GS01 0163
Analysis of Microarray Data

Keith Baggerly and Kevin Coombes
Section of Bioinformatics
Department of Biostatistics and Applied Mathematics
UT M. D. Anderson Cancer Center
kabagg@mdanderson.org
kcoombes@mdanderson.org

15 September 2005

Lecture 5: R, Objects and Affymetrix Arrays

- R Data Structures
- Bioconductor Packages
- Microarray Data Structures
- Affymetrix Data in BioConductor
- Processing Affymetrix data
- Quantification = summarization
- More about reading Affymetrix data

R Data Structures

We've seen that R has a bunch of useful functions, and we can see how these would have helped S (and then R) to catch on. But there was more; we remarked on how S allowed one to think about data in a more coherent fashion. Let's think about that a bit more.

Consider x. Initially, this symbol has no meaning; we must assign something to it.

```
x <- 2
```

In the process of assignment, we have created an object with the name of x. This object has the value 2, but there are other things about objects: they have properties, or attributes.

Some Basic Attributes

```
> mode(x)
‘‘numeric’’
> storage.mode(x)
‘‘double’’
> length(x)
[1] 1
```

These are attributes that x has by default, but we can give it others.

What’s In A Name?

Initially, nothing:

```
> names(x)
NULL
```

but we can change this

```
> names(x) <- c(‘‘A’’)
> x
A
2
```

This element of x now has a name! If something has a name, we can call it by name.

Calling Names

```
> x[1]
A
2
> x[''A'']
A
2
```

Admittedly, this isn't that exciting here, but it can get more interesting if things get bigger and the names are chosen in a more enlightening fashion.

Let's assign a matrix of values to x, and see if we can make the points clearer.

So, how do we assign a matrix? Well, there may be a function called matrix. Let's find out.

Matrix X: TMTOWTDI!

There's more than one way to do it...

```
> help(matrix)
```

```
Usage
matrix(data = NA, nrow = 1, ncol = 1,
    byrow = FALSE, dimnames = NULL)
```

even the arguments to the function have names!

Arguments to a function can be supplied by position, or by name.

Matrix X: TMTOWTDI!

We're going to assign the numbers 1 through 12. That means we need to get these numbers. Some ways to do that:

```
> 1:12
> c(1:12)
> c(1:6, c(7:12))
> 1:12.5
> seq(from=1, to=12, by=1)
> seq(1, 12, 1)
> seq(1, 12)
> seq(by=1, to=12, from=1)
```

Matrix X: TMTOWTDI!

```
> x <- matrix(1:12,3,4)
> x <- matrix(data = 1:12, nrow = 3, ncol = 4)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

The numbers in brackets suggest how things should be referred to now:

```
> x[2,3]
[1] 8
> x[2,]
[1] 2  5  8 11
```

Matrix X: TMTOWTDI!

```
> x[,3]
[1] 7 8 9
> x[3,1:2]
[1] 3 6
> x[3,c(1,4)]
[1]  3 12
> x[2, x[2,] > 6]
[1]  8 11
```

But what about names?

Naming x

```
> rownames(x)
NULL
```

Naming x

```
> rownames(x)
NULL


> rownames(x) <- c("Gene1","Gene2","Gene3")
> x
      [,1] [,2] [,3] [,4]
Gene1    1    4    7   10
Gene2    2    5    8   11
Gene3    3    6    9   12
```

Naming x

```
> colnames(x) <- c("N01","N02","T01","T02")
> x
      N01 N02 T01 T02
Gene1   1   4   7  10
Gene2   2   5   8  11
Gene3   3   6   9  12
```

One more thing – names can be inherited!

Naming x

```
> colnames(x) <- c("N01","N02","T01","T02")
> x
      N01 N02 T01 T02
Gene1   1   4   7  10
Gene2   2   5   8  11
Gene3   3   6   9  12
```

One more thing – names can be inherited!

```
> x[‘‘Gene2’’,]
      N01 N02 T01 T02
Gene2   2   5   8  11
```

On Beyond Matrices

Ok, we’ve gone from scalar to vector to matrix, attaching names as we go, with the goal of keeping associated information together. So far, we’ve done this with numbers, but we could use character strings instead:

```
> letters[1:3]
‘‘a’’ ‘‘b’’ ‘‘c’’
> x <- letters[1];
> x <- letters[1:3];
> x <- matrix(letters[1:12],3,4);
```

but we can’t easily mix data of different modes

```
> x <- c(1,’’a’’);
> x
‘‘1’’ ‘‘a’’
```

Mixing Modes in Lists

In R, a list can have components that are of different modes and even different sizes:

```
x <- list(teacher=''Keith'',n_students=14,
          grades=letters[c(1:4,6)])
x
$teacher
[1] ''Keith''
$n_students
[1] 14
$grades
[1] ''a'' ''b'' ''c'' ''d'' ''f''
```

Note that we named the components of the list at the same time that we created it. Many functions in R return answers as lists.

Extracting Items From Lists

If we want to access the first element of x, we might try using the index or the name in single brackets:

```
> x[1]
$teacher
[1] ‘‘Keith’’
> x[‘‘teacher’’]
$teacher
[1] ‘‘Keith’’
```

These don’t quite work. The single bracket extracts a component, but keeps the same mode; what we have here is a list of length 1 as opposed to a character string. Two brackets, on the other hand...

Extracting Items From Lists

```
> x[[1]]
[1] ‘‘Keith’’
> x[[‘‘teacher’’]]
[1] ‘‘Keith’’
```

The double bracket notation can be rather cumbersome, so there is a shorthand notation involving the dollar sign:

```
> x$teacher
[1] ‘‘Keith’’
```

This method has the advantage that using names keeps the goals clear.

Lists with Structure

Now, there are some very common types of structured arrays. The most common is simply a table, where the rows correspond to individuals and the columns correspond to various types of information (potentially of multiple modes). Because we want to allow for multiple modes, we can construct a table as a list, but this list has a constraint imposed on it – all of its components must be of the same length. This is similar in structure to the idea of a matrix that allows for multiple modes. This structure is built into R as a `data frame`.

This structure is important for data import.

Reading Data Into R

While we can simply type stuff in, or use `source()` to pull in small amounts of data we've typed into a file, what we really want to do is to read a big table of data. R has several functions that allow us to do this, including `read.table()`, `read.delim()`, and `scan()`.

We can experiment by using some of the files that we generated in dChip for the first HWK.

We could load the sample info file, and the list of filtered genes. Then we could use the sample info values to suggest how to contrast the expression values in the filtered gene table. Let's try this.

Importing our dChip Data

I exported all of the dChip quantifications to a single file. The file has a header row, with columns labeled “probe set”, “gene”, “Accession”, “LocusLink”, “Description” and then “N01” and so on, 1 column per sample. We can read this into R as follows:

```
> singh_dchip_data <-
    read.delim(c("../SinghProstate/Singh_’’,
                ‘‘Prostate_dchip_expression.xls"));
> class(singh_dchip_data)
[1] "data.frame"
> dim(singh_dchip_data)
[1] 12625  108
```

The number of columns is a bit odd...

More on Importing

If we invoke `help(read.delim)`, help pops up for `read.table`. The former is a special case of the latter. Let's take a look at bits of the usage lines for each:

```
read.table(file, header = FALSE, sep = "", quote = "\
           row.names, col.names, as.is = FALSE, na.st
           colClasses = NA, nrows = -1,
           skip = 0, check.names = TRUE, fill = !blar
           strip.white = FALSE, blank.lines.skip = TF
           comment.char = "#")

read.delim(file, header = TRUE, sep = "\t", quote=
           "\"", dec=".", fill = TRUE, ...)
```

Note the default function arguments!

Speeding Up Import

Reading the documentation suggests a few speedups:

- we can use comment.char = "", speeding things up
- we can use nrows = 12626, for better memory usage
- we could shift to using scan() (use help!).

```
singh_dchip_data <-
    read.delim(c("../SinghProstate/Singh_Prostate’’
                 ,’’_dchip_expression.xls")
               , comment.char = ""
               , nrows = 12626
    );
```

is indeed faster!

Is This What We Want?

All of the expression data is now nicely loaded in a data frame. But this data frame really breaks into two parts quite nicely – gene information, and expression values. If we split these apart, then the expression value matrix has 102 columns, corresponding to the sample info entries quite nicely.

```
singh_annotation <- singh_dchip_data[,1:5];
singh_dchip_expression <-
    as.matrix(singh_dchip_data[,6:107]);
rownames(singh_dchip_expression) <-
    singh_annotation\$probe.set;
```

Grab the Sample Info Too

What are the columns in my sample info file?

```
scan name        sample name     type
    run_date_block  cluster_block
N01__normal      N01     N       2       2
```

(the last two you might not have).

```
singh_sample_info <-
    read.delim("../SinghProstate/sample_info_2.txt"
               , comment.char = ""
               , nrows = 103
    );
```

Test Something Interesting

In the first homework, we saw that the data split into two clusters that didn't agree well with the tumor/normal split. It might very well be that there was some type of batch effect in addition to the biological split of interest.

Can we factor the batch effect out? If we know what the batch split is, we can first fit a model using just the batches, subtract the fit off, and then fit a model using the tumor/normal split on what remains.

Tumor vs Normal

```
singh_probeset_lm <-
    lm(singh_dchip_expression[
            singh_annotation$probe.set
            == "31539_r_at",]
        ~ singh_sample_info$type
    );
singh_probeset_anova <-
    anova(singh_probeset_lm);
```

Tumor vs Normal (cont)

```
> singh_probeset_anova
Analysis of Variance Table

Response: singh_dchip_expression[
    singh_annotation$probe.set == "31539_r_at",]

            Df  Sum Sq Mean Sq F value  Pr(>F)
$type        1   71.42   71.42  5.3748 0.02247 *
Residuals  100 1328.81   13.29
---
Signif. codes:  0 ’***’ 0.001 ’**’ 0.01 ’*’ 0.05 ’.’
```

T vs N, After Blocking

```
singh_probeset_lm_full <-
    lm(singh_dchip_expression[
            singh_annotation$probe.set
            == "31539_r_at",]
        ~ singh_sample_info$cluster.block
        + singh_sample_info$type
    );
singh_probeset_anova_full <-
    anova(singh_probeset_lm_full);
```

T vs N, After Blocking (cont)

```
> singh_probeset_anova_full
Analysis of Variance Table

Response: singh_dchip_expression[
    singh_annotation$probe.set == "31539_r_at",]

