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

14 September 2006

Lecture 5: R, Objects and Affymetrix Arrays

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

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

Beyond Matrices

We have 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"
```

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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.

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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...

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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.

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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.

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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.

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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...

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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 = "\"", dec = ".", row.names, col.names,
   as.is = FALSE, na.strings = "NA", colClasses = NA,
   nrows = -1, skip = 0, check.names = TRUE,
   fill =  !blank.lines.skip, strip.white = FALSE,
   blank.lines.skip = TRUE, comment.char = "#")

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

Note the default function arguments!

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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!

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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;
```

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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
    );
```

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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.

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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);
```

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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 ’.’
```

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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);
```

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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 ’.’
```

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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.

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

Bioconductor Packages

You will need the following packages from the Bioconductor web site. Use the items “Select repositories...” and “Install package(s)...” on the “Packages” menu 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

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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

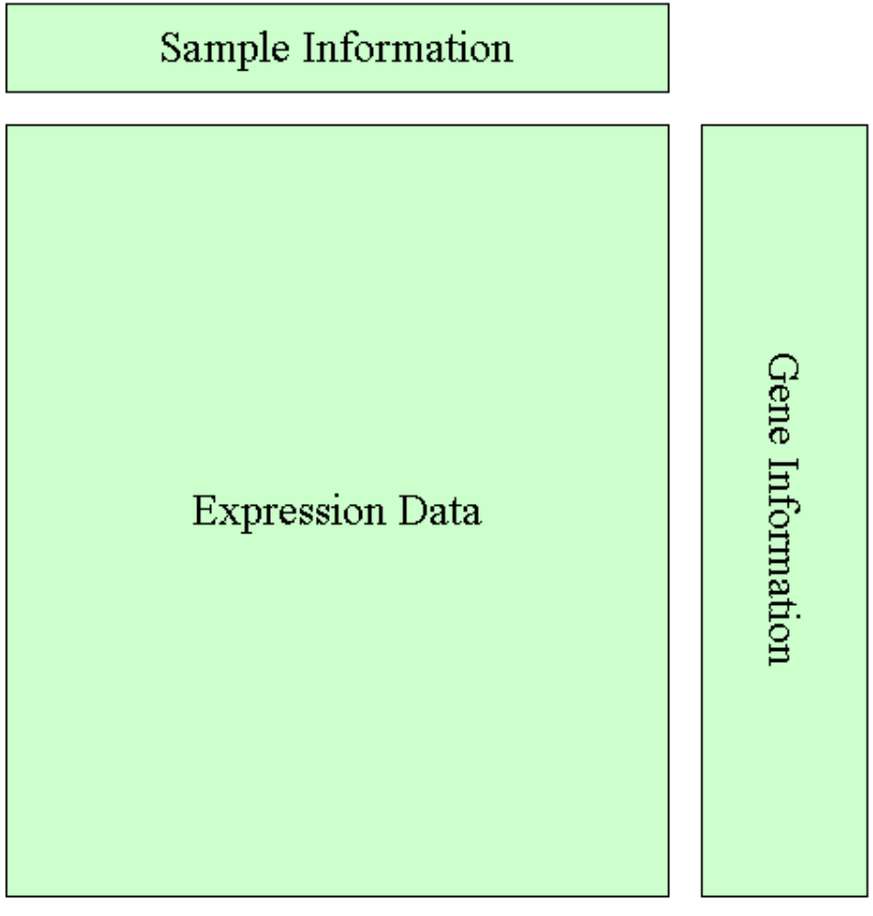
widgetTools：Creates an interactive tcltk widget

DynDoc：Dynamic document tools

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

Microarray Data Structures

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



© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

Experiment/Sample Information

In even the simplest experimental designs, where we want to find out which genes are differentially expressed between two types of samples, we at least have to be told which samples are of which type. In more complicated experimental designs, we may be interested in a number of additional factors. For example, in a study comparing cancer patients to healthy individuals, we may want to record the age and sex of the study subjects. In animal experiments, there may be a variety of different treatments that have to be recorded.

The R object that holds this kind of information is a `data.frame`. Conceptually, a `data.frame` is just a two-dimensional table. By convention, they are arranged so that each row corresponds to an experimental sample and each column corresponds to one of the interesting factors.

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

Example of a data.frame

Array	Age	Sex	Status
a1	41	M	cancer
a2	64	F	cancer
a3	56	M	healthy
a4	48	F	healthy

Data frames are particularly useful for this purpose in R, because they can hold textual factors as well as numeric ones. For most array sudies, it is best to create a table of the interesting information and store it in a separate file. If you create the table in a spreadsheeet program (like Excel), you should store it as a text file in “tab-separated-value” format. That is, each row holds the information from one experiment, and column entries are separated by tab characters.

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

Phenotypes

You can create a data frame in R from a file in tab-separate-value format using the `read.table` command. (You can also create them directly, as illustrated later.)

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.

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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(spl='Sample Name',
                  stat='Cancer Status'))
```

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

```
> pheno
    phenoData object with 2 variables and 8 cases
    varLabels
        spl : Sample Name
        stat : 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
```

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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
        spl : Sample Name
        stat : Cancer Status
```

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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.

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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.

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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.

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

Demonstration data

Note: If you are trying to follow along and have not yet obtained some CEL files, the `affydata` package includes demonstration data from 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.

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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
```

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

Looking at the experimental design

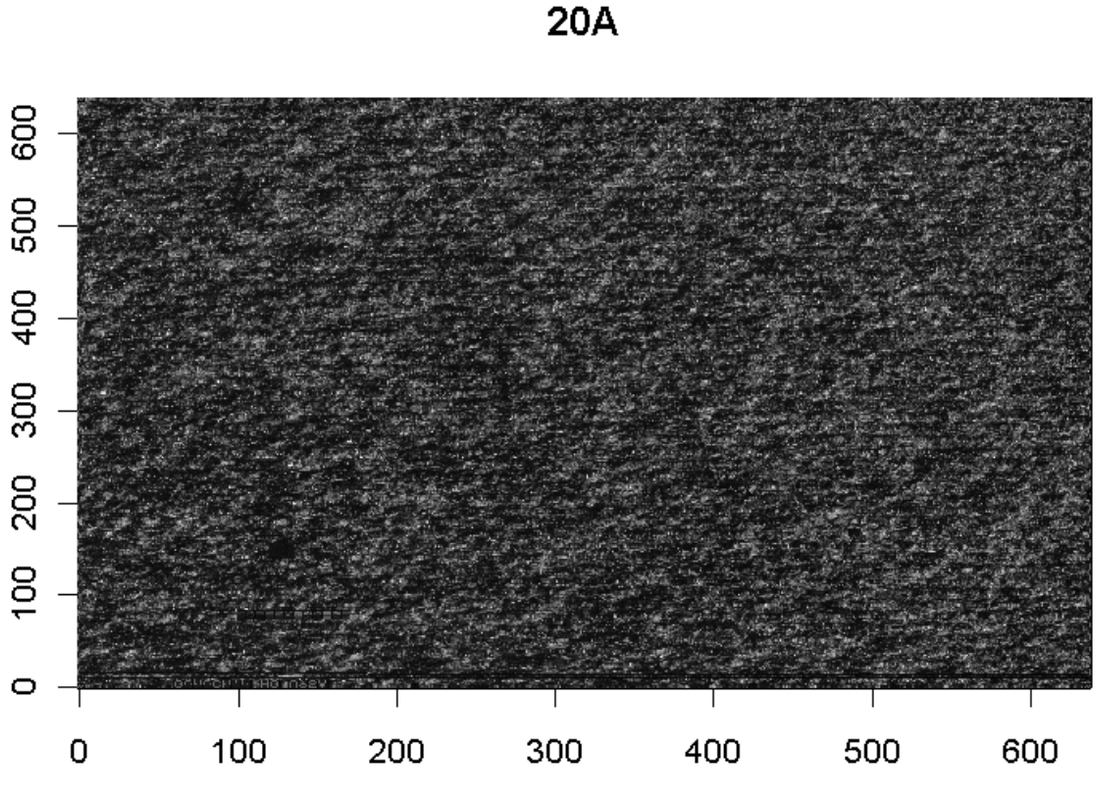
You can see what the experiments are by looking at the phenotype information.

```
> 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 hybi
      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
```

