Detecting spatial artefacts on BeadArrays using BASH

Mark Dunning and Jonathan Cairns

October 14, 2008

Introduction

Data

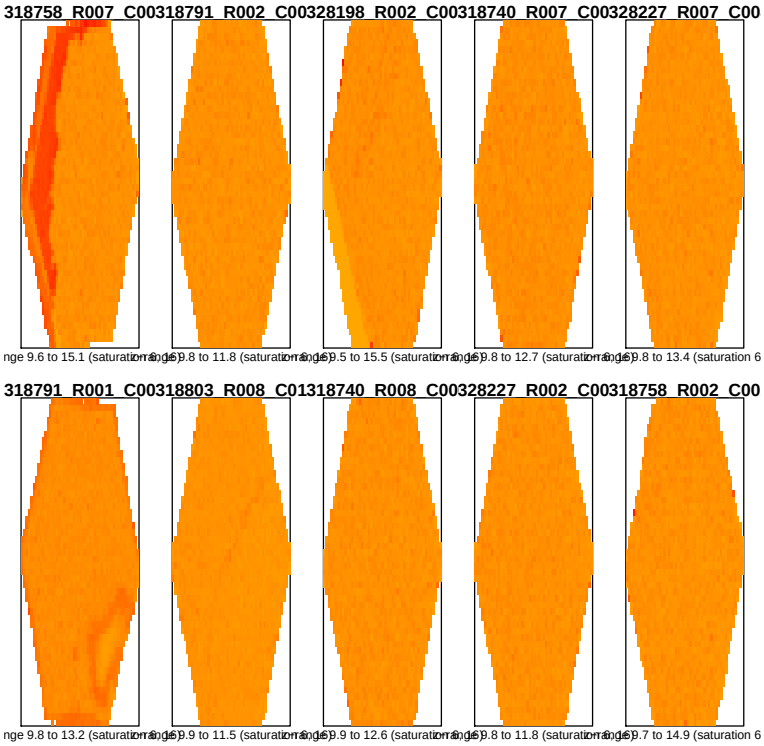
The raw data you will need for this vignette are made available with the beadarray package. The original bead-level data are available in the archive `SAMExample.zip` (46 Mb) which can be downloaded from
`http://www.compbio.group.cam.ac.uk/Resources/illumina/`. This file contains tiffs and text files from 10 hexagonal BeadArrays from several SAMs in an experiment which aimed to measure gene expression differences between cell lines. These data can be read in by using the function `readIllumina`. For information on how to use this function and for bead-level analysis in general, see the vignette on bead-level data. For covenience, we will use the BeadLevelList object distributed with the package.

```
> data(BLData)
```

