GS01 0163
Analysis of Microarray Data

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

18 September 2007

Lecture 6: More on R and Affymetrix Arrays

- Beyond Matrices
- The Reproducibility Problem
- Installing TeX
- Reading Data Into R
- Bioconductor Packages
- Microarray Data Structures
- Affymetrix Data in BioConductor

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]

[1] "a" "b" "c"

> x <- letters[1]
> x <- letters[1:3]
> x <- matrix(letters[1:12], 3, 4)
```

Mixing Modes in Lists

In R, we cannot easily mix data of different modes in a vector or matrix:

```
> x <- c(1, "a")
> x

[1] "1" "a"
```

However, a list can have (named) 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...

```
> x[[1]]

[1] "Keith"

> x[["teacher"]]

[1] "Keith"
```

The double bracket notation can be cumbersome, so there is a shorthand notation with the dollar sign. Using names keeps the goals clear.

```
> x$teacher

[1] "Keith"
```

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. Before looking at that, however, we are going to revisit the notion of reproducibility of our analyses.

The Reproducibility Problem

1. Researcher contacts analyst: "I just read this interesting paper. Can you perform the same analysis on my data?"

2. Analyst reads paper. Finds algorithms described by biologists in English sentences that occupy minimal amount of space in the methods section.

3. Analyst gets public data from the paper. Takes wild guesses at actual algorithms and parameters. Is unable to reproduce reported results.

4. Analyst considers switching to career like bicycle repair, where reproducibility is less of an issue.

Alternate Forms of the Same Problem

1. Remember that microarray analysis you did six months ago? We ran a few more arrays. Can you add them to the project and repeat the same analysis?

2. The statistical analyst who looked at the data I generated previously is no longer available. Can you get someone else to analyze my new data set using the same methods (and thus producing a report I can expect to understand)?

3. Please write/edit the methods sections for the abstract/paper/grant proposal I am submitting based on the analysis you did several months ago.

The Code/Documentation Mismatch

Most of our analyses are performed using R. We can usually find an R workspace in a directory containing the raw data, the report, and one or more R scripts.

There is no guarantee that the objects in the R workspace were actually produced by those R scripts. Nor that the report matches the code. Nor the R objects.

Because R is interactive, unknown commands could have been typed at the command line, or the commands in the script could have been cut-n-pasted in a different order.

This problem is even worse if the software used for the analysis has a fancy modern GUI. It is impossible to document how you used the GUI in such a way that someone else could produce the exact same results—on the same data—six months later.

The Solution: Sweave

$$\text{Sweave} = \text{R} + \text{LaTeX}.$$

This talk was prepared using Sweave. So was this standard report.

If you already know both R and LaTeX, then the ten-second version of this talk takes only two slides:

1. Prepare a LaTeX document. Give it an "Rnw" extension instead of "tex". Say it is called "myfile.Rnw"

2. Insert an R code chunk starting with <<>>=

3. Terminate the R code chunk with an "at" sign (@) followed by a space.

Using Sweave

To produce the final document

1. In an R session, issue the command

   ```
   Sweave("myfile.Rnw")
   ```

   This executes the R code, inserts input commands and output computations and figures into a LaTeX file called "myfile.tex".

2. In the UNIX or DOS window (or using your favorite graphical interface), issue the command

   ```
   pdflatex myfile
   ```

   This produces a PDF file that you can use as you wish.

Viewing The Results

Here is a simple example, showing how the R input commands can generate output that is automatically included in the LaTeX output of Sweave.

```
> x <- rnorm(30)
> y <- rnorm(30)
> mean(x)

[1] 0.2279967

> cor(x, y)

[1] 0.3408799
```

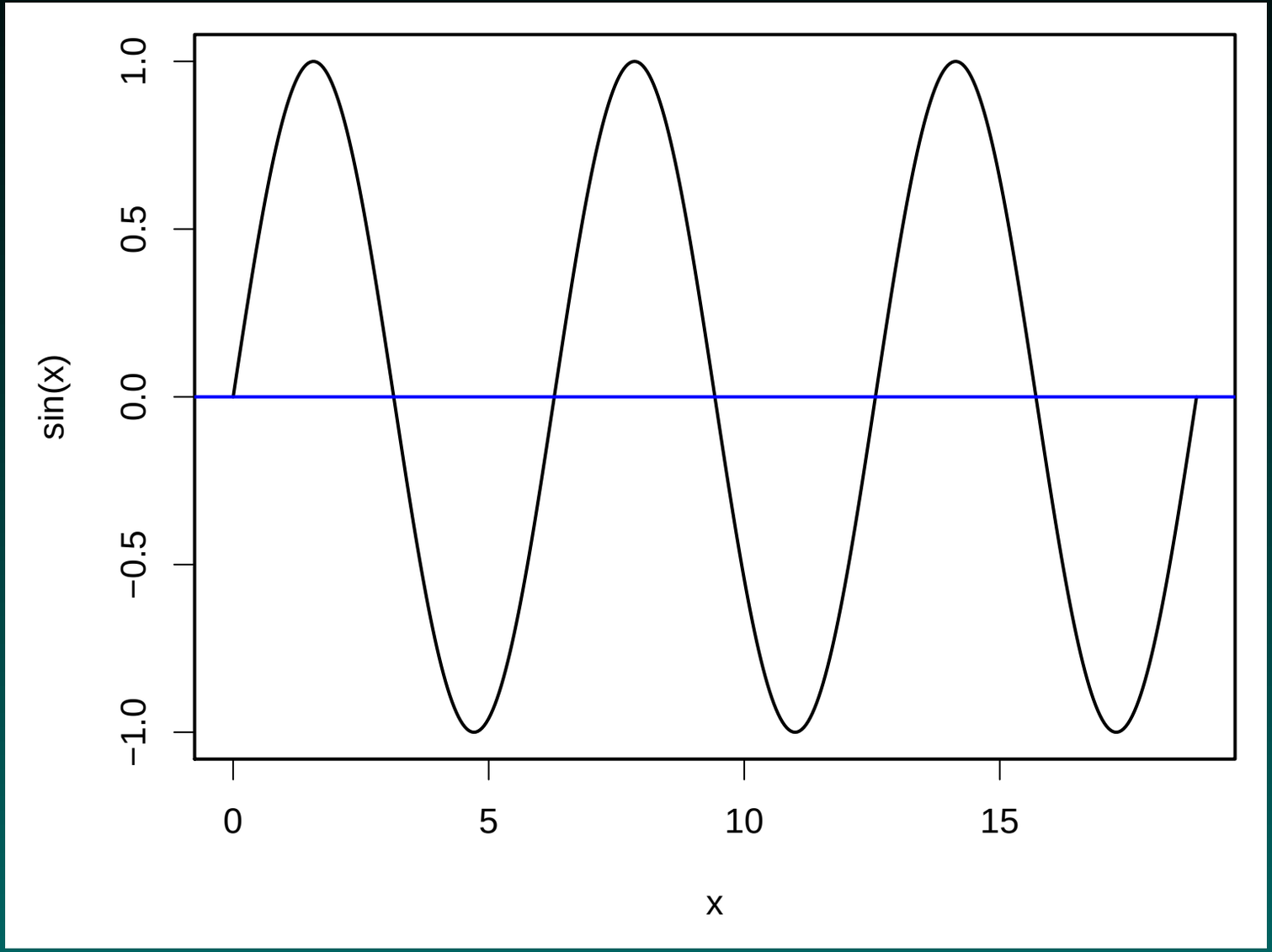
A Figure

Next, we are going to insert a figure. First, we can look at the R commands that are used to produce the figure.