           Df Sum Sq Mean Sq F value    Pr(>F)
$block      1 404.97  404.97 40.6399 5.85e-09 ***
$type       1   8.75    8.75  0.8779   0.3511
Residuals  99 986.51    9.96
---
Signif. codes:  0 ’***’ 0.001 ’**’ 0.01 ’*’ 0.05 ’.’
```

Hasn't Someone Done This?

Other people have thought about the data structures that might be natural for microarray data. In particular, a lot of these functions are collected at Bioconductor.

Let's try to grab some of the packages and functions that will help with this type of analysis.

Bioconductor Packages

You will need the following packages from the Bioconductor web site. Use the menu item "Packages" $->$ "Install package(s) from BioConductor..." to get them.

reposTools : Repository tools for R

Biobase : Base functions for BioConductor

affy : Methods for Affymetrix oligonucleotide arrays

affydata : Affymetrix data for demonstration purposes

affypdnn : Probe dependent nearest neighbor (PDNN) for the affy package

Bioconductor Widget Packages

In order to use some of the graphical tools that make it easier to read Affymetrix microarray data and construct sensible objects describing the experiments, you will also need the following packages from the Bioconductor web site.

tkWidgets : R based Tk widgets

widgetTools : Creates an interactive tcltk widget

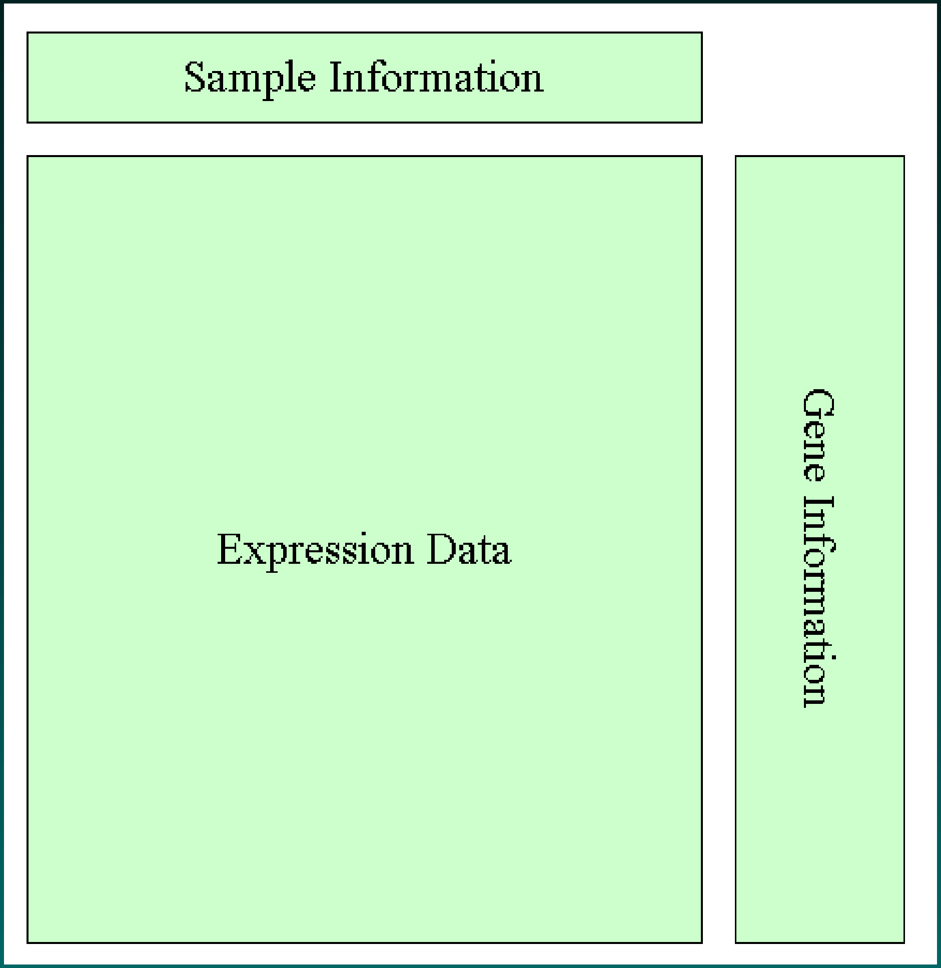
DynDoc : Dynamic document tools

Microarray Data Structures

Recap: What information do we need in order to analyze a collection of microarray experiments?

Microarray Data Structures

Recap: What information do we need in order to analyze a collection of microarray experiments?



Phenotypes

The `Biobase` package in BioConductor views the sample information as an extension of the notion of a data frame, which they call a `phenoData` object. In their conception, this object contains the "phenotype" information about the samples used in the experiment. The extra information in a `phenoData` object consist of optional "long" labels that can be used to identify the covariates (or factors) in the columns.

Mock data

Let’s create a fake data set. We pretend we have measured 200 genes in 8 experimental samples, the first four of which are healthy and the last four are cancer patients.

```
> fake.data <- matrix(rnorm(8*200), ncol=8)
> sample.info <- data.frame(
+    spl=paste('A', 1:8, sep=''),
+    stat=rep(c('healthy', 'cancer'), each=4)
```

At this point, we have a matrix containing fake expression data and a data fame containing two columns (“spl” and “stat”). Let’s create a `phenoData` object with more intelligible labels:

```
> pheno <- new("phenoData", pData=sample.info,
+  varLabels=list('Sample Name', 'Cancer Status'))
```

```
> pheno
    phenoData object with 2 variables and 8 cases
    varLabels
        : Sample Name
        : Cancer Status
> pData(pheno)
  spl    stat
1  A1  cancer
2  A2  cancer
3  A3  cancer
4  A4  cancer
5  A5 healthy
6  A6 healthy
7  A7 healthy
8  A8 healthy
```

ExprSets

The object in BioConductor that links together a collection of expression data and its associated sample information is called an `exprSet`.

```
> my.experiments <- new("exprSet",
+    exprs=fake.data, phenoData=pheno)
> my.experiments
Expression Set (exprSet) with
  200 genes
  8 samples
     phenoData object with 2 variables and 8 cases
     varLabels
        : Sample Name
        : Cancer Status
```

Warning

If you create a real `exprSet` this way, you should ensure that the columns of the data matrix are in exactly the same order as the rows of the sample information data frame; the software has no way of verifying this property without your help.

You'll also need to put together something that describes the genes used on the microarrays.

Where is the gene information?

The `exprSet` object we have created so far lacks an essential piece of information: there is nothing to describe the genes. One flaw in the design of BioConductor is that it allows you to completely separate the biological information about the genes from the expression data. (This blithe acceptance of the separation is surprisingly common among analysts.)

Each `exprSet` includes a slot called `annotation` , which is a character string containing the name of the environment that holds the gene annotations.

We'll return to this topic later to discuss how to create these annotation environments.

Optional parts of an `exprSet`

In addition to the expression data (`exprs`) and the sample information (`phenoData`), each `exprSet` includes several optional pieces of information:

annotation name of the gene annotation enviroment

se.exprs matrix containing standard errors of the expression estimates

notes character string describing the experiment

description object of class MIAME describing the experiment

Affymetrix Data in BioConductor

For working with Affymetrix data, BioConductor includes a specialized kind of `exprSet` called an `AffyBatch`. To create an `AffyBatch` object from the CEL files in the current directory, do the following:

```
> library(affy)  # load the affy library
> my.data <- ReadAffy() # read CEL data
```

You may have to start by telling R to use a different working directory to find the CEL files; the command to do this is `setwd`.

```
> setwd("/my/celfiles") # point to the CEL files
```

Paths in R are separated by forward slashes (/) not backslashes (\); this is a common source of confusion.

Demonstration data

Note: If you are trying to follow along and have not yet obtained some CEL files, the `affydata` package includes demonstration data fom a dilution experiment. You can load it by typing

```
> library(affydata)
> data(Dilution)
```

These commands will create an AffyBatch object called `Dilution` that you can explore.

Peeking at what’s inside

BioConductor will automatically build an object with the correct gene annotations for the kind of array you are using the first time you access the data; this may take a while, since it downloads all the information from the internet. So, don’t be surprised if it takes a few minutes to display the response to the command

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

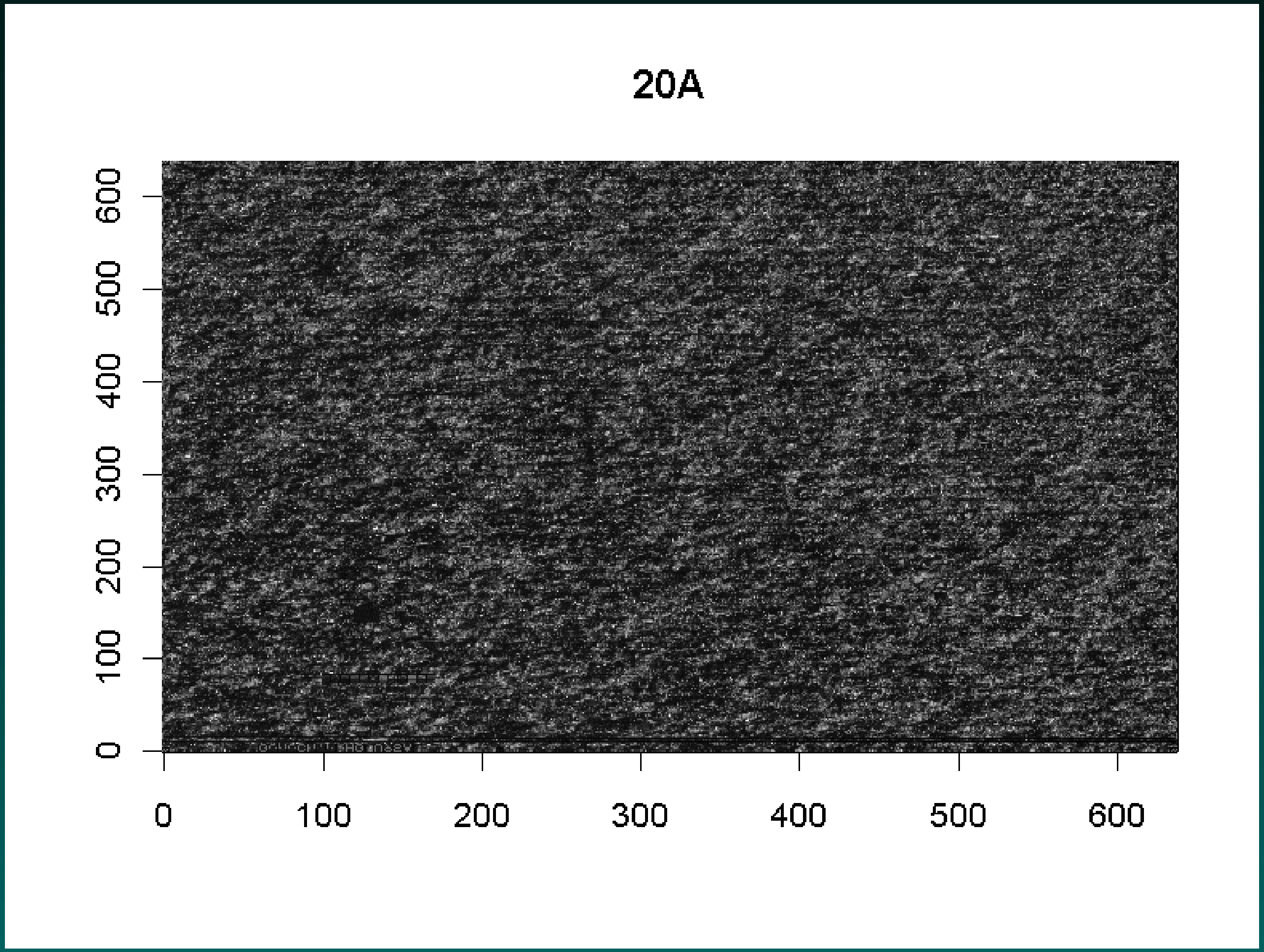
Looking at the experimental design

You can see what the experiments are by looking at the phenotype information.