Motivation for using BASH

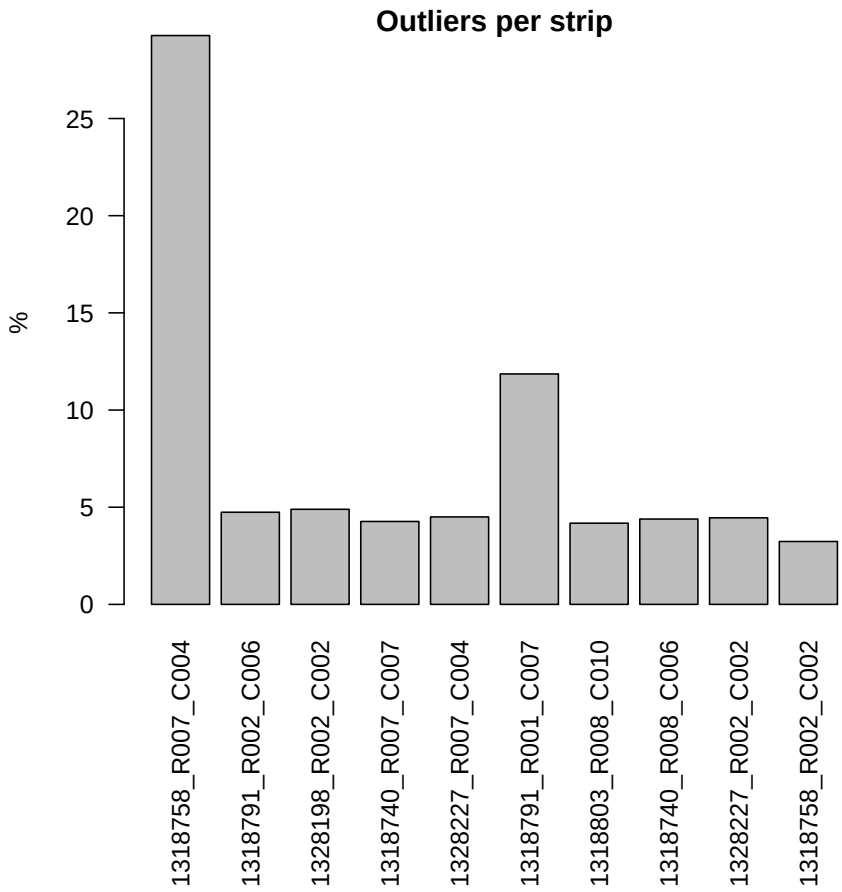
Spatial artefacts on the array surface can occur from mis-handling or scanning problems. Image plots can be used to identify these artefacts. This kind of visualisation is not possible when using the summarised BeadStudio output, as the summary values are averaged over spatial positions. Image plots in R are also more convenient than scrutinising the original TIFFs, as multiple arrays can be visualised on the one page. Here we plot imageplots of the foreground intensities for all 10 arrays.

```
> par(mfrow = c(2, 5))
> zlim = c(6, 16)
> an = arrayNames(BLData)
> for (i in 1:10) {
+     imageplot(BLData, array = i, nrow = 50, ncol = 50,
+         high = "red", low = "yellow", zlim = zlim, main = an[i])
+ }
```



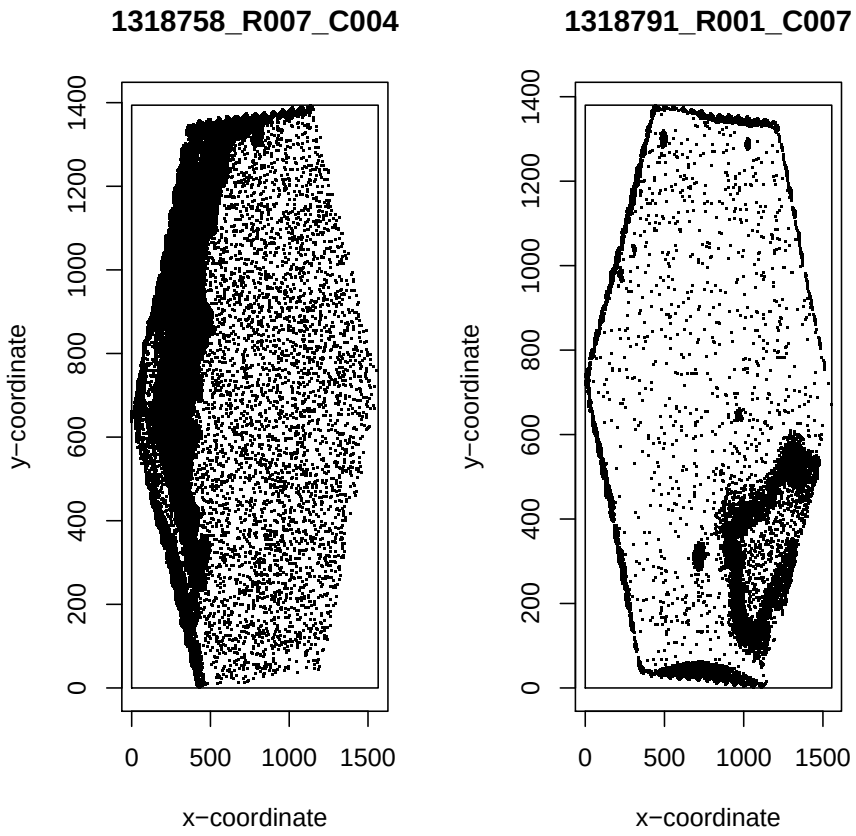
Recall that the BeadArray technology includes around 30 replicates of each bead type on every array. By default, BeadStudio removes outliers greater than 3 median absolute deviations (MADs) from the median prior to calculating the bead summary values. In beadarray, the `findAllOutliers` function identifies outlier beads on a given strip or array using this criterion, by returning a vector of row indices of the outliers. An excessively high number of outliers can be indicative of a poor quality array. The locations of outliers can then be plotted.

```
> outliers = NULL
> for (i in 1:10) {
+     outliers[i] = length(findAllOutliers(BLData, array = i))
+ }
> par(mai = c(2, 1, 0.2, 0.1))
> barplot(outliers/numBeads(BLData) * 100, main = "Outliers per strip",
+     ylab = "%", las = 2, names = an)
```