```
> x <- seq(0, 6 * pi, length = 450)
> par(bg = "white", lwd = 2, cex = 1.3, mai = c(1.2,
+     1.2, 0.2, 0.2))
> plot(x, sin(x), type = "l")
> abline(h = 0, col = "blue")
```

On the next slide, we can look at the actual figure. (Part of the point of this example is to illustrate that you can separate the input from the output. You can even completely hide the input in the source file and just include the output in the report.)

Sine Curve



A Table

```
> library(xtable)
> x <- data.frame(matrix(rnorm(12), nrow = 3,
+     ncol = 4))
> dimnames(x) <- list(c("A", "B", "C"), c("C1",
+     "C2", "C3", "C4"))
> tab <- xtable(x, digits = c(0, 3, 3, 3, 3))
> tab
```

	C1	C2	C3	C4
A	1.052	−0.720	0.015	−0.595
B	0.159	−1.059	0.321	1.753
C	−0.539	0.530	−0.734	0.119

A Table, Repeated

Again, we want to point out that you can show the results—including tables—without showing the commands that generate them.

	C1	C2	C3	C4
A	1.052	−0.720	0.015	−0.595
B	0.159	−1.059	0.321	1.753
C	−0.539	0.530	−0.734	0.119

Sweave Details

Next, we are going to leave this PDF presentation for a while to look at a very similar document in a more typical print format, and then look at the Rnw file that was used to produce it.

Next, we will talk about how to install LaTeX so that you can start using these ideas yourself.

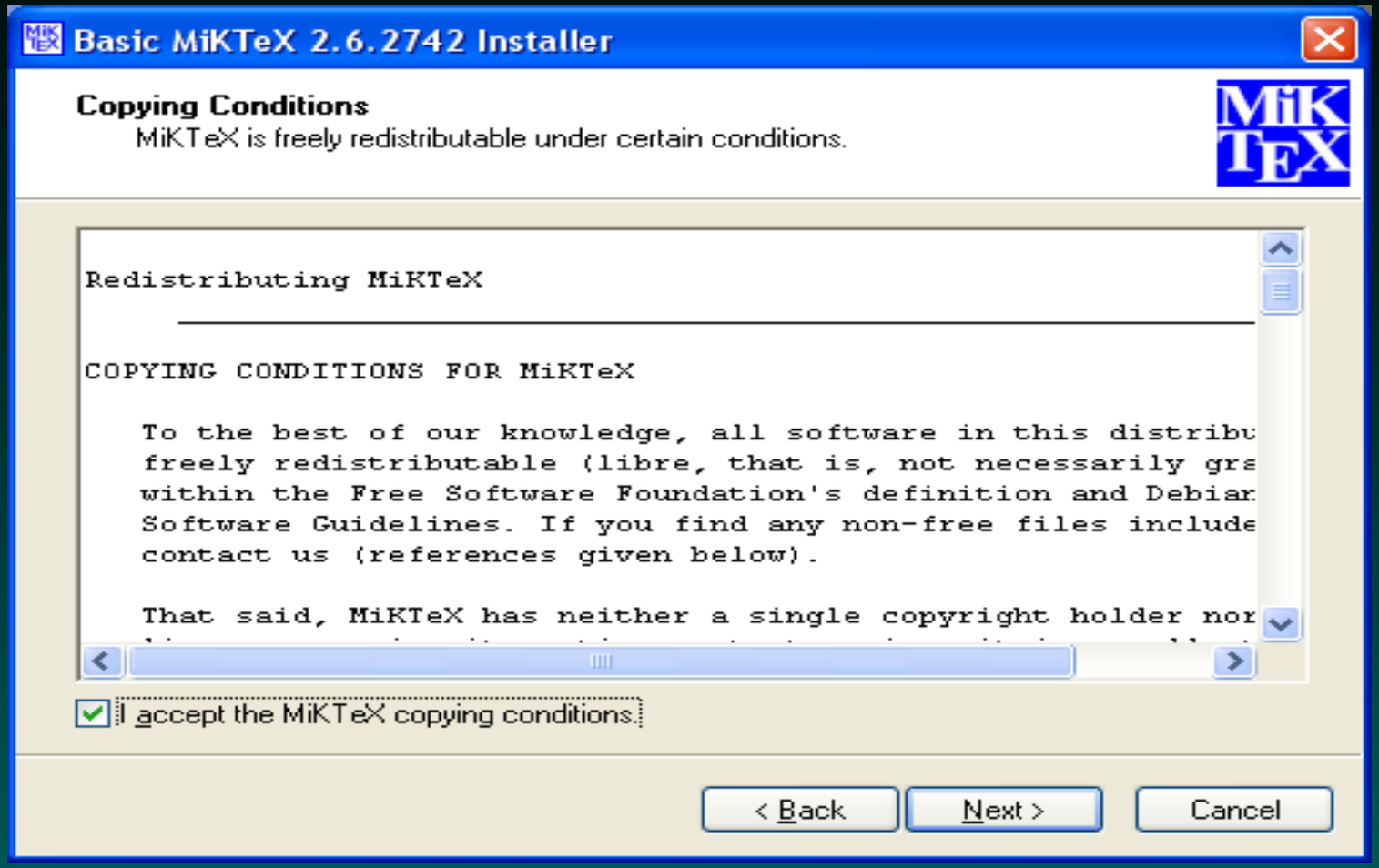
