Textual description of affy

Laurent Gautier, Rafael Irizarry, Leslie Cope, and Ben Bolstad

November 12, 2002

Contents

1 Introduction

The `affy` package is part of the Bioconductor[1] project. It is meant to be an extensible, interactive environment for data analysis and exploration of Affymetrix oligonucleotide array probe level data.

The software utilities provided with the Affymetrix software suite summarizes the probe set intensities to form one *expression measure* for each gene. The expression measure is the data available for analysis. However, as pointed out by Li and Wong (2001), much can be learned from studying the individual probe intensities, or as we call them, the *probe level data*. This is why we developed this package. The package includes plotting functions for the probe level data useful for quality control, RNA degradation assessments, different probe level normalization and background correction procedures, and flexible functions that permit the user to convert probe level data to expression measures. The package includes utilities for computing MAS 4.0's AvDiff (Affymetrix, 1999), MAS 5.0's signal (Affymetrix, 2001), DChip's MBEI (Li and Wong, 2001), and RMA (Irizarry et al., 2003b).

We assume that the reader is already familiar with oligonucleotide arrays and with the design of the Affymetrix GeneChip arrays. If you are not, we recommend the Appendix of the Affymetrix MAS manual Affymetrix (1999, 2001).

The following terms are used throughout this document:

probe oligonucleotides of 25 base pair length used to probe RNA targets.

perfect match probes intended to match perfectly the target sequence.

PM intensity value read from the perfect matches.

mismatch the probes having one base mismatch with the target sequence intended to account for non-specific binding.

MM intensity value read from the mis-matches.

probe pair a unit composed of a perfect match and its mismatch.

affyID an identification for a probe set (which can be a gene or a fraction of a gene) represented on the array.

probe pair set PMs and MMs related to a common *affyID*.

CEL **files** contain measured intensities and locations for an array that has been hybridized.

CDF **file** contain the information relating probe pair sets to locations on the array.

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

Section 2 described the main differences between version 1.0 and this version. Section 3 describes a quick way of getting started and getting expression measures. Section 4 describes the different classes defined in the package. Section 5 describes normalization routines. Section 6 describes the different classes in the package. 7 describes our strategy to map probe locations to probe set membership. Section 8 describes how to change the default options on the package.

Note: If you use this package please cite Irizarry et al. (2003a).

2 Changes from Version 1.0

What's new?

- Faster reading functions (type `?read.affybatch`)
- Widgets for reading phenotypic and MIAME information and choices of preprocessing when computing expression measures. (`?read.phenoData, ?read.MIAME, ?expresso`)
- No need to read in *CDF* files.
- More efficient expression measure functions. (`?expresso`, `?express`).
- Very fast RMA (`?rma`).
- MAS 5.0 available (`?expresso`).
- RNA degradation assessment. (`?AffyRNAdeg`)

In version 2.0, we have implemented various of the procedures described in the MAS 5.0 manual. Throughout the package we use `mas` to denote the procedures described in the MAS 5.0 software.

The main difference between Version 1.0 and version 2.0 is that the user no longer needs to provide the *CDF* files. We now provide a more efficient way of obtaining this information. Data packages containing the necessary CDF information can be obtained from `http://www.bioconductor.org/data/cdfenvs/cdfenvs.html`. Simply download as many of these *cdf environments* as you need and install them. The affy package will know where to look. If you are using the **HGU95Av2** or **HGU133A** chip the information is included in the affy package and you do not need to download further packages. You can also create your own cdf environments. See Section 7 for information on how the environments work. A cdf environment making package is available from the Bioconductor web site `www.bioconductor.org`.

The new version provides a unified approach to working with probe level data. *AffyBatch* is the main class the user will manipulate. We believe it combines the simplicity of the former *Plob* with the flexibility of the former *Cel.container*. As before, it bundles

the data from a *batch* of experiments. The class *Cel* models the data from a single experiment (coming from a CEL file). The classes *Cdf* contains the information of *CDF* file, the class *ProbeSet* contain PM and MM intensities for a particular probe set. Beginners need do not understand these classes. However, they are briefly described in Section 6.

There are some minor differences in what you can do but little functionality has disappeared. Much has been added!

3 Getting Started: From probe level data to expression values

The first thing you need to do is **load the package**.

```
R> library(affy) ##load the affy package
```

And download and install the cdf environment you need from `http://www.bioconductor.org/data/cdfenvs/cdfenvs.html` (except if you are using the HGU95Av2 or HGU133A chip or making your own).

3.1 Quick start

If all you want is to go from probe level data (*Cel* files) to expression measures here are some quick ways.

The quickest way of reading in data and getting expression measures is the following:

1. Create a directory, move all the relevant *CEL* files to that directory
2. Start R in that directory.
3. If using Microsoft Windows make sure your working directory contains the *Cel* files (the function `getwd`, `setwd`, and `list.celfiles` are useful for this).
4. Load the library.

   ```
   R> library(affy) ##load the affy package
   ```

5. Read in the data and create an expression, using RMA for example.

   ```
   R> Data <-  ReadAffy() ##read data in working directory
   R> eset <- rma(Data)
   ```

The `rma` function was written in C for speed and efficiency. It uses the expression measure described in Irizarry et al. (2003b).

For other popular methods use `expresso` instead of `rma`. For example for MAS 5.0 signal you can use

```
R> eset <- expresso(Data, normalize=FALSE, bgcorrect.method="mas",
                    pmcorrect.method="mas",summary.method="mas")
```

For creating your own expression measures you can use `express`

```
R> Data <-  ReadAffy() ##read data in working directory
R> eset <- expres(Data, summary.method=function(x) apply(x,2,median))
```

In all the above examples, the variable `eset` is an object of class `exprSet` described in the Biobase vignette. Many of the packages in Bioconductor work on objects of this class. See the genefilter and geneplotter packages for some examples.

If you want to use some other analysis package you can write out the expression values to file using the following command:

```
R> write.exprs(eset, file="mydata.txt")
```

3.2 Reading CEL file information

The function `ReadAffy` is quite flexible. It lets you specify the filenames, phenotypic, and MIAME information. You can enter them by reading files (see the help file) or widgets (you need to have the tkWidgets package installed and working)

```
R> Data <-  ReadAffy(widget=TRUE) ##read data in working directory
```

This function call will pop-up a file browser widget, see Figure 1, that provides an easy way of choosing cel files.

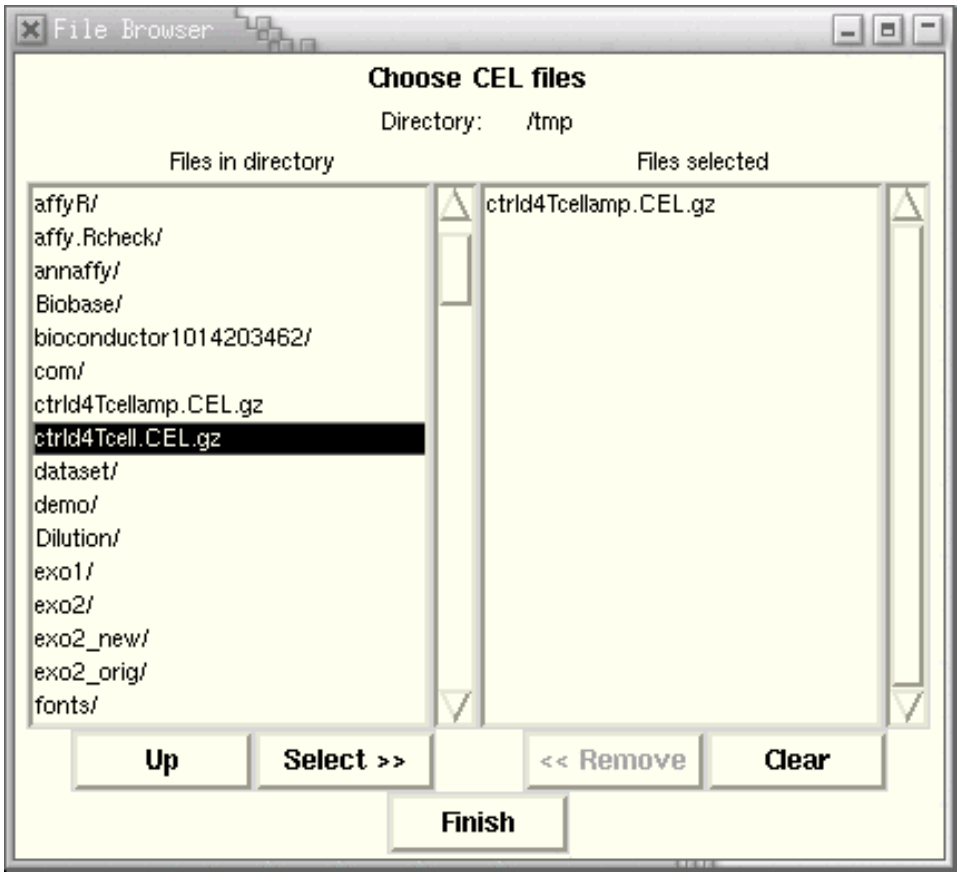

Figure 1: Graphical display for selecting *CEL* files. This widget is part of the *tkWidgets* package.