```
> phenoData(Dilution)
    phenoData object with 3 variables and 4 cases
    varLabels
      liver: amount of liver RNA hybridized to array
      sn19: amount of central nervous system RNA hyb
      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
```

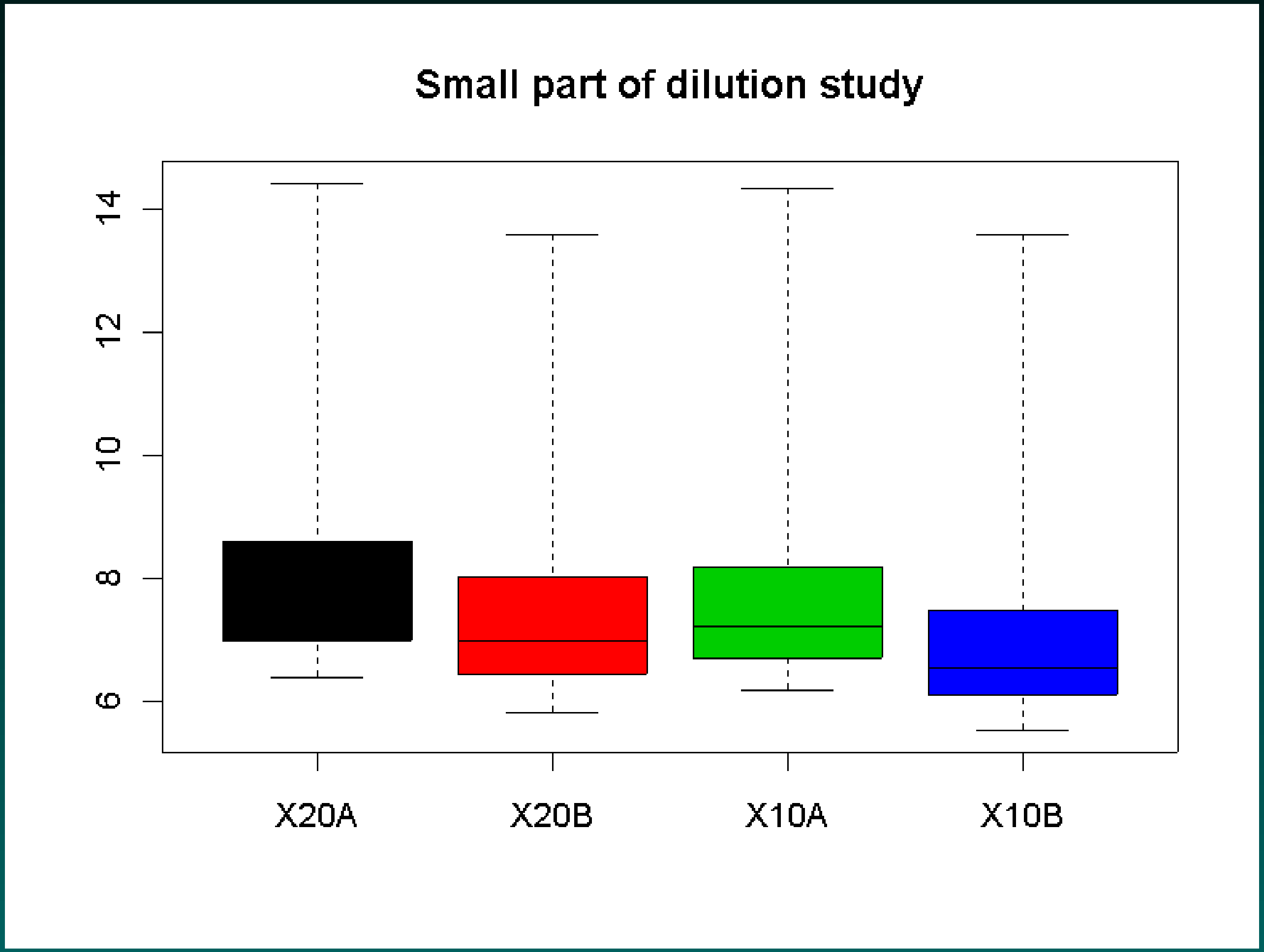
A first look at an array

```
> image(Dilution[,1])
```



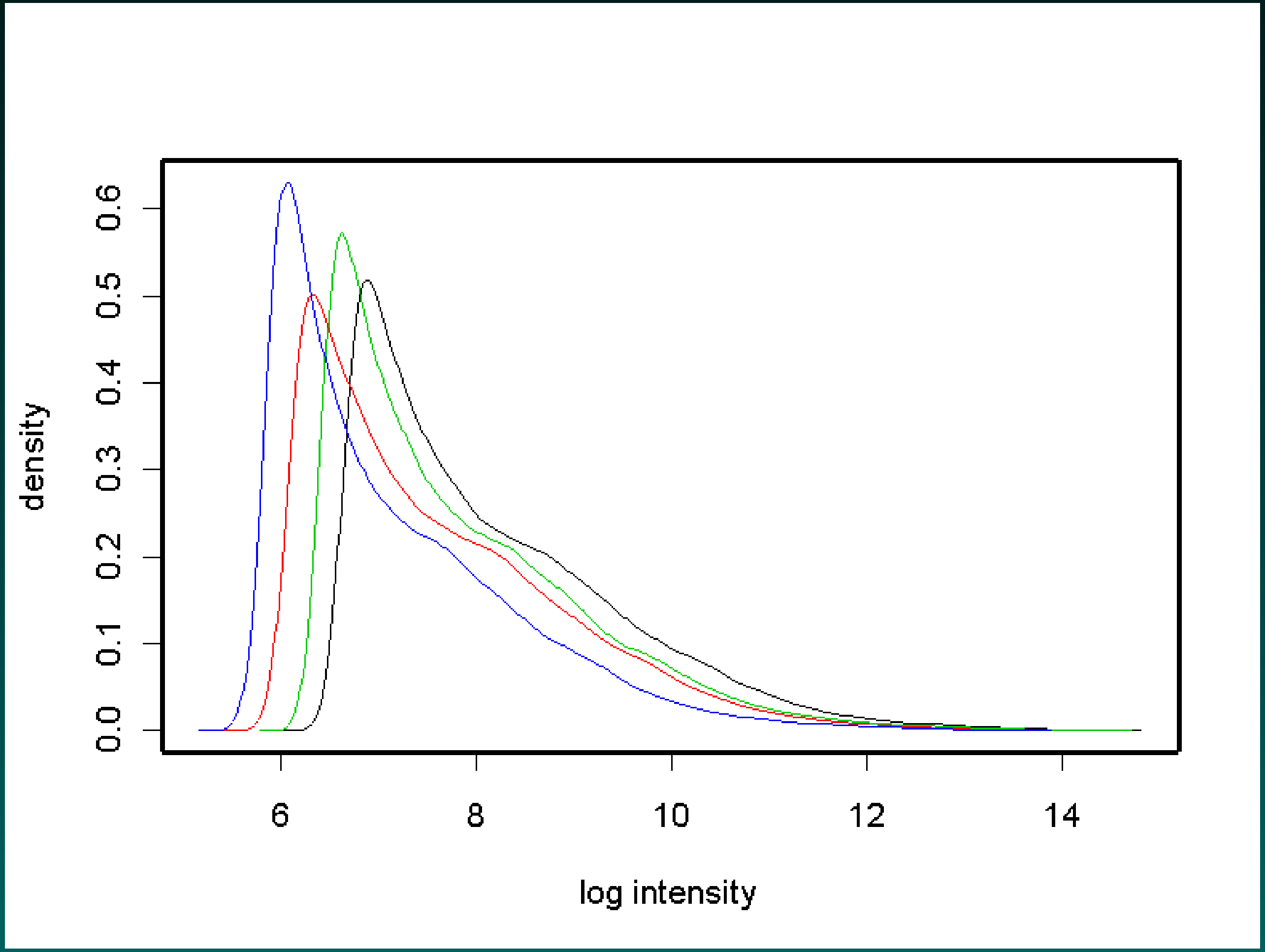
A summary view of four images

```
> boxplot(Dilution, col=1:4)
```



The distribution of feature intensities

```
> hist(Dilution, col=1:4, lty=1)
```



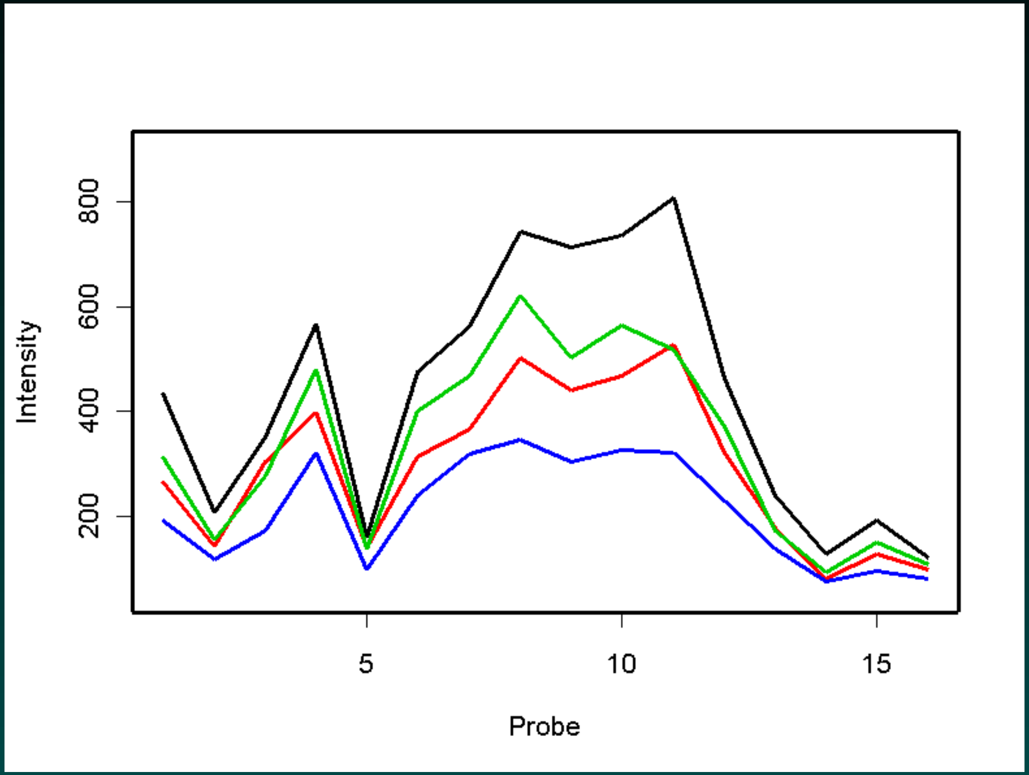
Examining individual probesets

The `affy` package in BioConductor includes tools for extracting individual probe sets from a complete `AffyBatch` object. To get at the probe sets, however, you need to be able to refer to them by their “name”, which at present means their Affymetrix ID.