Finally, we will look at methods for reading microarray data into R so we can start analyzing it.

Installing TeX

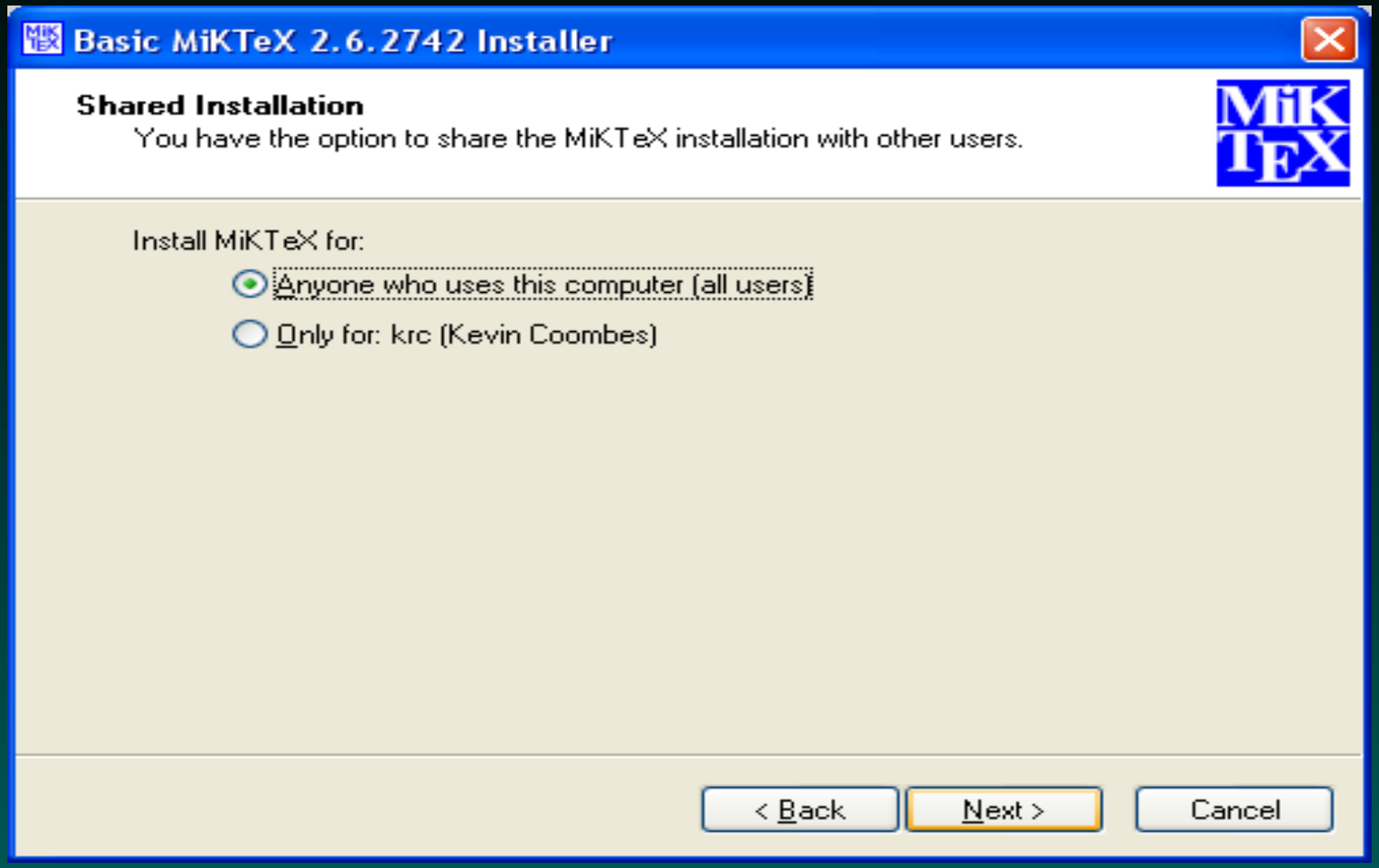
The standard version of TeX or LaTeX for Windows is MiKTeX, which is available at http://www.miktex.org. The current version is 2.6. You should download the "Basic MiKTeX 2.6" installer. The file you get when you do this is called `basic-miktex-2.6.2742.exe`. Keep track of where you save this file (your desktop will work just fine) and then double-click on the resulting icon to start the installation.

According to the Comprehensive TeX Archive Network (CTAN, http://www.ctan.org), the standard version of TeX or LaTeX for Macintosh computers is gwTeX, and they provide a link to the download and installation page. Since I have never installed this version, you will have to figure out how to get it yourself....

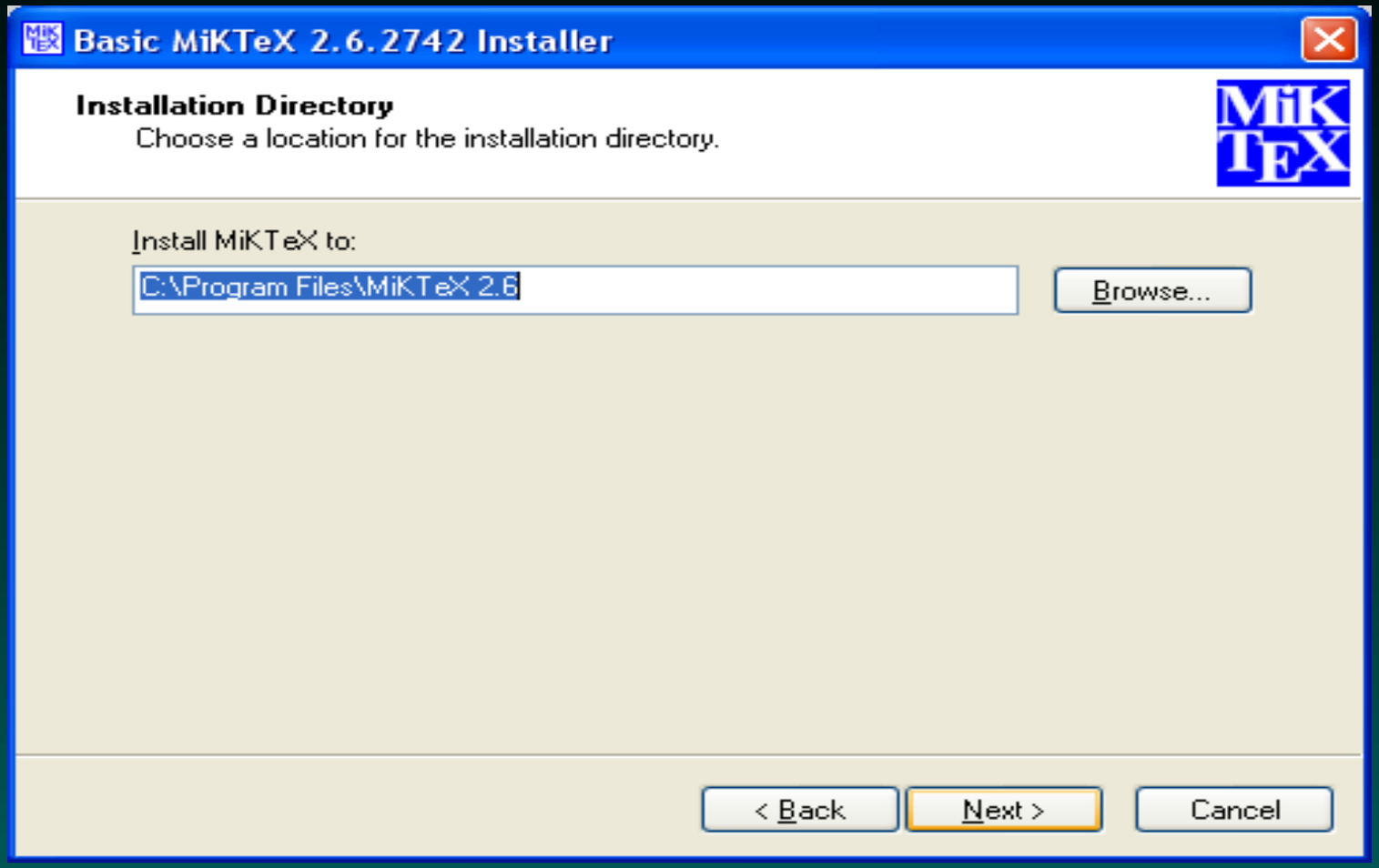
The MiKTeX Installer



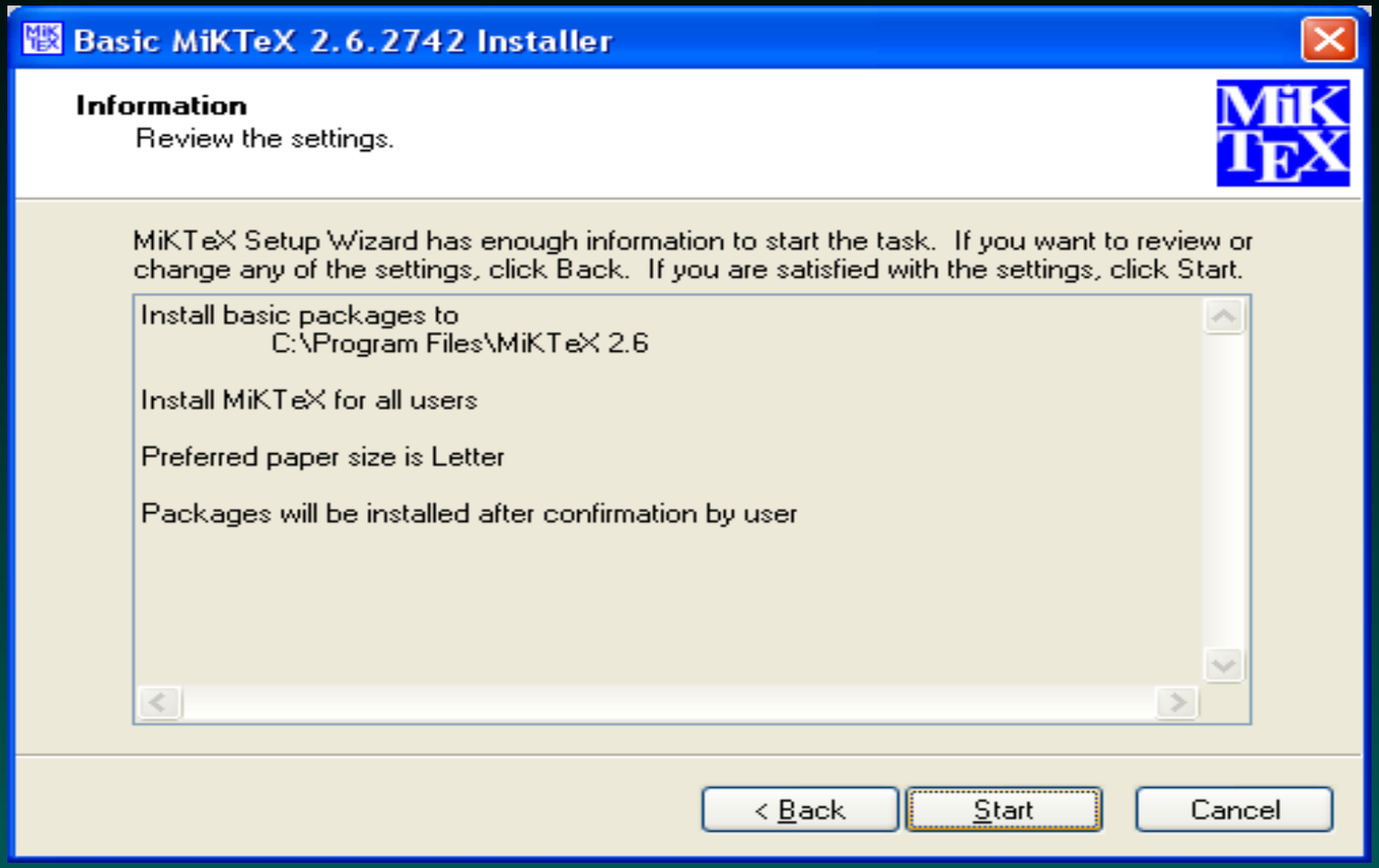
The MiKTeX Installer



The MiKTeX Installer



The MiKTeX Installer



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 often 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.

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 = "\"", 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!

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 '.' 0.1 ' ' 1
```