Next, a widget (not shown) permits the user to enter sample names for the arrays and descriptions for these samples. In the next step a series of three widgets permits the user to enter phenotypic or covariate information. See Figure 2.

Finally the a widget is presented for the user to enter MIAME information. Seen in Figure 3.

Notice that it is not necessary to use widgets to enter this information. Please read the help file for more information on how to read it from flat files or to enter it by hand.

The function `ReadAffy` is a wrapper for the functions `read.affybatch`, `tkSampleNames`, `read.phenoData`, and `read.MIAME`. The function `read.affybatch` has some nice feature that make it quite flexible. For example, the `compression` argument permit the user to read compressed *CEL* files. The argument *compress* set to *TRUE* will inform the readers that your files are compressed and let you read them while they remain compressed. The compression formats *zip* and *gzip* are known to be recognized.

A comprehensive description of all these options is found in the help file:

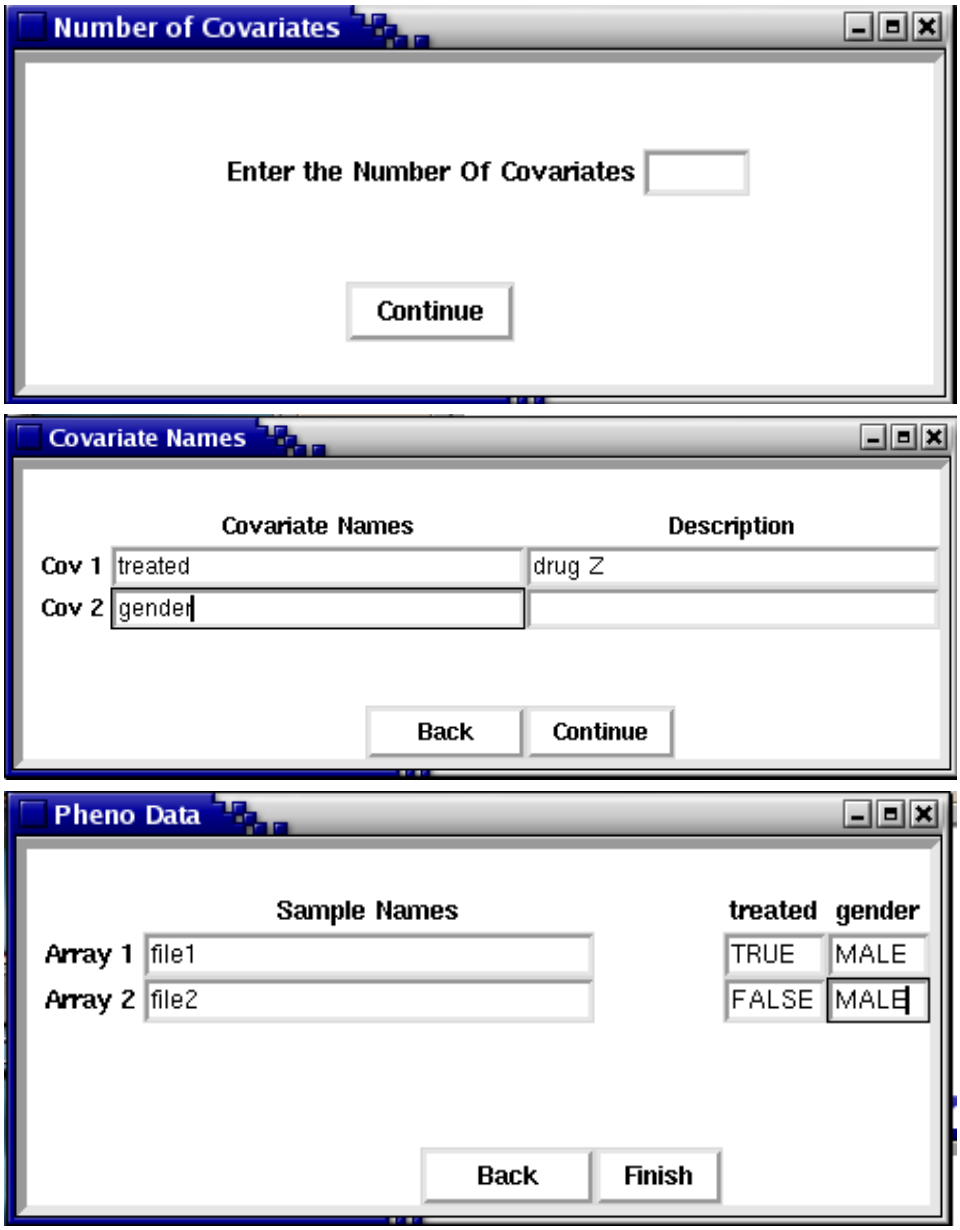

Figure 2: Graphical display for entering phenoData This widget is part of the *tkWidgets* package.

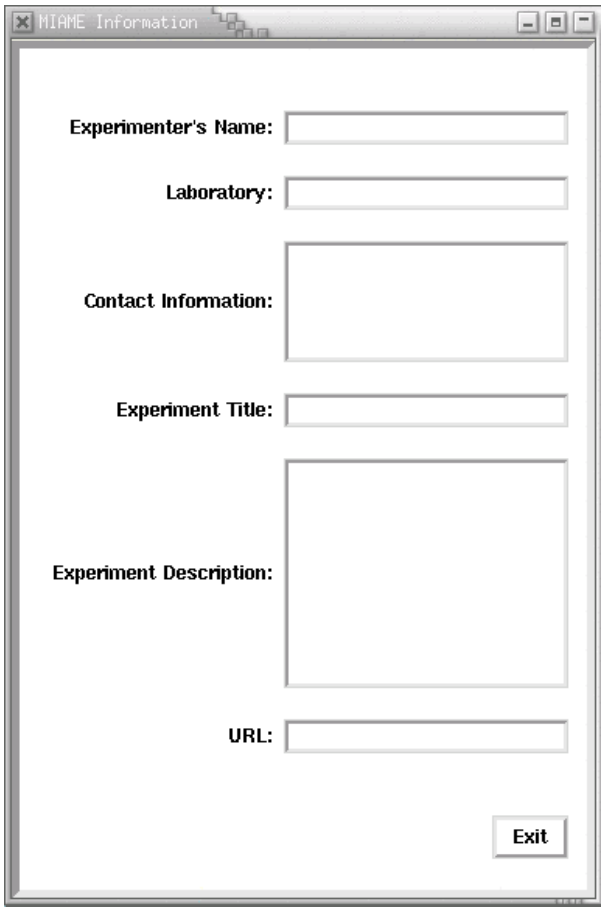

Figure 3: Graphical display for entering *MIAME* informations. This widget is part of the *tkWidgets* package.

```
R> ?read.affybatch
R> ?read.phenoData
R> ?read.MIAME
```

3.3 Expression measures

The most common operation is certainly to convert probe level data to expression values. Typically this is achieved through the following sequence:

1. reading in probe level data.
2. background correction.
3. normalization.
4. probe specific background correction, e.g. subtracting MM.
5. summarizing the probe set values into one expression measure and, in some cases, a standard error for this summary.

We detail what we believe is a good way to proceed below. As mentioned the function `expresso` provides many options. For example,

```
R> eset <- expresso(affybatch, normalize.method="qspline", bg.method="rma",
                    summary.method="liwong")
```

This will store expression values, in the object `eset`, as an object of class `exprSet` (see the Biobase package). You can either use R and the Bioconductor packages to analyze your expression data or if you rather use another package you can write it out to a tab delimited file like this

```
R> write.exprs(eset, file="mydata.txt")
```

In the `mydata.txt` file, row will represent genes and columns will represent samples/arrays. The first row will be a header describing the columns. The first column will have the *affyID*s. The `write.exprs` function is quite flexible on what it writes (see the help file).

The function `rma` computes the RMA expression measure. Because it is hard coded in C it is quite fast and efficient. The function `express` lets the user define their own functions and send them as arguments.

The function *expresso* performs the steps background correction, normalization, probe specific correction, and summary value computation. This function too has several parameters to facilitate your work. We now show this using an `AffyBatch` included in the package for examples. The `data(affybatch.example)` is used to load these data.

background.method . The background correction method to use. The available methods are

```
> bgcorrect.methods
```

```
[1] "mas"  "none" "rma"
```

normalize.method . The normalization method to use. The available methods can be queried by using *normalize.methods*.

```
> data(affybatch.example)
> normalize.methods(affybatch.example)
```

```
[1] "constant"         "contrasts"        "invariantset"     "loess"
[5] "qspline"          "quantiles"        "quantiles.robust"
```

pmcorrect.method The method for probe specific correction. The available methods are

```
> pmcorrect.methods
```

```
[1] "mas"        "pmonly"     "subtractmm"
```

summary.method . The summary method to use. The available methods are

```
> express.summary.stat.methods
```

```
[1] "avgdiff"      "liwong"       "mas"          "medianpolish" "playerout"
```

Here use `mas` to refer to the methods described in the Affymetrix manual version 5.0.

widget Making the `widget` argument `TRUE`, will let you select missing parameters (like the normalization method, the background correction method or the summary method). Figure 4 presents the widgets for the selection of a different methods.

```
R> expresso(affybatch.example, widget=TRUE)
```

3.4 MAS 5.0

To obtain MAS 5.0 use

```
eset <- expresso(affybatch.example, normalize=FALSE, bgcorrect.method="mas",
                 pmcorrect.method="mas",summary.method="mas")
eset <- affy.scalevalue.exprSet(eset)
```

Notice that normalization occurs after we obtain expression measures. The function `affy.scalevalue.exprSet` performs a normalization similar to that described in the MAS 5.0 manual.