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

A first look at an array

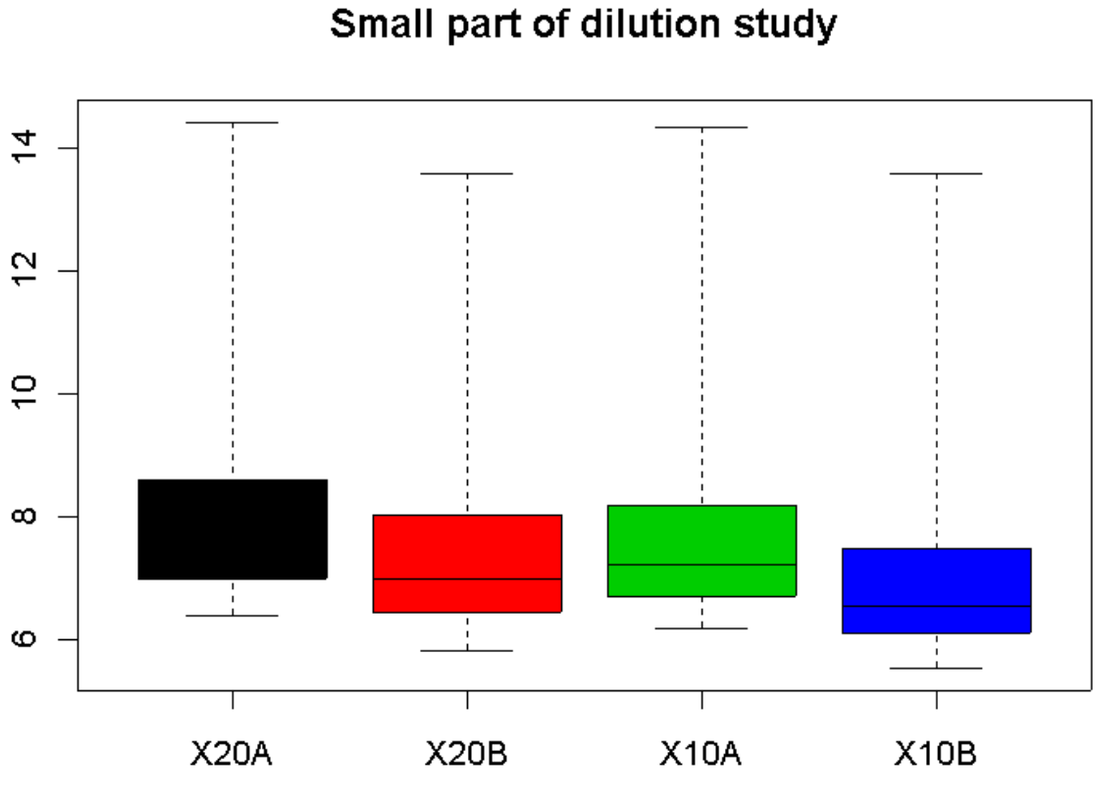
```
> image(Dilution[,1])
```



© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

A summary view of four images

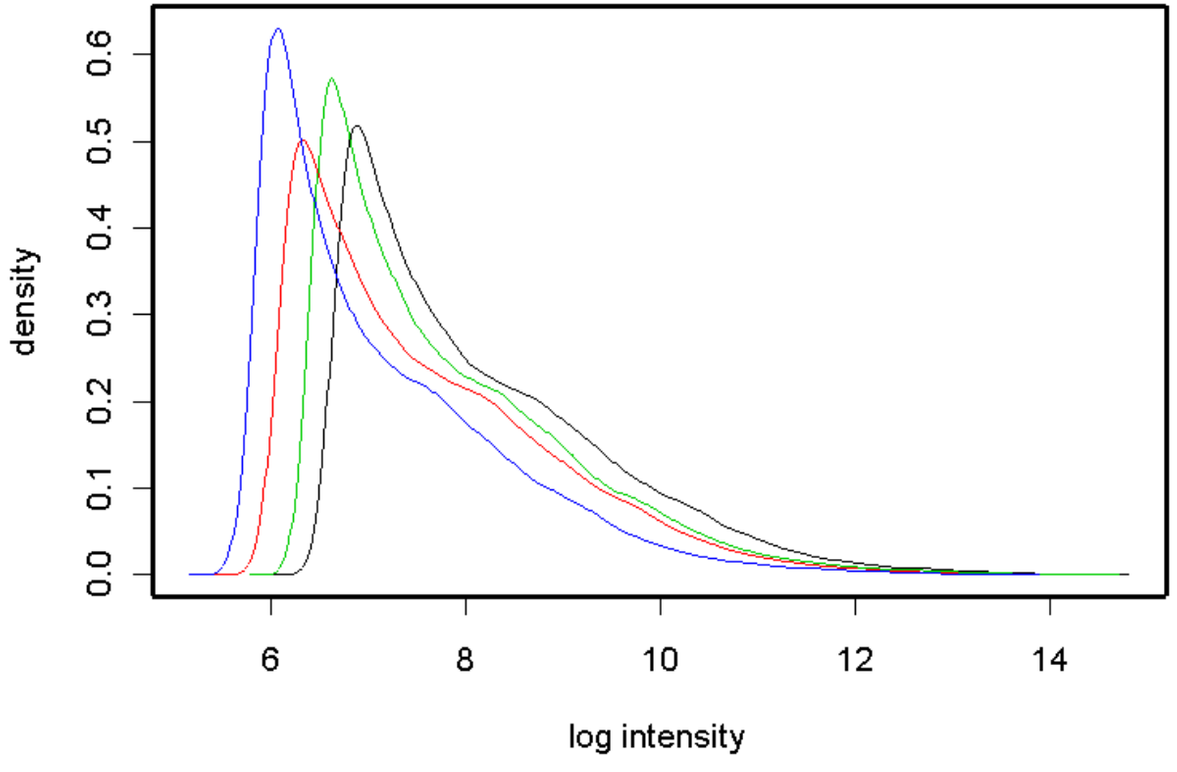
```
> boxplot(Dilution, col=1:4)
```



© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

The distribution of feature intensities

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



© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

Examining individual probesets

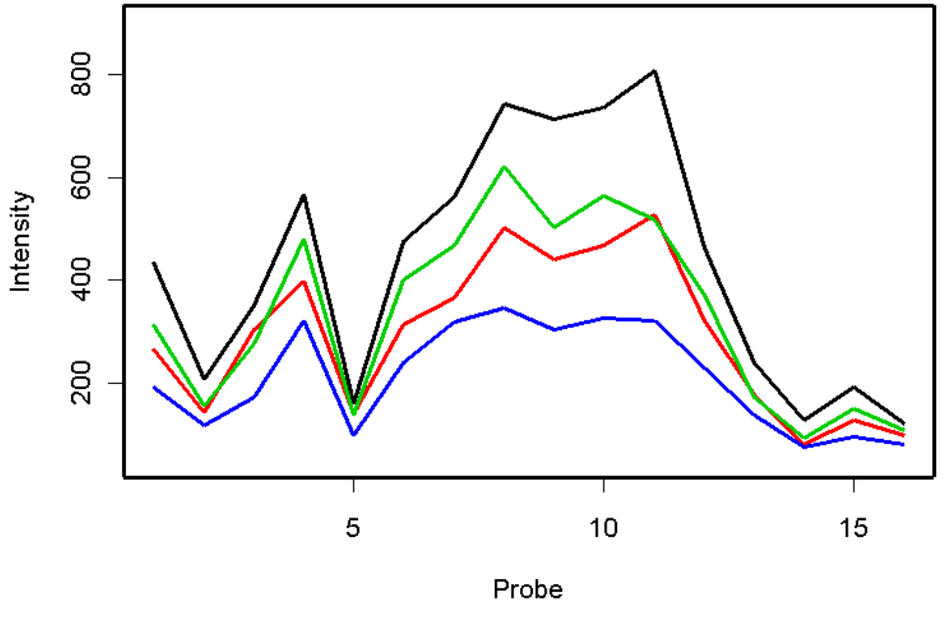
The `affy` package in BioConductor includes tools for extracting individual probe sets from a complete `AffyBatch` object. To