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 '.' 0.1 ' ' 1
```

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 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

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.

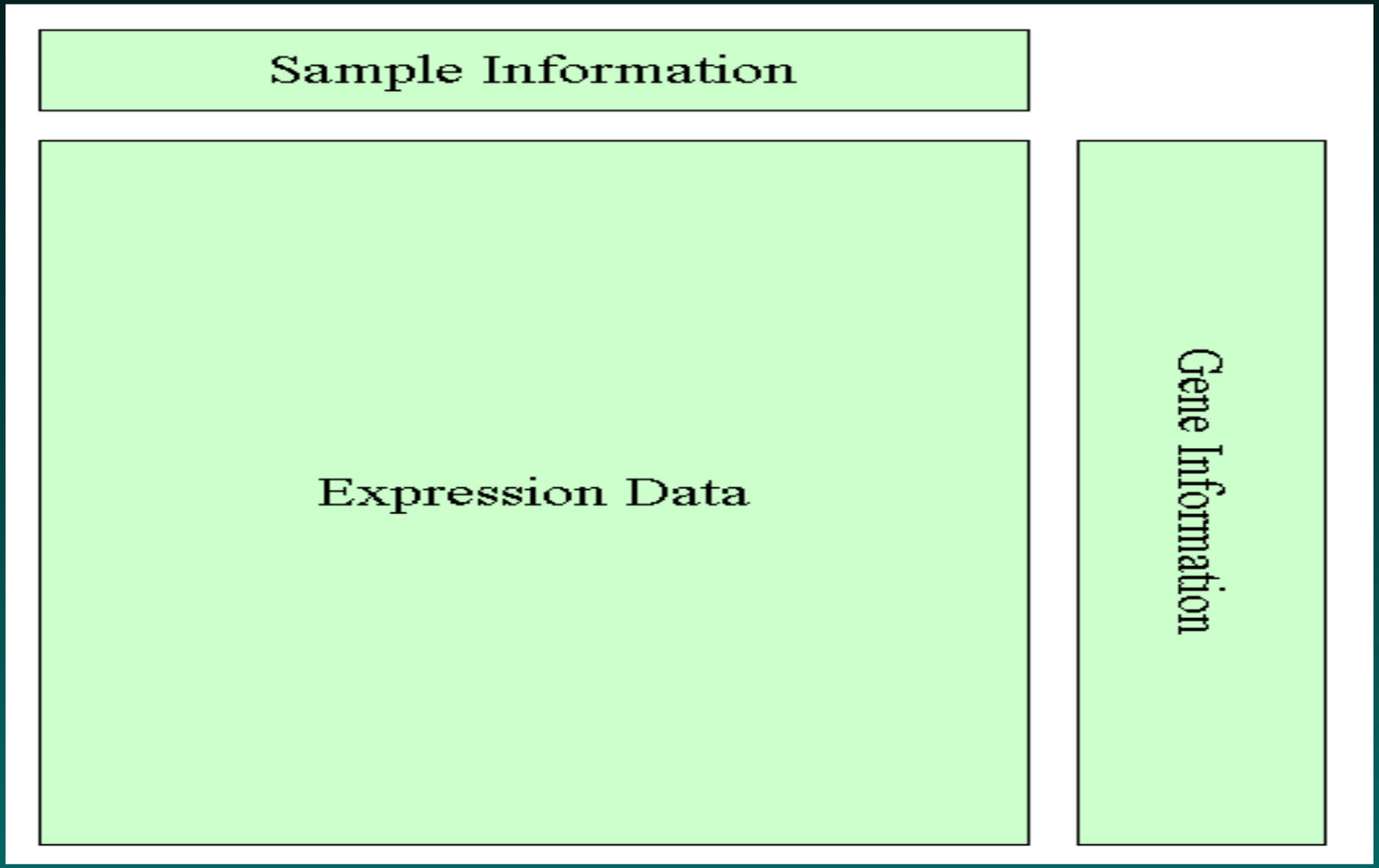
tkWidgets : R based Tk widgets

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?



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.

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.

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.

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('cancer', 'healthy'), 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'))
```

```
> 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
```