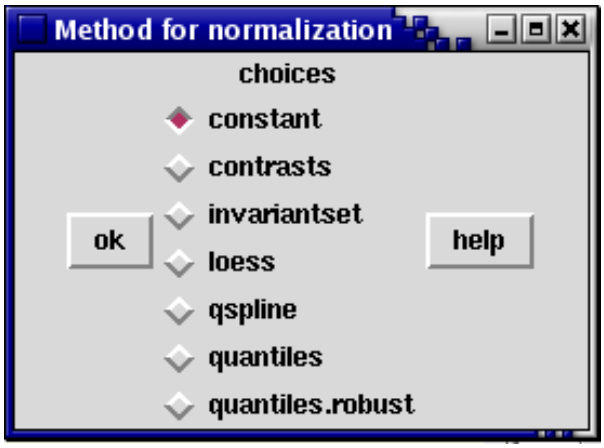

Figure 4: Graphical display for selecting a normalization method.

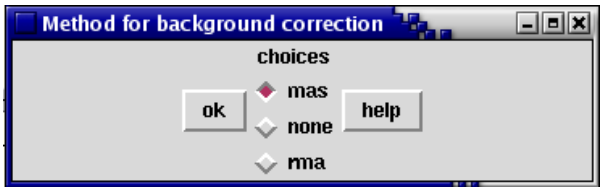

Figure 5: Graphical display for selecting a background correction method.

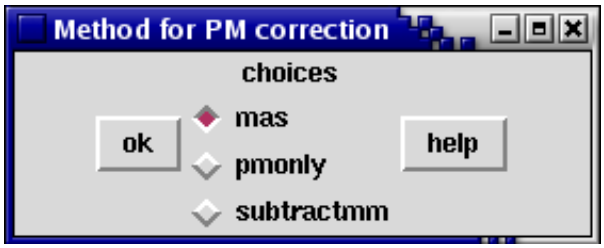

Figure 6: Graphical display for selecting a method for PM value adjustement.

Figure 7: Graphical display for selecting a method for computing an summary expression value from the probe intensities.

3.5 Li and Wong's MBEI (dchip)

To obtain a measure similar to Li and Wong's MBEI one can use

```
%<<>>=
eset <- expresso(affybatch.example, normalize.method="invarianteset",
                 bg.correct=FALSE,
                 pmcorrect.method="pmonly",summary.method="liwong")
%@
```

This gives the current PM-only default. The reduced model (previous default) can be obtained using `pmcorrect.method="subtractmm"`.

3.6 C implementation of RMA

One of the quickest ways to compute expression using the `affy` package is to use the `rma` function. We have found that this method allows a user to compute the RMA expression measure in a matter of minutes for datasets that may have taken hours in previous versions of `affy`. The function serves as an interface to a hard coded C implementation of the RMA method Irizarry et al. (2003b). Generally, the following would be sufficient to compute RMA expression measures:

```
> eset <- rma(affybatch.example)
```

Currently the `rma` function implements RMA in the following manner

1. Probe specific correction of the PM probes using a model based on observed intensity being the sum of signal and noise
2. Normalization of corrected PM probes using quantile normalization Bolstad et al. (2002)

3. Calculation of Expression measure using median polish.

The `rma` function is likely to be improved and extended in the future as the RMA method is fine-tuned.

4 Quality Control through Data Exploration

Several of the functions for plotting summarized probe level data are useful for diagnosing problems with the data. The plotting functions `boxplot` and `hist` have methods for `AffyBatch` objects. Each of these functions presents side-by-side graphical summaries of intensity information from each array. Important differences in the distribution of intensities are often evident in these plots. The function mva.pairs (applied to `pm(afbatch)`), offers pairwise graphical comparison of intensity data. These plots can be particularly useful in diagnosing problems in replicate sets of arrays.

For the users convenience we have included a sample data set containing part of the data from a Dilution experiment. The full data is publicly available from Gene Logic `http://qolotus02.genelogic.com/datasets.nsf/` and described in Irizarry et al. (2003b). The help file obtained via `?Dilution` describes the data set.

```
> data(Dilution)
> Dilution

AffyBatch object
size of arrays=640x640 features (12805 kb)
cdf=HG_U95Av2 (12625 affyids)
number of samples=4
number of genes=12625
annotation=hgu95av2
notes=
```

This will create the `Dilution` object of class `AffyBatch`. `print` (or `show`) will display summary information. These objects represent data from one experiment. The `AffyBatch` class combines the information of various *CEL* files with a common *CDF* file. This class is designed to keep information of one experiment. The probe level data is contained in this object.

The data in `Dilution` is a small sample of probe sets from 2 sets of duplicate arrays hybridized with different concentrations of the same RNA. This information is part of the `AffyBatch` and can be accessed with the `phenoData` and `pData` methods:

```
> phenoData(Dilution)

          phenoData object with 3 variables and 4 cases
          varLabels
                 liver: amount of liver RNA hybridized to array in micrograms
                 sn19: amount of central nervous system RNA hybridized to array in micro
                 scanner: ID number of scanner used

> pData(Dilution)
```

```
    liver sn19 scanner
20A    20    0       1
20B    20    0       2
10A    10    0       1
10B    10    0       2
```

4.1 Accessing PM and MM Data

The PM and MM intensities and corresponding *affyID* can be accessed with the `pm`, `mm`, and `probeNames` methods. These will be matrices with rows representing probe pairs and columns representing arrays. The gene name associated with the probe pair in row i can be found in the ith entry of the vector returned by `probeNames`.

```
> Index <- c(1, 2, 3, 100, 1000, 2000)
> pm(Dilution)[Index, ]

            20A   20B   10A   10B
1000_at1  468.8 282.3 433.0 198.0
1000_at2  430.0 265.0 308.5 192.8
1000_at3  182.3 115.0 138.0  86.3
1006_at4  264.0 151.0 167.0 103.3
1057_at8  152.3 113.0 135.0  88.8
1114_at10 275.0 155.5 194.3 124.5

> mm(Dilution)[Index, ]

             20A   20B   10A   10B
1000_at1  1123.5 673.0 693.5 434.5
1000_at2   259.0 175.3 194.0 110.3
1000_at3   160.0  95.0 119.3  72.5
1006_at4   180.3 102.5 109.0  74.0
1057_at8   178.8 126.8 156.3  83.5
1114_at10  478.0 284.0 305.0 212.3

> probeNames(Dilution)[Index]

[1] "1000_at" "1000_at" "1000_at" "1006_at" "1057_at" "1114_at"
```

`Index` contains six arbitrary probe positions.

Notice that the column names of PM and MM matrices are the sample names and the row names are the *affyID*, e.g. `1000_at` and `1006_at` together with the probe number (related to position in the target sequence).

```
> sampleNames(Dilution)
```

```
[1] "20A" "20B" "10A" "10B"
```

Quick example: To see what percentage of the MM are larger than the PM simply type

```
> mean(mm(Dilution) > pm(Dilution))
```

```
[1] 0.2746048
```

The `pm` and `mm` functions can be used to extract specific probe set intensities.

```
> gn <- geneNames(Dilution)
> pm(Dilution, gn[1000])
```

```
              20A   20B    10A   10B
189_s_at1   118.0  69.5   79.0  59.3
189_s_at2   711.8 435.3  561.8 282.8
189_s_at3   894.0 592.3  720.5 364.0
189_s_at4   184.0 123.0  185.5 101.8
189_s_at5   281.0 187.8  246.0 133.3
189_s_at6   581.0 361.3  425.0 239.0
189_s_at7   140.0 100.0  106.3  79.5
189_s_at8   384.3 252.0  269.0 167.0
189_s_at9  1136.0 693.3  827.8 442.0
189_s_at10  258.8 143.0  206.0 136.5
189_s_at11 1616.8 975.3 1020.5 625.3
189_s_at12  354.0 250.0  282.0 171.0
189_s_at13  340.0 196.3  251.0 133.0
189_s_at14 1244.5 716.0  712.8 476.0
189_s_at15  119.0  79.8   95.0  63.3
189_s_at16  116.0  90.3   93.3  63.5
```

The method `geneNames` extracts the unique *affyID*s. Also notice that the 1000th probe set is different from the 1000th probe! The 1000th probe is not part of the the 1000th probe set.

