Description of simpleaffy: easy analysis routines for Affymetrix data

Crispin J Miller

October 13, 2014

Contents

1 Introduction

The *simpleaffy* package is part of the Bioconductor[1] project. It it was written to provide a starting point for exploring Affymetrix data, and to provide functions for some of the most common tasks we found ourselves doing over and over again. It is based on the *affy* package, which does does most of the hard work.

affy provides a variety of functions for processing Affymetrix data, with many more in *affycomp*. Even so, some tasks (such as computing t-tests and fold changes between replicate groups, plotting scatter-plots and generating tables of annotated 'hits') require a bit of coding, and some of the most commonly used functions can be a bit slower than

[1] `http://www.bioconductor.org/`

we would like. This package aims to provide high-level methods to perform these routine analysis tasks, and many of them have been re-implemented in C for speed.

Since *simpleaffy* is written over the top of the *affy* package, a basic understanding of the library and its vignette is a good idea. We also assume that the reader knows how the Affymetrix system works. If not, a brief introduction can be found at `http://bioinf.picr.man.ac.uk/`; a more detailed description is in the Affymetrix MAS manual at `http://www.affymetrix.com`.

2 Reading in data and generating expression calls

The first thing you need to do is to get R to use the *simpleaffy* package by telling it to load the library:

```
> library(simpleaffy) ##the affy package also gets loaded
> library(affy)
```

(NB: The examples in the vignette are hypothetical - we are putting together a data package containing a complete experimental dataset, when that's done, we will use these as the basis for the example code in this document).

Affymetrix data is generated by processing an image of the microarray (stored in a .DAT file) to produce a .CEL file, which contains, for each probe on the array, a single number defining its intensity. These are combined for each *probeset* using an algorithm such as RMA or MAS 5.0, to generate an expression level call for each transcript represented on the chip. Both the *affy* and *simpleaffy* packages work on the data in .CEL files, so we need to load them into R. In order to compute means, fold changes and t-tests, *simpleaffy* needs to know about the replicates in your experiment, so we must also load some descriptive data that says which arrays were replicates and also something about the different experimental conditions you were testing. This means that *simpleaffy* needs *two things*:

1. your .CEL files, and
2. a white-space delimited file describing the samples that went on them.

By default, this file is called *covdesc*. The first column should have no header, and contains the names of the .CEL files you want to process. Each remaining column is used to describe something in the experiment you want to study. For example you might have a set of chips produced by treating a cell line with two drugs. Your *covdesc* file might look like something like this:

	treatment
ctrl1.cel	n
ctrl2.cel	n
ctrl3.cel	n
a1.cel	a
a2.cel	a
a3.cel	a
b1.cel	b
b2.cel	b
b3.cel	b
ab1.cel	a.b
ab2.cel	a.b

Sooo, the easiest way to get going is it:

1. Create a directory, move all the relevant *CEL* files to that directory
2. Create a *covdesc* file and put it in the same directory
3. If using linux/unix, start R in that directory.
4. If using the Rgui for Microsoft Windows make sure your working directory contains the *Cel* files (use "File -> Change Dir" menu item).
5. Load the library.

```
> library(simpleaffy) ##load the simpleaffy package
>                     ##(which loads the affy package too)
```

6. Read in the data and generate expression calls, (using RMA), for example.

```
R> library(simpleaffy)
R> raw.data <-  read.affy() ##read data in working directory
R> x.rma <- call.exprs(raw.data,"rma")
R> # alternatively, use MAS 5.0: x.mas <- call.exprs(raw.data,"mas5")
```

Take a look at the help files for a more detailed description of these functions (i.e. ?read.affy or ?call.exprs).

The function `justMAS` provides a faster implementation of the MAS 5.0 expression summary algorithm written in C. (described in: Hubbell et al. (2002) Robust Estimators for expression analysis. Bioinformatics 18(12) 1585-1592), and in Affymetrix's 'Statistical Algorithms Description Document' that can be found on their website at `http://www.affymetrix.com`). As with any implementation of an algorithm, variations can occur. The simpleaffy website, `http://bioinf.picr.man.ac.uk/simpleaffy`, describes what testing was done: you should be aware of these differences, and if in any doubt, use MAS5.0 or GCOS to generate your data.

By default, `justMAS` is used by the `call.exprs` to generate the MAS5 calls, to use the `expresso` version specify 'mas5-R' instead of 'mas5' when you invoke `call.exprs`.