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
```

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 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.

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
```

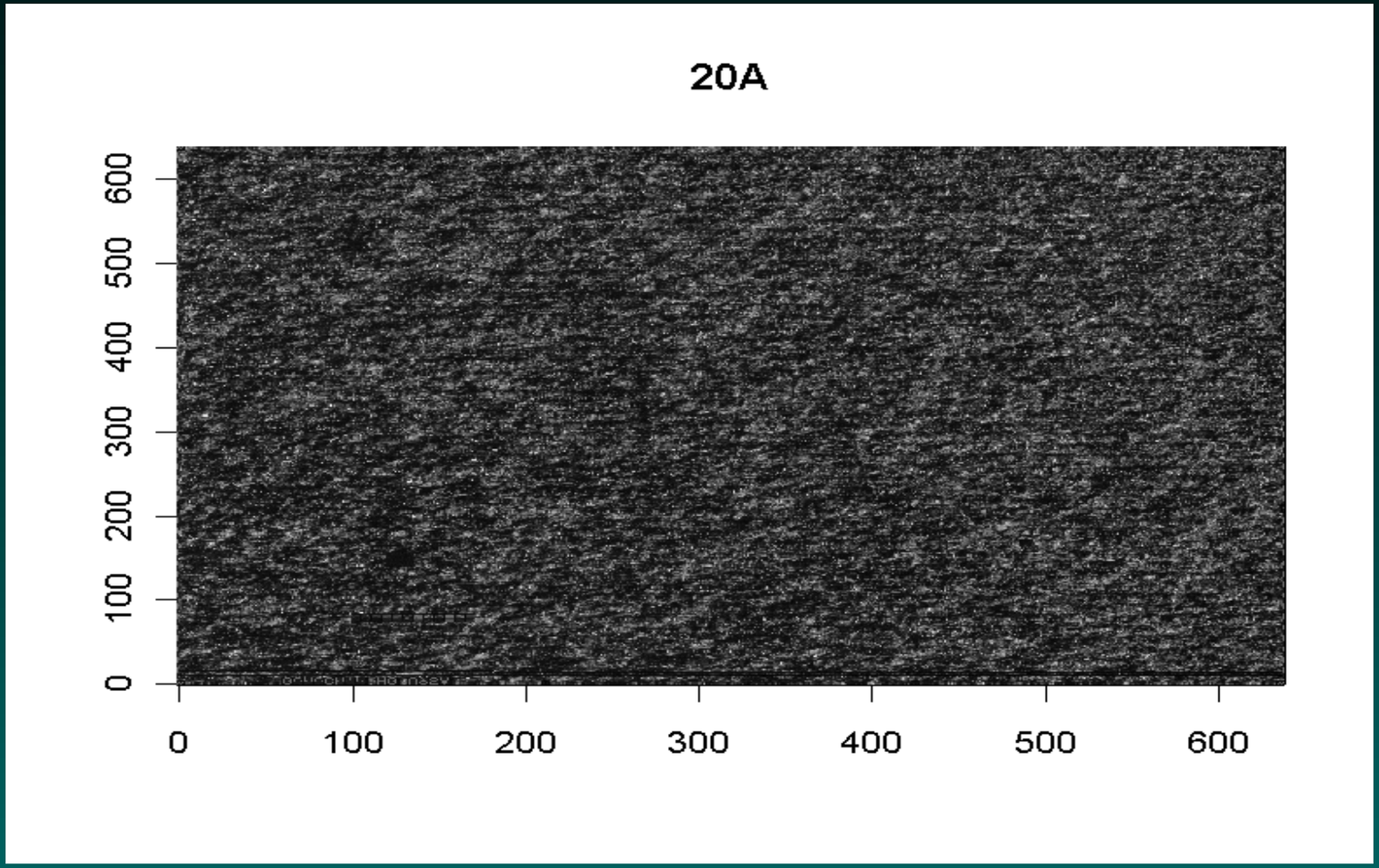
Looking at the experimental design

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 in micro
      sn19: amount of central nervous system RNA hybidized to
      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