We now plot the locations of two arrays that seem to have obvious spatial artefacts in the imageplots produced previously.

```
> par(mfrow = c(1, 2))
> o = findAllOutliers(BLData, array = 1)
> plotBeadLocations(BLData, array = 1, BeadIDs = o, pch = ".",
+     main = an[1])
> o = findAllOutliers(BLData, array = 6)
> plotBeadLocations(BLData, array = 6, BeadIDs = o, pch = ".",
+     main = an[6])
```



In this example, the regions of the arrays which appear to be spatial artefacts are also flagged as outliers, which would be removed by BeadStudio before creating summarised values for each bead type. It is important to note that if a very large fraction of the array is affected, the converse may occur, i.e. the 'good' section of the array may appear to be the outlier region, which users need to be aware of.

The number of outliers could be used as an ad-hoc criterion for removing poor quality arrays. However, upon closer inspection, it is often found that not all beads inside an obvious spatial artefact are called outliers by Illumina. This is because spatial information is not taken into account when they determine outliers. Moreover, Illumina exclude outliers on the unlogged scale, and are therefore more likely to exclude obsevations above the median than below.

The basics of BASH

BASH uses the methodology developed for the Harshlight package, but altered to exploit the availability of replicated observations on the same array. The algorithm first identifies *Extended defects*, where an array has gradual but significant shifts across the surface. BASH also seeks to find more localized artifacts on arrays by classifying features that have unusual intensities as outliers and then finding outliers close to each other on the array. Two separate algorithms then search for areas with a larger numbers of outliers than would be expected by chance (Diffuse Defects) and large connected clusters of outliers (Compact defects). The random nature (both in position and numbers of each feature type) of Illumina arrays mean that the Harshlight algorithm must proceed in a different way to the original Harshlight implementation. Whereas Affymetrix probes have replicates on other arrays, Illumina beads

are replicated on the same array. We can therefore generate an error image based on how much each bead differs from the median of its replicates' intensities, instead of replicates on other arrays. Having performed manipulations to the error image, we can then find outliers on this image by bead type, determining which beads are more than 3 Median Absolute Devations, or MADs, from the median. Finally, since Illumina arrays are randomly arranged and use a hexagonal grid rather than rectangular, BASH has it's own method for creating networks of beads on the array

We will now apply BASH to the set of 10 arrays. A convenient `BASH` function is provided that will seek all three types of defect on a specified set of arrays. Note that BASH does not automatically remove any areas within spatial artefacts, but merely reports whether each bead is found inside an artefact or not. Thus, the user may decide to exclude the beads or not. Hence, the output object contains a vector of weights for each array (with the same length as the number of beads) with 0 indicating that a bead lies within a compact of diffuse defect. The extended scores for each array are also given and can be another indicator of array quality. Note that although we are illustrating BASH using a gene expression array, it can be applied to other types of Illumina array that can be stored in a *BeadLevelList* object. Although for two-colour arrays there may be improvements possible by considering the intensitites in both channels.

The output of BASH is quite verbose as it reports which defect it is looking for (diffuse, compact or extended) and how many beads identified at each stage. In the following we only show output for arrays of interest.