The methods `boxplot`, `hist`, and `image` are useful for quality control. Figure 8 shows histograms of PM intensities for the 1st and 2nd array of the `affybatch.example` also included in the package

4.2 Histograms, Images, and Boxplots

Figure 8 provides evidence of saturation in that we see two small "bumps" near the largest value.

```
> data(affybatch.example)
> hist(affybatch.example[1:2])
```

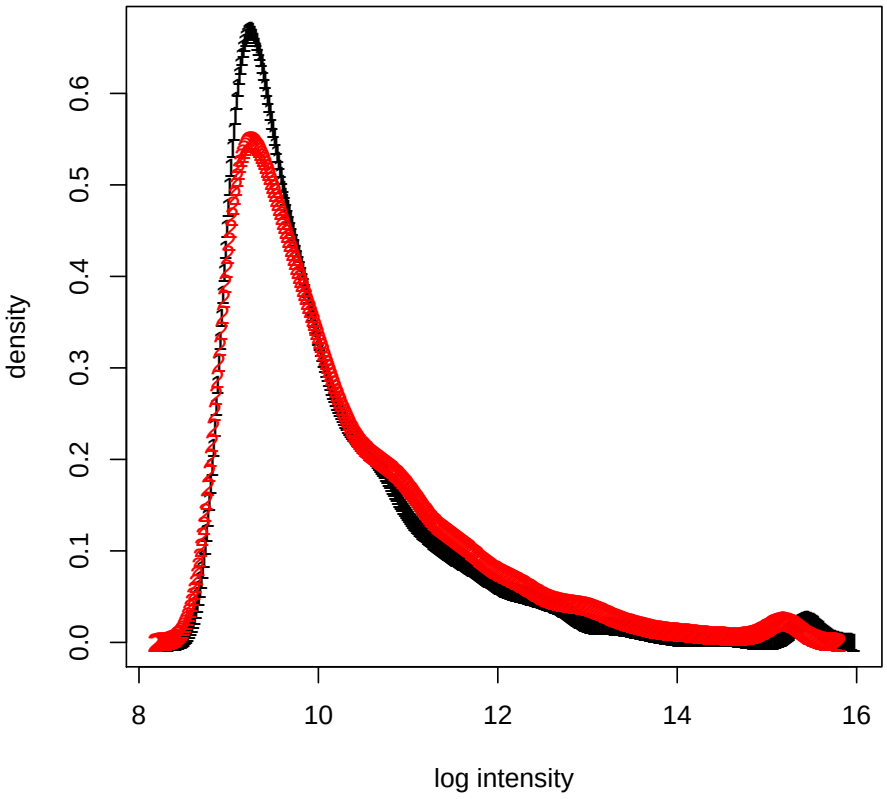

Figure 8: Histogram of PM intensities for 1st and 2nd array

As seen in the previous example, the sub-setting method `[` can be used to extract specific arrays. **NOTE: Sub-setting is different in version 2.0. One can no longer subset by gene. We can only define subsets by one dimension: the columns, i.e. the arrays.**

The method `image()` can be used to detect spatial artifacts. By default we look at log transformed intensities. This can be changed through the `transfo` argument.

```
> data(affybatch.example)
> par(mfrow = c(2, 2))
> image(affybatch.example)
```

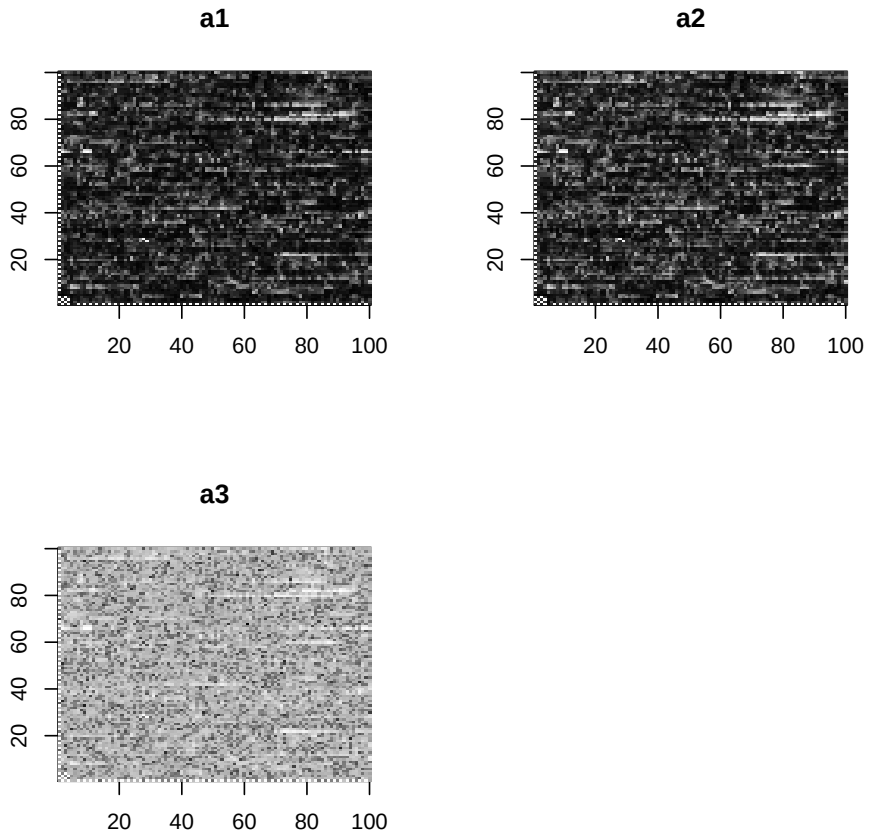

Figure 9: Image of the log intensities.

These images are quite useful for quality control. We recommend examining these images as a first step in data exploration.

The method `boxplot` can be used to show PM, MM or both intensities. As discussed

```
> par(mfrow = c(1, 1))
> boxplot(Dilution, col = c(2, 2, 3, 3))
```

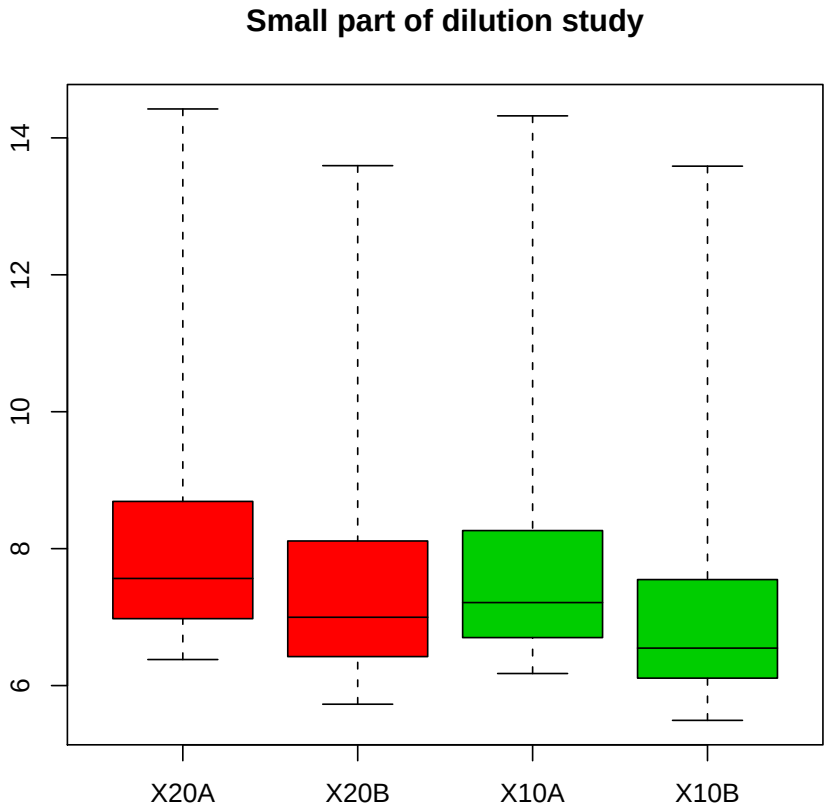

Figure 10: Boxplot of arrays in dilution data.

in the next section this plot shows that we need to normalize these arrays.