get at the probe sets, however, you need to be able to refer to them by their “name”, which at present means their Affymetrix ID.

```
> 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...).

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

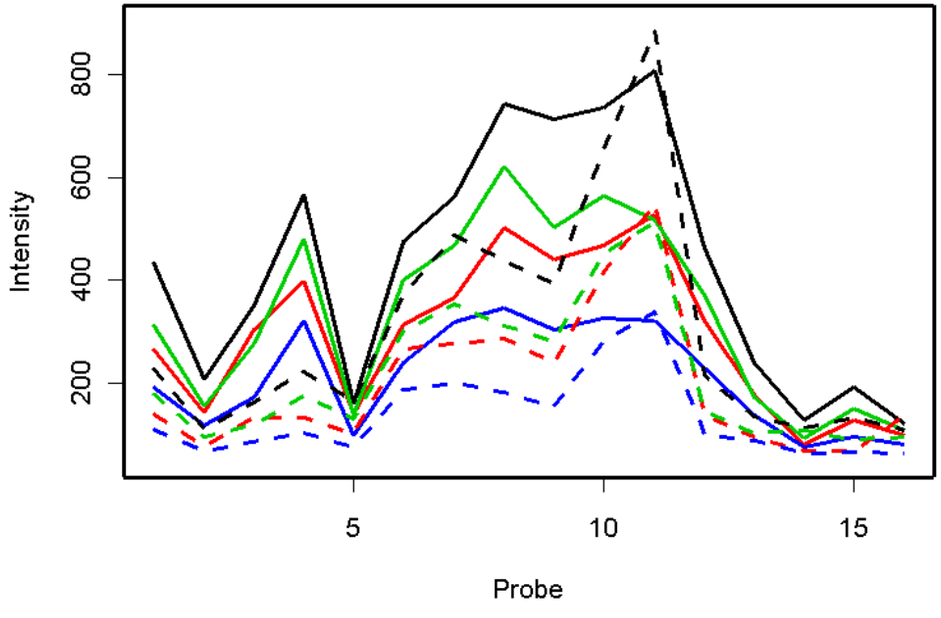
A probeset profile in four arrays




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

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

Examining individual probesets

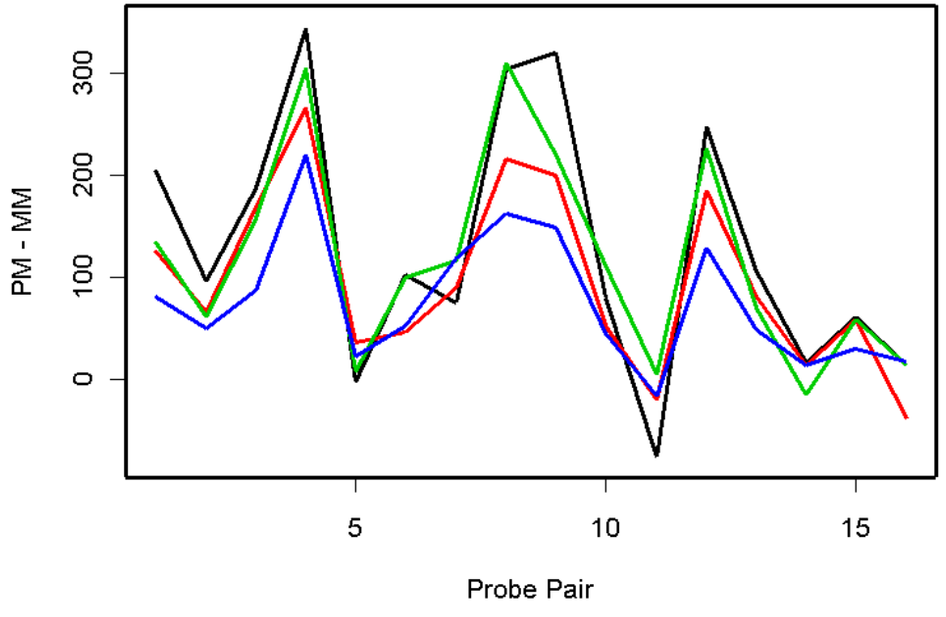


Let's add the mismatch probes to the graph:

```
> for (i in 1:4) lines(pm(ps)[,i], col=i, lty=2)
```

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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)
```

