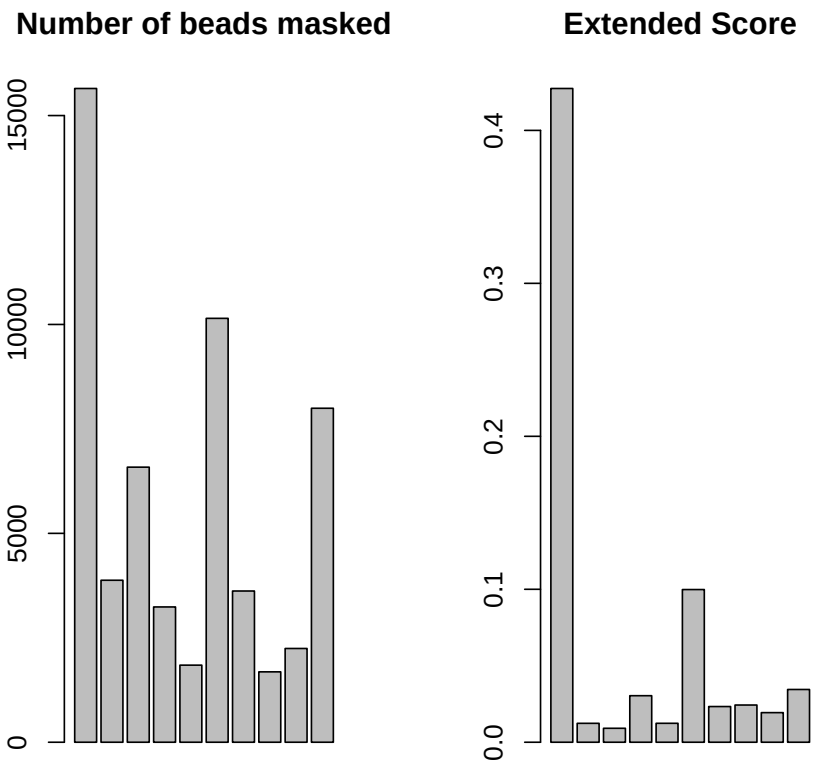
```
> bsh = BASH(BLData, array = 1:10)
```

The result of bash includes quality control scores; the number of beads masked in total and the extended score.

```
> bsh$QC
```

```
   BeadsMasked ExtendedScore
1        15648   0.427391452
2         3877   0.012364884
3         6584   0.009120325
4         3238   0.030427706
5         1845   0.012382269
6        10146   0.099817347
7         3620   0.023329529
8         1685   0.024332186
9         2244   0.019361307
10        7994   0.034485072
```

```
> par(mfrow = c(1, 2))
> barplot(bsh$QC[, 1], main = "Number of beads masked")
> barplot(bsh$QC[, 2], main = "Extended Score")
```




### 6.1 Saving the output of BASH

The weights themselves can be stored using setWeights. The QC can be appended to the sectionData slot of BLData

```
> for (i in 1:10) {
+     BLData = setWeights(BLData, wts = bsh$wts[[i]], array = i)
+ }
> BLData = insertSectionData(BLData, what = "BASHQC", data = bsh$QC)
```

## 7 Using control information

Illumina have designed a number of control probes for each expression platform. For expression arrays, we store the ArrayAddressIDs of the control probes for in the ExpressionControlData object. Otherwise a data frame may be used to define these ids. As the example data described in this vignette were derived using an obsolete technology, we have stored the control information with the package in the `controlProfile` object. ArrayAddressIDs are listed in the first column, and the type of control in the second column. Objects of this form can be used in various quality assessment functions in beadarray.

```
> data(controlProfile)
> head(controlProfile)
```