4.3 RNA degradation plots

The functions `AffyRNAdeg`, `summaryAffyRNAdeg`, and `plotAffyRNAdeg` aid in assessment of RNA quality. Individual probes in a probeset are ordered by location relative to the 5′ end of the targeted RNA molecule.Affymetrix (1999) Since RNA degradation typically starts from the 5′ end of the molecule, we would expect probe intensities to be systematically lowered at that end of a probeset when compared to the 3′ end. On each chip, probe intensities are averaged by location in probeset, with the average taken over probesets. The function `plotAffyRNAdeg` produces a side-by-side plots of these means, making it easy to notice any 5′ to 3′ trend. The function `summaryAffyRNAdeg` produces a single summary statistic for each array in the batch, offering a convenint measure of the severity of degradation and significance level. For an example

```
> deg <- AffyRNAdeg(affybatch.example)
> names(deg)
```

```
[1] "N"              "sample.names"    "means.by.number" "ses"
[5] "slope"          "pvalue"
```

does the degradation analysis and returns a list with various components. A summary can be obtained using

```
> summaryAffyRNAdeg(deg)
```

```
a1 a2 a3
slope -0.225 -0.156 -0.137
pvalue 0.0177 0.0107 0.0187
```

Finally a plot can be created using `plotAffyRNAdeg`, see Figure 11.

```
> plotAffyRNAdeg(deg)
```

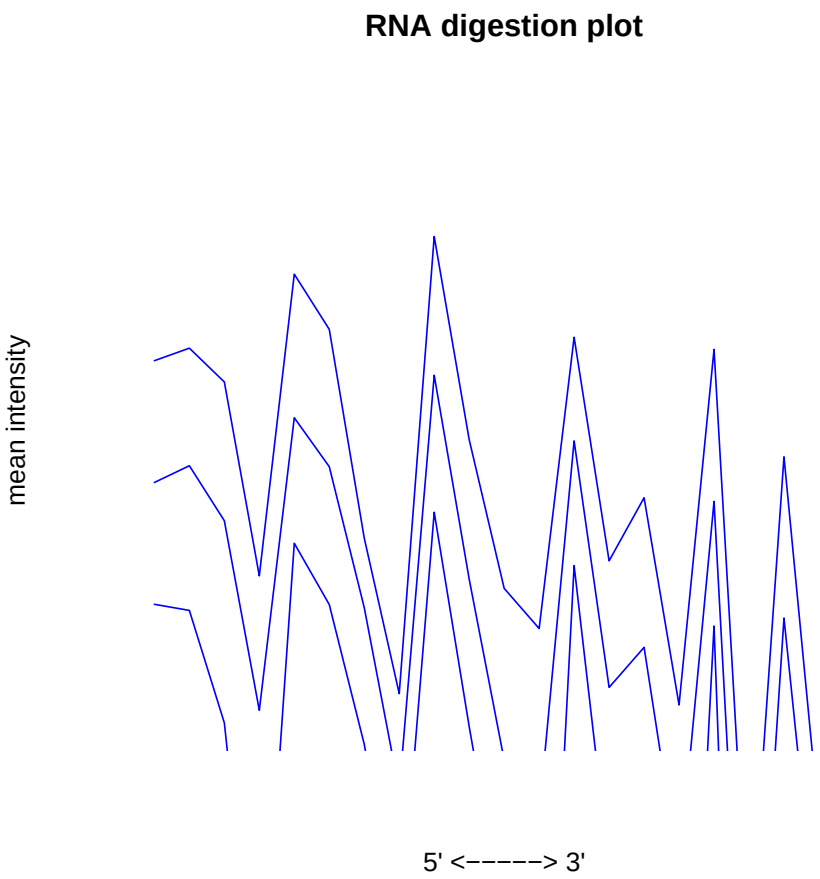

Figure 11: Side-by-side plot produced by plotAffyRNAdeg.

az

5 Normalization

Various researchers have pointed out the need for normalization of Affymetrix arrays. See for example Bolstad et al. (2002). Let's look at an example. The first two arrays in `Dilution` are technical replicates (same RNA), so the intensities obtained from these should be about the same. The second 2 are also replicates. The second arrays are hybridized to twice as much RNA so the intensities should be in general bigger. However, notice that the scanner effect is stronger than the RNA concentration effect.

```
> pData(Dilution)
```

```
    liver sn19 scanner
20A    20    0       1
20B    20    0       2
10A    10    0       1
10B    10    0       2
```

The `boxplot` method for the `AffyBatch` class, shown in Figure 10, shows this is the case.

Figure 10 shows the need for normalization. For example arrays scanned using scanner 1 are globally larger than those scanned with 2.

Another way to see that normalization is needed is by looking at log ratio versus average log intensity (MVA) plots. The method `mva.pairs` will show all MVA plots of each pairwise comparison on the top right half and the interquartile range (IQR) of the log ratios on the bottom left half. For replicates and cases where most genes are not differentially expressed, we want the cloud of points to be around 0 and the IQR to be small.

The method `normalize` lets one normalize the data.

```
> normalized.Dilution <- merge(normalize(Dilution[1:2]), normalize(Dilution[3:4]))
```

We normalize the two concentration groups separately. Notice the function `merge` permits us to put together two `AffyBatch` objects.

Various methods are available for normalization (see the help file). The default is quantile normalization (Bolstad et al., 2002)). All the available methods are obtained using this function:

```
> normalize.methods(Dilution)
```

```
[1] "constant"         "contrasts"        "invariantset"     "loess"
[5] "qspline"          "quantiles"        "quantiles.robust"
```

and can be called using the `method` argument of the normalize function.

Figures 13 and 14 show the boxplot and mva pairs plot after normalization. The normalization routine seems to correct the boxplots and mva plots.

```
> gn <- sample(geneNames(Dilution), 100)
> pms <- pm(Dilution[3:4], gn)
> mva.pairs(pms)
```

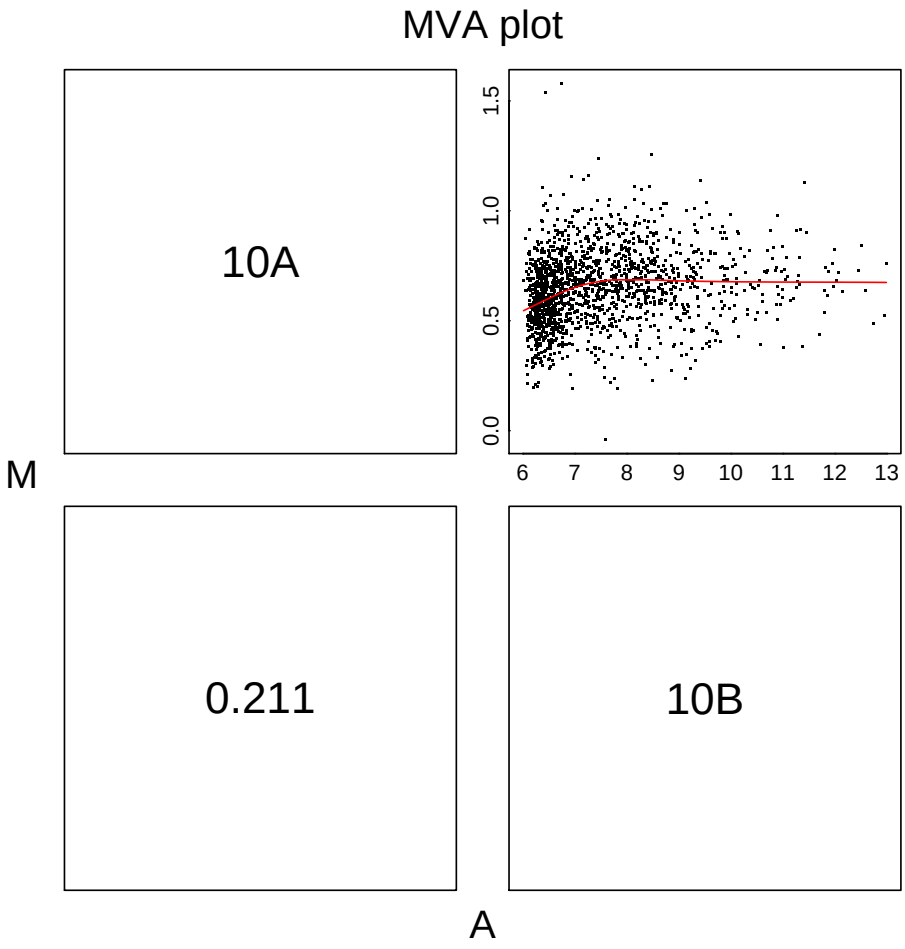

Figure 12: MVA pairs for first three arrays in dilution data

```
> boxplot(normalized.Dilution, col = c(2, 2, 3, 3))
```

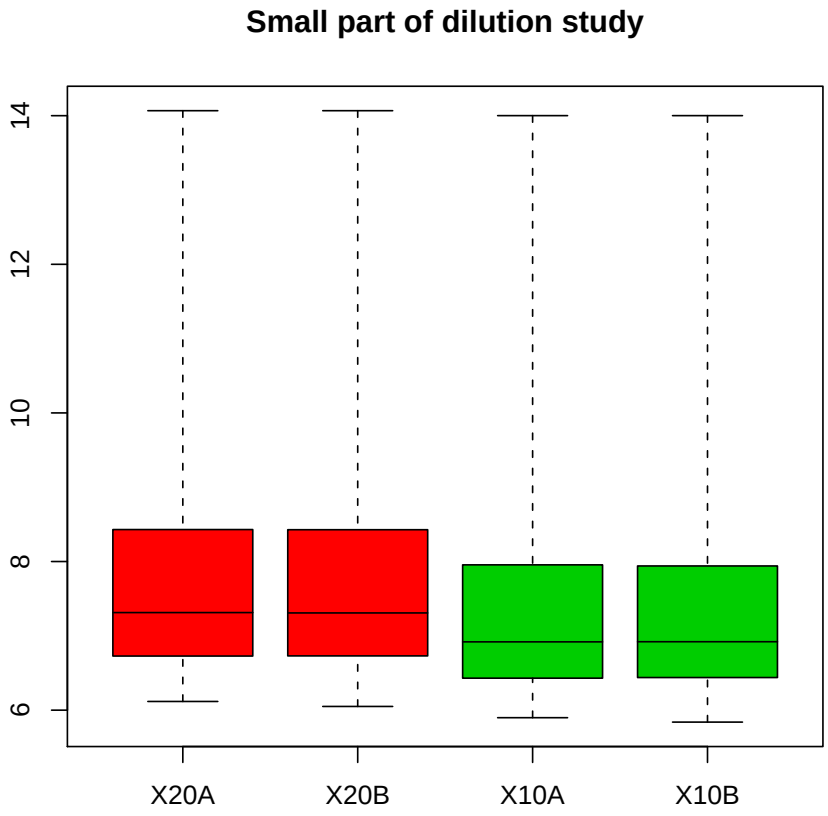

Figure 13: Boxplot of first arrays in normalized dilution data.

```
> pms <- pm(normalized.Dilution[3:4], gn)
> mva.pairs(pms)
```