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

Array 1 :
Generating neighbours matrix... done.
Compact analysis... done. 13567 beads identified.
Extended analysis... done. Value is 0.4272985 .
Diffuse analyis... done. 2047 beads identified.
Weights found. Total no of defective beads: 15567
Array 2 :
Generating neighbours matrix... done.
Compact analysis... done. 0 beads identified.
Extended analysis... done. Value is 0.01234762 .
Diffuse analyis... done. 4123 beads identified.
Weights found. Total no of defective beads: 4123
Array 6 :
Generating neighbours matrix... done.
Compact analysis... done. 8820 beads identified.
Extended analysis... done. Value is 0.0998686 .
Diffuse analyis... done. 1390 beads identified.
Weights found. Total no of defective beads: 10182
Array 7 :
Generating neighbours matrix... done.
Compact analysis... done. 501 beads identified.
Extended analysis... done. Value is 0.02312396 .
Diffuse analyis... done. 3251 beads identified.
Weights found. Total no of defective beads: 3686

> names(bashOutput)

[1] "wts"  "ext"  "call"

> length(bashOutput$wts)
```

```
[1] 10
```

```
> numBeads(BLData)
```

```
 [1] 49777 49777 49777 49777 49777 49777 49777 49777 49777 49777
```

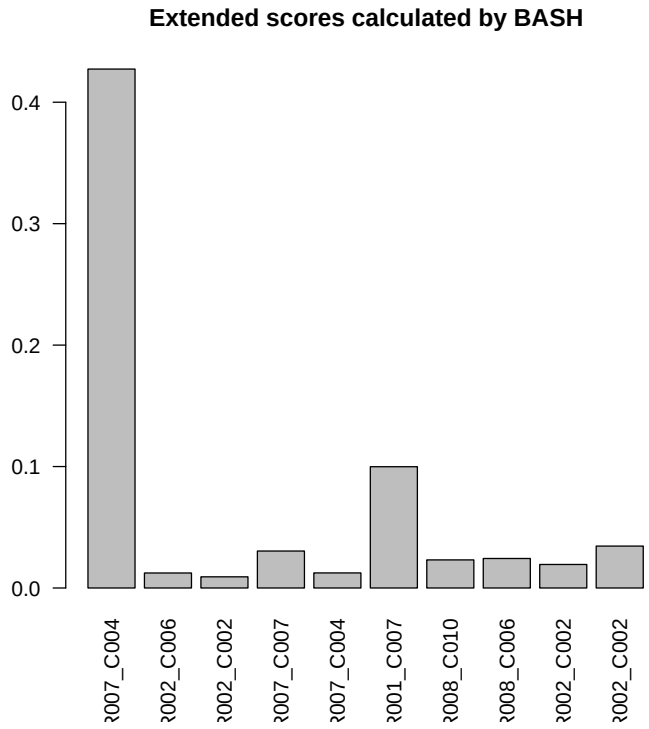
```
> table(bashOutput$wts[1])
```

```
    0     1
15567 34210
```

```
> bashOutput$ext
```

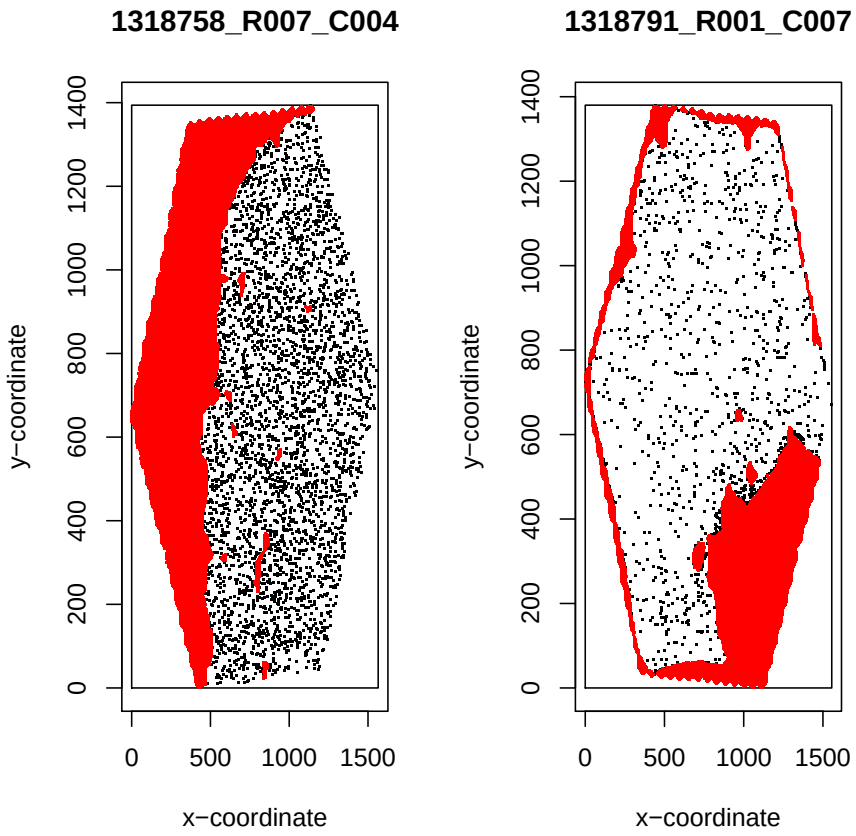
```
 [1] 0.42729849 0.01234762 0.00918654 0.03043032 0.01240257 0.09986860
 [7] 0.02312396 0.02428851 0.01936169 0.03451333
```