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

RNA degradation

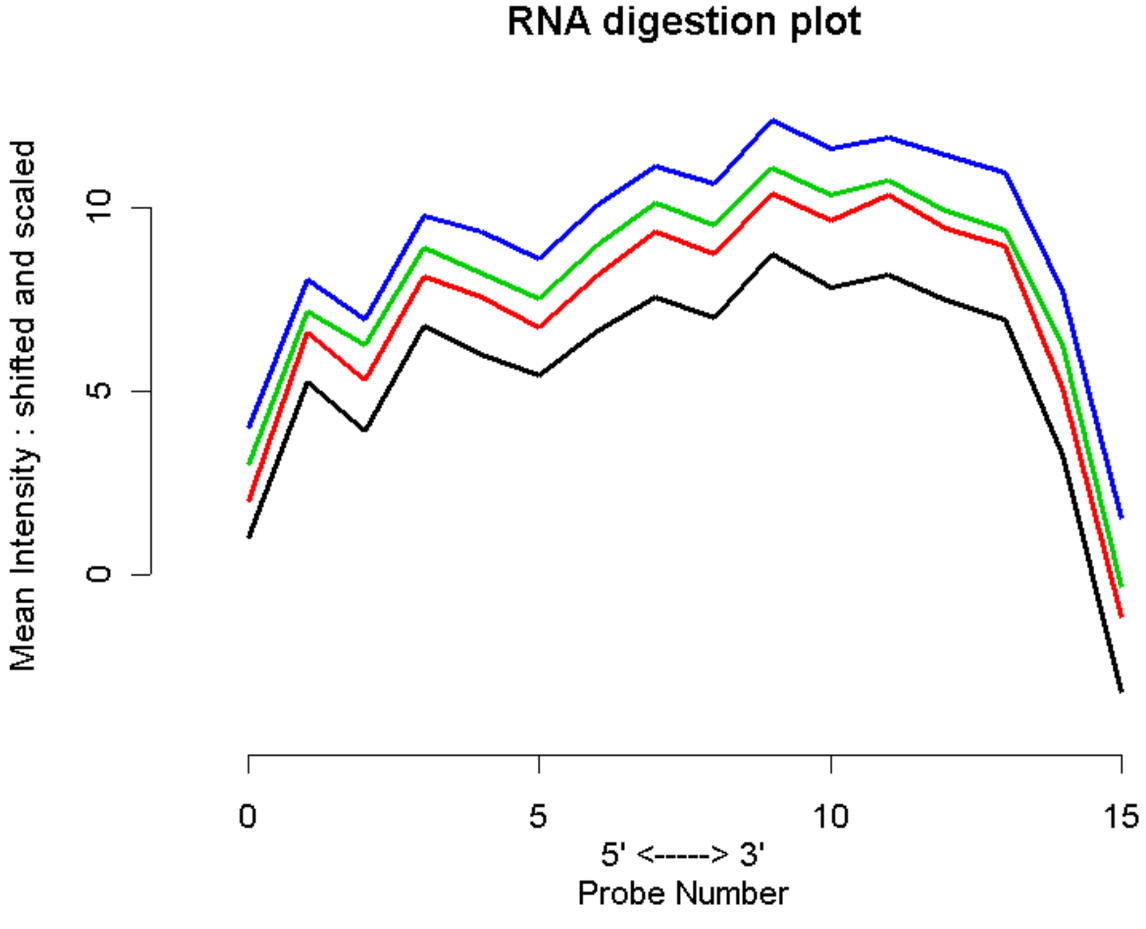
Individual (perfect match) probes in each probe set are ordered by location relative to the 5' end of the targeted mRNA molecule. We also know that RNA degradation typically starts at the 5' end, so we would expect probe intensities to be lower near the 5' end than near the 3' end.

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.

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

Visualizing RNA degradation

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



© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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

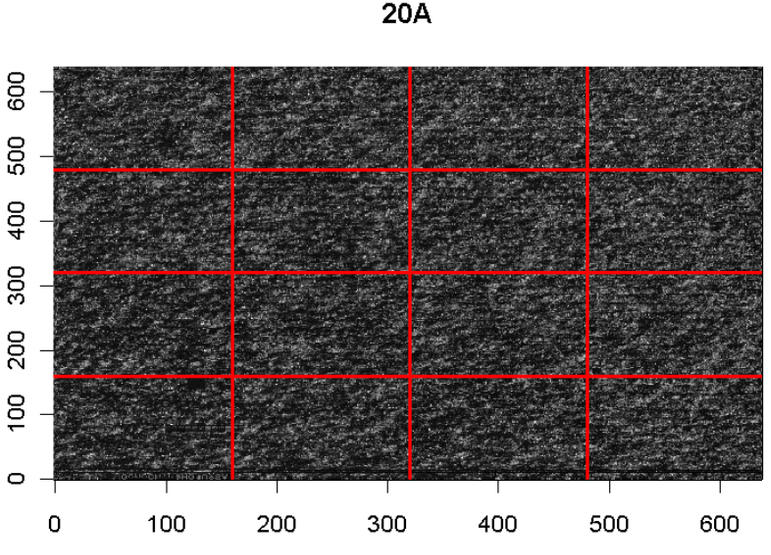
rma Use the algorithm from the current version of RMA

rma2 Use the algorithm from an older version of RMA

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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.



© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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

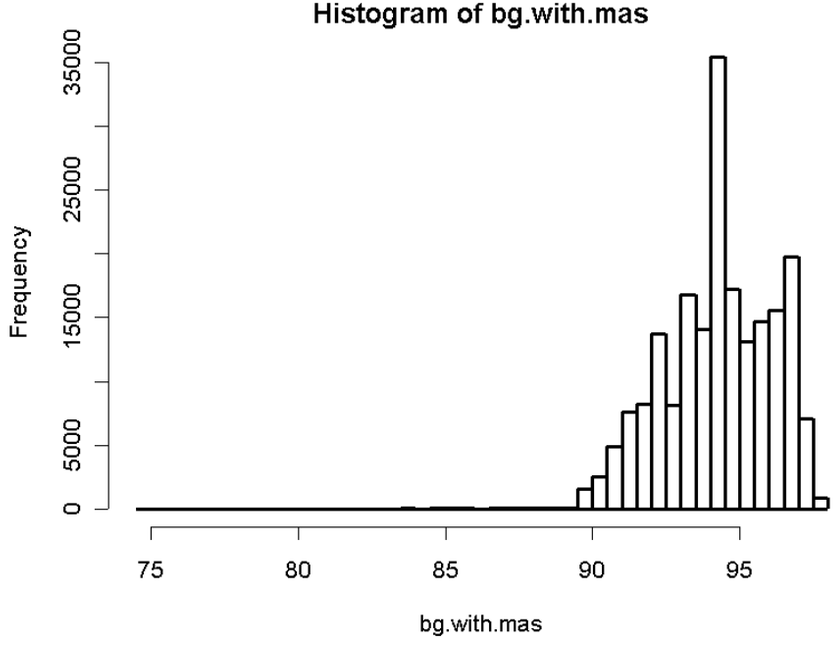
$$E(s|X = x) = a + b\frac{\phi(a/b) - \phi((x-a)/b)}{\Phi(a/b) + \Phi((x-a)/b) - 1}$$