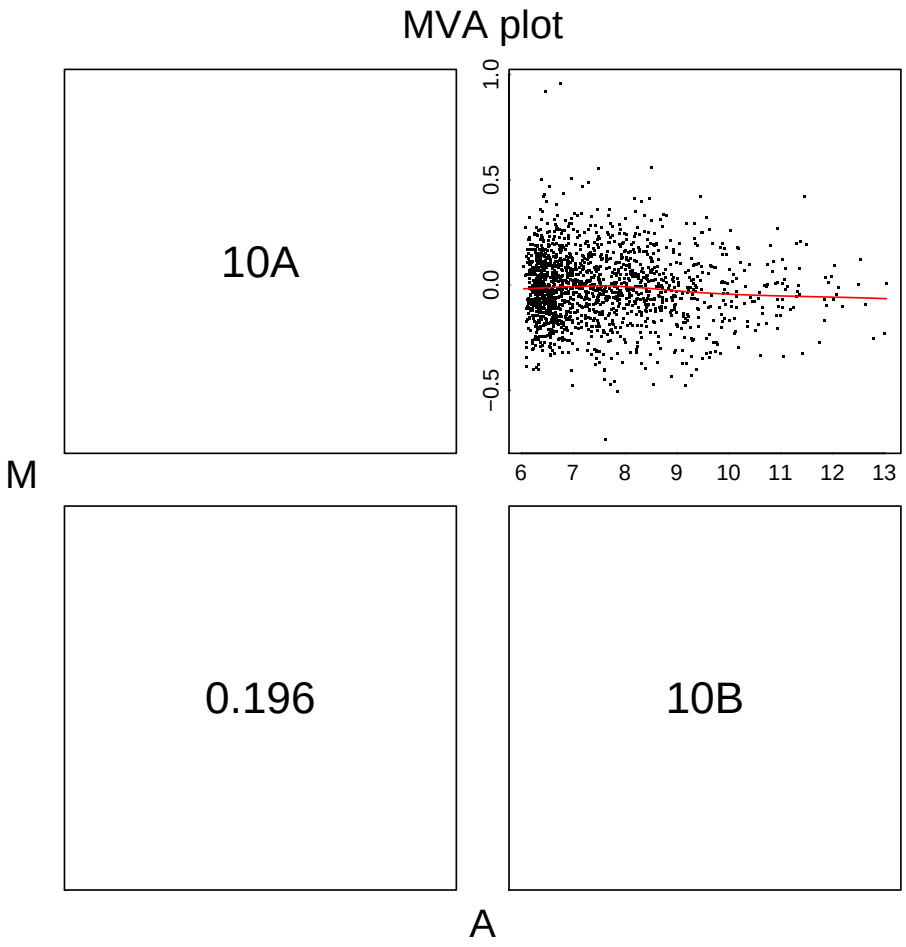

Figure 14: MVA pairs for first two replicate arrays in normalized dilution data

6 Classes

`AffyBatch` is the main class in this package. There are three other auxiliary classes that we also describe in this Section.

6.1 AffyBatch

The AffyBatch class has slots to keep all the probe level information for a batch of *Cel* files, which usually represent an experiment. It also stores phenotypic and MIAME information as does the `exprSet` class in the Biobase package (the base package for Bioconductor). In fact, `AffyBatch` extends `exprSet`.

The `exprs` slot contains the a matrix with the columns representing the intensities read from the different arrays. The rows represent the *cel* intensities for all position on the array. The cel with physical coordinates (x, y) will be in row $i = x \times$ `nrow` $y + 1$. The `ncol` and `nrow` slots contain the physical rows of the array. Notice that this is different from the dimensions of the `exprs` matrix. The number of row of the `exprs` matrix is equal to `ncol`$\times$`nrow`. For consistency with the previous version the accessor method `intensity` exists for obtaining the `exprs` slot.

The `cdfName` slot contains the necessary information for the package to find the locations of the probes for each probe set. See Section 7 for more on this.

6.2 ProbeSet

The `ProbeSet` class holds the information of all the probes related to an *affyID*. The components are `pm` and `mm`.

The method `probeset` extracts probe sets from `AffyBatch` objects. It takes as arguments an `AffyBatch` object and a vector of *affyIDs* and returns a list of objects of class `ProbeSet`

```
> gn <- geneNames(Dilution)
> ps <- probeset(Dilution, gn[1:2])
> show(ps[[1]])

ProbeSet object:
  id=1000_at
  pm= 16 probes
```

The `pm` and `mm` methods can be used to extract these matrices (see below).

This function is general in the way it defines a probe set. The default is to use the definition of a probe set given by Affymetrix in the CDF file. However, the user can define arbitrary probe sets. The argument `locations` lets the user decide the row numbers in the `intensity` that define a probe set. For example, we are interested in redefining the `1000_at` and `1001_at` probe sets, we could do the following:

First, define the locations of the PM and MM on the array of the `1000_at` and `1001_at` probe sets

```
> mylocation <- list("1000_at" = cbind(pm = c(1, 2, 3), mm = c(4,
+     5, 6)), "1001_at" = cbind(pm = c(4, 5, 6), mm = c(1, 2, 3)))
```

The first column of the matrix defines the location of the PMs and the second column the MMs.

Now we are ready to extract the `ProbSet`s using the `probeset` function:

```
> ps <- probeset(Dilution, genenames = c("1000_at", "1001_at"),
+     locations = mylocation)
```

Now, `ps` is list of `ProbeSet`s. We can see the PMs and MMs of each component using the `pm` and `mm` accessor methods.

```
> pm(ps[[1]])

        20A   20B    10A   10B
[1,]  149.0 112.0  129.0  60.0
[2,] 1153.5 575.3 1262.3 564.8
[3,]  142.0  98.0  128.0  56.0

> mm(ps[[1]])

      20A 20B  10A 10B
[1,] 1051 597 1269 570
[2,]   91  77   90  46
[3,]  136 133  117  62

> pm(ps[[2]])

      20A 20B  10A 10B
[1,] 1051 597 1269 570
[2,]   91  77   90  46
[3,]  136 133  117  62

> mm(ps[[2]])

        20A   20B    10A   10B
[1,]  149.0 112.0  129.0  60.0
[2,] 1153.5 575.3 1262.3 564.8
[3,]  142.0  98.0  128.0  56.0
```

This can be useful in situations where the user wants to determine if leaving out certain probes improves performance at the expression level. It can also be useful to combine probes from different human chips, for example by considering only probes common to both arrays.

Users can also define their own environment for probe set location mapping. More on this in Section 7.

An example of a `ProbeSet` is included in the package. A spike in data set is included in the package in the form of a list of `ProbeSet`s. The help file describes the data set. Figure 15 uses this data set to demonstrate that the MM also detect transcript signal.

6.3 Cel

Our package defines a class for storing what we consider to be relevant information in a CEL file. The mean intensities (and if desired the SD) are stored in matrices. The `i,j` entry in these matrices contain the probe intensity and SD in position `i,j` on the array. These objects also contain information on the position of masked and outlier probes, the name of the array, and relevant information added by the user. Since it is not clear how the pixel level SD information is useful, the default behavior was set not to load it.

The slots are:

intensity Object of class "matrix" containing intensity values in a matrix of dimension (nrow,ncol). The position in the matrix represents the physical position on the array

sd Object of class "matrix" containing the standard deviation for the intensity values obtained from CEL files.

name Object of class "character" the name given to this CEL file data. The function `read.celfile` uses the file name by default.

cdfName Name of corresponding *CDF* file.

masks Object of class "matrix" containing the coordinates of masked measurements.

outliers Object of class "matrix" containing the coordinates of outlier measurements.

history Object of class "list" containing details about where the file.