```

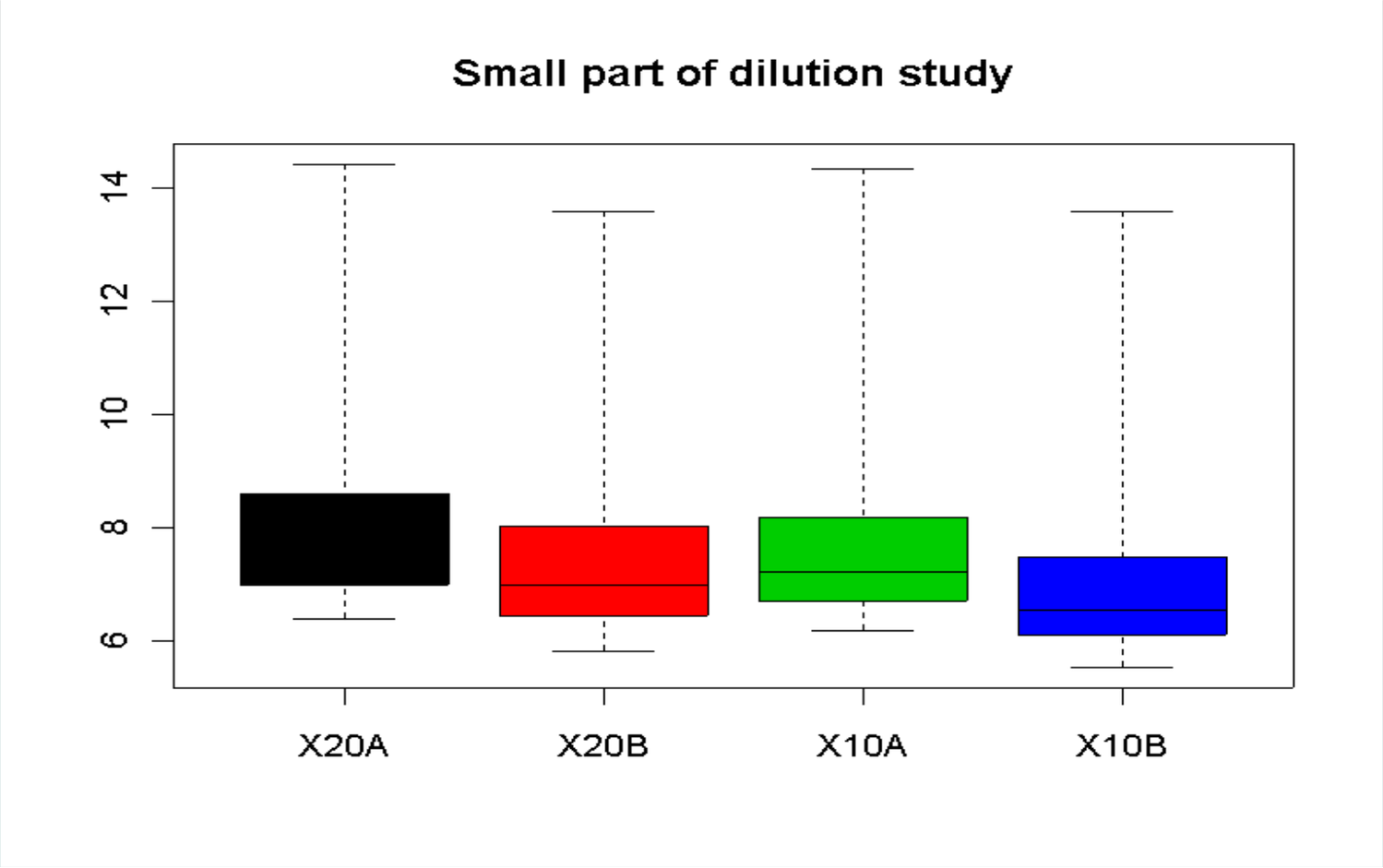
A first look at an array

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



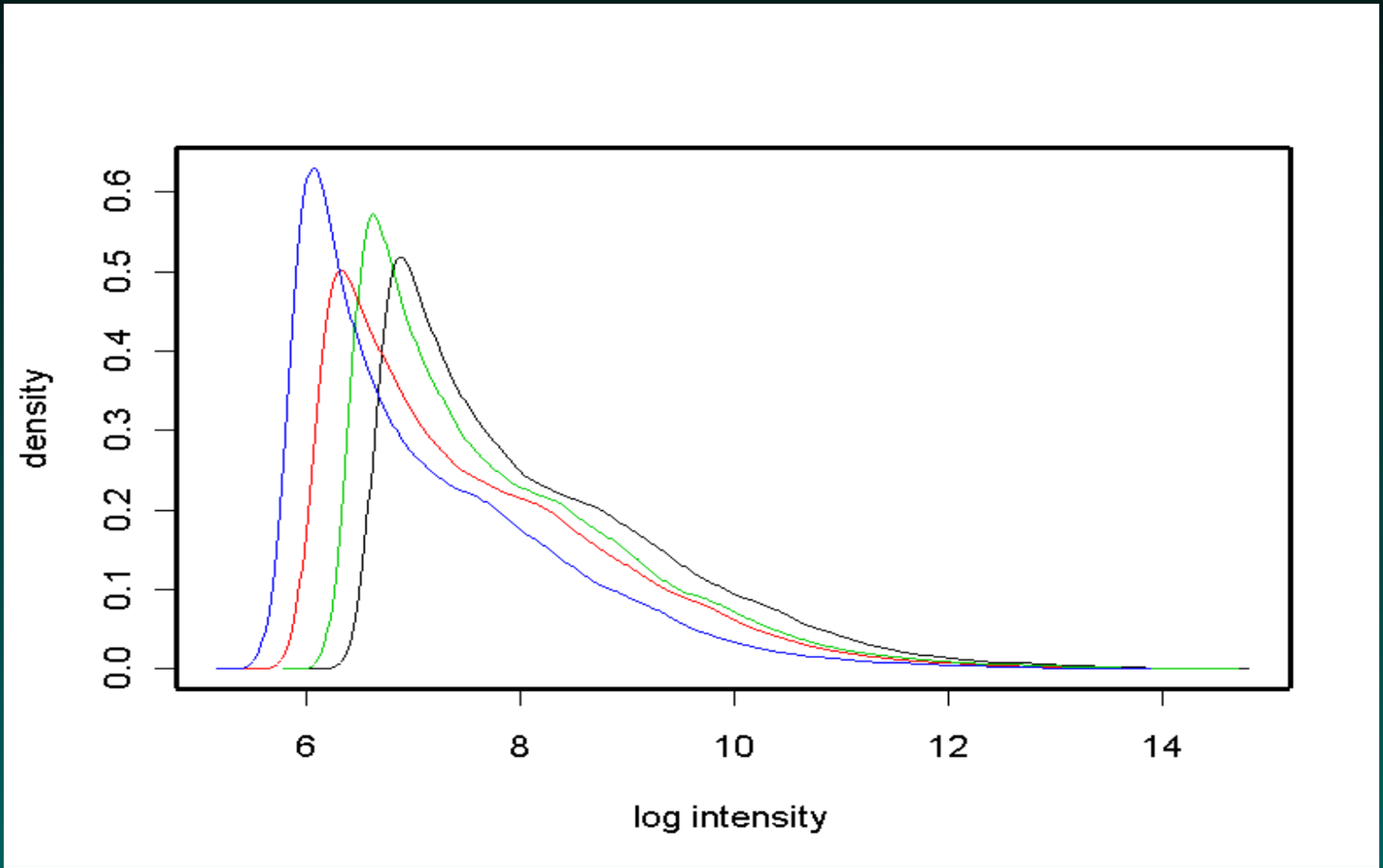
A summary view of four images

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



The distribution of feature intensities

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



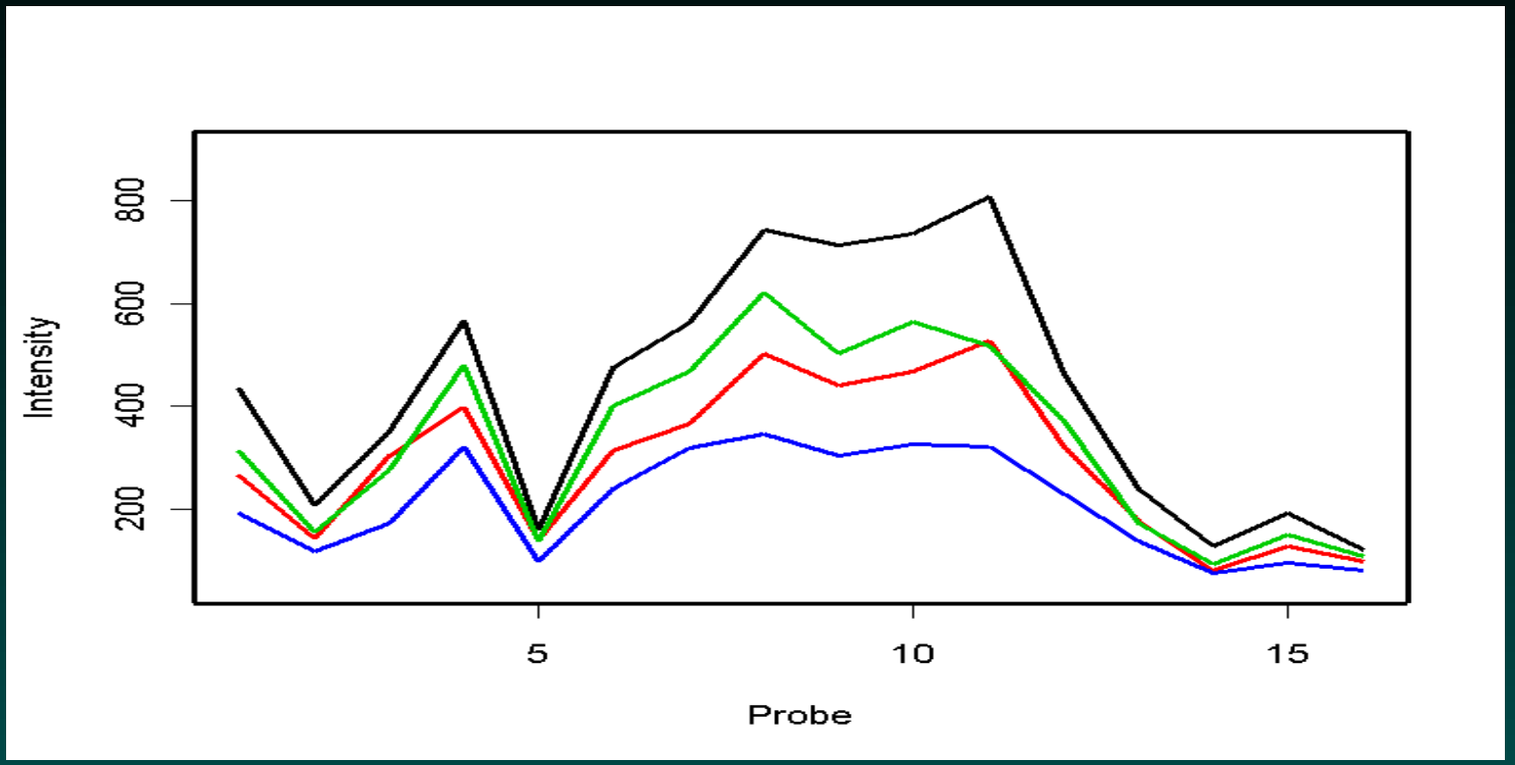
Examining individual probesets

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]]
```