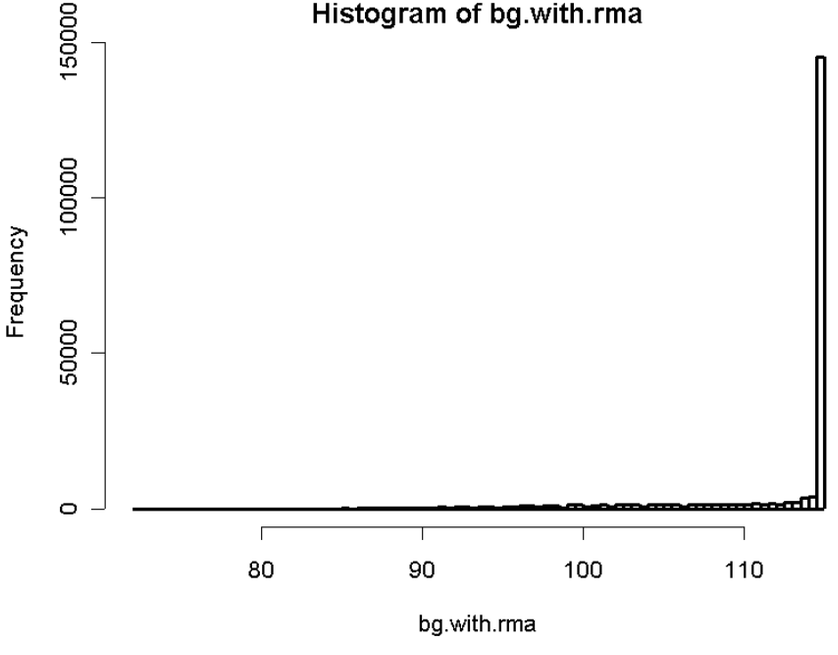
where $b = \sigma$ and $a = s - \mu - \lambda\sigma^2$.

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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)
```





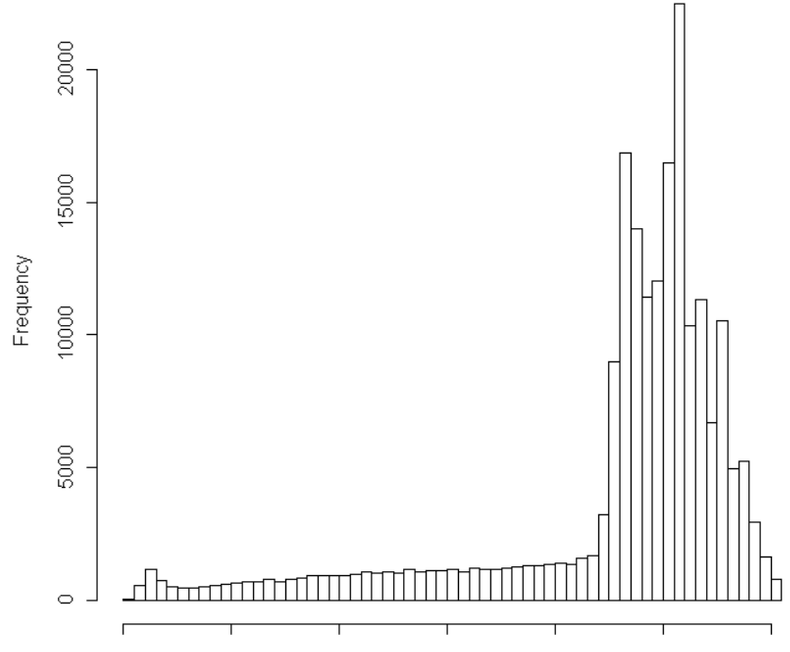
© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

```
> 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
```

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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.



© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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"
```

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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"
```

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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
```

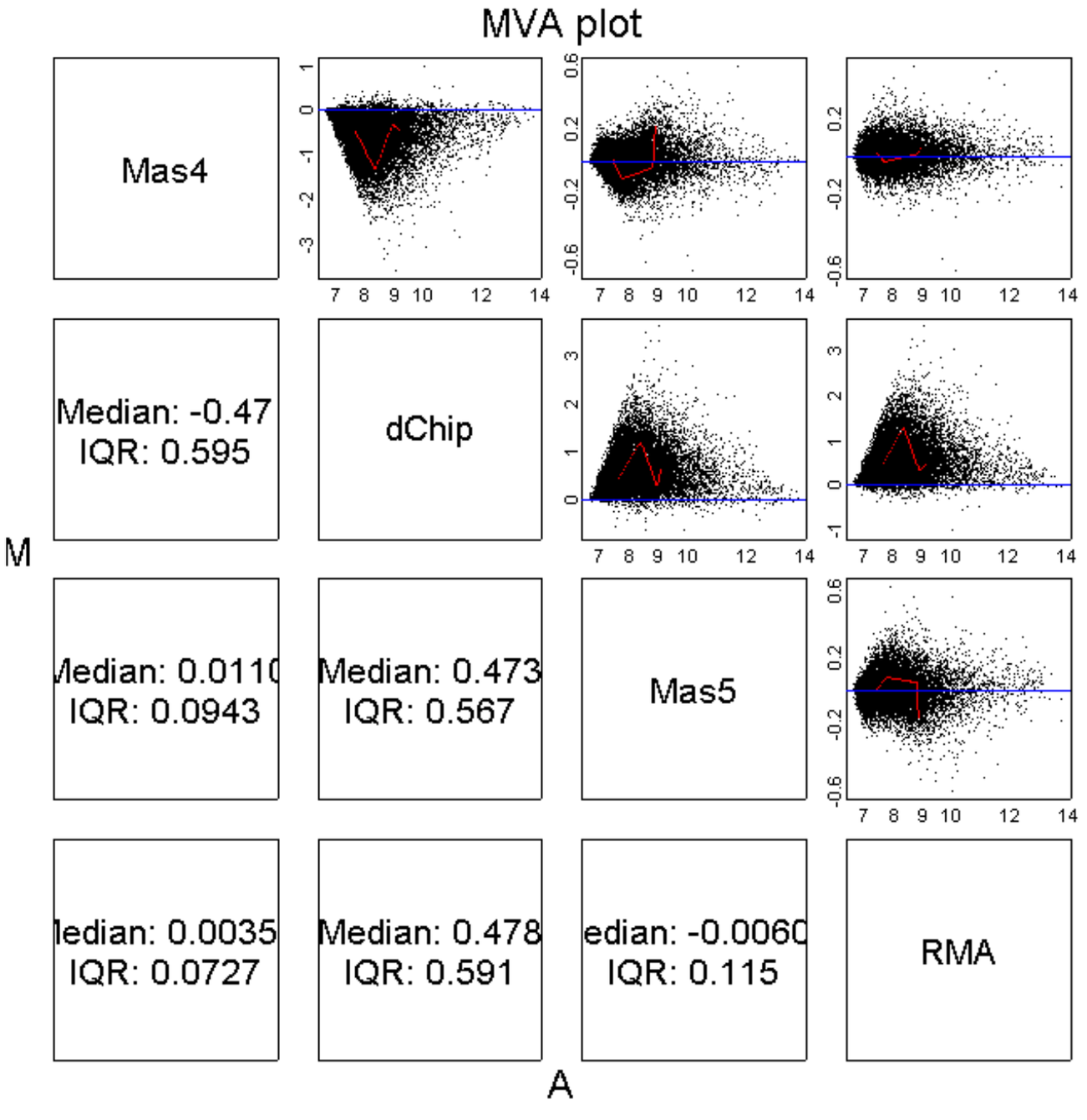
© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