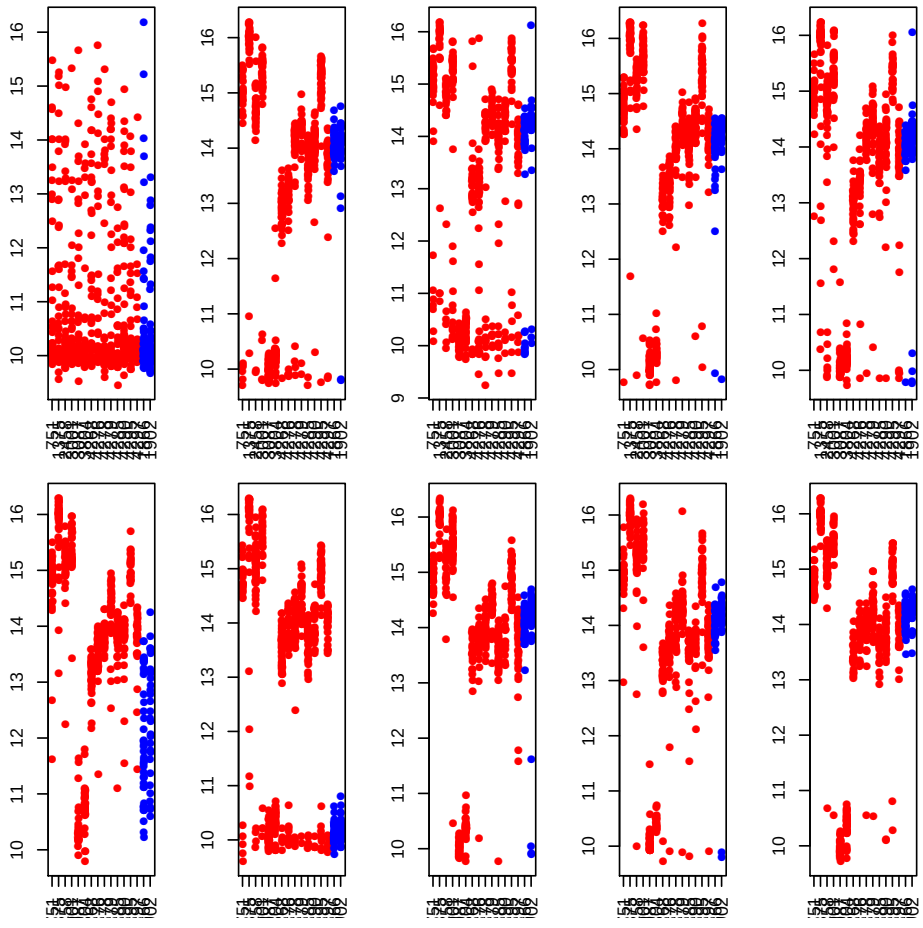
```
  ArrayAddressID ControlType
1           6124    labeling
2           6125    labeling
3           6126    labeling
4           6130    labeling
5           6131    labeling
6           6136    labeling
```

```
> table(controlProfile[, 2])
```