```
> geneNames(Dilution)[1:3]
[1] "100_g_at" "1000_at" "1001_at"
> random.affyid <- sample(geneNames(Dilution), 1)
> # random.affyid <- ’34803_at’
> ps <- probeset(Dilution, random.affyid)[[1]]
```

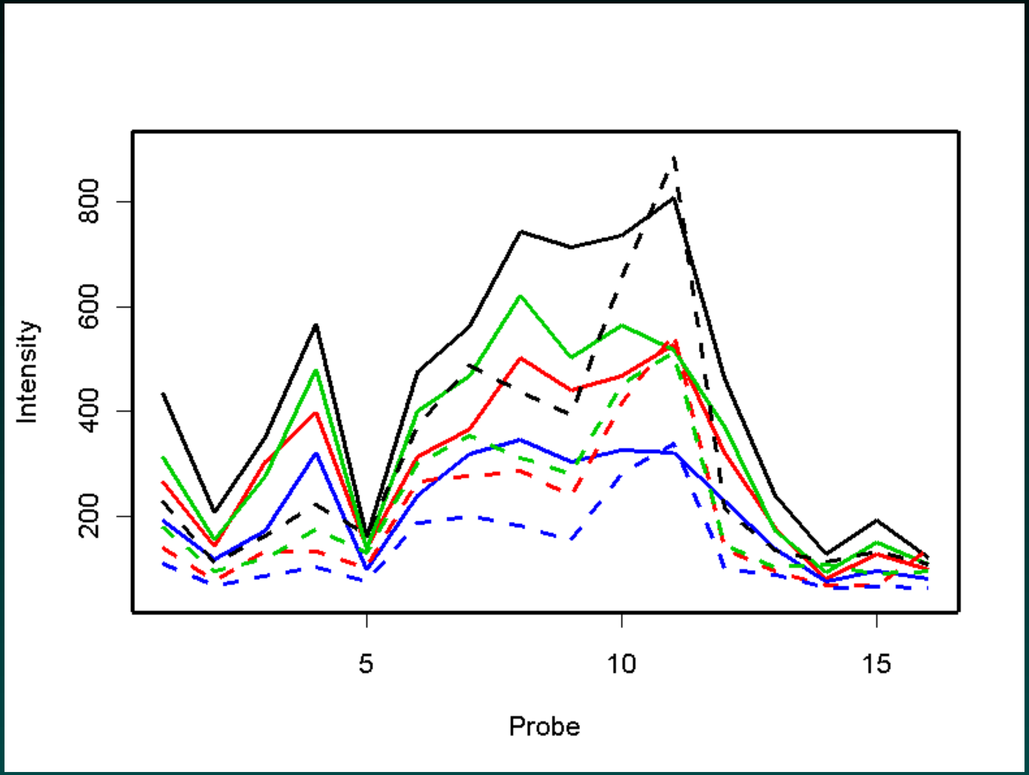
The `probeset` function returns a list of probe sets; the mysterious stuff with the brackets takes the first element from the list (which only had one...).

A probeset profile in four arrays



```
> plot(c(1,16), c(50, 900), type='n',
+    xlab='Probe', ylab='Intensity')
> for (i in 1:4) lines(pm(ps)[,i], col=i)
```

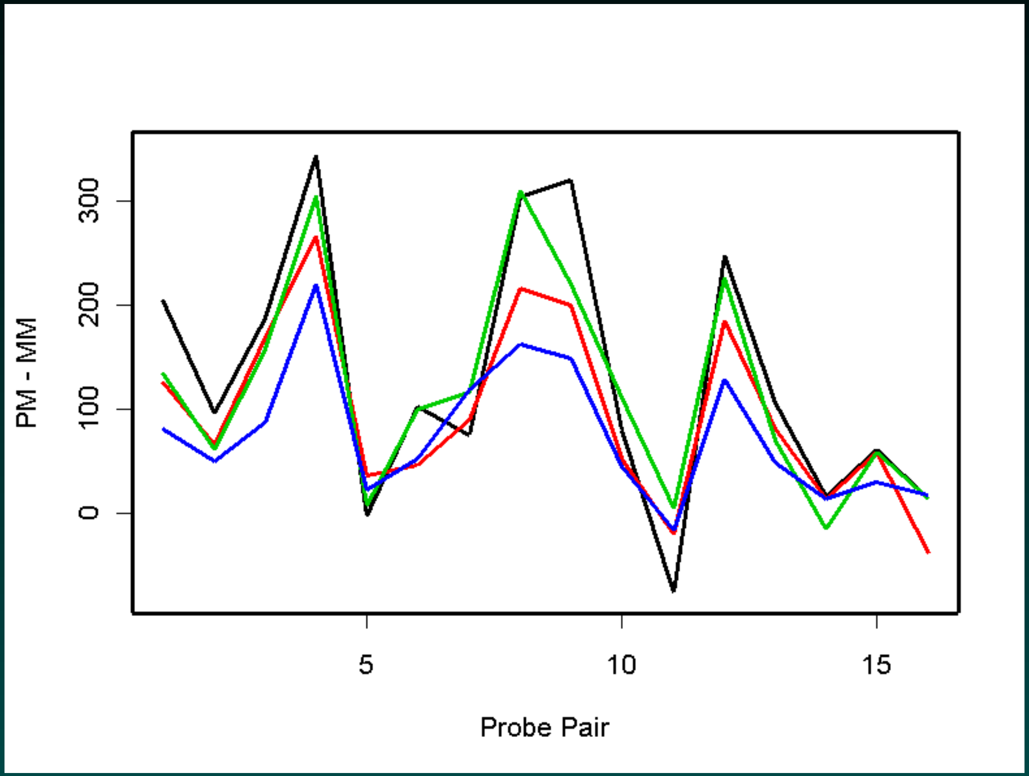
Examining individual probesets



Let’s add the mismatch probes to the graph:

```
> for (i in 1:4) lines(pm(ps)[,i], col=i)
```

PM – MM



```
> plot(c(1,16), c(-80, 350), type='n',
+   xlab='Probe Pair', ylab='PM - MM)
> temp <- pm(ps) - mm(ps)
> for (i in 1:4) lines(temp[,i], col=i)
```

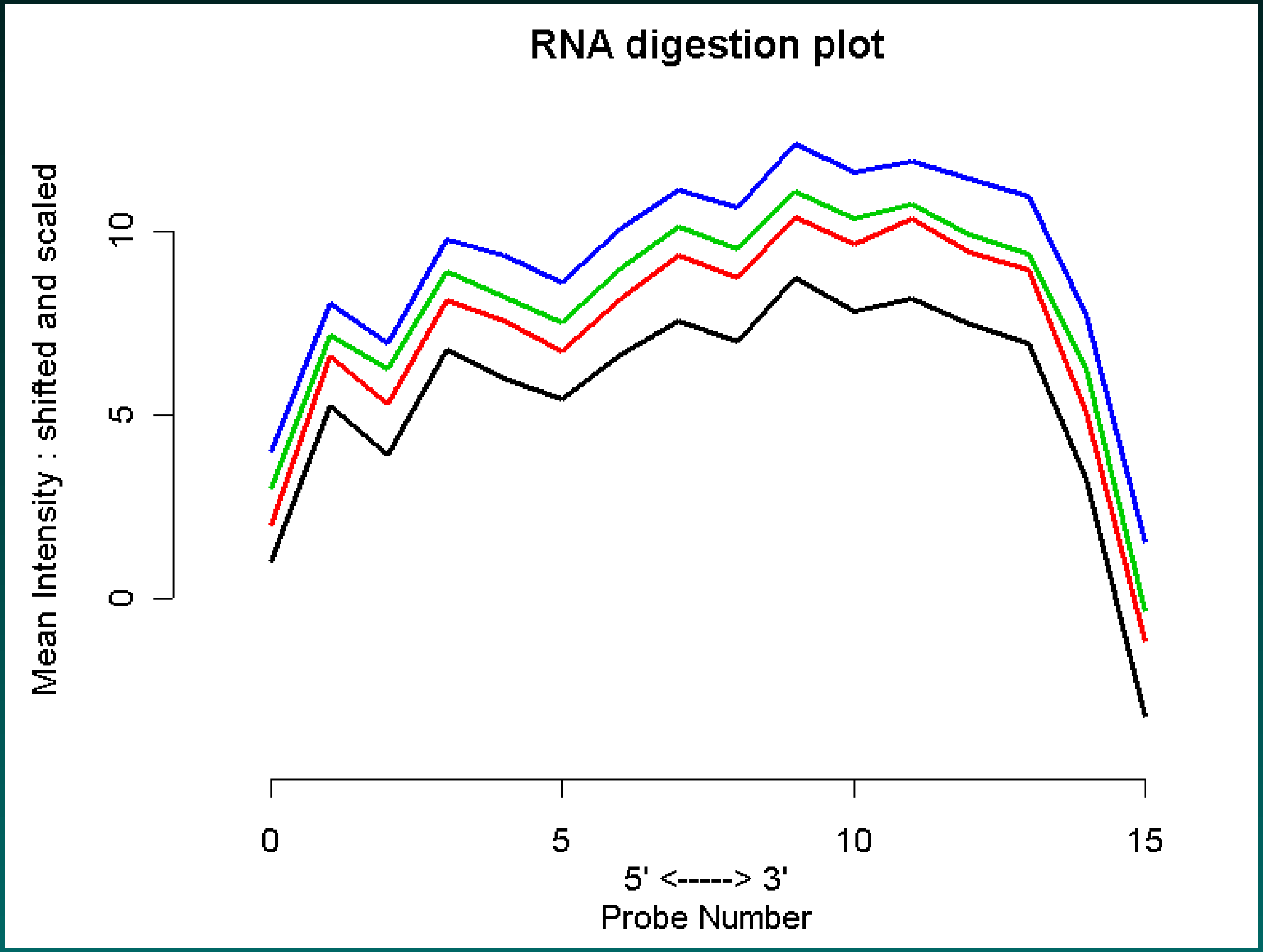
RNA degradation

Individual (perfect match) probes in each probe set are ordered by location relative to the 5’ end of the targeted mRNA molecule. We also know that RNA degradation typically starts at the 5’ end, so we would expect probe intensities to be lower near the 5’ end than near the 3’ end.

The `affy` package of BioConductor includes functions to summarize and plot the degree of RNA degradation in a series of Affymetrix experiments. These methods pretend that something like “the fifth probe in an Affymetrix probe set” is a meaningful notion, and they average these things over all probe sets on the array.

Visualizing RNA degradation