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')
```

© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

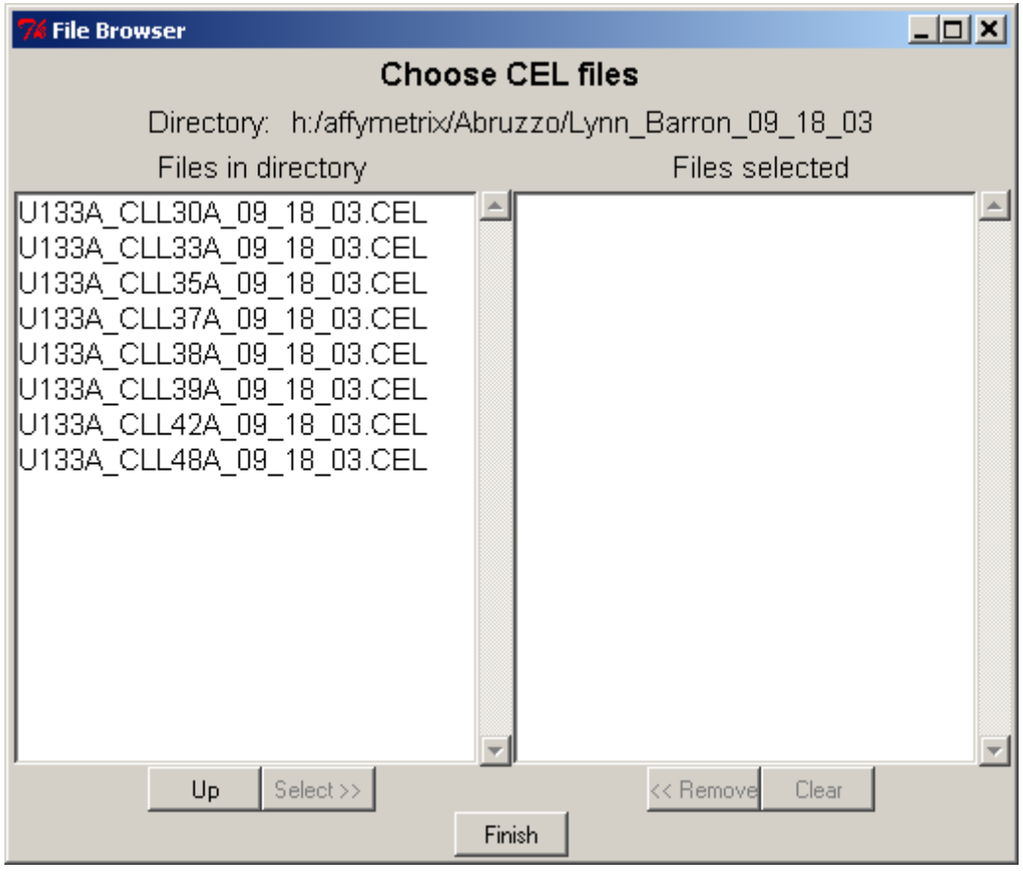

MVA plot
Mas4
Median: -0.47
IQR: 0.595
dChip
Median: 0.0110
IQR: 0.0943
Median: 0.473
IQR: 0.567
Mas5
Median: 0.0035
IQR: 0.0727
Median: 0.478
IQR: 0.591
Median: -0.0060
IQR: 0.115
RMA
M
A

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.

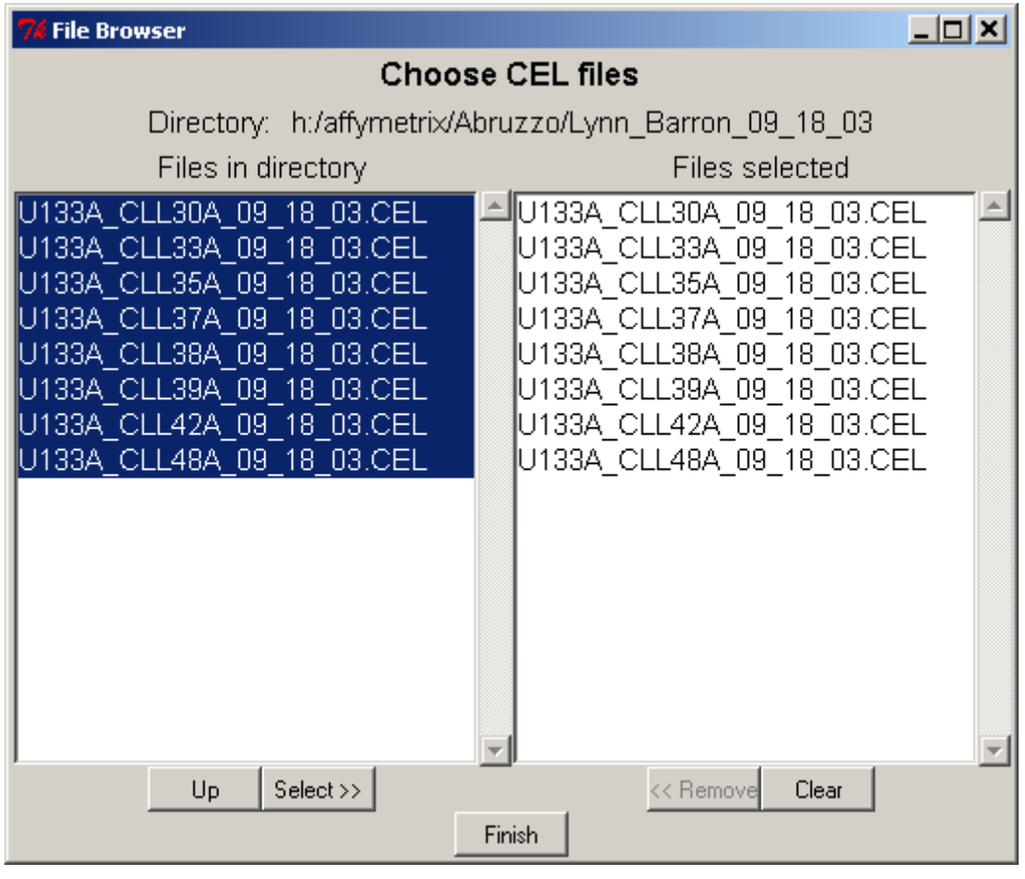
© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

Affy Widgets



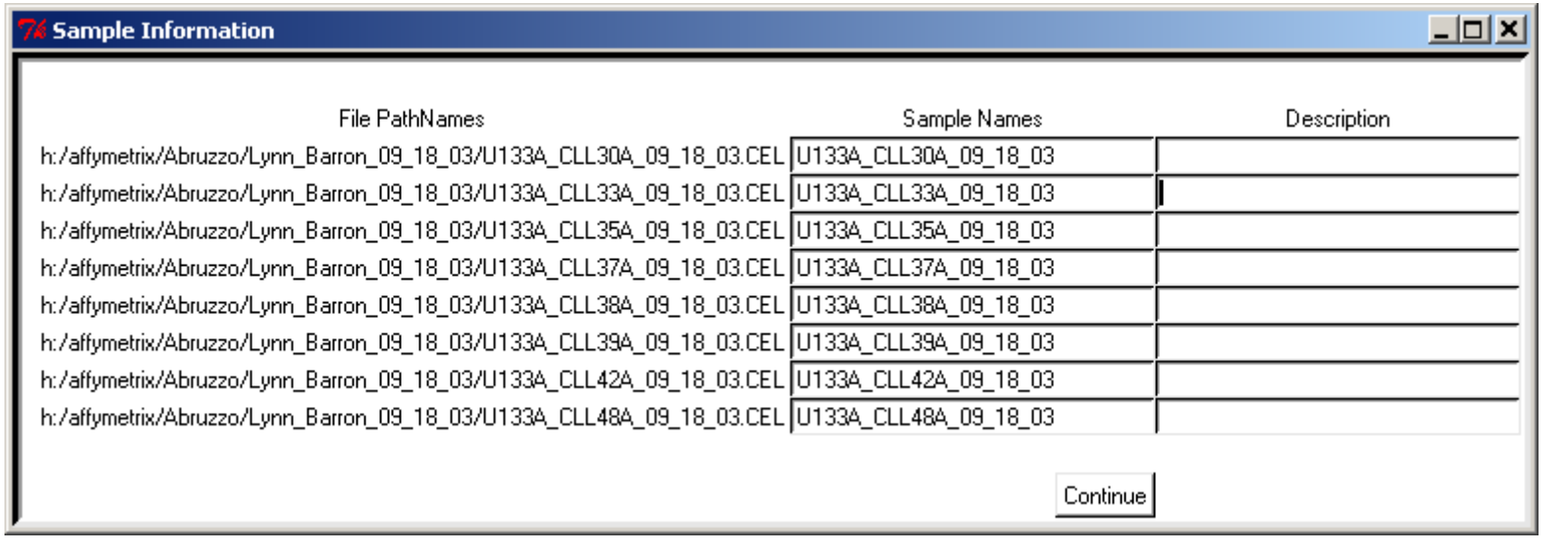
© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

Affy Widgets



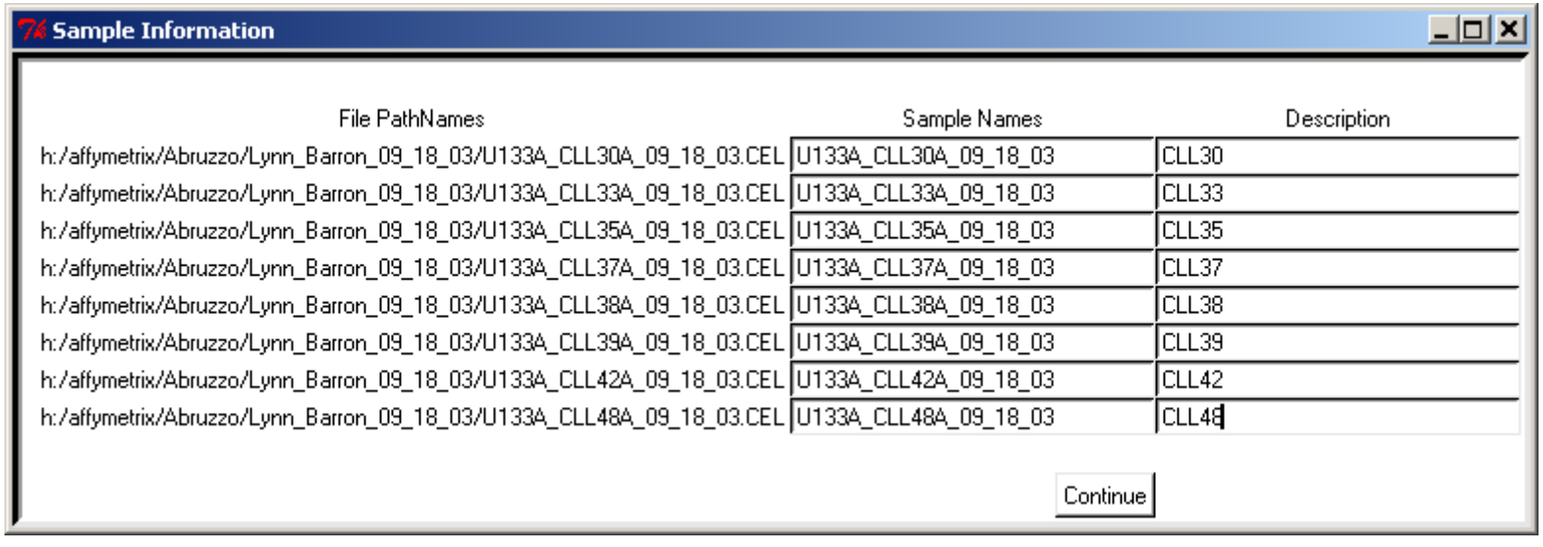
© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

Affy Widgets



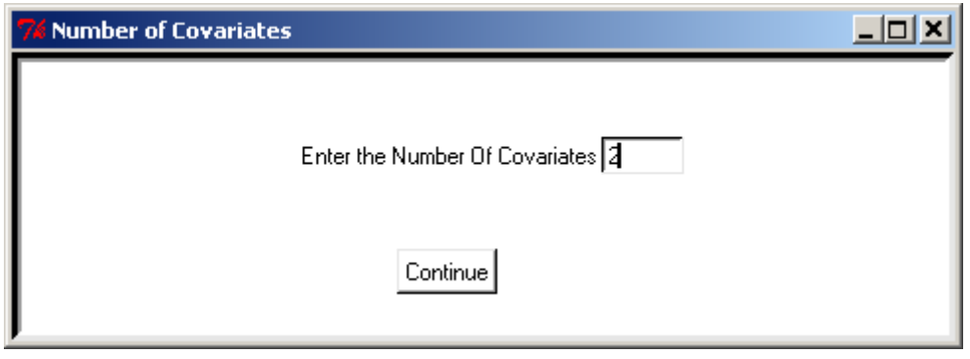
© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

Affy Widgets



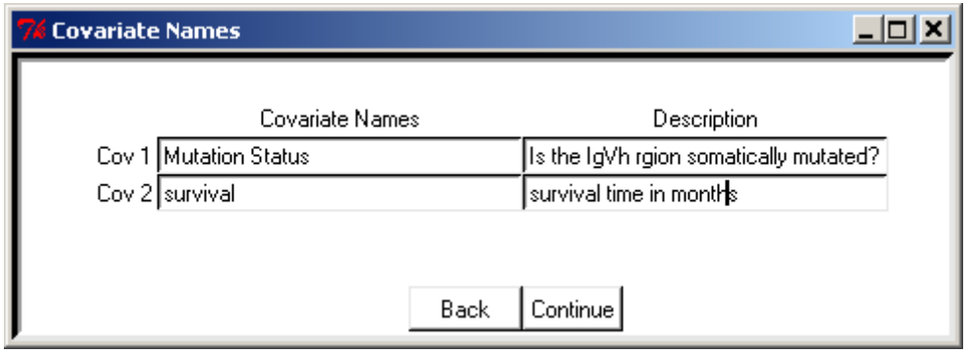
© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

Affy Widgets



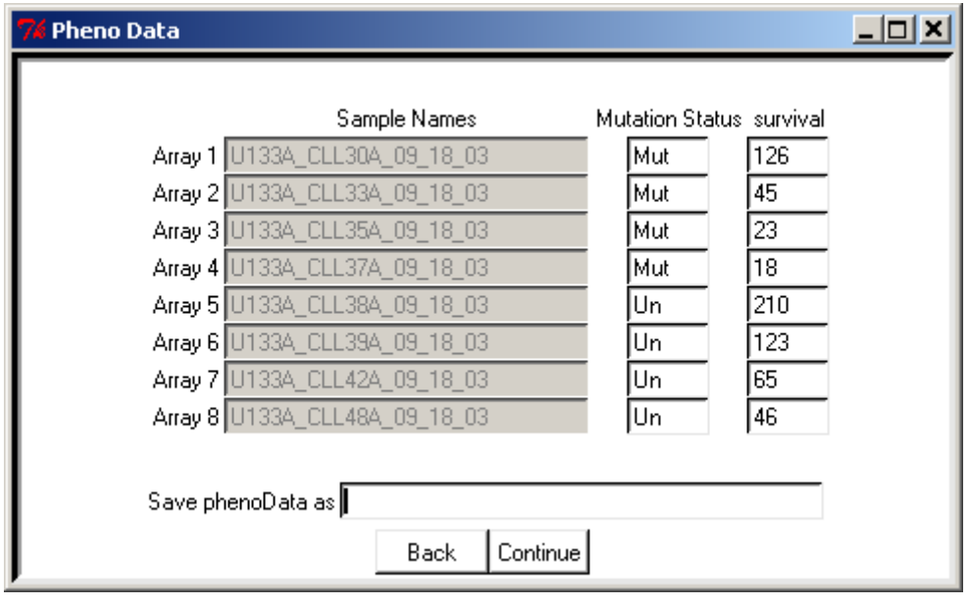
© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

Affy Widgets



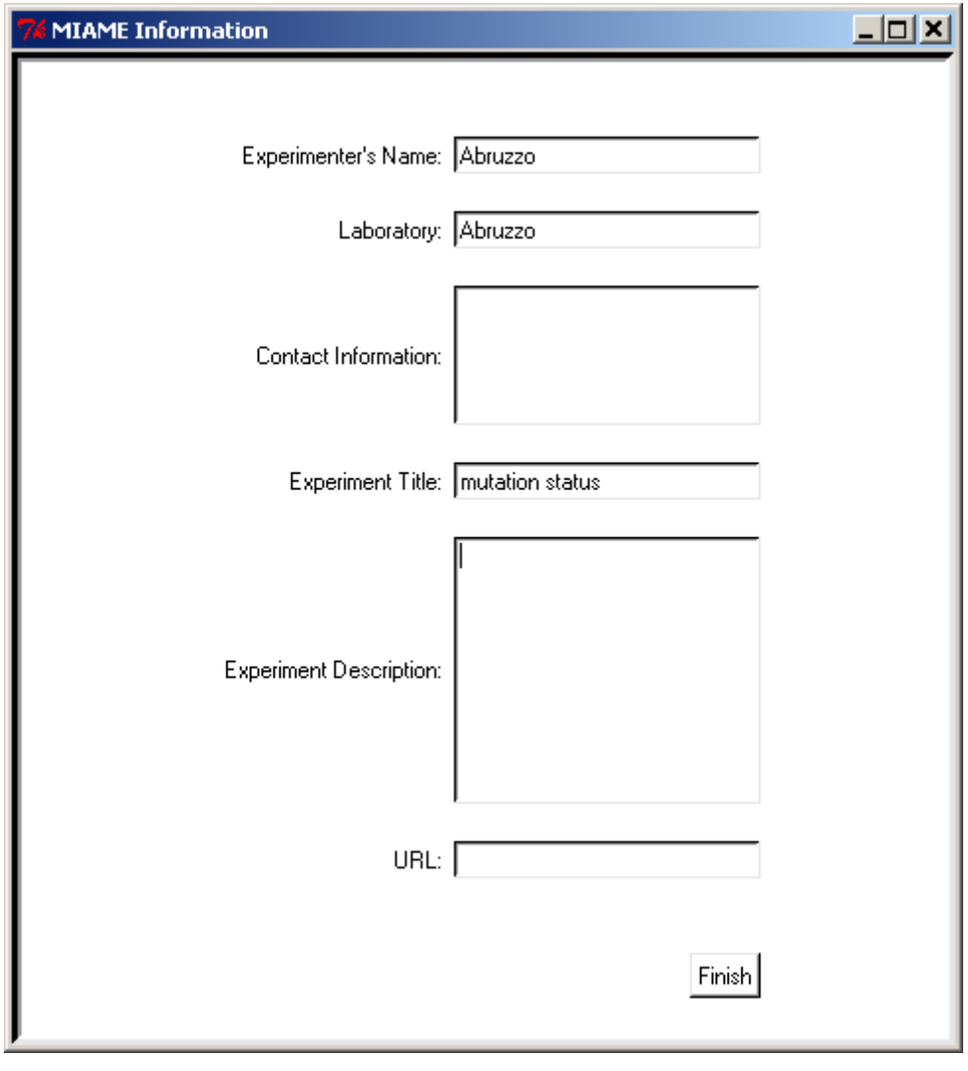
© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

Affy Widgets



© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly

Affy Widgets



© Copyright 2004–2006 Kevin R. Coombes and Keith A. Baggerly