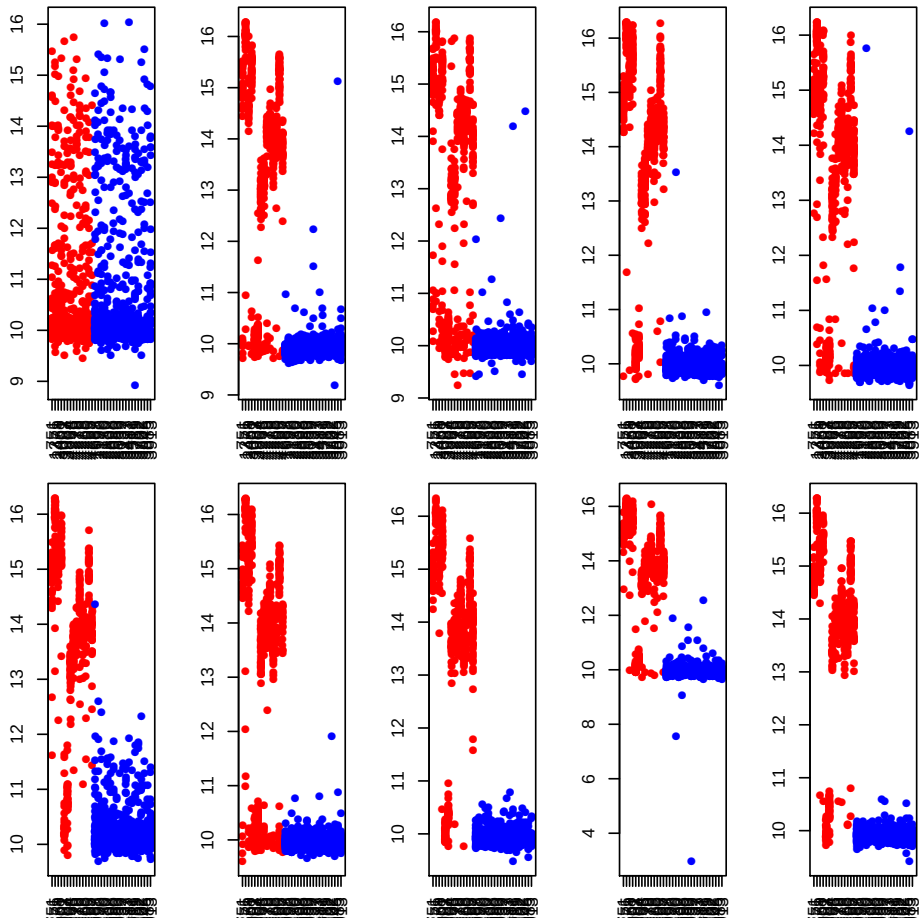
```
             biotin             cy3_hyb high_stringency_hyb        housekeeping
                  2                   6                   1                  14
           labeling  low_stringency_hyb            negative               other
                  8                   8                  19                  18
```

Two particular controls on expression arrays are housekeeping and biotin controls. With the `poscontPlot` function, we can plot the intensities of any ArrayAddressIDs that are annotated as housekeeping or Biotin in the corresponding entry in `ExpressionControlData` or `controlProfile`. The `poscontPlot` is flexible in allowing other "tags" in the `controlProfile`, in the example below we configure the plot to show both housekeeping and negative controls in the same plot.

```
> par(mfrow = c(2, 5), mar = c(2, 2, 2, 2))
> for (i in 1:10) {
+     poscontPlot(BLData, array = i, controlProfile = controlProfile,
+         ylim = c(10, 16))
+ }
```



```
> par(mfrow = c(2, 5), mar = c(2, 2, 2, 2))
> for (i in 1:10) {
+     poscontPlot(BLData, array = i, controlProfile = controlProfile,
+         positiveControlTags = c("housekeeping", "negative"),
+         ylim = c(10, 16))
+ }
```



With knowledge of which ArrayAddressIDs match 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 taken for a specified array, using intensities of all beads that correspond to the control type. Note that these summaries may not correspond to similar quantities reported in Illumina's BeadStudio software, as the BeadStudio summaries are produced after removing outliers (see later).

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.

```
> quickSummary(BLData, array = 1, reporterIDs = controlProfile[,
+     1], reporterTags = controlProfile[, 2])

$biotin
[1] 10.67205

$cy3_hyb
[1] 10.88688

$high_stringency_hyb
[1] 11.12719

$housekeeping
[1] 11.06371
```

```
$labeling
[1] 10.7935

$low_stringency_hyb
[1] 11.04094

$negative
[1] 10.89803

$other
[1] 11.04473
```

```
> qcReport = makeQCTable(BLData, controlProfile = controlProfile)
> head(qcReport)[, 1:5]
```

```
                  Mean:biotin Mean:cy3_hyb Mean:high_stringency_hyb
1318758_R007_C004    10.67205     10.88688                 11.12719
1318791_R002_C006    13.95192     10.73725                 14.68147
1328198_R002_C002    13.53781     10.92058                 14.16797
1318740_R007_C007    13.99586     10.85723                 15.38247
1328227_R007_C004    13.83850     10.78821                 14.67338
1318791_R001_C007    12.14608     10.47578                 14.77956
                  Mean:housekeeping Mean:labeling
1318758_R007_C004          11.06371     10.793501
1318791_R002_C006          13.34732      9.928735
1328198_R002_C002          12.99986     10.081458
1318740_R007_C007          13.88669      9.987988
1328227_R007_C004          13.60260      9.960814
1318791_R001_C007          13.80687     10.306656
```

```
> BLData = insertSectionData(BLData, what = "BeadLevelQC", data = qcReport)
> names(BLData@sectionData)
```

```
[1] "Targets"     "BASHQC"      "BeadLevelQC"
```