```
> degrade <- AffyRNAdeg(Dilution)
> plotAffyRNAdeg(degrade)
```



Processing Affymetrix data

BioConductor breaks down the low-level processing of Affymetrix data into four steps. The design is highly modular, so you can choose different algorithms at each step. It is highly likely that the results of later (high-level) analyses will change depending on yopur choices at these steps.

- Background correction
- Normalization (on features)
- PM-correction
- Summarization

Background correction

The list of available background correction methods is stored in a variable:

```
> bgcorrect.methods
[1] "mas"  "none" "rma"  "rma2"
```

So there are four methods:

none Do nothing

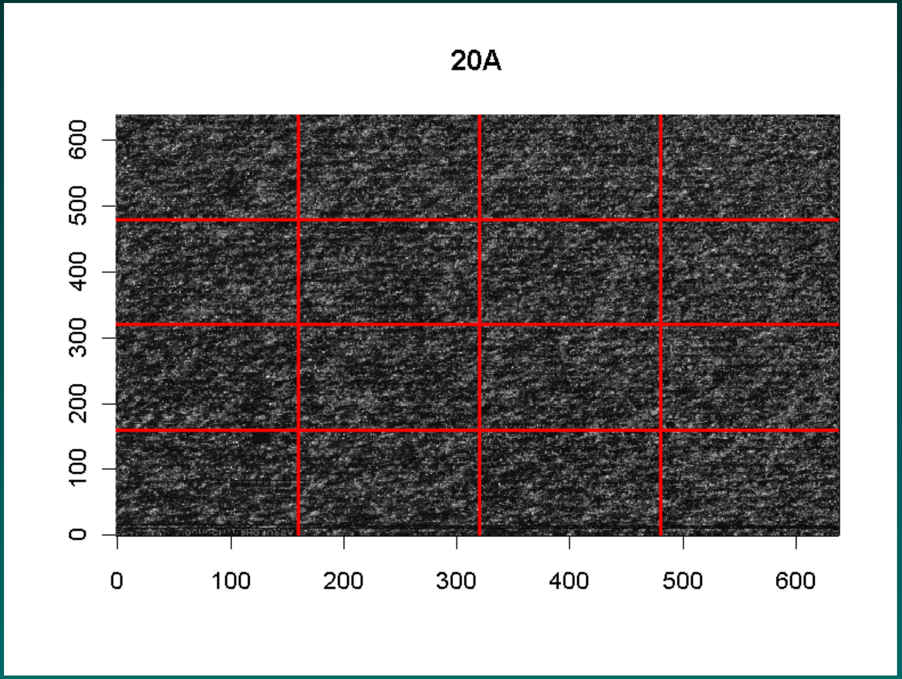
mas Use the algorithm from MAS 5.0

rma Use the algorithm from the current version of RMA

rma2 Use the algorithm from an older version of RMA

Background correction in MAS 5.0

MAS 5.0 divides the microarray (more precisely, the CEL file) into 16 regions. In each region, the intensity of the dimmest 2% of features is used to define the background level. Each probe is then adjusted by a weighted average of these 16 values, with the weights depending on the distance to the centroids of the 16 regions.



Background correction in RMA

RMA takes a very different approach to background correction. First, only PM values are adjusted, the MM values are not changed at all. Second, they try to model the distribution of PM intensities statistically as a sum of

- exponential signal with mean λ
- normal noise with mean μ and variance σ^2 (truncated at 0 to avoid negatives).

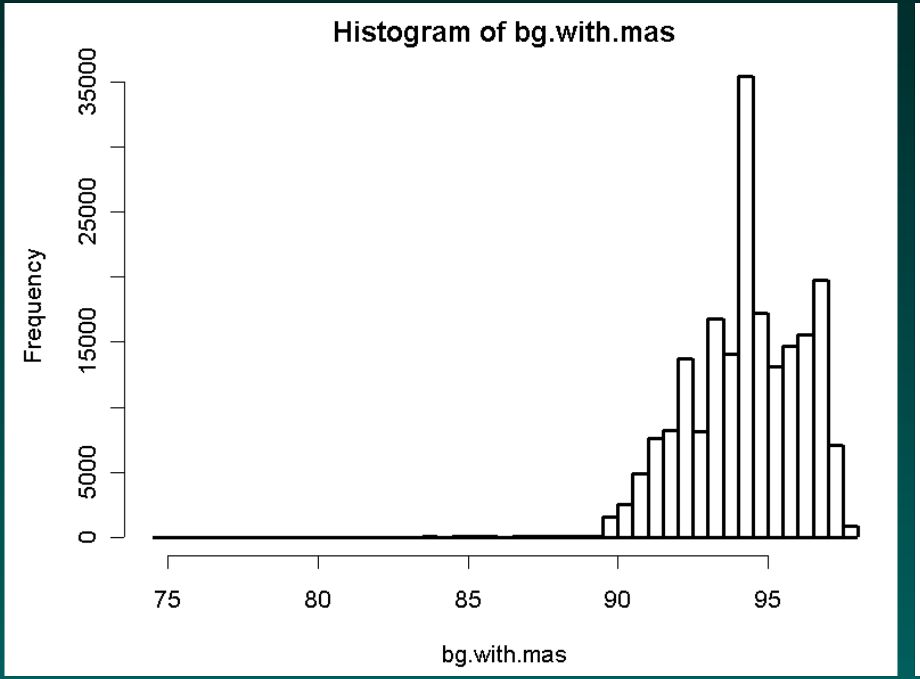
If we observe a signal $X = x$ at a PM feature, we adjust it by

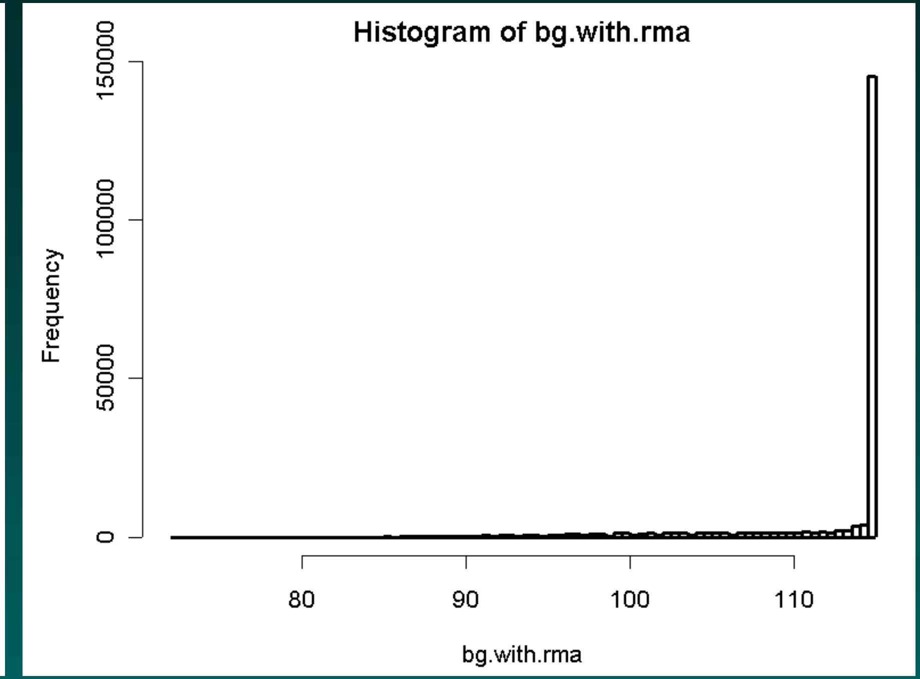
$$E(s|X = x) = a + b\frac{\phi(a/b) - \phi((x-a)/b)}{\Phi(a/b) + \Phi((x-a)/b) - 1}$$

where $b = \sigma$ and $a = s - \mu - \lambda\sigma^2$.

Comparing background methods

```
> d.mas <- bg.correct(Dilution[,1], "mas")
> d.rma <- bg.correct(Dilution[,1], "rma")
> bg.with.mas <- pm(Dilution[,1]) - pm(d.mas)
> bg.with.rma <- pm(Dilution[,1]) - pm(d.rma)
```

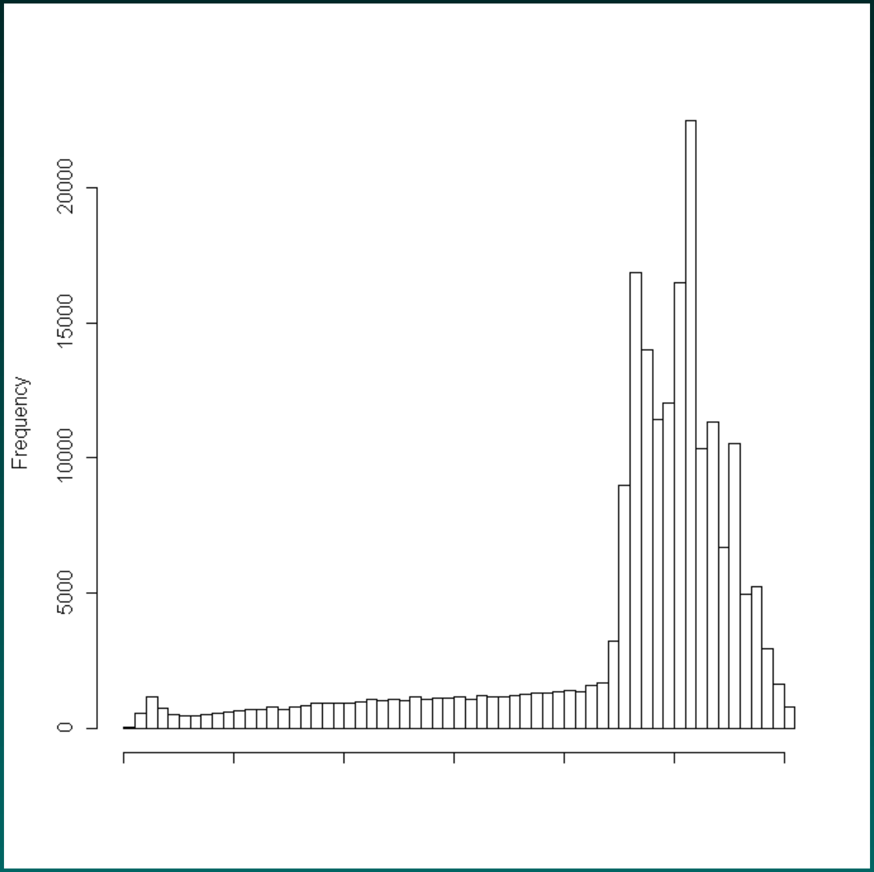




```
> summary(bg.with.mas)
Min.   :74.53
1st Qu.:93.14
Median :94.35
Mean   :94.27
3rd Qu.:95.80
Max.   :97.67
> summary(bg.with.rma)
Min.   : 72.4
1st Qu.:113.7
Median :114.9
Mean   :112.1
3rd Qu.:114.9
Max.   :114.9
```

Difference in background estimates

On this array, RMA gives slightly larger background estimates, and gives estimates that are more nearly constant across the array. The overall differences can be displayed in a histogram.



Quantification = summarization

I'm going to avoid talking about normalization and PM correction for the moment, and jump ahead to summarization. As we have explained previously, this step is the critical final component in analyzing Affymetrix arrays, since it's the one that combines all the numbers from the PM and MM probe pairs in a probe set into a single number that represents our best guess at the expression level of the targeted gene.

The available summarization methods, like the other available methods, can be obtained from a variable.