The function `read.celfile()` is used to read *CEL* file information into R. An object of class *Cel* is returned. In this example we read in a *CEL* file to create a *Cel* object:

```
R> cel <- read.celfile("filename.cel")
```

The following useful methods are available for this class:

image Display an *image* of the data in the `Cel` object. Among the other parameters, `transfo=log` can be convenient.

```
> data(SpikeIn)
> pms <- pm(SpikeIn)
> mms <- mm(SpikeIn)
> par(mfrow = c(1, 2))
> concentrations <- matrix(as.numeric(sampleNames(SpikeIn)), 20,
+     12, byrow = TRUE)
> matplot(concentrations, pms, log = "xy", main = "PM", ylim = c(30,
+     20000))
> lines(concentrations[1, ], apply(pms, 2, mean), lwd = 3)
> matplot(concentrations, mms, log = "xy", main = "MM", ylim = c(30,
+     20000))
> lines(concentrations[1, ], apply(mms, 2, mean), lwd = 3)
```

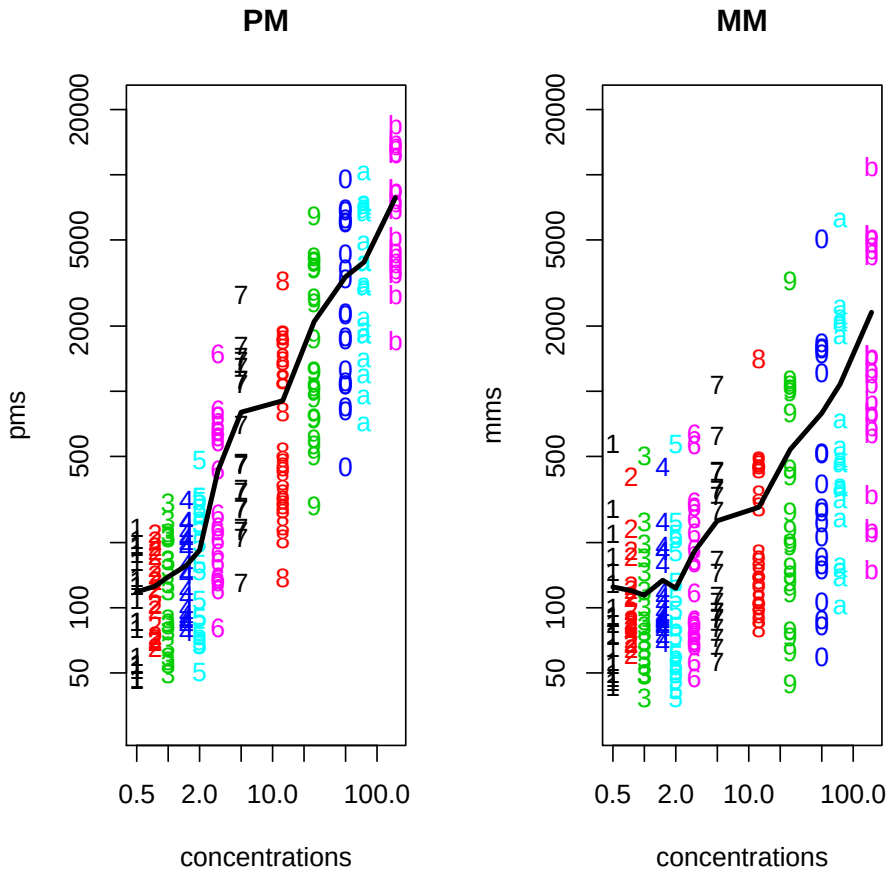

Figure 15: PM and MM intensities plotted against SpikeIn concentration

show Outputs few general facts about the object on the console.

Notice that the `AffyBatch` contains enough information to create a `Cel` object for each array. We can extract this using

```
> mycelfile <- affybatch.example[[1]]
```

Now `mycelfile` is of class `Cel`.

By default the method `image()` creates an image of the intensities. Figure 16 shows some results from the the `image` method.

```
> image(mycelfile, sub = "raw values")
```

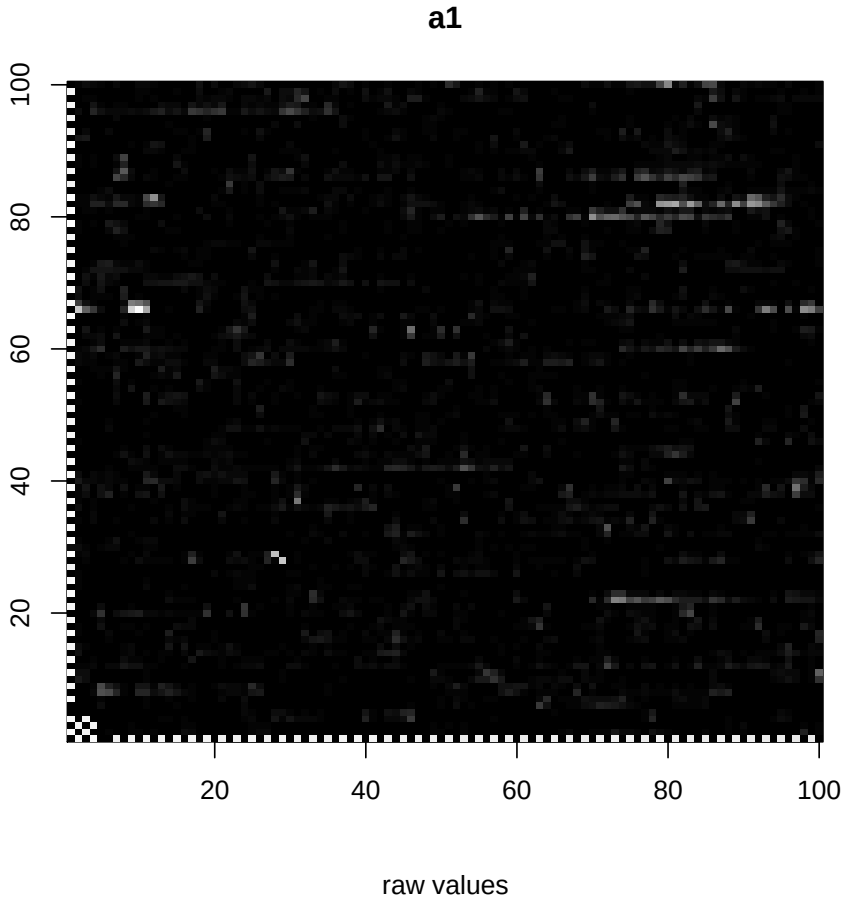

Figure 16: Images of PM intensities from a section of a cel file using different colors and concentrations.

6.4 Cdf

The *Chip Definition Files* (*CDF* files) are used to store information related to Affymetrix's GeneChip arrays (a type of high density oligonucleotide expression array sold by the manufacturer). All the arrays belonging to a given type will share this same information. As the quantity of information in a *CDF* can be rather large, this is an important point. This is kept in the design of the package, as there will be only one `Cdf` object in memory, to which will refer the corresponding `Cel` files.

For instance, if during an experiment a total of 30 arrays of type *Hu6800* are used, the information relative to the array type is common to all the chips. Knowing that *CDF* files are usually of size 20 Megabytes, illustrates the reason for avoiding having multiple instances of this information.

The `Cdf` class is designed to store the information in a *CDF* file. The slots are:

name Each *probe* (or *feature*) on an array is an oligonucleotide from a larger sequence of nucleic acids (generally a gene or a fraction of a gene, we refer to as a probe pairs set). All these *large sequences* were given a unique name by the chip manufacturer. We call this name *affyID*. By design of the arrays several probes correspond to different parts of the same larger sequence, hence have the same name. Names are stored as factors, and the corresponding factors labels (or levels) are found in the attribute `name.levels`.

name.level See previous item.

pbase In the *CDF* files, the column called *PBASE* holds one of the nucleic acid letters. From trials and errors [2] , the *p* was guessed to stand for *probe*.

pbase.levels The four levels for `pbase` are the four letters used to designate nucleic acids.

tbase In the *CDF* files, the column called *TBASE* holds also a nucleic acid letter. From trials and errors where the *t* was assumed to stand for *tbase*.

tbase.levels The four same letters than for `pbase.levels` are found here.

atoms Each *probe pair* in a *probe pair set* is given a unique integer as an identifier. This number is refereed as the *atom* number. The corresponding *perfect matches* and *mismatches* are found by using this number.

The function `read.cdffile` can be used to create an object of class `Cdf` from a file:

```
R> cdf <- read.cdffile("filename.CDF")
```

This function has various useful arguments. For example, the `compress` argument permits you to specify if the file you are reading is compressed. See the help file for more details.

[2] Comparing the letter between *PBASE* and *TBASE*, it appeared that two cases could appear.

7 Location ProbeSet Mapping

On Affymetrix GeneChip arrays, several probes are used to represent genes in the form of probe sets. From a *CEL* file we get for each physical location, or cel, (defined by x and y coordinates) an intensity. The *CEL* file also contains the name of the *CDF* file needed for the location-probe-set mapping. The *CDF* files store the probe set related to each location on the array. We store this mapping information in *R* environments. For each *CDF* file there is package, available from `http://www.bioconductor.org/data/cdfenvs/cdfenvs.html`, that contains exactly one of these environments. In our packages we store the x and y coordinates as one number ($x\times$ `nrow` $+y+1$).