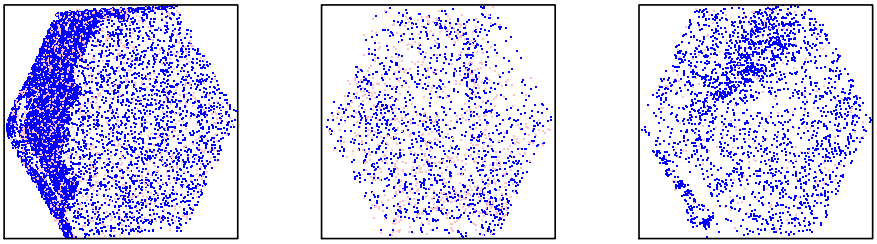
The generation of QA plots for all sections in the beadLevelData object is provided by the expressionQCPipeline function. Results are generated in a directory of the users choosing. This report may be generated at any point of the analysis. If the overWrite paramater is set to FALSE, then any existing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

# 8 Outlier removal and plotting

Before combining the observations for each bead-type on an array, Illumina remove any observations with outlying intensity (more than 3 median absolute deviations from the median). This step can be repeated in beadarray and it is sometimes useful to see where these outliers are located on the array surface. Often, they will coincide with beads masked by BASH or with any spatial artefacts that may be seen.

Users are able to define their own functions to identify outliers. Such functions must take a list of intensities and corresponding ArrayAddressIDs and return indices of which observations are found to be outliers.

```
> par(mfrow = c(1, 3), mar = c(2, 2, 2, 2))
> outlierplot(BLData, array = 1, horizontal = FALSE)

11946  outliers found on the section

> outlierplot(BLData, array = 2, horizontal = FALSE)

2269  outliers found on the section

> outlierplot(BLData, array = 3, horizontal = FALSE)

2886  outliers found on the section
```



# 9 Summarization

The summarization procedure

```
> myMean = function(x) mean(x, na.rm = TRUE)
> mySd = function(x) sd(x, na.rm = TRUE)
> greenChannel = new("illuminaChannel", logGreenChannelTransform,
+     illuminaOutlierMethod, myMean, mySd, "G")
> system.time(BSData <- sectionLevelSummary(BLData, list(greenChannel),
+     silent = TRUE))

Making  summary object
   user  system elapsed
  6.680   0.070   6.762

> BSData

ExpressionSetIllumina (storageMode: list)
assayData: 1472 features, 10 samples
  element names: exprs, se.exprs, NoBeads
protocolData: none
phenoData: none
featureData
  featureNames: 0 2 ... 6140 (1472 total)
  fvarLabels: ProbeID
  fvarMetadata: labelDescription
experimentData: use 'experimentData(object)'
Annotation:
QC Information
```

```
 Available Slots:  exprs se.exprs NoBeads controlType
  featureNames:
  sampleNames:
```

```
> sessionInfo()
```

```
R version 2.12.0 Under development (unstable) (2010-08-01 r52659)
Platform: x86_64-unknown-linux-gnu (64-bit)

locale:
 [1] LC_CTYPE=en_GB.utf8       LC_NUMERIC=C
 [3] LC_TIME=en_GB.utf8        LC_COLLATE=C
 [5] LC_MONETARY=C             LC_MESSAGES=en_GB.utf8
 [7] LC_PAPER=en_GB.utf8       LC_NAME=C
 [9] LC_ADDRESS=C              LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_GB.utf8 LC_IDENTIFICATION=C

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

other attached packages:
[1] beadarray_1.99.1 hwriter_1.2      Biobase_2.9.2

loaded via a namespace (and not attached):
[1] tools_2.12.0
```