`justMAS` has not been tested on every chip type available - the majority of development has been on HGU95A arrays and newer. For more details of what chips it has been tested on, how the testing was done, and the results of the comparisons, see our website: `http://bioinformatics.picr.man.ac.uk/simpleaffy`.

3 Quality Control

One of the nice things about the Affymetrix platform is the collection of QC metrics (and accompanying guidelines) that are available. These may help flag up arrays that may have problems, and are described in detail in the Affymetrix *Data Analysis Fundamentals* manual, which can be found on their website at `http://www.affymetrix.com`.

Also look at the document 'QC and simpleaffy', which accompanies this one. It discusses QC metrics in significantly more detail than this vignette. A brief introduction can be found below...

The function `qc` generates the most commonly used metrics:

1. Average background
2. Scale factor
3. Number of genes called present (see the note on detection p-values) described below.
4. 3' to 5' ratios

All of these stats are parameters computed for/from the MAS 5.0 algorithm. Affy's QC requires that the Scale Factor for all chips are within 3-fold of one another, and that the average background and percent present calls for each chip are 'similar'. Affy chips also use probes at the 3' and 5' ends of the (generally) GAPDH and beta-actin genes to measure RNA quality, and additional probes spiked in during the latter stages of the sample preparation process are used to verify hybridisation efficiency. These probes (BioB,BioC,BioD,and CreX) should be present in increasing intensity.

The function `qc` produces an object of class 'QCStats' containing QC metrics for each array in a project. It takes both processed and raw data, since it makes use of numbers generated during the mas 5.0 expression calling algorithm. Alternatively, it can be called with just the raw data, in which case it calls `call.exprs(x,"mas5")` internally. For example,

```
R> x <- read.affy("covdesc");
R> x.mas5 <- call.exprs(x,"mas5");
R> qcs <- qc(x,x.mas5);
```

Note that we have used `call.exprs(x,"mas5")` to normalise the data using the mas 5.0 algorithm. This is important because `call.exprs` stores some additional data in the AffyBatch object's `description@preprocessing` slot:

```
R> x.mas5@description@preprocessing
```

`qc` returns an object containing scale-factors, % present, average, mininimum, maximum and mean background intensities,and bioB, bioC, bioD and creX present calls (1=present;0=not present). It also stores 3', 5' and M values for the QC probes. `ratios(qc)` generates a table of qc ratios for these probes. See ?qc for more details.

For some arrays, there are more than one probeset that target the gapdh and beta-actin genes. In this situation, we've attempted to make a sensible choice as to which probeset to use. To find out which probesets are used for your arrays use the methods `qc.get.ratios`, `qc.get.probes` and `qc.get.spikes`.

The data file `qcs` contains an example QCStats object for a 25 array project comparing different protocols for processing MCF7 and MCF10A data. This dataset is interesting because some of the arrays have poor qc.

```
> data(qcs)
```

The QCStats object (returned by qc), or what we just loaded with `data`, has a number of accessor functions to get at its values:

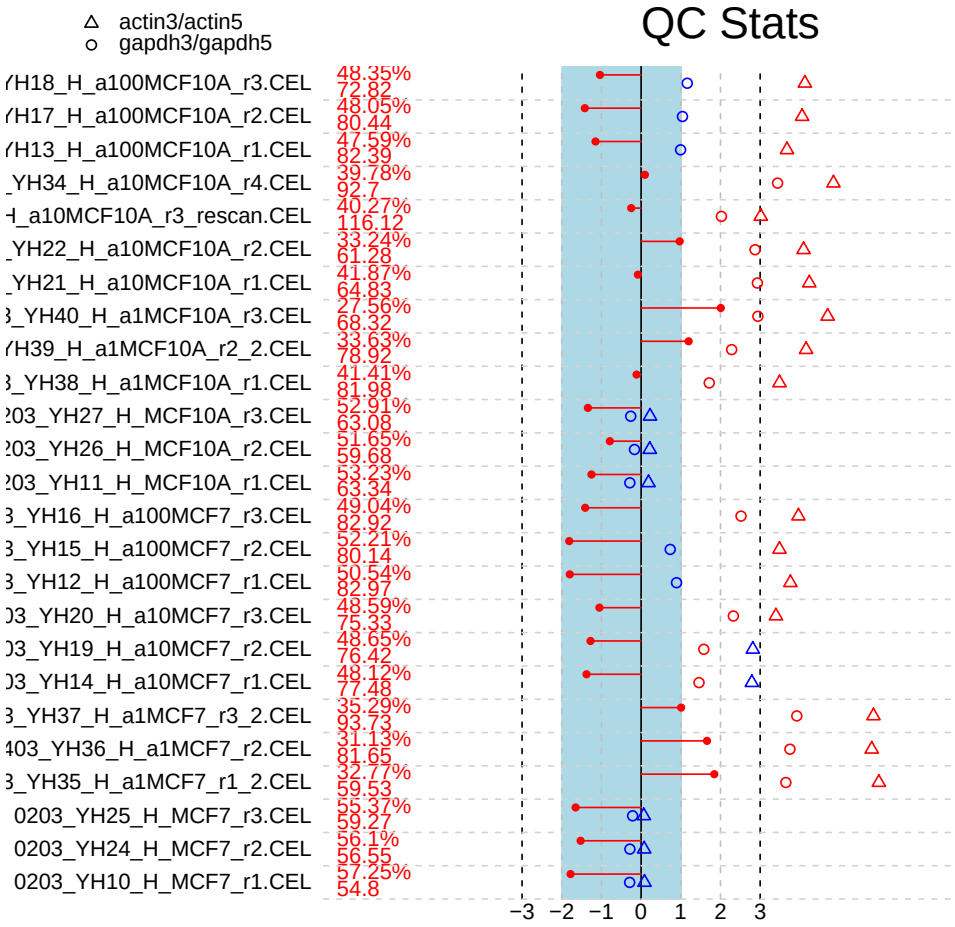
```
> ratios(qcs)
> avbg(qcs)
> maxbg(qcs)
> minbg(qcs)
> spikeInProbes(qcs)
> qcProbes(qcs)
> percent.present(qcs)
> plot(qcs)
> sfs(qcs)
> target(qcs)
> ratios(qcs)
```

Many of these also have 'set' methods - see section 5.4 for more details.

The qc functions and detection p value code have not been tested on every chip type available - the majority of development has been on HGU95A arrays and newer. Again, for more details of what chips it has been tested on, how the testing was done, and the results of the comparisons, see the simpleaffy website.

A plot of qc data can also be obtained by `plot(qc)`.

```
>  plot(qcs)
```



Each array is represented by a seperate line in the figure. The central vertical line corresponds to 0 fold change, the dotted lines on either side correspond to up and down regulation. The blue bar represents the region in which all arrays have scale factors within, by default, three-fold of each other. Its position is found by calculating the mean scale factor for all chips and placing the center of the region such that the borders are -1.5 fold up or down from the mean value.

Each array is plotted as a line from the 0-fold line to the point that corresponds to its scale factor. If the ends of all of the lines are in the blue region, their scale-factors are compatible. The lines are coloured blue if OK, red if not.

The figure also shows (for most arrays) GAPDH and beta-actin 3′/5′ratios. These are represented as a pair of points for each chip. Affy state that beta actin should be within 3, gapdh around 1. Any that fall outside these thresholds (1.25 for gapdh) are coloured red; the rest are blue.

Written along the left hand side of the figure are the number of genes called present on each array and the average background. These will vary according to the samples being processed, and Affy's QC suggests simply that they should be similar. If any chips have significantly different values this is flagged in red, otherwise the numbers are

displayed in blue. By default, 'significant' means that %-present are within 10% of each other; background intensity, 20 units. These last numbers are somewhat arbitrary and may need some tweaking to find values that suit the samples you're dealing with, and the overall nature of your setup.

Finally, if BioB is not present on a chip, this will be flagged by printing 'BioB' in red.

In short, everything in the figure should be blue - red highlights a problem!

4 Filtering by expression measures

When R loaded the .CEL files, it also used the data in the covdesc file to define which experimental groups the chips belonged to. The `get.array.subset` function makes it easy to select a subest of arrays from the experiment. For example,

```
R> get.array.subset(x.rma,"treatment",c("a","n"))
```

will return an `ExpressionSet` containing just the chips corresponding to treatment with drug 'a' or or with no drug at all.