```
express.summary.stat.methods
[1] "avgdiff"      "liwong"       "mas"
    "medianpolish" "playerout"
```

Including the PDNN method

The implementation of the PDNNmethod is contianed in a separate package. When you load the package libary, it updates the list of available methods.

```
> library(affypdnn)
registering new summary method ’pdnn’.
registering new pmcorrect method ’pdnn’
  and ’pdnnpredict’.
> express.summary.stat.methods
[1] "avgdiff"      "liwong"       "mas"
[4] "medianpolish" "playerout"    "pdnn"
```

expresso

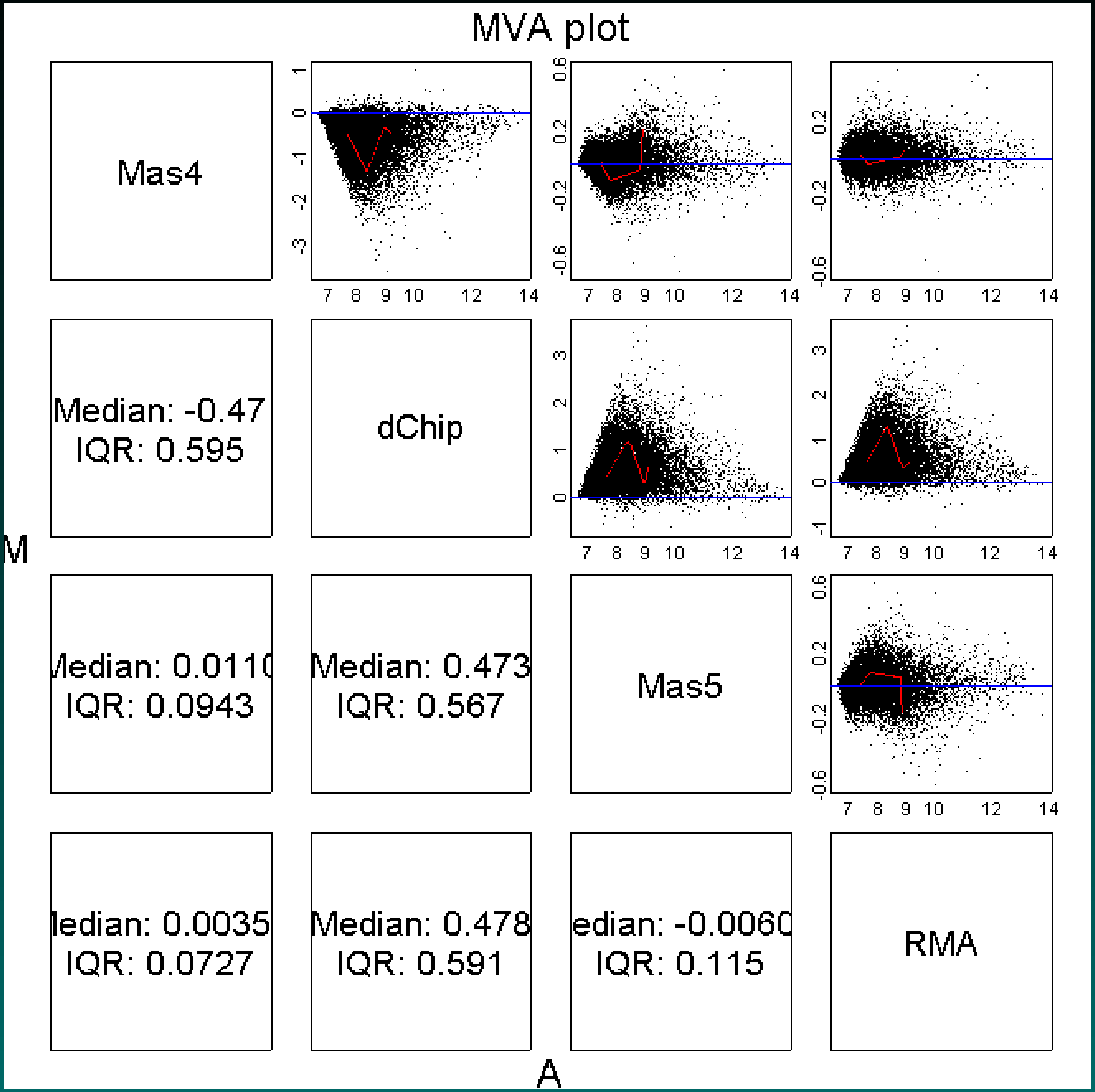
The recommended way to put together all the steps for processing Affymetrix arrays in BioConductor is with the function `expresso`. Here's an example that blocks everything except the summarization:

```
> tempfun <- function(method) {
+   expresso(Dilution, bg.correct=FALSE,
+   normalize=FALSE, pmcorrect.method="pmonly",
+   summary.method=method)
+ }
> ad <- tempfun("avgdiff") # MAS4.0
> al <- tempfun("liwong")  # dChip
> am <- tempfun("mas")     # MAS5.0
> ap <- tempfun("pdnn")    # PDNN
> ar <- tempfun("medianpolish") # RMA
```

M-versus-A plots

Instead of plotting two similar things on the usual x and y axes, plot the average ($(x+y)/2$) along the horizontal axis and the difference ($y-x$) along the vertical axis. The `affy` package includes a function called `mva.pairs` to make it easier to generate these plots. We're going to use this to compare the different quantification/summary methods.

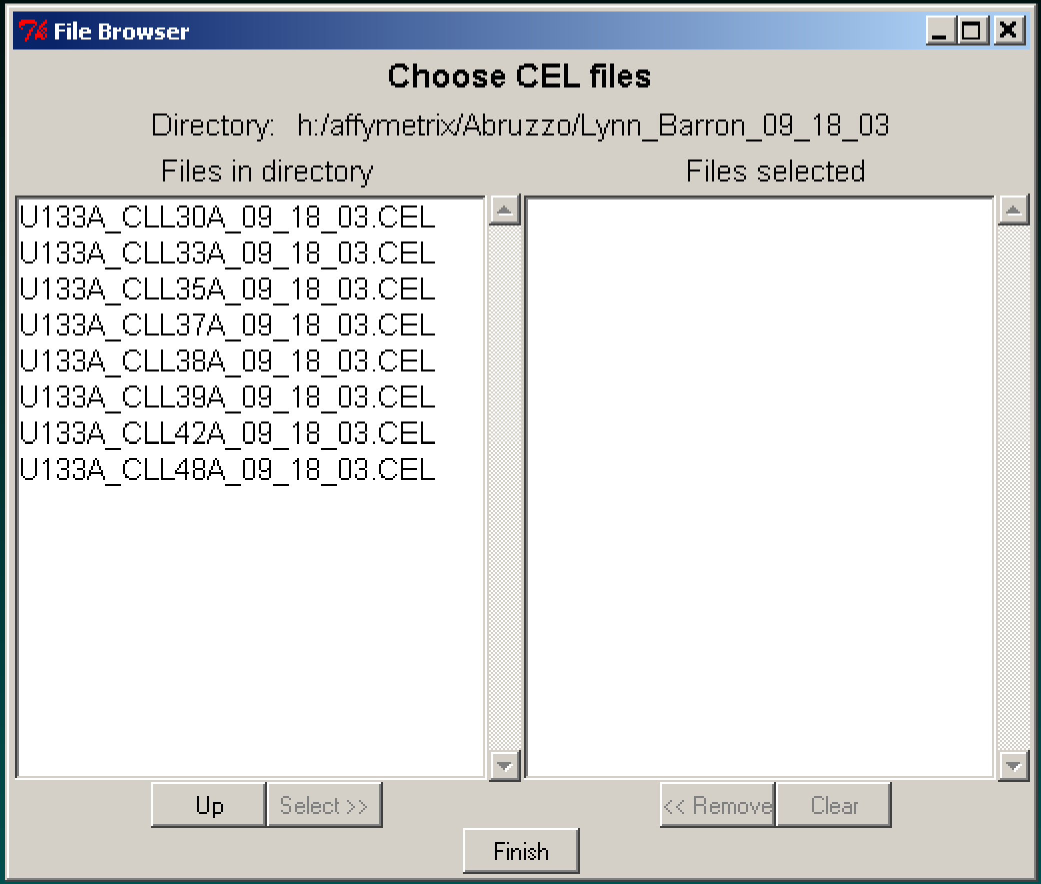
```
> temp <- data.frame(exprs(ad)[,1], exprs(al)[,1],
+   exprs(am)[,1], 2^exprs(ar)[,1])
> dimnames(temp)[[2]] <- c('Mas4', 'dChip',
+   'Mas5', 'RMA')
```


MVA plot
Mas4
dChip
Mas5
RMA
Median: -0.47
IQR: 0.595
Median: 0.0110
IQR: 0.0943
Median: 0.473
IQR: 0.567
Median: 0.0035
IQR: 0.0727
Median: 0.478
IQR: 0.591
Median: -0.0060
IQR: 0.115
M
A

More about reading Affymetrix data

The BioConductor `affy` package includes a graphical interface to make it easier to read in Affymetrix data and contruct `AffyBatch` objects.

Affy Widgets

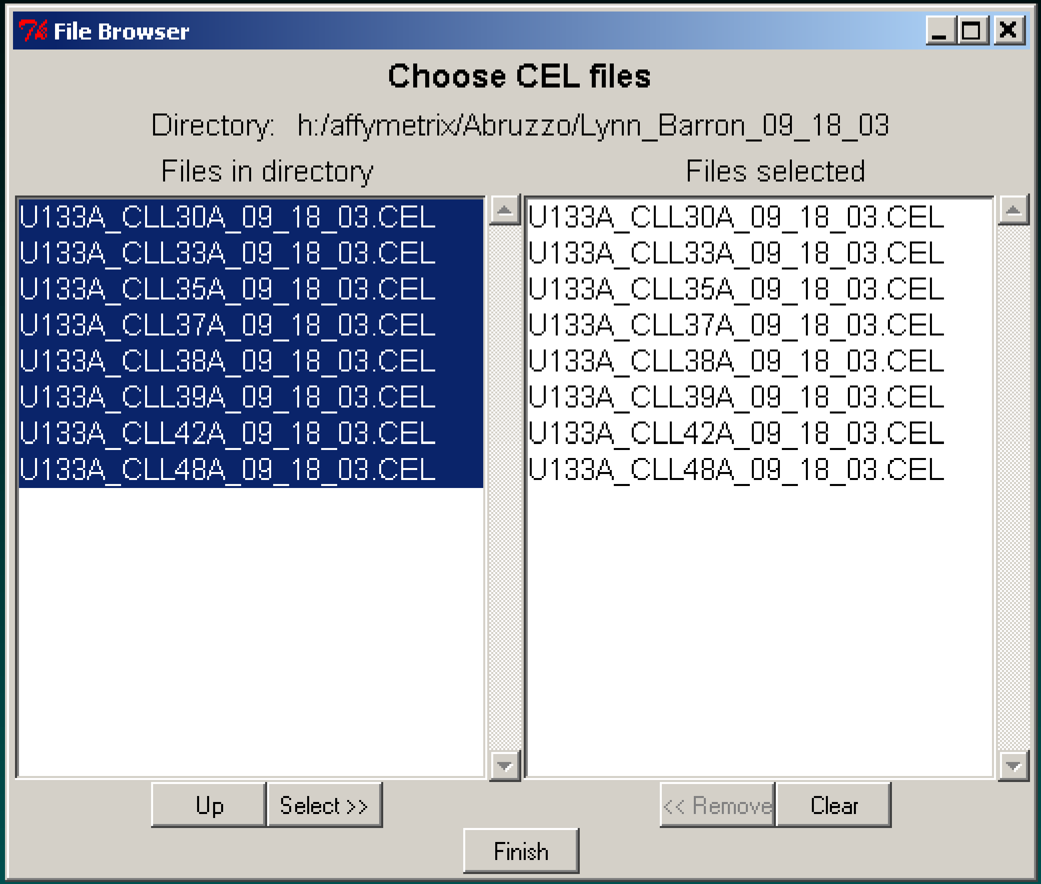


Choose CEL files

Directory: h:/affymetrix/Abruzzo/Lynn_Barron_09_18_03

Files in directory	Files selected
U133A_CLL30A_09_18_03.CEL	
U133A_CLL33A_09_18_03.CEL	
U133A_CLL35A_09_18_03.CEL	
U133A_CLL37A_09_18_03.CEL	
U133A_CLL38A_09_18_03.CEL	
U133A_CLL39A_09_18_03.CEL	
U133A_CLL42A_09_18_03.CEL	
U133A_CLL48A_09_18_03.CEL	

Affy Widgets

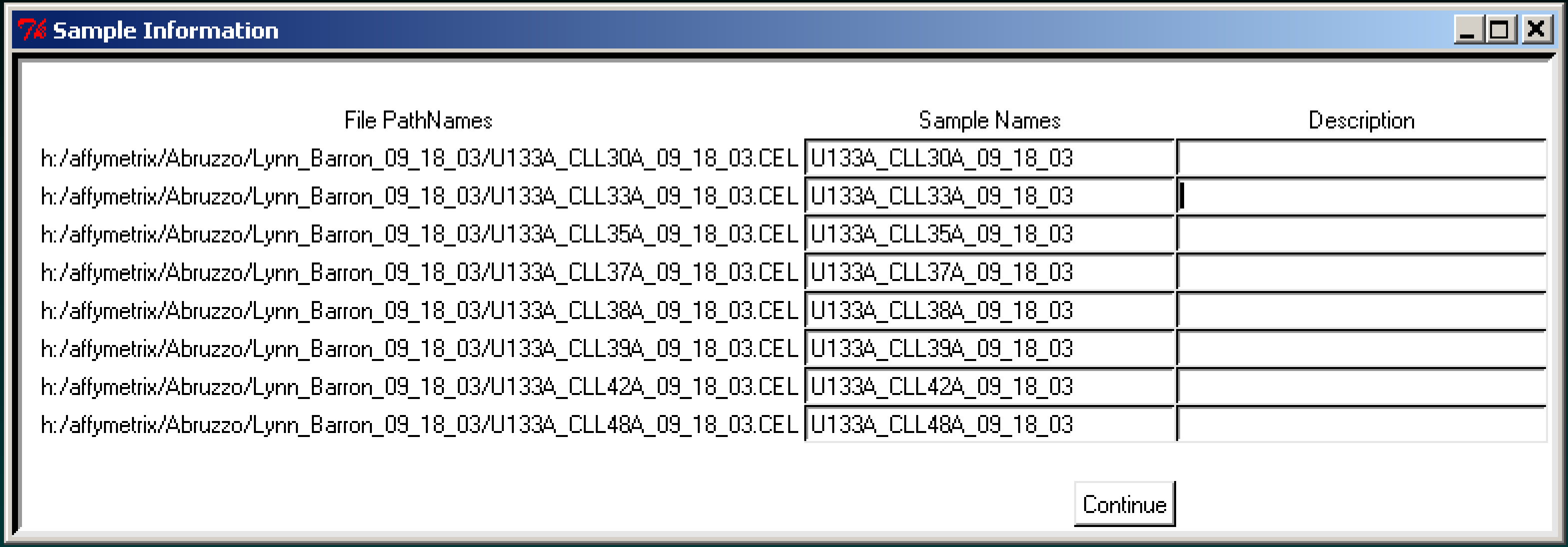


Choose CEL files

Directory: h:/affymetrix/Abruzzo/Lynn_Barron_09_18_03

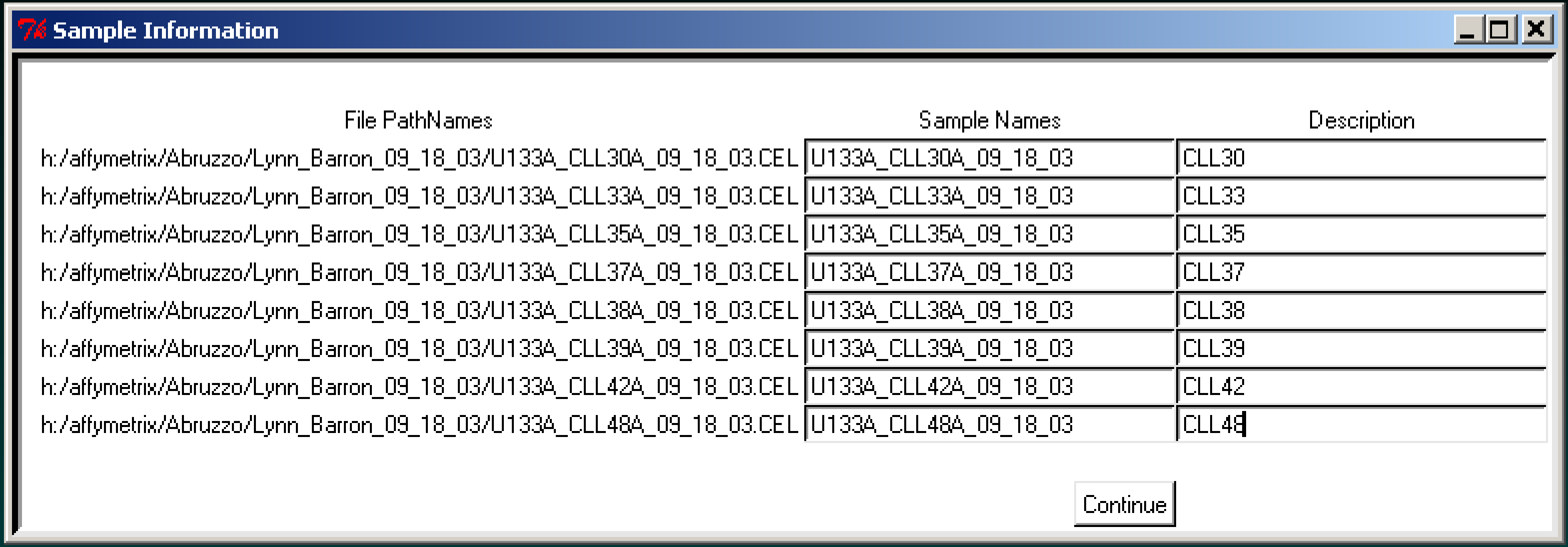
Files in directory	Files selected
U133A_CLL30A_09_18_03.CEL	U133A_CLL30A_09_18_03.CEL
U133A_CLL33A_09_18_03.CEL	U133A_CLL33A_09_18_03.CEL
U133A_CLL35A_09_18_03.CEL	U133A_CLL35A_09_18_03.CEL
U133A_CLL37A_09_18_03.CEL	U133A_CLL37A_09_18_03.CEL
U133A_CLL38A_09_18_03.CEL	U133A_CLL38A_09_18_03.CEL
U133A_CLL39A_09_18_03.CEL	U133A_CLL39A_09_18_03.CEL
U133A_CLL42A_09_18_03.CEL	U133A_CLL42A_09_18_03.CEL
U133A_CLL48A_09_18_03.CEL	U133A_CLL48A_09_18_03.CEL

Affy Widgets



File PathNames	Sample Names	Description
h:/affymetrix/Abruzzo/Lynn_Barron_09_18_03/U133A_CLL30A_09_18_03.CEL	U133A_CLL30A_09_18_03	