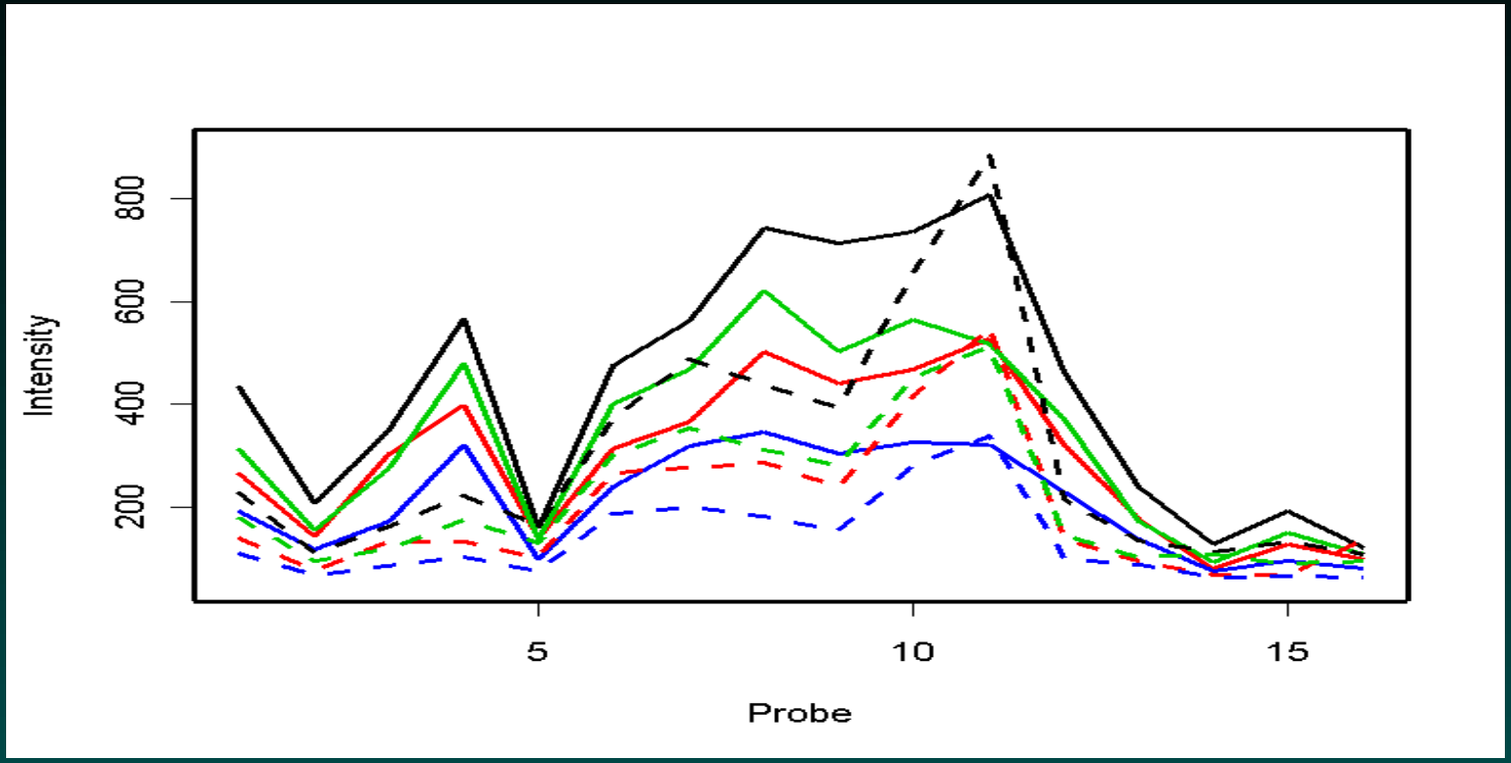
The probeset function returns a list of probe sets; the mysterious stuff with the brackets takes the first element from the list (which only had one...).