In instances of *AffyBatch*, the *cdfName* slot gives the name of the appropriate *CDF* file for arrays represented in the `intensity` slot. The functions `read.celfile`, `read.affybatch`, and `ReadAffy` extract the *CDF* filename from the *CEL* files being read. Each *CDF* file corresponds to exactly one environment. The function `cleancdfname` converts the Affymetrix given *CDF* name to a Bioconductor environment and annotation name. Here are two examples:

These give environment names:

```
> cat("HG_U95Av2 is", cleancdfname("HG_U95Av2"), "\n")

HG_U95Av2 is hgu95av2cdf

> cat("HG-133A is", cleancdfname("HG-133A"), "\n")

HG-133A is hg133acdf
```

This gives annotation name:

```
> cat("HG_U95Av2 is", cleancdfname("HG_U95Av2", addcdf = FALSE),
+     "\n")

HG_U95Av2 is hgu95av2
```

The HGU95Av2 and HGU133A environments are available with the package. In the following, we load the environment, look at the names for the first 5 objects defined in the environment, and finally look at the first object in the environment:

```
> data(hgu95av2cdf)
> ls(hgu95av2cdf)[1:5]

[1] "1000_at"   "1001_at"   "1002_f_at" "1003_s_at" "1004_at"

> get(ls(hgu95av2cdf)[1], hgu95av2cdf)
```

```
          pm     mm
 [1,] 358160 358800
 [2,] 118945 119585
 [3,] 323731 324371
 [4,] 223978 224618
 [5,] 313420 314060
 [6,] 349209 349849
 [7,] 199525 200165
 [8,] 213669 214309
 [9,] 236739 237379
[10,] 298099 298739
[11,] 282744 283384
[12,] 281443 282083
[13,] 349198 349838
[14,] 297953 298593
[15,] 317054 317694
[16,] 404069 404709
```

The package needs to know what locations correspond to which probe sets. The `cdfName` slot contains the necessary information to find the environment with this location information. The method `getCdfInfo` takes as an argument an `AffyBatch` and returns the necessary environment. If `x` is an `AffyBatch`, this function will look for an environment with name `cleancdfname(x@cdfName)`. Here are two examples.

The call to `data` loads an `AffyBatch` and also an environment that corresponds to a small corner of the HG6800 chip, which we created.

```
> data(affybatch.example)
> print(affybatch.example@cdfName)

[1] "corner.Hu6800.example"

> myenv <- getCdfInfo(affybatch.example)
> ls(myenv)[1:5]

[1] "A28102_at"   "AB000114_at" "AB000115_at" "AB000220_at" "AB002314_at"
```

Now lets look at `Dilution`

```
> Dilution@cdfName

[1] "HG_U95Av2"

> myenv <- getCdfInfo(Dilution)
> ls(myenv)[1:5]
```

```
[1] "1000_at"   "1001_at"   "1002_f_at" "1003_s_at" "1004_at"
```

By default we search for the environment first in the global environment, then in a packages named `cleancdfname(x@cdfName)`, and finally in the `data` directory of the `affy` package. This order can be changed through the options (see Section 8).

Various methods exist to obtain locations of probes as demonstrated in the following examples:

```
> Index <- pmindex(Dilution)
> names(Index)[1:2]

[1] "1000_at" "1001_at"

> Index[1:2]

$"1000_at"
 [1] 358160 118945 323731 223978 313420 349209 199525 213669 236739 298099
[11] 282744 281443 349198 297953 317054 404069

$"1001_at"
 [1] 340142 236569 327449 203508 300798 276193 354374 400320 250783 379851
[11] 365637 144611 120239 189384 182903 299352
```

`pmindex` returns a list with probe set names as names and locations in the components. We can also get specific probe sets:

```
> pmindex(Dilution, genenames = c("1000_at", "1001_at"))

$"1000_at"
 [1] 358160 118945 323731 223978 313420 349209 199525 213669 236739 298099
[11] 282744 281443 349198 297953 317054 404069

$"1001_at"
 [1] 340142 236569 327449 203508 300798 276193 354374 400320 250783 379851
[11] 365637 144611 120239 189384 182903 299352

> pmindex(Dilution, genenames = c("1000_at"), xy = TRUE)

$"1000_at"
        x   y
 [1,] 400 560
 [2,] 545 186
 [3,] 531 506
 [4,] 618 350
```

```
 [5,] 460 490
 [6,] 409 546
 [7,] 485 312
 [8,] 549 334
 [9,] 579 370
[10,] 499 466
[11,] 504 442
[12,] 483 440
[13,] 398 546
[14,] 353 466
[15,] 254 496
[16,] 229 632
```

The `xy` argument permits us to figure out the original x and y coordinates of the probes. The locations are ordered from 5' to 3' on the target transcript. The function `mmindex` performs in a similar way:

```
> mmindex(Dilution, genenames = c("1000_at", "1001_at"))

$"1000_at"
 [1] 358800 119585 324371 224618 314060 349849 200165 214309 237379 298739
[11] 283384 282083 349838 298593 317694 404709

$"1001_at"
 [1] 340782 237209 328089 204148 301438 276833 355014 400960 251423 380491
[11] 366277 145251 120879 190024 183543 299992

> mmindex(Dilution, genenames = c("1000_at"), xy = TRUE)

$"1000_at"
        x   y
 [1,] 400 561
 [2,] 545 187
 [3,] 531 507
 [4,] 618 351
 [5,] 460 491
 [6,] 409 547
 [7,] 485 313
 [8,] 549 335
 [9,] 579 371
[10,] 499 467
[11,] 504 443
[12,] 483 441
```

```
[13,] 398 547
[14,] 353 467
[15,] 254 497
[16,] 229 633
```

They both use the method `indexProbes`

```
> indexProbes(Dilution, which = "pm")[1]

$"1000_at"
 [1] 358160 118945 323731 223978 313420 349209 199525 213669 236739 298099
[11] 282744 281443 349198 297953 317054 404069

> indexProbes(Dilution, which = "mm")[1]

$"1000_at"
 [1] 358800 119585 324371 224618 314060 349849 200165 214309 237379 298739
[11] 283384 282083 349838 298593 317694 404709

> indexProbes(Dilution, which = "both")[1]

$"1000_at"
 [1] 358160 118945 323731 223978 313420 349209 199525 213669 236739 298099
[11] 282744 281443 349198 297953 317054 404069 358800 119585 324371 224618
[21] 314060 349849 200165 214309 237379 298739 283384 282083 349838 298593
[31] 317694 404709
```

The `which="both"` options returns the location of the PMs followed by the MMs.

8 Configuring the package options

Package-wide options can be configured. We will show how to configure the options with a first example: the compression of files. If you are always compressing your CEL files, you might find annoying to specify it each you read them. I can be specified once for all in the options.

```
> opt <- getOption("BioC")
> affy.opt <- opt$affy
> names(affy.opt)

[1] "compress.cdf" "compress.cel" "use.widgets"  "probesloc"

> affy.opt$compress.cel <- TRUE
```

The second example explains how the finding locations of probes environment works.

```
> opt <- getOption("BioC")
> affy.opt <- opt$affy
> print(affy.opt$probesloc)

[[1]]
[[1]]$what
[1] "environment"

[[1]]$where
<environment: R_GlobalEnv>


[[2]]
[[2]]$what
[1] "package"

[[2]]$where
NULL

[[2]]$probesloc.autoload
[1] TRUE


[[3]]
[[3]]$what
[1] "data"

[[3]]$where
[1] "affy"
```

The option *probesloc* is a list. Each element of the list is it self a list with two elements *what* and *where*. When looking for the information related to the locations of the probes on the array, the elements in the list will be looked at sequentially. The first one leading to the information is used (an error message is returned if none permits to find the information). The element *what* can be one of *package*, *environment*.

References

Affymetrix. *Affymetrix Microarray Suite User Guide*. Affymetrix, Santa Clara, CA, version 4 edition, 1999.

Affymetrix. *Affymetrix Microarray Suite User Guide*. Affymetrix, Santa Clara, CA, version 5 edition, 2001.

B.M. Bolstad, R.A. Irizarry, M. Åstrand, and T.P. Speed. A comparison of normalization methods for high density oligonucleotide array data based on variance and bias. Unpublished Manuscript, 2002.

Rafael A. Irizarry, Laurent Gautier, and Leslie M. Cope. *The Analysis of Gene Expression Data: Nethods and Software*, chapter 4. Spriger Verlag, 2003a.

Rafael A. Irizarry, Bridget Hobbs, Francois Collin, Yasmin D. Beazer-Barclay, Kristen J. Antonellis, Uwe Scherf, and Terence P. Speed. Exploration, normalization, and summaries of high density oligonucleotide array probe level data. *Biostatistics*, 2003b. To appear.

C. Li and W.H. Wong. Model-based analysis of oligonucleotide arrays: Expression index computation and outlier detection. *Proceedings of the National Academy of Science U S A*, 98:31–36, 2001.