```
> barplot(bashOutput$ext, main = "Extended scores calculated by BASH",
+     las = 2, names = an)
```



If the weights are found to be suitable, the `setWeights` function can be used to store them in a modifed *BeadLevelList* object. The beads that have been masked can then be viewed by using the `showArrayMask` function. The plot produced shows the outliers calculated by Illumina in black, and the beads masked by BASH in red.

```
> BLData.BASHed = setWeights(BLData, bashOutput$wts, array = 1:10)
> par(mfrow = c(1, 2))
> showArrayMask(BLData.BASHed, array = 1)
> showArrayMask(BLData.BASHed, array = 6)
```



Notice that BASH tends to mask more beads in the obvious artefacts that we can see by eye, and also does not mask spurious outliers found on the rest of the array. Thus, we should expect more reliable expression measures after using BASH. If the mask that BASH produces is not deemed to be adequate, then it may be modified by a number the interactive tools `addArrayMask` and `clearArrayMask` where the user can select which beads to remove by hand.

There are also a number of ways that BASH may be modified to better suit the data being analyzed. If we feel that the diffuse analysis is picking too many small areas on the array we can change the `diffn` parameter which defines how many MADs away from the median an intensity must be for it to be labelled an outlier when finding outliers on the diffuse error image. Another parameter of interest is `diffsig`, which is the significance level of the binomial test performed in the diffuse analysis. `bgcorr` is used in diffuse analysis to attempt to compensate for the background varying across an array. For example, on a SAM array this should be left at "median", or maybe even switched to "none", but if analysing a large beadchip then you might consider setting this to "medianMAD".

The three types of defect may be detected separately using the functions `BASHCompact`, `BASHDiffuse` and `BASHExtended`, which will return a vector of bead IDs that are masked or the extended score. The extended defects step can also be skipped by setting the `extended` parameter to FALSE. It is a good idea to experiment with BASH and see which settings seem to work best for the data and the call to the function can be retrieved at any time by accessing the $ call item in the BASH output.

```
> output <- BASH(BLData, diffsig = 1e-05, array = 1)

Array 1 :
Generating neighbours matrix... done.
```

```
Compact analysis... done. 13567 beads identified.
Extended analysis... done. Value is 0.4272985 .
Diffuse analyis... done. 1733 beads identified.
Weights found. Total no of defective beads: 15272
```

```
> output <- BASH(BLData, diffn = 5, array = 1)
```

```
Array 1 :
Generating neighbours matrix... done.
Compact analysis... done. 13567 beads identified.
Extended analysis... done. Value is 0.4272985 .
Diffuse analyis... done. 1033 beads identified.
Weights found. Total no of defective beads: 14600
```

```
> output <- BASH(BLData, bgcorr = "none", array = 1)
```

```
Array 1 :
Generating neighbours matrix... done.
Compact analysis... done. 13567 beads identified.
Extended analysis... done. Value is 0.4272985 .
Diffuse analyis... done. 0 beads identified.
Weights found. Total no of defective beads: 13567
```