The function `pairwise.comparison` allows you to take a subset of chips and perform the following analyses on it:

1. find means of the data,
2. compute log2 fold changes between the means,
3. compute a t-test between the groups,
4. possibly compute MAS5.0 style detection p values and Present/Marginal/Absent calls.

it returns a 'PairComp' object containing the results of the analysis:

```
R> results <- pairwise.comparison(
     x.rma,                  ## processed data
     "treatment",            ## the factor in covdesc
                             ## to use to group arrays
     c("n","a"),             ## groups to compare
     raw.data       ## for PMA calls
   )
```

This function is implemented in C for speed - and does everything in one go. There are no individual functions for fold-change and p-value, because it works out quicker simply to get everything and to discard the stuff you don't need.

Note that detection p values are computed using the function `detection.p.val`. This makes use of two parameters (alpha1 and alpha2) that are, like the control probes described above, dependent on the array type you're analysing. Use `getAlpha1` and `getAlpha2` to find out what values are being used for your array.

If you only have two groups of replicates - a simple control v. treatment experiment, for example, things are even easier, you do not have to specify the members to compare since the function can work it out from the group:

```
R> results <- pairwise.comparison(
     x.rma,                  ## processed data
     "treatment",            ## the factor in covdesc
     spots=raw.data          ## for PMA calls
   )
R> ## Find the 100 most changing genes
R> sort(abs(fc(results)),decreasing=TRUE)[1:100]
```

Since `call.exprs` *always* returns logged data, `pairwise.comparison` expects logged data by default – this can be changed with the `logged` parameter.

Averages can be calculated 3 ways:

1. From the unlogged values – i.e. log2(mean(replicates)).
2. From the logged values – i.e. mean(log2(replicates)).
3. From the median – i.e log2(median(replicates))

..and then the log2(fold change) values are simply worked out from these.

By default, unlogged values are used, but the parameter, `method`, allows you to change this by specifying method='unlogged','logged' or 'median'.

Note that if you have no replicates in one or both of your experimental groups - so that you have only one control or treatment chip, for example, the function returns a p-score of 0.0 for each t-test comparison *rather than generating an error* complaining that there are not enough observations.

The function `pairwise.filter` takes the output of `pairwise.comparison` and filters it for significantly changing genes:

```
R> # find genes expressed with an intensity
R> # greater than 10 on at least 6 chips,
R> # that show a fold change greater than 1.5
R> # and are significantly different
R> # between groups with a t-test p.value
R> # of 0.001 or better
R> significant <- pairwise.filter(
     results,
```

```
     min.exp=log2(10),
     min.exp.no=6, fc=log2(1.5),
     tt= 0.001)
```

If the pairwise comparison object was created from MAS 5.0 data, it can be filtered by Present Marginal Absent calls. Each gene is tested to see how many arrays it is called present on and this value used to decide whether the transcript passes the filter. Two ways of doing this are possible: the first simply sees if the transcript is Present on more than `min.present.no` arrays in the experiment. To do this, the parameter `present.by.group` must be set to FALSE.

The second makes use of the replicate groups used to calculate the `pairwise.comparison` object. In this case, the transcript must be present by more than `min.present.no` arrays in at least one of the two experimental groups. For this `present.by.group` must be TRUE. This is useful because it provides a way of trying to identify transcripts that have gone from 'off' to 'on', i.e. from being called Absent in one set to Present in the other, as well as those that are Present on all arrays.

For both methods of filtering, if `min.present.no="all"` is specified instead of a number , the transcript must be present in "all" arrays of a replicate group (or the whole experiment, depending on `present.by.group`).

For example:

```
R> # find genes present on all chips,
R> # that show a fold change greater than 1.5
R> # and are significantly different
R> # between groups with a t-test p.value
R> # of 0.001 or better
R> significant <- pairwise.filter(
     results,
     min.present.no="all",
     present.by.group=F, fc=log2(1.5),
     tt= 0.001)

R> # find genes present on all chips in at least one replicate group,
R> # that show a fold change greater than 1.5
R> # and are significantly different
R> # between groups with a t-test p.value
R> # of 0.001 or better
R> significant <- pairwise.filter(
     results,
     min.present.no="all",
     present.by.group=T, fc=log2(1.5),
     tt= 0.001)
```

4.1 Paired replicates

In the above example, we simply lumped all the replicates together when we calculated the t-tests. Ideally, we would like to design experiments so that each treatment is matched by a control sample that mirrors the protocols and processes it was subjected to as closely as possible.

This is particularly important if we are going to be processing replicates on separate days, or perhaps with different operators. A protocol that requires incubation on the bench for a couple of hours, say, might behave differently on a hot day and we would like the control samples to pick this sort of thing up.