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)
```

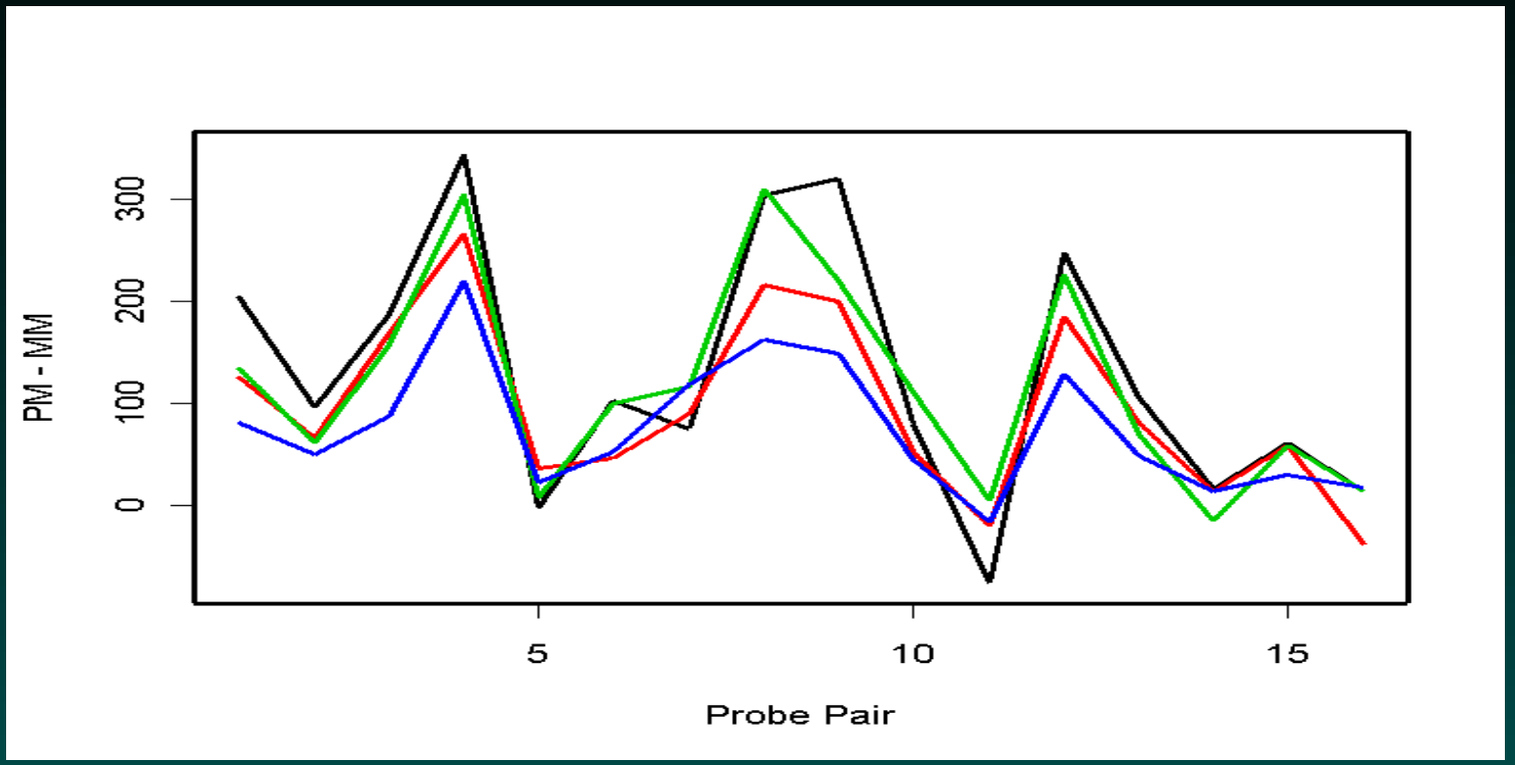
Examining individual probesets



Let's add the mismatch probes to the graph:

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

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)
```