```
> output$call
```

```
BASH(BLData = BLData, array = 1, bgcorr = "none")
```

```
> output <- BASH(BLData, extended = FALSE, array = 1)
```

```
Array 1 :
Generating neighbours matrix... done.
Compact analysis... done. 13567 beads identified.
Diffuse analyis... done. 2047 beads identified.
Weights found. Total no of defective beads: 15567
```

Appendix - More detailed description of BASH functions

Generating neighbourhood networks and error images

`generateNeighbours` determines, for each bead on the array, which beads are next to it. It assumes that the beads are in a hexagonal lattice. The algorithm used first links each bead to its 6 closest neighbours. It then removes the longest link if its squared length is more than `thresh` multiplied by the squared length of the next longest link. A similar process is applied to the 2nd and 3rd longest links. Finally, any one way links are removed (i.e. a link between two beads is only preserved if each bead considers the other to be its neighbour). To ease computation, the algorithm only computes neighbours of beads in a square window of side length 2*(window) which travels across the array. Beads in a margin around the square, of width `(margin)`, are also considered as possible neighbours. The Neighbours matrix is designed for use with the BASH functions and should not need to be run by the user.

`generateE` creates an error image, usually based on bead residuals. This output can then be fed into `BASHDiffuse` or `BASHExtended`. If the parameter `what` is `residG`, `residR`, or `residM`, then residuals are calculated based on `method`. For other values of `what`, the residuals are not calculated. We then apply a "background filter" to this data, using the function `BGFilter` with arguments `bgfilter` and `invasions` - see its help file for more details. The background filter subtracts an estimate of the local background of the error image, and/or scales by the local MAD. This step is disabled by using `bgfilter = "none"`.

Compact defect detection

`BASHCompact` finds "compact defects" on an array. A compact defect is defined as a large connected cluster of outliers. This function first finds the outliers on an array. This is done via the function `findAllOutliers`. Next, using the Neighbours matrix and a Flood Fill algorithm, it determines which beads are in large connected clusters of outliers (of size larger than `cutoff`). These beads are then temporarily removed and the process repeated with the remaining beads. The repetition continues until either no large clusters of outliers remain, or until we have repeated the process `maxiter` times (and in this case, a warning will be given). In this way, we obtain a list of defective probes. Finally, we "close" the image, to fill in small gaps in the defect image. This consists of a "dilation" and an "erosion". In the dilation, we expand the defect image, by adding beads adjacent to defective beads into the defect image. This is repeated `cinvasions` times. In the erosion, we contract the defect image, by removing beads adjacent to non-defective beads from the defect image. (Erosion of the defect image is equivalent to a dilation of the non-defective image).

Diffuse defect detection

`BASHDiffuse` finds "diffuse defects" on an array. A diffuse defect is defined as a region containing an unusually large number of (not necessarily connected) outliers. Firstly, we consider the error image `E`, and find outlier beads on this image. Outliers for a particular bead type are determined using a 3 MAD cut-off from the median. We now consider an area around each bead (known as the "kernel"). The kernel is found by an invasion process using the neighbours matrix - we choose the beads which can be reached from the central bead in `cinvasions` steps. We count how many beads are in the kernel, and how many of these are marked as outliers. Using a binomial test, we work out if there are significantly more outliers in the kernel than would be expected if the outliers were equally distributed over the entire array. If so, then the central bead is marked as a diffuse defect. Lastly, we run a clustering algorithm and a closing algorithm similar to those in `BASHCompact`.

```
> sessionInfo()

R version 2.8.0 alpha (2008-10-04 r46598)
i386-pc-mingw32

locale:
LC_COLLATE=English_United Kingdom.1252;LC_CTYPE=English_United Kingdom.1252;LC_MONETARY=English_United

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

other attached packages:
 [1] beadarray_1.9.9.1    sma_0.5.15           hwriter_0.93
 [4] affy_1.19.4          preprocessCore_1.3.4 affyio_1.9.1
 [7] geneplotter_1.19.6   annotate_1.19.2      xtable_1.5-4
[10] AnnotationDbi_1.3.12 RSQLite_0.7-0        DBI_0.2-4
[13] lattice_0.17-15      Biobase_2.1.7        limma_2.15.14
[16] weaver_1.7.0         codetools_0.2-1      digest_0.3.1

loaded via a namespace (and not attached):
[1] grid_2.8.0         KernSmooth_2.22-22 RColorBrewer_1.0-2
```

Acknowledgements

BASH was developed by Jonathan Cairns and Andy Lynch with contributions from Mark Dunning and Matthew Ritchie. We are grateful to Barbara Stranger and Matthew Forrest for allowing us to use their data in this tutorial.