The `pairwise.comparison` function lets us do a t-test with paired replicates, by specifying the ordering of replicates within groups.

```
R> results <- pairwise.comparison(
    x.rma,                  ## processed data
    "treatment", c("n","a"),## groups to compare
    raw.data,
    a.order=c(1,3,2),       ## a.1 matches b.1
    b.order=c(1,2,3))       ## a.3 matches b.2 etc...
R> ## Find the 100 most changing genes
R> sort(abs(fc(results)),decreasing=TRUE)[1:100]
```

Here, we have specified the order the replicate samples should be compared to each other - so that replicate 3 in group a is compared to replicate 2 in group b, and so on.

Again, the help pages for these functions explain more about them and the values they return.

5 Viewing results

The function `trad.scatter.plot` does a scatter plot between a pair of vectors:

```
R> trad.scatter.plot(exprs(x.rma)[,1],exprs(x.rma)[,3],
                     fc.line.col="lightblue",col="blue");
R> trad.scatter.plot(exprs(x.rma)[,2],exprs(x.rma)[,4],
                     add=T,col="red");
R> legend(2,12,c("Control v. treatment rep 1","Control v. treatment rep 2"),
                  col=c("blue","red"),pch=20)
```

Generic plot functions also exist for `pairwise.comparison` objects.

`plot(pairwise.comparison)` will do a straight scatter plot of the means of the two replicate groups. In addition the parameter `type` can be used:

```
R> plot(results,type="scatter") #scatter plot
R> plot(results,type="ma")      #M v A  plot
R> plot(results,type="volcano") #volcano plot
```

If PMA calls are available, the function will colour the points as follows:

1. Red – all present
2. Orange – all present in one group or the other
3. Yellow – all that remain

If a second `pairwise.comparison` object is supplied, then these points are drawn (by default) as blue circles. This allows the results of a pairwise.filter to be identified on the graph - eg.:

```
R> # find genes present on all chips in at least one replicate group,
R> # that show a fold change greater than 1.5
R> # and are significantly different
R> # between groups with a t-test p.value
R> # of 0.001 or better
R> results <- pairwise.comparison(
    x.rma,                  ## processed data
    "treatment", c("n","a"),## groups to compare
    raw.data,
    a.order=c(1,3,2),       ## a.1 matches b.1
    b.order=c(1,2,3))       ## a.3 matches b.2 etc...
R> significant <- pairwise.filter(
     results,
     min.present.no="all",
     present.by.group=T, fc=log2(1.5),
     tt= 0.001)
R> plot(results,significant)
```

5.1 Heatmaps

Simpleaffy also has a pair of functions for plotting simple heatmaps, `hmap.eset` and `hmap.pc` for AffyBatch and PairComp objects, respectively. The simplest use is as follows:

```
R> #generate a heatmap of the first 100 genes in an expression set.
R> eset <- read.affy()
R> eset.rma <- call.exprs(eset)
R> hmap.eset(eset.rma,1:100)
```

You can specify colours using the 'col' parameter either by giving the function a vector of colors (e.g. rainbow(21)) or by the following strings:

1. "bwr" – from blue to red via white

2. "rbg" – from red to green via black

3. "ryw" – from red to white via yellow

By default, colouring is such that the colours are scaled so that all the data in the heatmap fits in the colour range supplied. Alternatively, minimum and maximum values for the colour range can be specified. Clustering functions or trees generated by hclust or dendrogram can also be provided – by default 1-Pearson correlation is used. See ?hmap.eset for more details.

If you have the results of a pairwise.filter, you can use this to select the samples and probesets to plot:

```
R> #generate a heatmap of the first 100 probesets in an expression set.
R> eset <- read.affy()
R> eset.rma <- call.exprs(eset)
R> pc <- pairwise.comparison(eset.rma,''group'',c(``a'',''b''))
R> pc.f <- pairwise.filter(pc,tt=0.001,fc=1)
R> hmap.pc(pc.f,eset.rma)
```

By default, this scales the colouring for each probeset in terms of its standard deviation. This is done as follows:

1. The replicate groups used for the pairwise comparison are found. The s.d. for each probeset is calculated for each group, and the results averaged to give a mean standard deviation for each probeset.

2. Data are clustered as before

3. Each probeset is scaled by it standard deviation, and coloured according to how many s.d.s it is from its mean.

4. The samples used for the pariwise comparison are plotted (by default)

The parameter 'scale' can be used to turn this scaling off, and spread defines how many standard deviations above and below the mean should be shown. see ?hmap.pc for more details.

5.2 Printing

Another thing that can sometimes be tricky is producing figures for papers and presentations. Two utility functions, `journalpng` and `screenpng` make it easy to generate .png files at 300dpi (huge) and 72dpi. Most journals accept .pngs, and they can be converted into other formats using a decent graphics package.

R uses the concept of a *device* to deal with graphics. When you start to plot a graph, it looks for a graphics device to print it on. If it can't find one that's already there, it

opens a new one, which by default corresponds to a window on the screen. In order to generate a *file* (rather than a window) containing our figure, we can use `journalpng` and `screenpng` to create a new graphics device that R can use instead. This device results in our graph being plotted to a file on disk - when you type something like:

```
R> journalpng(file="results/figure1.png");
R> trad.scatter.plot(exprs(x.rma)[,1],exprs(x.rma)[,3],fc.line.col="lightblue");
R> dev.off();
```

a new png file is created, the scatterplot is printed into that file and then, when dev.off() is called, the data is saved. Note that because R is plotting to our `journalpng` device rather than the screen, we won't see any pictures (or anything happen at all).

5.3 Generating a table of results

It would also be nice to know what our changing genes actually do. The function `get.annotation` takes a list of probe set names along with a string specifying the type of array we are looking at, and uses these to look up annotation for our data. The resulting dataframe, when saved as a tab delimited .XLS file, loads into excel, with hyperlinks to the NCBI's Unigene and LocusLink databases.

Two functions `results.summary` and `write.annotation` are also useful - the former generates a summary table with expression data and associated annotation, the latter spits it out in tab delimited format (see `?get.annotation` for more details):

```
R> x     <- read.affy()
R> x.rma <- call.exprs(x,"rma")
R> pw       <- pairwise.comparison(x.rma,"drug",c("formulaX","nothing"))
R> pw.filtered <- pairwise.filter(pw)
R> summary     <- results.summary(pw.filtered,"hgu133a")
R> write.annotation(file="spreadsheet.xls",summary)
```

5.4 Specifying alternate QC configurations

If when running `qc`, one of the other qc functions, or `detection.p.val`, you get an error because simpleaffy doesn't recgnize the type of array in your project, or because, for example, you are using custom arrays, you need to specify your own QC parameters. These include alpha1 and alpha2 values for the array, the name of the spike control probesets and the pairs of probesets used to calculate the 5'/3' ratios. These can be then specified in a config file. This, for example, is the configuration for the hgu133 plus2 array:

```
array hgu133plus2cdf
alpha1 0.05
```

```
alpha2 0.065
spk bioB AFFX-r2-Ec-bioB-3_at
spk bioC AFFX-r2-Ec-bioC-3_at
spk bioD AFFX-r2-Ec-bioD-3_at
spk creX AFFX-r2-P1-cre-3_at
ratio actin3/actin5 AFFX-HSAC07/X00351_3_at AFFX-HSAC07/X00351_5_at
ratio actin3/actinM AFFX-HSAC07/X00351_3_at AFFX-HSAC07/X00351_M_at
ratio gapdh3/gapdh5 AFFX-HUMGAPDH/M33197_3_at AFFX-HUMGAPDH/M33197_5_at
ratio gapdh3/gapdhM AFFX-HUMGAPDH/M33197_3_at AFFX-HUMGAPDH/M33197_M_at
```

The file must be called <cdfname>.qcdef, where cdfname is the 'clean' cdfname produced by calling `cleancdfname` on the `cdfName` of the appropriate AffyBatch object. For example:

```
R> n <- cdfName(eset)
R> cleancdfname(n)

[1] "hgu133plus2cdf"
```

The first token on each line specfies what the rest of the line describes. Allowable tokens are: 'array' (array name), 'alpha1', 'alpha2' (alpha values), 'spk' (the name of a spike probeset – must be followed by one of 'bioB','bioC', 'bioD' or 'creX'), or 'ratio' (to specify pairs to comput 3'/5' or 3'/M metrics). 'ratio' must be followed by the name of the ratio, which must be of the form 'name1/name2', where name1 and name2 are textual identifiers suitable for displaying in the qc plot, for example. This must then be followed by two probeset ids.

Once the file is created, the QC environment can be set up with a call to `setQCEnvironment(array,path)` specifying the appropriate cdfname and the path to the directory containing the file.

MORE on changes to follow...