h:/affymetrix/Abruzzo/Lynn_Barron_09_18_03/U133A_CLL33A_09_18_03.CEL	U133A_CLL33A_09_18_03	
h:/affymetrix/Abruzzo/Lynn_Barron_09_18_03/U133A_CLL35A_09_18_03.CEL	U133A_CLL35A_09_18_03	
h:/affymetrix/Abruzzo/Lynn_Barron_09_18_03/U133A_CLL37A_09_18_03.CEL	U133A_CLL37A_09_18_03	
h:/affymetrix/Abruzzo/Lynn_Barron_09_18_03/U133A_CLL38A_09_18_03.CEL	U133A_CLL38A_09_18_03	
h:/affymetrix/Abruzzo/Lynn_Barron_09_18_03/U133A_CLL39A_09_18_03.CEL	U133A_CLL39A_09_18_03	
h:/affymetrix/Abruzzo/Lynn_Barron_09_18_03/U133A_CLL42A_09_18_03.CEL	U133A_CLL42A_09_18_03	
h:/affymetrix/Abruzzo/Lynn_Barron_09_18_03/U133A_CLL48A_09_18_03.CEL	U133A_CLL48A_09_18_03	

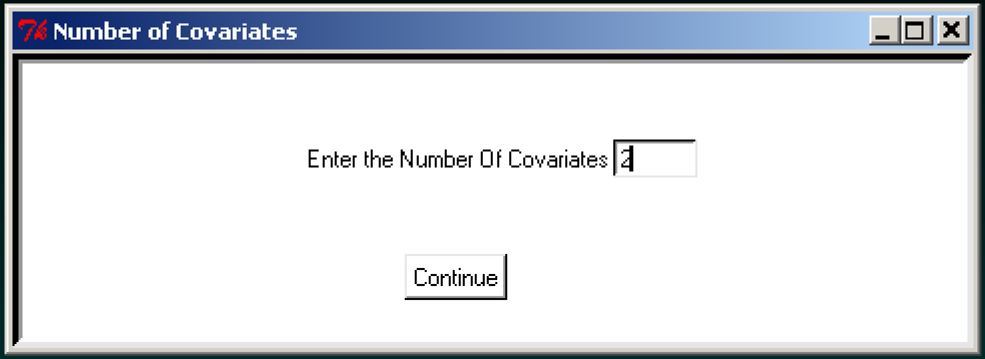
Affy Widgets



Sample Information

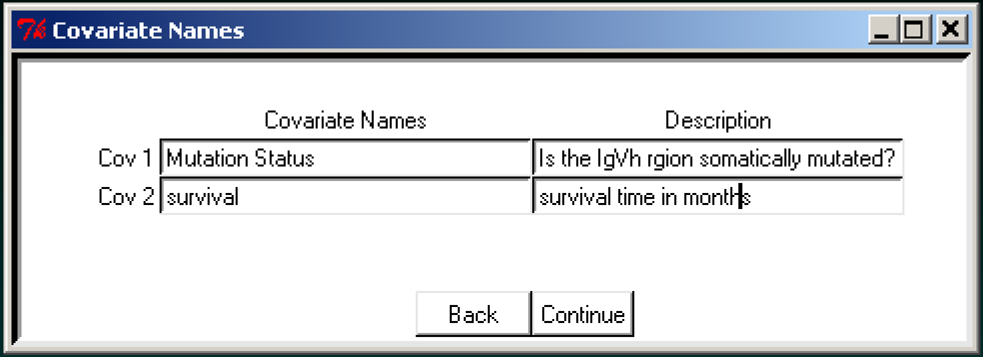
File PathNames	Sample Names	Description
h:/affymetrix/Abruzzo/Lynn_Barron_09_18_03/U133A_CLL30A_09_18_03.CEL	U133A_CLL30A_09_18_03	CLL30
h:/affymetrix/Abruzzo/Lynn_Barron_09_18_03/U133A_CLL33A_09_18_03.CEL	U133A_CLL33A_09_18_03	CLL33
h:/affymetrix/Abruzzo/Lynn_Barron_09_18_03/U133A_CLL35A_09_18_03.CEL	U133A_CLL35A_09_18_03	CLL35
h:/affymetrix/Abruzzo/Lynn_Barron_09_18_03/U133A_CLL37A_09_18_03.CEL	U133A_CLL37A_09_18_03	CLL37
h:/affymetrix/Abruzzo/Lynn_Barron_09_18_03/U133A_CLL38A_09_18_03.CEL	U133A_CLL38A_09_18_03	CLL38
h:/affymetrix/Abruzzo/Lynn_Barron_09_18_03/U133A_CLL39A_09_18_03.CEL	U133A_CLL39A_09_18_03	CLL39
h:/affymetrix/Abruzzo/Lynn_Barron_09_18_03/U133A_CLL42A_09_18_03.CEL	U133A_CLL42A_09_18_03	CLL42
h:/affymetrix/Abruzzo/Lynn_Barron_09_18_03/U133A_CLL48A_09_18_03.CEL	U133A_CLL48A_09_18_03	CLL48

Affy Widgets



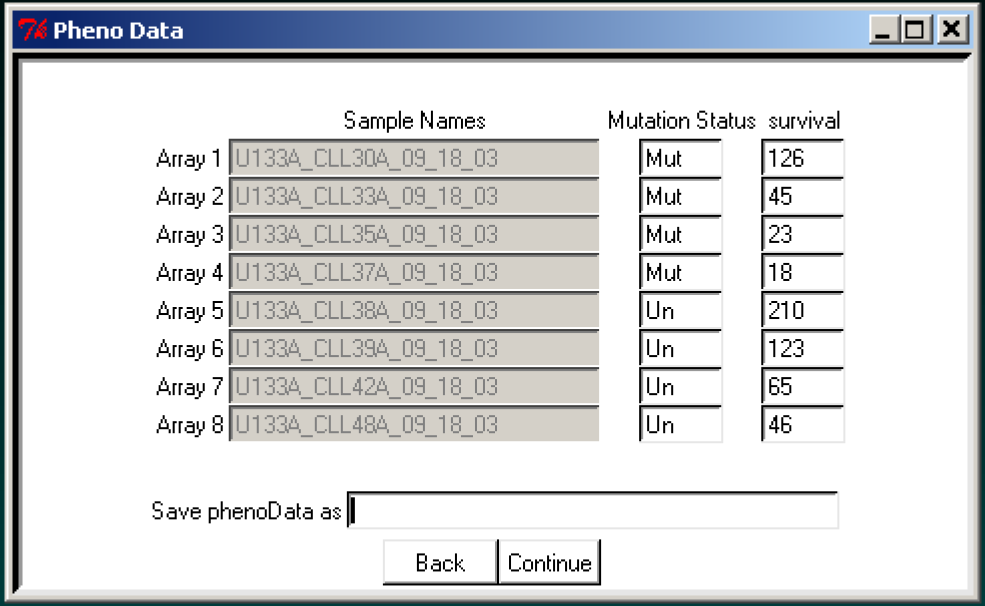
Enter the Number Of Covariates 2

Affy Widgets



	Covariate Names	Description
Cov 1	Mutation Status	Is the IgVh rgion somatically mutated?
Cov 2	survival	survival time in months

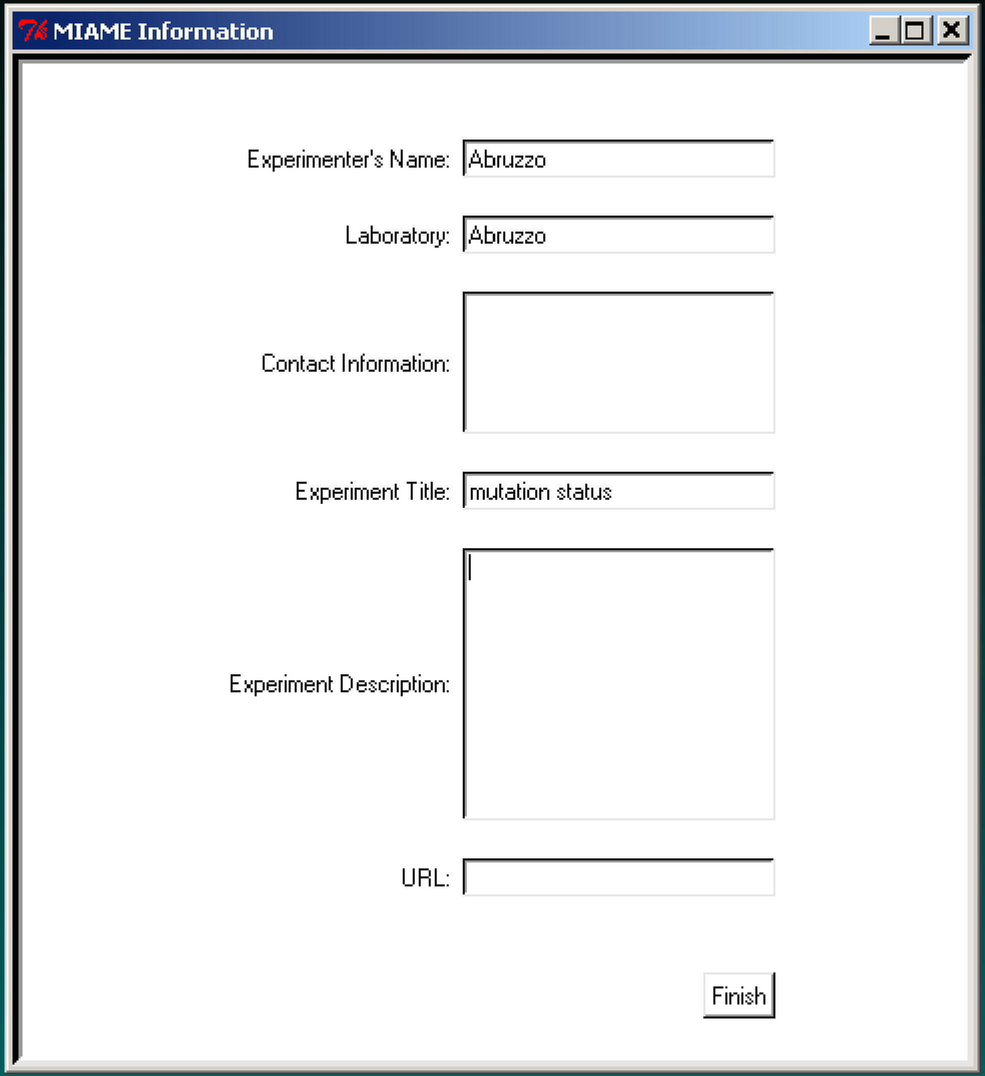
Affy Widgets



	Sample Names	Mutation Status	survival
Array 1	U133A_CLL30A_09_18_03	Mut	126
Array 2	U133A_CLL33A_09_18_03	Mut	45
Array 3	U133A_CLL35A_09_18_03	Mut	23
Array 4	U133A_CLL37A_09_18_03	Mut	18
Array 5	U133A_CLL38A_09_18_03	Un	210
Array 6	U133A_CLL39A_09_18_03	Un	123
Array 7	U133A_CLL42A_09_18_03	Un	65
Array 8	U133A_CLL48A_09_18_03	Un	46

Save phenoData as

Affy Widgets



Experimenter's Name: Abruzzo

Laboratory: Abruzzo

Contact Information:

Experiment Title: mutation status

Experiment Description:

URL: