# Introduction to R for Biologists

*David Wheeler, CCR, NCI*

*1/29/2015*

## Contents



---

R is a statistical program that can be used interactively to explore data or used as a programming language in which to implement complex data analyses. It is extensible via *libraries* which can add both new functions for analysis and new data for reference. For biologists, the Bioconductor framework provides the foundation for almost a thousand libraries that allow biologists to analyze high throughput data of many types.

**The Central Repository for R and its Numerous Libraries:**

`http://cran.r-project.org/`

**A Quick Reference PDF**

`http://cran.r-project.org/doc/contrib/Short-refcard.pdf`

## The Virtues of R

- Elegant syntax for both interactive use and programming
- Vectorized methods that minimize the need for user-specified loops
- Simple methods to produce a wide array of stunning graphics
- Easily extensible with hundreds of libraries, e.g. *Bioconductor*
- Extensive help system

## The R Environment

### Functions Used

- `save.image()` to save your current *Environment* containing all your data into the file *.RData*

R is started within a terminal by typing `R`. This starts the program, causes several basic packages to be loaded, and puts you into the global R *Environment*.

An R *Environment* is a set of variables (also called *objects* in R parlance) and their values. When you start R, you are operating within the global *Environment*. You can think of your global R *Environment* as the equivalent of a virtual disk. You can list the variables in it and create new variables but when you leave R, unless you save the *Environment*, you will lose everything. Fortunately, you can save the *Environment* at any time during an R session using `save.image()`. In addition, R offers to save the *Environment* for you whenever you leave. On your disk, the copy of the *Environment* saved by R when you reply "yes" to the prompt or use `save.image()` appears as the compressed image, *.RData*. This *.RData* image is loaded automatically the next time you start R. A common practice among R users is to take advantage of this system by making a new directory for each R project. In that case, each project has its own *.RData* file and that file is automatically loaded when you start R in the project directory.

Some R packages use smaller *Environments* within the global *Envirnoments* as an efficient way to package large data sets. We will see an example of this use when we explore *Bioconductor*.

### The Initial R display

R is a terminal application. You type in commands at the console and usually receive the output via the console. Exceptions to this are graphical output, which is sent to a separate window, and output that you have explicitly redirected to a file.

When R starts, it prints its version number, a bit of propaganda, and some hints as to how to get started. One suggestion R makes is to try out some *demos*. As an example, after class try `demo(graphics)` for a good introduction to the graphical capabilities of R.

```
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
```

### A First Look at Some Data

### Functions Used

- `data()` # to see and import sample data sets
- `head()` # to see the first lines of a variable
- `ls()` # to list the objects in the work space
- `search()` # to see the packages in R's search path
- `tail()` # to see the last lines of a variable

Initially, R loads some basic packages (the terms *packages* and *libraries* are used interchangeably) that provide its core functions. The `search()` function shows the packages that are currently in the search path, i.e.. the packages that have been loaded. Typing `search()` at the start of a session will produce something similar to the following:

```
search()
```

```
## [1] ".GlobalEnv"        "package:stats"     "package:graphics"
## [4] "package:grDevices" "package:utils"     "package:datasets"
## [7] "package:methods"   "Autoloads"         "package:base"
```

The package *base* gives us over 1100 functions including `which()`, `which.max()`, `table()`, `apply()`, `summary()`, `sum()`, `min()`. `max()`, `sample()`, all the basic object constructors such as `matrix()`, and finally, the all-important `c()` function to combine objects into a vector.

The package *stats* provides over 400 statistical functions including `t.test()`, `hclust()`, `cutree()`, `dist()`, `runif()`, and `rnorm()`.

The package *utils* provides another 200 functions including such essentials as `head()`, `tail()`, `write.table()`, `read.table()`, `history()`, `install.packages()` and the very friendly `help()` function.

The package *graphics* provides staples such as `plot()`, `boxplot()`, `barplot()`, and `hist()`.

Listing these packages with, e.g., `ls(package:graphics)` is a very good way to discover some gems.

Some packages also come with sample data sets. To see these, use `data()`.

```
data()
```

To load a particular data set, use e.g. `data(DNase)`. The data is more than one screen long so we can limit the display to the first 6 lines using `head()`. Also try `tail()` and `summary()`.

```
data(DNase)  # import `DNase into your Environment

ls()  # the new data set now shows up in the listing of files
```

```
## [1] "DNase"
```

```
head(DNase)  # see the first 6 lines
```

```
##   Run       conc density
## 1   1 0.04882812   0.017
## 2   1 0.04882812   0.018
## 3   1 0.19531250   0.121
## 4   1 0.19531250   0.124
## 5   1 0.39062500   0.206
## 6   1 0.39062500   0.215
```

```
tail(DNase)  # see the last 6 lines
```

```
##     Run   conc density
## 171  11  3.125   0.994
## 172  11  3.125   0.980
## 173  11  6.250   1.421
## 174  11  6.250   1.385
## 175  11 12.500   1.715
## 176  11 12.500   1.721
```

```
summary(DNase)  # get a statistical summary
```

```
##       Run          conc            density
##  10     :16   Min.   : 0.04883   Min.   :0.0110
##  11     :16   1st Qu.: 0.34180   1st Qu.:0.1978
##  9      :16   Median : 1.17188   Median :0.5265
##  1      :16   Mean   : 3.10669   Mean   :0.7192
##  4      :16   3rd Qu.: 3.90625   3rd Qu.:1.1705
##  8      :16   Max.   :12.50000   Max.   :2.0030
##  (Other):80
```

### Accessing Your Command History

### Functions Used

- `history()` # to see your command history
- `loadhistory()` # to load a previously saved command history
- `savehistory()` # to save your command history

In the terminal, you can move back through your command history using the up and down cursor keys. The `history()` function can also be used to get a list of previous commands (25 by default). The history can be saved and saved histories retrieved. When you save your current *Environment* at the end of an R session, your command history is also saved in a separate file called *.Rhistory*.

```
history(5) # see your last 5 commands
```

```
savehistory("history1.Rhistory") # save all the commands issued during this session to a file
```

```
loadhistory("history1.Rhistory") # load the saved history into the current session--overwrites current h
```

### Ending a Session

### Functions Used

- `quit()` # exit R and optionally save your *Environment*

You type `quit()` to exit R. At this point, R will offer to save your current *Environment*. You should generally reply "y" to this prompt. You can always delete the resulting *.RData* file later if you don't want it. There are a number of command line options you can give R to change default behaviors, such as loading *.RData* at the start of a session. For instance, starting R with `R --vanilla` will prevent R from loading anything from previous sessions and will eliminate the offer to save anything at the end of the session. A normal exit will look something like this:

```
quit()
```

```
Save workspace image to ~/.RData? [y/n/c]:
```

### Viewing and manipulating variables

#### Functions Used

- `ls()` # to list the variables in the global *Environment*
- `cat()` # to print the raw contents of a object
- `rm()` # to delete an object

R provides a few *unix-like* commands for viewing and manipulating variables.

```
ls()  # get a listing of variables and functions
```

```
## [1] "DNase"
```

```
NewVar = "abc"  # create a new variable

ls()  # you should now see NewVar in the listing
```

```
## [1] "DNase"  "NewVar"
```

```
NewVar  # see NewVar--it is a vector of length 1
```

```
## [1] "abc"
```

```
cat(NewVar)  # print the contents of the character string in NewVar
```

```
## abc
```

```
rm(NewVar)  # remove NewVar

ls()  # it should be gone now
```

```
## [1] "DNase"
```

#### Saving R Objects for Later Use

The object that show up in your workspace when you use `ls()` can be saved to disk at any time using `save.image()`, which saves everything, or `save()` which saves only what you specify. The objects are saved as compressed images and cannot be read by other programs. To save as text you must use a function such as `write.table()`, discussed later.

#### Function Used

- `save.image()` # to save your workspace image
- `save()` # to save one or more objects to a file as a compressed image
- `load()` # to load objects from a previously saved compressed image

```
save.image() # saves everything into the file .RData

save(c(a,b,c,d,e,f),file="objects.R") # saves the objects a-f into a file called "objects.R"

load("objects.R") # loads previously saved objects
```

**Help!**

**Functions Used**

- `apropos` # to get a list of functions and other objects containing a text string
- `example()` # to retrieve and run code that demonstrates the use of a function
- `help()` # to view the help pages for a function or other object

R's help system is quite good. There are many ways to get help. In the example below, we just want to run a T-test but have no clue as to where to begin:

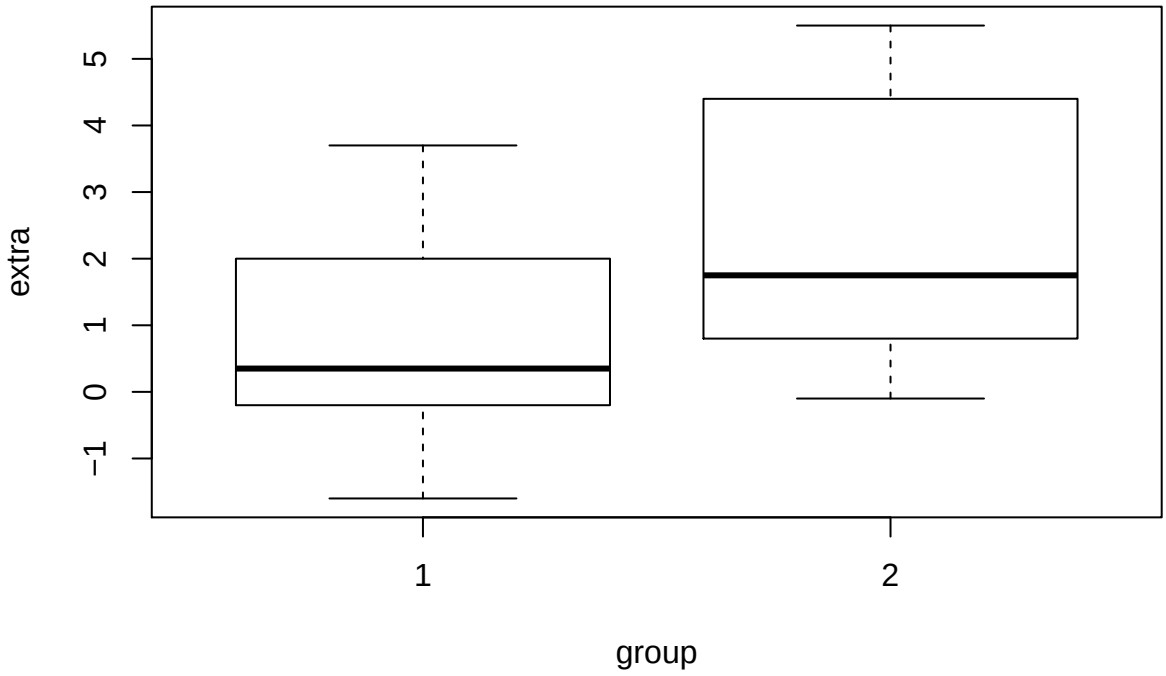
```
apropos("test") #if you don't know the exact name, generate some possibilites...
```

```
##  [1] "ansari.test"             "bartlett.test"
##  [3] "binom.test"              "Box.test"
##  [5] "chisq.test"              "cor.test"
##  [7] "file_test"               "fisher.test"
##  [9] "fligner.test"            "friedman.test"
## [11] "kruskal.test"            "ks.test"
## [13] "mantelhaen.test"         "mauchly.test"
## [15] "mcnemar.test"            "mood.test"
## [17] "oneway.test"             "pairwise.prop.test"
## [19] "pairwise.t.test"         "pairwise.wilcox.test"
## [21] "poisson.test"            "power.anova.test"
## [23] "power.prop.test"         "power.t.test"
## [25] "PP.test"                 "prop.test"
## [27] "prop.trend.test"         "quade.test"
## [29] "shapiro.test"            "testInheritedMethods"
## [31] "testPlatformEquivalence" "testVirtual"
## [33] "t.test"                  ".valueClassTest"
## [35] "var.test"                "wilcox.test"
```

```
help(t.test) #...then pick one from the list

?t.test #...or use the shorthand form

example("t.test") # to get an example of the use of the function
```

```
##
## t.test> require(graphics)
##
## t.test> t.test(1:10, y = c(7:20))      # P = .00001855
##
##  Welch Two Sample t-test
##
## data:  1:10 and c(7:20)
## t = -5.4349, df = 21.982, p-value = 1.855e-05
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -11.052802  -4.947198
## sample estimates:
## mean of x mean of y
```

```
##       5.5      13.5
##
##
## t.test> t.test(1:10, y = c(7:20, 200)) # P = .1245    -- NOT significant anymore
##
##  Welch Two Sample t-test
##
## data:  1:10 and c(7:20, 200)
## t = -1.6329, df = 14.165, p-value = 0.1245
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -47.242900   6.376233
## sample estimates:
## mean of x mean of y
##   5.50000  25.93333
##
##
## t.test> ## Classical example: Student's sleep data
## t.test> plot(extra ~ group, data = sleep)
```




```
##
## t.test> ## Traditional interface
## t.test> with(sleep, t.test(extra[group == 1], extra[group == 2]))
##
##  Welch Two Sample t-test
##
## data:  extra[group == 1] and extra[group == 2]
## t = -1.8608, df = 17.776, p-value = 0.07939
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -3.3654832  0.2054832
## sample estimates:
```

```
## mean of x mean of y
##      0.75      2.33
##
##
## t.test> ## Formula interface
## t.test> t.test(extra ~ group, data = sleep)
##
##  Welch Two Sample t-test
##
## data:  extra by group
## t = -1.8608, df = 17.776, p-value = 0.07939
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -3.3654832  0.2054832
## sample estimates:
## mean in group 1 mean in group 2
##            0.75            2.33
```

### Loading Packages

### Functions Used

- `library()` # to load and installed package
- `search()` # to see which packages are already loaded

R is extensible via packages. As we've seen, `search()` tells us what is already loaded. A related function `library()` can tell us what is available on our computer (these are the packages that are installed locally) and subsequently load packages to make their functions and data available in the current session. Installing packages from within R is accomplished using the `install.packages()` function, e.g. `install.packages("gplots")`.

```
library()  #see what is already installed locally
```

```
## Warning in library(): library '/usr/lib/R/site-library' contains no
## packages
```

```
library("Biobase")  #load an installed library
```

```
## Loading required package: BiocGenerics
## Loading required package: parallel
##
## Attaching package: 'BiocGenerics'
##
## The following objects are masked from 'package:parallel':
##
##     clusterApply, clusterApplyLB, clusterCall, clusterEvalQ,
##     clusterExport, clusterMap, parApply, parCapply, parLapply,
##     parLapplyLB, parRapply, parSapply, parSapplyLB
##
## The following object is masked from 'package:stats':
##
##     xtabs
##
```

```
## The following objects are masked from 'package:base':
##
##     anyDuplicated, append, as.data.frame, as.vector, cbind,
##     colnames, do.call, duplicated, eval, evalq, Filter, Find, get,
##     intersect, is.unsorted, lapply, Map, mapply, match, mget,
##     order, paste, pmax, pmax.int, pmin, pmin.int, Position, rank,
##     rbind, Reduce, rep.int, rownames, sapply, setdiff, sort,
##     table, tapply, union, unique, unlist, unsplit
##
## Welcome to Bioconductor
##
##     Vignettes contain introductory material; view with
##     'browseVignettes()'. To cite Bioconductor, see
##     'citation("Biobase")', and for packages 'citation("pkgname")'.
```

```
search()  #you should now see this newly loaded library in your search path
```

```
##  [1] ".GlobalEnv"           "package:Biobase"      "package:BiocGenerics"
##  [4] "package:parallel"     "package:stats"        "package:graphics"
##  [7] "package:grDevices"    "package:utils"        "package:datasets"
## [10] "package:methods"      "Autoloads"            "package:base"
```

```
data(package = "Biobase")  #check to see if the library *Biobase* comes with any sample data


data(geneData)  #it does, so load geneData--we'll be using this data later
```

### Logging a Session to a File

#### Functions Used

- `sink()` # to log console output to a file

R allows you to redirect console output to a file, or send output to a file in addition to continuing console output.

```
# log everything that you do to 'logfile.log' but also continue (split=TRUE)
# printing to the screen. Don't overwrite any previous contents
# (append=TRUE)

sink("logfile.log", split = TRUE, append = TRUE)


sink()  #close the logfile when you are finished
```

### Running External R Code

#### Functions Used

- `cat()` # to print the raw, unformatted contents of an object
- `source()` # to read and execute external R code

- `ls()` # to list the variables in the workspace

External R code can be read into an R session. The code can be read from a local file or from a remote location on the Web:

```
# use the *cat()* command export some R code to a file.  The code creates
# 'test.matrix'. Save as 'test.R'

cat("test.matrix=matrix(1:1000,100,10)", file = "test.R")


source("test.R")  #now, read and execute the R source file we have just created


ls()  #verify that 'test.matrix' has been created
```

```
## [1] "DNase"       "geneData"    "test.matrix"
```

## The Elements of R: Atomic Types

### Functions Used

- `c()` # to combine objects into a vector
- `sum()` # to take the sum of the values in an object
- `class()` # to see the class of an object

These are the most basic elements used in R. The more complex **objects** (a designation that includes but is not limited to variables and functions) that you will use in your analyses are built from these. The `class()` function returns a object's type. Determining the *class* of an object is very important when things go wrong as many errors stem from using the wrong type of object as an argument to a function.

- numeric
- integer
- missing values
- character
- logical

### Numeric

Both the integer **10** and the float **3.14159** are of the same type, **numeric**.

```
10
```

```
## [1] 10
```

```
3.14159
```

```
## [1] 3.14159
```

But ranges, which are indicated using a `:`, are of class **integer**.

```
r = 10

R = 1:10

class(r)
```

```
## [1] "numeric"
```

```
class(R)
```

```
## [1] "integer"
```

Logical values are represented with the constants `TRUE` and `FALSE`. These evaluate to `1` and `0` respectively. Logical expressions are frequently used to subset data.

### Logical

```
TRUE == 0  # this is FALSE
```

```
## [1] FALSE
```

```
TRUE == 1  # this is TRUE
```

```
## [1] TRUE
```

```
FALSE == 0  # this is also TRUE
```

```
## [1] TRUE
```

```
FALSE == 1  # but this is FALSE
```

```
## [1] FALSE
```

```
10 >= 9  # TRUE
```

```
## [1] TRUE
```

Here we use the `c()` function to combine a series of expressions into a single vector. When we do so, we get a vector of logical values. Taking the sum using `sum()` gives us the count of `TRUE` values.

```
a = c(1 > 2, 3 < 10, 1 == 1, 7 == 6)

a
```

```
## [1] FALSE  TRUE  TRUE FALSE
```

```
sum(a)
```

```
## [1] 2
```

The **AND** and **OR** operators are `&` and `|`, respectively.

```
10 >= 9 & 10 < 10
```

```
## [1] FALSE
```

```
10 == 20 | 10 == 10
```

```
## [1] TRUE
```

The NOT operator is `!`

```
!(10 == 20 | 10 == 10)
```

```
## [1] FALSE
```

Logicals can be used to restrict a function to act on a subset of a vector. This technique works because the logical operation generates a vector of logicals, one for each element in the vector. If this vector is then used as a subscript, all `TRUE` indices will be kept while all `FALSE` indices will be dropped.

```
b = c(1:3, -5:0)  # make a little vector

b  # here is 'b'
```

```
## [1]  1  2  3 -5 -4 -3 -2 -1  0
```

```
b > 0  # for each 'b' we have one logical--all the 'FALSE' indices will be dropped if we use this as a
```

```
## [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
b[b > 0]  # use only the positive values
```

```
## [1] 1 2 3
```

```
b  # here is 'b' again
```

```
## [1]  1  2  3 -5 -4 -3 -2 -1  0
```

```
b < 0  # here is another logical vector of the same length as 'b'
```

```
## [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
```

```r
b[b < 0]  # use only the negative values
```

```
## [1] -5 -4 -3 -2 -1
```

```r
sum(b[b < 0])  # sum only the negatives
```

```
## [1] -15
```

### Missing Values

### Functions Used

- `is.na()` # to identify NA's within an object
- `anyNa()` # to determine if there is at least one NA within an object
- `mean()` # to take the mean of the values in an object

Missing values get a value of `NA` (Not Available), another built-in constant like `TRUE` and `FALSE`. You can test for these using `is.na()` and you can ignore them in some functions using `na.rm=TRUE` as a function argument.

```r
nana = c(1, 2, 3, 4, 5, 6, NA, NA, 9, 10)

is.na(nana[1])  # is the first element a NA?
```

```
## [1] FALSE
```

```r
is.na(nana[7])  # is the second element a NA?
```

```
## [1] TRUE
```

```r
is.na(nana)  # generate a logical value for each member of nana
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE
```

```r
anyNA(nana)  # return `TRUE` if any member of `nana` is NA
```

```
## [1] TRUE
```

```r
mean(nana)  # generates a NA if any element of `nana` is NA
```

```
## [1] NA
```

```r
mean(nana, na.rm = TRUE)  # ignores the NA's
```

```
## [1] 5
```

### Character

### Functions Used

- `c()` # to combine objects into a vector
- `length()` # to get the length of an object
- `nchar()` # to get the number of characters in a character string
- `paste()` # to create a single character string by joining together character strings and numbers with spaces
- `paste0()` # like `paste()` but joins without spaces

Strings are *character* types. You can get their lengths using `nchar()`. Note that `length()` returns the length of the variable itself rather than the length of the string it contains.

```
a = "ABCDEFG"  # define a string

b = "HIJ"  # define another

length(a)  # get the length of 'a'
```

```
## [1] 1
```

```
nchar(a)  # get the number of characters in the string containined in 'a'
```

```
## [1] 7
```

```
c = c(a, b)  # combine the strings into a single vector

length(c)  # get the length of a vector containing two strings
```

```
## [1] 2
```

```
nchar(c)  # get a vector containing the number of characters in each member of 'c(a,b)'
```

```
## [1] 7 3
```

A couple of pre-defined *character* constants are worth mentioning; `letters`, and `LETTERS`. These are useful for labeling arrays of variables on the fly.

```
letters[1:10]
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

```
LETTERS[1:10]
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"
```

The `paste()` function combines character strings and numerics with a space character and returns a single character string. To combine without the intervening space, use `paste0()`.

```
prefix = "I've always said that "

suffix = ", and that's a fact."

paste0(prefix, "pi,", pi, ", is a pre-defined constant in R", suffix)
```

```
## [1] "I've always said that pi,3.14159265358979, is a pre-defined constant in R, and that's a fact."
```

## The Elements of R: More Complex Types

| Object | Composition |
|---|---|
| vector | an array of elements |
| list | container for other elements |
| matrix | vector with RxC elements |
| data.frame | list with N columns of identical length |
| factor | group identifier |
| function | block of R commands |

### Vectors

### Functions Used

- `c()` # to combine objects into a vector
- `length()` # to get the length of an object

Working in R is largely about the manipulation of vectors. The result of the operation $1 + 1$ is a vector of length 1. The variable $a$ below is also a vector of length 1. So is the result of `length(a)`. Individual elements of a vector are specified within square brackets, e.g. `a[1]`. Indices for vectors and all other array-like objects start at `1`.

```
1 + 1
```

```
## [1] 2
```

```
a = 1 + 1

a
```

```
## [1] 2
```

```
a[1]
```

```
## [1] 2
```

```
length(a)
```

```
## [1] 1
```

We can construct a vector with arbitrary values using the `c()` function and we can use such a vector within square brackets to specify arbitrary elements of another vector.

```
b = c(1, 2, -30, 40, -50)

b
```

```
## [1]   1   2 -30  40 -50
```

```
b[1]
```

```
## [1] 1
```

```
b[2]
```

```
## [1] 2
```

```
b[c(3, 5)]
```

```
## [1] -30 -50
```

Ranges of integers can be generated using the `:` operator and then used to specify the indices of a vector.

```
2:4
```

```
## [1] 2 3 4
```

```
b[2:4]
```

```
## [1]   2 -30  40
```

And one can nest elements, e.g. a range nested within the `c()` construct:

```
b[c(1:3, 5)]
```

```
## [1]   1   2 -30 -50
```

### Matrices

### Functions Used

- `attributes()` # to list the attributes of an object (e.g. names, dim)
- `c()` # to combine objects into a vector
- `colSums()` # to sum over the columns of a matrix or data.frame
- `dim()` # to get the dimensions of a matrix or data.frame
- `matrix()` # to construct a new matrix
- `rowSums()` # to sum over the rows of a matrix or data.frame
- `sum()` # to get the sum of the values in an object

Matrices are vectors of length $R * C$ , where $R$ and $C$ are rows and columns, respectively. The numbers of rows and columns are returned and set using the `dim()` function. The elements of a *matrix* are accessed using a comma-separated row and column identifier with square brackets.

```
M = 1:12  # generate some data for a matrix

attributes(M)  # this is just a vector so it has no `attributes`
```

```
## NULL
```

```
dim(M)  # an attribute of a matrix is that his has dimensions--this vector has no dimensions
```

```
## NULL
```

```
N = matrix(data = M, nrow = 3, ncol = 4)  # create a matrix by specifying numbers for rows and columns,

N  # take a look
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

```
dim(N)  # N has dimensions
```

```
## [1] 3 4
```

```
N[1, 2]  # a single element
```

```
## [1] 4
```

```
N[, 3]  # all the rows in column 3
```

```
## [1] 7 8 9
```

```r
N[2, ]  # all the columns in row 2
```

```
## [1]  2  5  8 11
```

```r
rowSums(N)  # we can now take row sums
```

```
## [1] 22 26 30
```

```r
colSums(N)  # or column sums
```

```
## [1]  6 15 24 33
```

```r
sum(N)  # or we can take the global sum
```

```
## [1] 78
```

### Lists

### Functions Used

- `attributes()` # to list the attributes of an object (e.g. names, dim)
- `c()` # to combine objects into a vector
- `head()` # to see the first lines of an object
- `list()` # to create a new list
- `matrix()` # to create a new matrix
- `names()` # to get the names of the elements of a list

The elements of a **list** can contain anything (**vectors**, **matrices**, **data.frames**, or other **lists**). You can add elements to and access the elements of a list using two operators; `[[]]`, and `$`. The first form is similar to a vector subscript and takes a number between the brackets. The `$` form is a special case of the `[[]]` form in which you access a single item its name.

```r
L = list()  # create an empty list using the 'list()' 'constructor'

L$first = c(1, 2, 3)  # add an item to the list and call it 'first'

L  # take a look
```

```
## $first
## [1] 1 2 3
```

```r
L[[2]] = c(4, 5, 6)  # add a second item but do not give it a name

L  #take a look
```

```
## $first
## [1] 1 2 3
##
## [[2]]
## [1] 4 5 6
```

```r
attributes(L)  # the list now has an attribute called 'names'
```

```
## $names
## [1] "first" ""
```

```r
names(L)  # we can get the name of 'L' using `names()`
```

```
## [1] "first" ""
```

```r
names(L)[2] = "second"  # we can also assign names this way

L  # take a look
```

```
## $first
## [1] 1 2 3
##
## $second
## [1] 4 5 6
```

```r
L$second  # now we can access the second element by name
```

```
## [1] 4 5 6
```

```r
M = matrix(1:200, 20, 10)  # create a matrix of 20 rows, 10 columns and load it with the integers 1-200

L$mat = M  # add the new matrix to our list and call it 'mat'

L  # take a look
```

```
## $first
## [1] 1 2 3
##
## $second
## [1] 4 5 6
##
## $mat
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    1   21   41   61   81  101  121  141  161   181
##  [2,]    2   22   42   62   82  102  122  142  162   182
##  [3,]    3   23   43   63   83  103  123  143  163   183
##  [4,]    4   24   44   64   84  104  124  144  164   184
##  [5,]    5   25   45   65   85  105  125  145  165   185
##  [6,]    6   26   46   66   86  106  126  146  166   186
##  [7,]    7   27   47   67   87  107  127  147  167   187
##  [8,]    8   28   48   68   88  108  128  148  168   188
##  [9,]    9   29   49   69   89  109  129  149  169   189
## [10,]   10   30   50   70   90  110  130  150  170   190
## [11,]   11   31   51   71   91  111  131  151  171   191
## [12,]   12   32   52   72   92  112  132  152  172   192
## [13,]   13   33   53   73   93  113  133  153  173   193
```

```
## [14,]   14   34   54   74   94  114  134  154  174   194
## [15,]   15   35   55   75   95  115  135  155  175   195
## [16,]   16   36   56   76   96  116  136  156  176   196
## [17,]   17   37   57   77   97  117  137  157  177   197
## [18,]   18   38   58   78   98  118  138  158  178   198
## [19,]   19   39   59   79   99  119  139  159  179   199
## [20,]   20   40   60   80  100  120  140  160  180   200
```

```
head(L$mat)  # see the first part of 'mat' in the list 'L'
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    1   21   41   61   81  101  121  141  161   181
## [2,]    2   22   42   62   82  102  122  142  162   182
## [3,]    3   23   43   63   83  103  123  143  163   183
## [4,]    4   24   44   64   84  104  124  144  164   184
## [5,]    5   25   45   65   85  105  125  145  165   185
## [6,]    6   26   46   66   86  106  126  146  166   186
```

### Data.Frames

### Functions Used

- `data.frame()` # to create a new data.frame
- `attributes()` # to list the attributes of an object (e.g. names, dim)
- `names()` # to list the names of the elements of a list or data.frame

A **data.frame** is a **list** consisting of elements treated as columns of identical length. Individual items within a **data.frame** can be accessed using the subscript notation of the **matrix**.

```
# create a data.frame with columns called 'A', 'twiceA', and 'noisyA', all
# of length 10

D = data.frame(A = 1:10, twiceA = 2 * 1:10, noisyA = 1:10 + rnorm(10))

D  # take a look
```

```
##     A twiceA    noisyA
## 1   1      2  3.268181
## 2   2      4  3.296129
## 3   3      6  2.503717
## 4   4      8  4.804190
## 5   5     10  4.334152
## 6   6     12  5.080558
## 7   7     14  7.229084
## 8   8     16  9.004516
## 9   9     18  9.035847
## 10 10     20 10.821371
```

```
D[[1]]  # get element 1 which is column 1
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
D[[2]]  # get element 2 which is column 2
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20
```

```
D[[3]]  # get element 2 which is column 3
```

```
##  [1]  3.268181  3.296129  2.503717  4.804190  4.334152  5.080558  7.229084
##  [8]  9.004516  9.035847 10.821371
```

```
attributes(D)  # the data.frame has a names attribute
```

```
## $names
## [1] "A"      "twiceA" "noisyA"
##
## $row.names
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $class
## [1] "data.frame"
```

```
names(D)  # get the names of 'D'
```

```
## [1] "A"      "twiceA" "noisyA"
```

```
D$noisyA  # get the column of data named 'noisyA'
```

```
##  [1]  3.268181  3.296129  2.503717  4.804190  4.334152  5.080558  7.229084
##  [8]  9.004516  9.035847 10.821371
```

### Custom Objects

### Functions Used

- `rnorm()` # to generate a set of random normal deviates that simulate noise
- `plot()` # to set up a graphics window and plot a set of points
- `lm()` # to fit a linear model to a set of (x,y) values
- `class()` # to show the class of an object
- `str()` # to show the internal structure of an object
- `attributes()` # to show the attributes of an object (e.g. names, dim)
- `abline()` # to draw a straight line by defining a slope and intercept
- `points()` # to plot points in an open graphics window without re-initializing the window as in `plot()`

Custom objects are returned by many functions in R. They often come in the form of lists of other objects and are away of delivering complicated output in a single package that we can dissect as we choose. Here we use `lm()` to create linear model object, *LinFit*, that contains a long list of statistics resulting from our fit to some quickly fabricated data. To specify the linear model, we use the ~ model operator which can be read as *is a function of*.

```r
noise = rnorm(1000, mean = 0, sd = 100)  # generate 1000 random deviates with Standard Deviation of 100

x = 1:1000  # generate a vector of integers running from 1 to 1000

y = 1:1000 + noise  # do the same here but add the 1000 elements of noise

plot(y ~ x)  # plot y 'as.a.function.of' x--`plot()` always creates a new plot

LinFit = lm(y ~ x)  # perform a linear fit to the data using the model 'y is a function of x'

class(LinFit)  # see what sort of object is returned by `lm()`
```

```
## [1] "lm"
```

```r
str(LinFit)  # see the internal structure of this innocent-looking variable
```

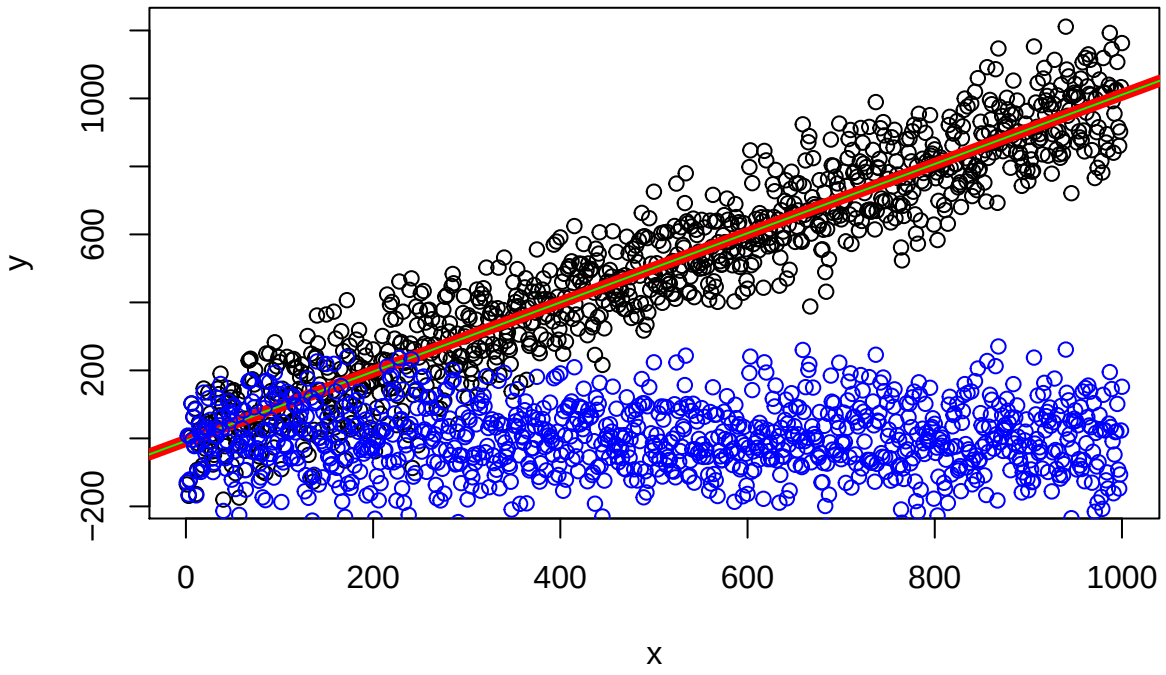
```
## List of 12
##  $ coefficients : Named num [1:2] -7.29 1.02
##   ..- attr(*, "names")= chr [1:2] "(Intercept)" "x"
##  $ residuals    : Named num [1:1000] -129.15 8.13 -164.83 -120.81 7.4 ...
##   ..- attr(*, "names")= chr [1:1000] "1" "2" "3" "4" ...
##  $ effects      : Named num [1:1000] -15885.4 9295.3 -161.5 -117.5 10.8 ...
##   ..- attr(*, "names")= chr [1:1000] "(Intercept)" "x" "" "" ...
##  $ rank         : int 2
##  $ fitted.values: Named num [1:1000] -6.27 -5.26 -4.24 -3.22 -2.2 ...
##   ..- attr(*, "names")= chr [1:1000] "1" "2" "3" "4" ...
##  $ assign       : int [1:2] 0 1
##  $ qr           :List of 5
##   ..$ qr   : num [1:1000, 1:2] -31.6228 0.0316 0.0316 0.0316 0.0316 ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:1000] "1" "2" "3" "4" ...
##   .. .. ..$ : chr [1:2] "(Intercept)" "x"
##   .. ..- attr(*, "assign")= int [1:2] 0 1
##   ..$ qraux: num [1:2] 1.03 1.05
##   ..$ pivot: int [1:2] 1 2
##   ..$ tol  : num 1e-07
##   ..$ rank : int 2
##   ..- attr(*, "class")= chr "qr"
##  $ df.residual  : int 998
##  $ xlevels      : Named list()
##  $ call         : language lm(formula = y ~ x)
##  $ terms        :Classes 'terms', 'formula' length 3 y ~ x
##   .. ..- attr(*, "variables")= language list(y, x)
##   .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. ..$ : chr [1:2] "y" "x"
##   .. .. .. ..$ : chr "x"
##   .. ..- attr(*, "term.labels")= chr "x"
##   .. ..- attr(*, "order")= int 1
##   .. ..- attr(*, "intercept")= int 1
##   .. ..- attr(*, "response")= int 1
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
```

```
##   .. ..- attr(*, "predvars")= language list(y, x)
##   .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##   .. .. ..- attr(*, "names")= chr [1:2] "y" "x"
##  $ model        :'data.frame':   1000 obs. of  2 variables:
##   ..$ y: num [1:1000] -135.42 2.87 -169.07 -124.03 5.2 ...
##   ..$ x: int [1:1000] 1 2 3 4 5 6 7 8 9 10 ...
##   ..- attr(*, "terms")=Classes 'terms', 'formula' length 3 y ~ x
##   .. .. ..- attr(*, "variables")= language list(y, x)
##   .. .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. .. ..$ : chr [1:2] "y" "x"
##   .. .. .. .. ..$ : chr "x"
##   .. .. ..- attr(*, "term.labels")= chr "x"
##   .. .. ..- attr(*, "order")= int 1
##   .. .. ..- attr(*, "intercept")= int 1
##   .. .. ..- attr(*, "response")= int 1
##   .. .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. .. ..- attr(*, "predvars")= language list(y, x)
##   .. .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##   .. .. .. ..- attr(*, "names")= chr [1:2] "y" "x"
##  - attr(*, "class")= chr "lm"
```

```
attributes(LinFit)  # see the attributes of the object
```

```
## $names
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "xlevels"       "call"          "terms"         "model"
##
## $class
## [1] "lm"
```

```
# extract the slope and intercept of the best fit line to draw the line on
# the plot

abline(a = LinFit$coefficients[1], b = LinFit$coefficients[2], col = "red",
    lwd = 6)

# this way of extracting the coefficients also works

abline(a = coefficients(LinFit)[1], b = coefficients(LinFit)[2], col = "green")

# add the residuals--do not use `plot()` again if you want to add to an
# existing plot

points(x, LinFit$residuals, col = "blue")  # plot the residuals along the bottom
```




**Factors**

**Functions Used**

- `as.numeric()` # to *coerce* an object to a numeric vector
- `boxplot()` # to plot data as a box an whisker plot given a raw data or a model
- `c()` # to combine objects into a vector
- `class()` # to see the class of an object
- `data.frame()` # to create a new data.frame
- `levels()` # to see or set the levels of a *factor*
- `str()` # to see the structure of an object
- `t.test()` # to perform a T-test

Factors are used to flag data items as belonging to a particular group (a level) for purposes of statistical comparison. Factors are actually implemented as arrays of integers but they are displayed as strings for human consumption.

Here we use a technique called *coersion* to get a factor as a set of integers with the function `as.numeric()`. Similar functions also exist to convert between other classes. These include `as.matrix()`, `as.data.frame()`, `as.character()`, and `as.factor()`.

```
F = factor(c("A", "B", "B", "A"))  # use the factor 'constructor' to make a factor with names given in

F  # take a look
```

```
## [1] A B B A
## Levels: A B
```

```
class(F)  # get the class to make sure we have a `factor`
```

```
## [1] "factor"
```

```r
levels(F)  # get the levels of F
```

```
## [1] "A" "B"
```

```r
str(F)  # see the structure which shows the levels both as labels and as the underlying integers
```

```
##  Factor w/ 2 levels "A","B": 1 2 2 1
```

```r
as.numeric(F)  # get the factor as a numeric vector
```

```
## [1] 1 2 2 1
```

Factors are used to group data for tests such as the T-test. The ~ operator used in the `t.test()` function defines the model to be tested–that **score** is a function of **group**. Let's use factors to analyze some data with a T-test.

```r
# create a data.frame with two columns called 'score' and 'group'

D = data.frame(score = c(100, 110, 95, 70, 75, 65), group = c("A", "A", "A",
    "B", "B", "B"))

D  # take a look
```

```
##   score group
## 1   100     A
## 2   110     A
## 3    95     A
## 4    70     B
## 5    75     B
## 6    65     B
```

```r
class(D$group)  # the characters in the 'group' data have been turned into factors automatically--data.
```

```
## [1] "factor"
```

```r
t.test(D$score ~ D$group)  # let score be a.function.of (that's the tilde) group in D and test the mean
```

```
##
##  Welch Two Sample t-test
##
## data:  D$score by D$group
## t = 6.0083, df = 3.448, p-value = 0.006133
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  16.06225 47.27109
## sample estimates:
## mean in group A mean in group B
##        101.6667         70.0000
```

Next, let's generate a corresponding box plot.

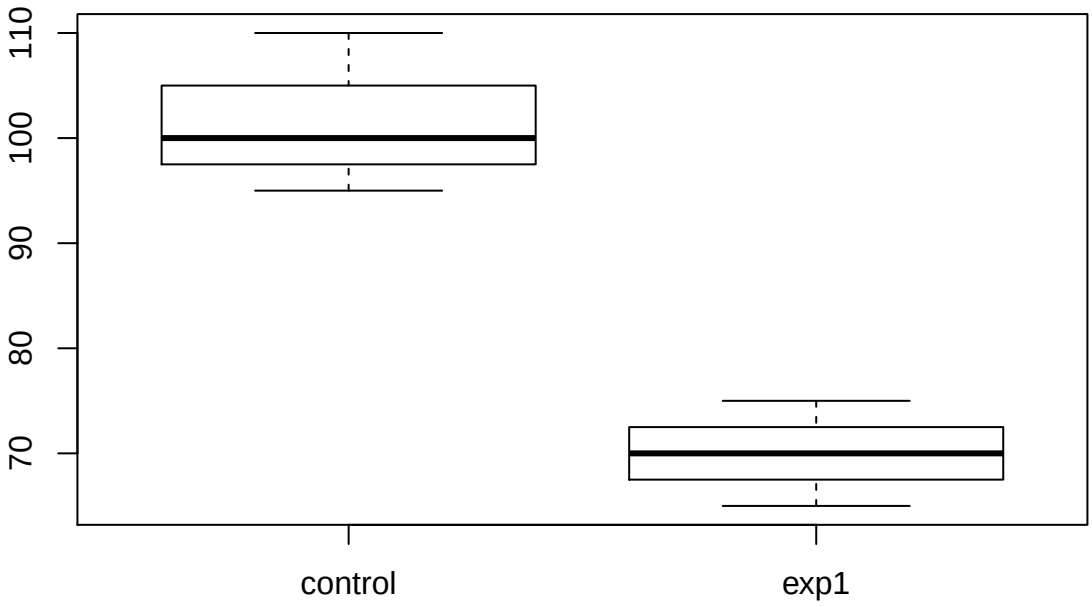
```
levels(D$group)  # remind ourselves of the levels
```

```
## [1] "A" "B"
```

```
levels(D$group) = c("control", "exp1")  #change the levels

levels(D$group)  #verify the change
```

```
## [1] "control" "exp1"
```

```
boxplot(D$score ~ D$group)  # display the data for the groups graphically
```




### Functions

### Functions Used

- `function()` # to begin a function block, defining parameters to be passed to the function
- `return()` # to end a function block, returning a value

Functions are a very convenient way to package parts of an analysis. Here is a simple function followed by a call to the function:

```
Fsq = function(x) {

    y = x^2

    return(y)

}

Fsq(3)
```

```
## [1] 9
```

## Procedures

Now that we have the basic elements in hand, it is time to get down to work with some common procedures. We'll begin with an essential procedure–reading and writing data. Will first make sure we have `geneData` and then write it to a file.

### Writing and Reading Data

The primary means of reading in and writing out data from within R are the `read.table()` and `write.table()` functions. The two functions handle tabular data and generally work without the need to specify data separators, however these separators can be specified using the parameter *sep*. Specialized variants such as `read.csv()` and `read.delim()` correspond to `read.table()` but with different default parameters.

### Functions Used

- `abs()` # to get the absolute value of number
- `data()` # to get some sample data for testing
- `head()` # to get a look at the first part of an object
- `length()` # to get the length of an object
- `read.table()` # to read tabular format data from a file
- `sum()` # to sum over the values in an object
- `write.table()` # to write a variable's data to a file in tabular format

```
data(geneData)  # add `geneData` to our environment

head(geneData, 2)  # check it--see only the first two lines
```

```
##                       A        B        C        D       E       F
## AFFX-MurIL2_at  192.742  85.7533 176.7570 135.5750 64.4939 76.3569
## AFFX-MurIL10_at  97.137 126.1960  77.9216  93.3713 24.3986 85.5088
##                       G       H       I        J       K       L       M
## AFFX-MurIL2_at  160.5050 65.9631 56.9039 135.6080 63.4432 78.2126 83.0943
## AFFX-MurIL10_at  98.9086 81.6932 97.8015  90.4838 70.5733 94.5418 75.3455
##                       N       O       P        Q       R       S       T
## AFFX-MurIL2_at  89.3372 91.0615 95.9377 179.8450 152.467 180.834 85.4146
## AFFX-MurIL10_at 68.5827 87.4050 84.4581  87.6806 108.032 134.263 91.4031
##                        U        V       W        X       Y        Z
## AFFX-MurIL2_at  157.98900 146.8000 93.8829 103.8550 64.4340 175.6150
## AFFX-MurIL10_at  -8.68811  85.0212 79.2998  71.6552 64.2369  78.7068
```

```
write.table(geneData, file = "test")  # write it in tabular form

NewVar = read.table(file = "test")  # now read it back into a new variable

head(NewVar, 2)  # take a look
```

```
##                       A        B        C        D       E       F
## AFFX-MurIL2_at  192.742  85.7533 176.7570 135.5750 64.4939 76.3569
## AFFX-MurIL10_at  97.137 126.1960  77.9216  93.3713 24.3986 85.5088
##                       G       H       I        J       K       L       M
```

```
## AFFX-MurIL2_at  160.5050 65.9631 56.9039 135.6080 63.4432 78.2126 83.0943
## AFFX-MurIL10_at  98.9086 81.6932 97.8015  90.4838 70.5733 94.5418 75.3455
##                        N       O       P        Q       R       S       T
## AFFX-MurIL2_at  89.3372 91.0615 95.9377 179.8450 152.467 180.834 85.4146
## AFFX-MurIL10_at 68.5827 87.4050 84.4581  87.6806 108.032 134.263 91.4031
##                         U        V       W        X       Y        Z
## AFFX-MurIL2_at  157.98900 146.8000 93.8829 103.8550 64.4340 175.6150
## AFFX-MurIL10_at  -8.68811  85.0212 79.2998  71.6552 64.2369  78.7068
```

How can we check that NewVar and geneData are identical, item by item?

```
# there are a number of ways

sum(geneData == NewVar) == length(geneData)  # why does this work?
```

```
## [1] TRUE
```

```
sum(abs(geneData - NewVar))  # why does this work?
```

```
## [1] 0
```

### Exploring and Summarizing Data

### Functions Used

- `class()` # get the class of an object
- `colnames()` # for both data.frames and matrices, get the column names
- `data.frame()` # create a new `data.frame`
- `dim()` # get the dimensions of a data.frame or matrix
- `dimnames()` # get both the row names and column names
- `head()` # see the first 6 lines
- `length()` # get the length–for a data.frame this is the number of columns
- `names()` # for a data.frame, get the column names (for a matrix, use colnames())
- `rownames()` # get the rownames of a data.frame or matrix
- `tail()` # see the last 6 lines
- `mean()` # get the mean of the values in an object
- `max()` # get the maximum
- `min()` # get the minimum
- `sd()` # get the Standard Deviation
- `as.matrix()` # coerce an object into a matrix
- `mode()` # get the underlying type of an object
- `summary()` # get a statistical summary for the data in an object

There are several functions that give you technical information about an R object.

```
M = data.frame(A = 1:10, B = 2 * (1:10), C = 2^(1:10))  # create a data.frame with three columns (varia


class(M)  # get its class
```

```
## [1] "data.frame"
```

```
str(M)  # see its internal structure
```

```
## 'data.frame':    10 obs. of  3 variables:
##  $ A: int  1 2 3 4 5 6 7 8 9 10
##  $ B: num  2 4 6 8 10 12 14 16 18 20
##  $ C: num  2 4 8 16 32 ...
```

```
object.size(M)  # find out how much memory it is using
```

```
## 1200 bytes
```

Others help you explore the content...

```
head(M)  # see the first 6 lines
```

```
##   A  B  C
## 1 1  2  2
## 2 2  4  4
## 3 3  6  8
## 4 4  8 16
## 5 5 10 32
## 6 6 12 64
```

```
tail(M)  # see the last 6 lines
```

```
##     A  B    C
## 5   5 10   32
## 6   6 12   64
## 7   7 14  128
## 8   8 16  256
## 9   9 18  512
## 10 10 20 1024
```

```
length(M)  # get the length--for a data.frame this is the number of columns
```

```
## [1] 3
```

```
dim(M)  # get the dimensions of a data.frame or matrix
```

```
## [1] 10  3
```

```
dimnames(M)  # get both the row names and column names as a list
```

```
## [[1]]
##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
##
## [[2]]
## [1] "A" "B" "C"
```

```r
names(M)  # for a data.frame, get the column names (for a matrix, use colnames())
```

```
## [1] "A" "B" "C"
```

```r
colnames(M)  # for both data.frams and matrices, get the column names
```

```
## [1] "A" "B" "C"
```

```r
rownames(M)  # get the rownames
```

```
##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
```

Why is the length of a data.frame equal to the number of columns?

```r
min(M)  # get the minimum value in 'M'
```

**Summarizing the Content of an Object**

```
## [1] 1
```

```r
max(M)  # get the maximum value in 'M'
```

```
## [1] 1024
```

```r
mean(M$B)  # get the mean of column 'B' in 'M'
```

```
## [1] 11
```

```r
sd(M$A)  # get the Standard Deviation of column 'A' in 'M
```

```
## [1] 3.02765
```

```r
summary(M)  # get a statistical summary for each column (variable or sample) in M
```

```
##        A               B              C
##  Min.   : 1.00   Min.   : 2.0   Min.   :   2.0
##  1st Qu.: 3.25   1st Qu.: 6.5   1st Qu.:  10.0
##  Median : 5.50   Median :11.0   Median :  48.0
##  Mean   : 5.50   Mean   :11.0   Mean   : 204.6
##  3rd Qu.: 7.75   3rd Qu.:15.5   3rd Qu.: 224.0
##  Max.   :10.00   Max.   :20.0   Max.   :1024.0
```

What happens if we try to take the mean of all columns of `M` directly?

```
mean(M)
```

```
## Warning in mean.default(M): argument is not numeric or logical: returning
## NA
```

```
## [1] NA
```

And now, **coercion** come in handy. We can **coerce** `M` from a data.frame into a **matrix**. The **coercion** technique works if the two objects have compatible structures. In this case, they do.

```
mean(as.matrix(M))
```

```
## [1] 73.7
```

The reason that we needed to **coerce** the `data.frame` is revealed using `mode()`. As mentioned earlier, a `data.frame` is really a special case of a `list`. It is not logical to take the mean of a **list** but one can certainly take the mean of a **numeric**.

```
mode(M)  # the data.frame is implemented as a list but appears matrix-like
```

```
## [1] "list"
```

```
mode(as.matrix(M))  # fortunantely, it is enough matrix-like that we can transform it into a matrix of
```

```
## [1] "numeric"
```

Taking the mean of one column of the **data.frame** `M` works because it is not a **list** but rather one element of a **list** which happens to be a **numeric**.

```
mode(M$A)
```

```
## [1] "numeric"
```

### Editing Data

### Functions Used

- `edit()` # invoke the editor but discard any changes on exit
- `emacs()` # invoke the 'emacs' editor on an object
- `fix()` # invoke the editor, make your changes and exit
- `pico()` # invoke "pico" on an object

You can edit data using R's primitive built-in GUI editor.

```
fix(M) # invoke the editor, make your changes and exit
```

```
edit(M) # invoke the editor but discard any changes on exit
```

```
N=edit(M) # invoke the default editor, but assign the changed version to another variable
```

You may also use a powerful console editor such as *pico*, or a full-featured graphical editor such as *emacs*.

```
M=pico(M) # invoke "pico", make changes and save them into "M" again on exit
```

```
M=emacs(M) # invoke the 'emacs' editor, and save your changes on exit
```

### Dealing with Missing Data

#### Functions Used

- `sum()` # to sum over the values in an object
- `is.na()` # to flag NA's (Not Available) values in an object

Missing data is problematic. For instance:

```
nana = c(1, 2, 3, 4, 5, 6, NA, NA, 9, 10)  # nana has to NA's

sum(nana)  # returns NA
```

```
## [1] NA
```

One way to fix this, as we've seen:

```
sum(nana, na.rm = TRUE)  # we can ignore NA's on a per function basis like this
```

```
## [1] 40
```

Another method is to put something in place of the missing data.

```
nana[is.na(nana)] = 0  # we can also change them into something else

nana  # take a look--the NA's are now zeros
```

```
##  [1]  1  2  3  4  5  6  0  0  9 10
```

Why does the second method shown above work?

### Restructuring Data

#### Functions Used

- `cbind()` # to append columns to a matrix or data.frame
- `class()` # see an object's class
- `data.frame()` # create a new data.frame()
- `head()` # see the first few lines of an object
- `rbind()` # to append rows to a matrix or data.frame

We can rearrange columns or create new objects from columns in existing objects

```
M = data.frame(A = 1:10, B = 101:110, C = 31:40)  # make a three column data.frame

head(M, 3)  # see the first three lines
```

```
##   A   B  C
## 1 1 101 31
## 2 2 102 32
## 3 3 103 33
```

```
N = data.frame(hiC = M$C, Aplus = M$A, BeBox = M$B)  # make another data.frame with the columns of 'M'

class(N)  # of course, this is a data.frame
```

```
## [1] "data.frame"
```

```
head(N, 2)  # see the first two lines of 'N'
```

```
##   hiC Aplus BeBox
## 1  31     1   101
## 2  32     2   102
```

Now try it a different way using a new function, `cbind()` which builds or adds to matrices or data.frames by tacking on columns side by side . There is a corresponding function called `rbind()` which tacks on more rows.

```
N = cbind(hiC = M$C, Aplus = M$A, BeBox = M$B)  # use `cbind()` to bind the colums together

head(N, 2)  # the result looks similar
```

```
##      hiC Aplus BeBox
## [1,]  31     1   101
## [2,]  32     2   102
```

```
class(N)  # but this is now a matrix rather than a data.frame
```

```
## [1] "matrix"
```

We can add another row to our matrix using `rbind()`.

```
N = rbind(N, c(1000, 2000, 3000))  # add one row, three columns wide

N  # see the result
```

```
##        hiC Aplus BeBox
##  [1,]   31     1   101
##  [2,]   32     2   102
##  [3,]   33     3   103
##  [4,]   34     4   104
##  [5,]   35     5   105
##  [6,]   36     6   106
```

```
##  [7,]   37     7   107
##  [8,]   38     8   108
##  [9,]   39     9   109
## [10,]   40    10   110
## [11,] 1000  2000  3000
```

After constructing the matrix above using `cbind`, how would you convert it into a `data.frame`?

```
N = as.data.frame(N)
```

We can delete a column in a matrix or data.frame using a negative subscript.

```
head(N, 2)  # see the first two lines
```

```
##   hiC Aplus BeBox
## 1  31     1   101
## 2  32     2   102
```

```
N = N[, -1]  # delete column 1--note the comma directly after the first bracket to indicate that the op

head(N, 2)  # verify the deletion
```

```
##   Aplus BeBox
## 1     1   101
## 2     2   102
```

Here we delete multiple rows using an array containing negative subscripts.

```
head(N)  # take a look at 'N'
```

```
##   Aplus BeBox
## 1     1   101
## 2     2   102
## 3     3   103
## 4     4   104
## 5     5   105
## 6     6   106
```

```
N = N[c(-1, -3, -6), ]  # remove rows 1, 3, and 6--note the comma prior to the last bracket to indicate

head(N)
```

```
##   Aplus BeBox
## 2     2   102
## 4     4   104
## 5     5   105
## 7     7   107
## 8     8   108
## 9     9   109
```

### Relabeling Data

### Functions Used

- `c()` # to combine objects into a vector
- `colnames()` # to get the column names of a matrix or data.frame
- `data.frame()` # to create a new data.frame
- `head()` # to take a look at the first lines of an object
- `names()` # to get the names of the elements of a list or data.frame
- `rownames()` # to get the rownames of a matrix or data.frame

We can relabel row and column names using `names()`, `colnames()`, and `rownames()`. These functions retrieve the existing names, if any, but also allow us to set the names.

```
M = data.frame(A = 1:10, B = 21:30, C = 101:110)  # generate a data.frame with columns 'A', 'B', and 'C

rownames(M)  # the row names are now numbers
```

```
##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
```

```
rownames(M) = paste("gene", 1:10, sep = "_")  # replace the row names with something more informative

rownames(M)  # verify the replacement
```

```
##  [1] "gene_1"  "gene_2"  "gene_3"  "gene_4"  "gene_5"  "gene_6"  "gene_7"
##  [8] "gene_8"  "gene_9"  "gene_10"
```

```
names(M)  # take a look at the column names which, by convention, are 'sample' names
```

```
## [1] "A" "B" "C"
```

```
names(M) = c("control", "cell_1", "cell_2")  # replace them with something more informative

names(M)  # take a look at the new names
```

```
## [1] "control" "cell_1"  "cell_2"
```

```
head(M, 3)  # take a look at the first three lines of the relabeled data.frame
```

```
##        control cell_1 cell_2
## gene_1       1     21    101
## gene_2       2     22    102
## gene_3       3     23    103
```

We can use the new row and column labels to access the data but we can still use numeric subscripts if we like.

```
M[c("gene_1", "gene_5"), c("control", "cell_2")]  # take the rows for two genes for the 'control' and '
```

```
##        control cell_2
## gene_1       1    101
## gene_5       5    105
```

```
M[c(1, 5), c(1, 3)]  # do the same using numeric subscripts
```

```
##        control cell_2
## gene_1       1    101
## gene_5       5    105
```

### Subsetting Data

### Functions Used

- `c()` # to combine objects into a vector
- `data.frame()` # to create a new data.frame
- `head()` # to see the first lines of an object
- `max()` # to get the maximum of the values in an object
- `tail()` # to see the last few lines of an object

There are many ways to extract subsets from vectors, matrices, or data.frames. Here are a few examples:

```
M = data.frame(A = 1:10, B = 2 * (1:10), C = 2^(1:10))  # generate a small matrix

M  # display the whole matrix
```

```
##     A  B    C
## 1   1  2    2
## 2   2  4    4
## 3   3  6    8
## 4   4  8   16
## 5   5 10   32
## 6   6 12   64
## 7   7 14  128
## 8   8 16  256
## 9   9 18  512
## 10 10 20 1024
```

```
H = head(M, 2)  # take only the first two rows, all columns

H  # 'H' now contains the first two rows of 'M'
```

```
##   A B C
## 1 1 2 2
## 2 2 4 4
```

```r
T = tail(M, 2)  # take only the last two rows, all columns

T  # 'T' now contains the last two lines of 'M'
```

```
##     A  B    C
## 9   9 18  512
## 10 10 20 1024
```

```r
M[5, ]  # take only row 5, all columns
```

```
##   A  B  C
## 5 5 10 32
```

```r
M[, 2]  # take only column 2, all rows
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20
```

```r
M[1:3, ]  # take rows 1-3, all columns
```

```
##   A B C
## 1 1 2 2
## 2 2 4 4
## 3 3 6 8
```

```r
M[c(1, 5, 9), ]  # take all columns for rows 1, 5, and 9
```

```
##   A  B   C
## 1 1  2   2
## 5 5 10  32
## 9 9 18 512
```

```r
M[c(1:3, 6), c("B", "C")]  # take only columns 'B' and 'C' in rows 1, 2, 3, and 6
```

```
##    B  C
## 1  2  2
## 2  4  4
## 3  6  8
## 6 12 64
```

```r
M[M$C > 10, ]  # take all columns for the rows in which the value in column 'C' is > 10
```

```
##     A  B    C
## 4   4  8   16
## 5   5 10   32
## 6   6 12   64
## 7   7 14  128
## 8   8 16  256
## 9   9 18  512
## 10 10 20 1024
```

```
M[M$C > max(M$B), ]  # take all the columns for the rows in which the value in column 'C' is greater th
```

```
##     A  B    C
## 5   5 10   32
## 6   6 12   64
## 7   7 14  128
## 8   8 16  256
## 9   9 18  512
## 10 10 20 1024
```

### Operating on Rows or Columns of Data

### Functions Used

- `apply()` # to apply a function to the rows or columns of a data.frame or matrix
- `colMeans()` # to take the mean of a column of a matrix or data.frame
- `colSums()` # to sum over the columns of a matrix or data.frame
- `head()` # to see the first lines of an object
- `median()` # get the median of the values in an object

There is a family of special functions for operating on columns of data such as `colSums()` and `colMeans()`. There are similar functions for rows, e.g. `rowSums()`, `rowMeans`.

```
head(geneData, 2)  # take a look at the data
```

```
##                        A        B        C        D       E       F
## AFFX-MurIL2_at   192.742  85.7533 176.7570 135.5750 64.4939 76.3569
## AFFX-MurIL10_at   97.137 126.1960  77.9216  93.3713 24.3986 85.5088
##                        G       H       I        J       K       L       M
## AFFX-MurIL2_at  160.5050 65.9631 56.9039 135.6080 63.4432 78.2126 83.0943
## AFFX-MurIL10_at  98.9086 81.6932 97.8015  90.4838 70.5733 94.5418 75.3455
##                        N       O       P        Q       R       S       T
## AFFX-MurIL2_at  89.3372 91.0615 95.9377 179.8450 152.467 180.834 85.4146
## AFFX-MurIL10_at 68.5827 87.4050 84.4581  87.6806 108.032 134.263 91.4031
##                         U        V       W        X       Y        Z
## AFFX-MurIL2_at  157.98900 146.8000 93.8829 103.8550 64.4340 175.6150
## AFFX-MurIL10_at  -8.68811  85.0212 79.2998  71.6552 64.2369  78.7068
```

```
colSums(geneData)  # generate column sums
```

```
##        A        B        C        D        E        F        G        H
## 166829.3 182053.0 162499.9 152959.7 175137.9 160830.8 158051.3 172265.5
##        I        J        K        L        M        N        O        P
## 173159.1 151272.5 194615.7 169560.6 141382.6 186999.0 148209.7 193736.1
##        Q        R        S        T        U        V        W        X
## 171587.1 142461.2 168612.8 183098.2 143974.6 146235.8 168407.3 170357.9
##        Y        Z
## 155219.8 186972.4
```

```
colMeans(geneData)  # generate column means
```

```
##        A        B        C        D        E        F        G        H
## 333.6586 364.1061 324.9998 305.9194 350.2758 321.6617 316.1027 344.5310
##        I        J        K        L        M        N        O        P
## 346.3181 302.5450 389.2314 339.1213 282.7653 373.9980 296.4194 387.4722
##        Q        R        S        T        U        V        W        X
## 343.1743 284.9225 337.2257 366.1964 287.9493 292.4716 336.8147 340.7157
##        Y        Z
## 310.4396 373.9449
```

The `apply()` function is more general as it allows one to apply a function to the rows or the columns of a`matrix` or `data.frame`. Similar functions `lapply()` and `sapply()` can be used on the elements of a `list`.

```
head(apply(geneData, 1, median))  # medians of the rows (second parameter=1)
```

```
##  AFFX-MurIL2_at AFFX-MurIL10_at  AFFX-MurIL4_at  AFFX-MurFAS_at
##        94.91030        85.26500        20.52205        12.85995
##  AFFX-BioB-5_at  AFFX-BioB-M_at
##        46.23955        49.71095
```

```
head(apply(geneData, 2, median))  # medians of the columns (second parameter=2)
```

```
##        A        B        C        D        E        F
## 79.85995 69.02255 79.29065 77.40295 65.89240 61.82655
```

### Graphing Data

### Functions Used

- `layout()` # set the layout of the graphics window
- `matrix()` # create a new matrix
- `barplot()` # create a bar plot from a vector, matrix, or data.frame
- `png()` # open a *PNG* file in which to store a graphic–until dev.off() graphic output is sent to this file
- `dev.off()` # write a graphic to a file and close the file

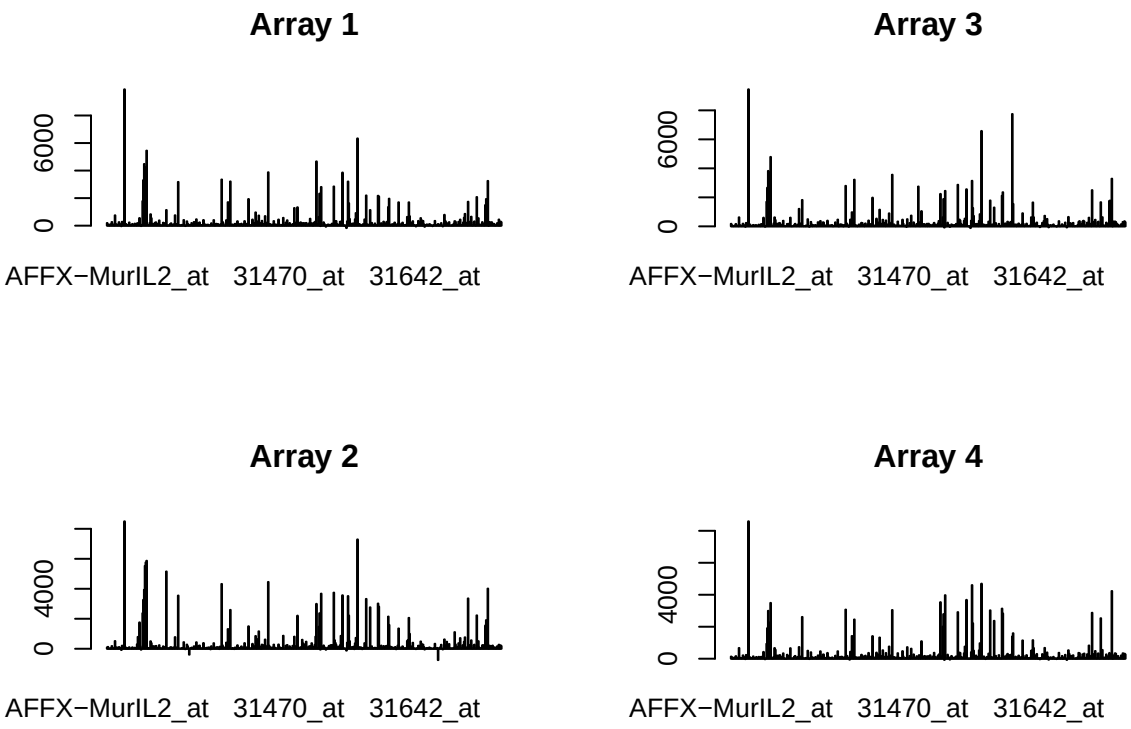
R can produce a variety of graphs. In most cases, the syntax required for a basic graph is simple and the result informative with intelligent defaults for labels. However, you can control nearly every aspect of the plot to produce just the graph you want. Here we'll make a quick set of 4 `barplot()`'s on one page. To do this we will use a simple `for` loop.

```
layout(matrix(c(1:4), 2, 2))  # use a matrix to specify that graphs will be placed into a 2 x 2 grid in
```

```
matrix(c(1:4), 2, 2)  # here is the layout matrix
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```
for (j in 1:4) {
    # let j vary over the range 1 to 4

    barplot(geneData[, j], main = paste("Array", j))  # barplot the data in all the rows of column 'j'
}
```

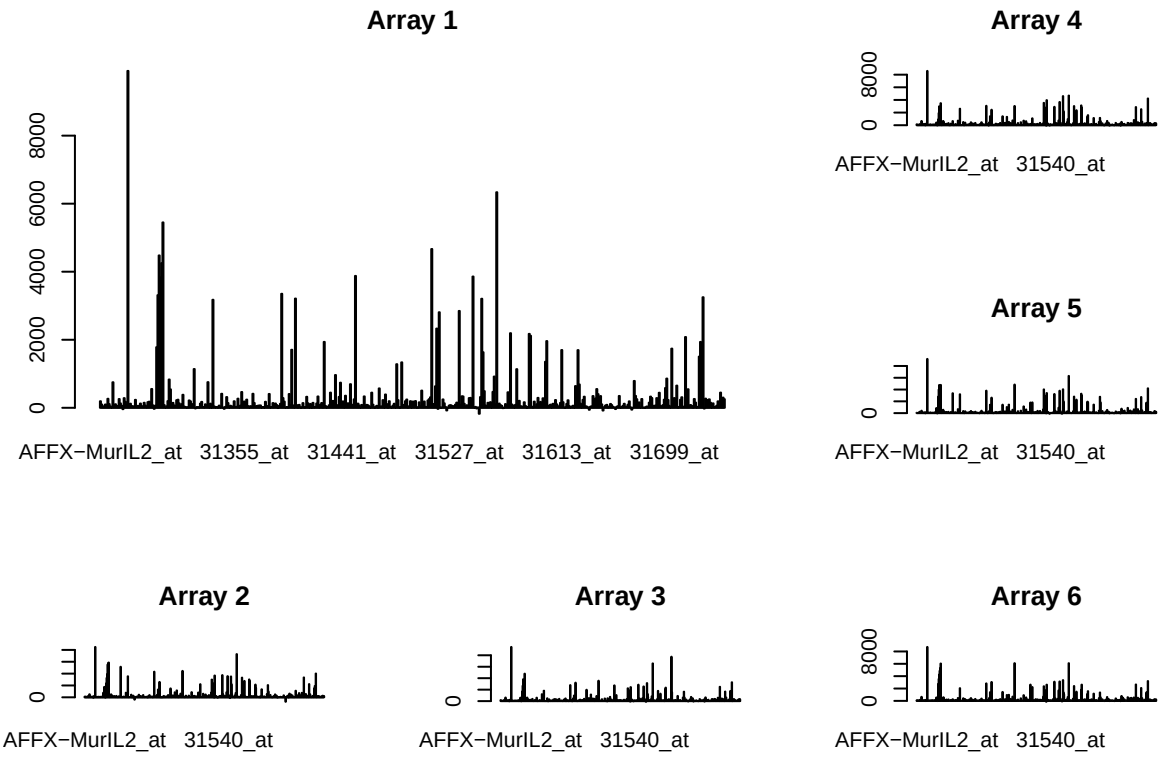



```
# partition the page into a 3 x 3 grid and give sectors 1,2,4,5 to plot 1;
# sector 3 to plot 2, sector 6 to plot 3, sector 7 to plot 4, sector 8 to
# plot 5 and sector 9 to plot 6

layout(matrix(c(1, 1, 2, 1, 1, 3, 4, 5, 6), 3, 3))

matrix(c(1, 1, 2, 1, 1, 3, 4, 5, 6), 3, 3)  # here is the layout matrix
```

### A More Complex Layout

```
##      [,1] [,2] [,3]
## [1,]    1    1    4
## [2,]    1    1    5
## [3,]    2    3    6
```

```
for (j in 1:6) {
    # let j vary over the range 1 to 6

    barplot(geneData[, j], main = paste("Array", j))  # barplot the data in all the rows of column 'j'

}
```




Plots can be redirected to a **PNG** or **PDF** file and re-sized in the process if need be.

```
png("plot1", width = 500, height = 500)  # open a graphics device that will put subsequent graphics int

dev.off()  # write the graphics file and close the graphics device
```

```
## pdf
##   2
```

### Performing a T-test

- `t.test()` # perform a T-test
- `data.frame()` # create a new data.frame
- `str()` # display the internal structure of an object
- `attr()` # extract data stored as an attribute within an object

We've already performed a couple of T-tests in passing, but let's revisit `t.test()` in a little more detail. The function `t.test()` returns an object with a complex structure much like the linear model (`lm()`) object we've already seen. With the `t.test()` return object there are several parameters that we may want to extract for further use.

```
D = data.frame(score = c(100, 110, 95, 70, 75, 65), group = c("A", "A", "A",
    "B", "B", "B"))  # create a little data

D  # check that it is OK
```

```
##   score group
## 1   100     A
```

```
## 2   110     A
## 3    95     A
## 4    70     B
## 5    75     B
## 6    65     B
```

```
T = t.test(score ~ group, D)  # run the T-test on the two groups, 'A', and 'B'

T  # display the result
```

```
##
##  Welch Two Sample t-test
##
## data:  score by group
## t = 6.0083, df = 3.448, p-value = 0.006133
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  16.06225 47.27109
## sample estimates:
## mean in group A mean in group B
##        101.6667         70.0000
```

```
str(T)  # get the structure of 'T'
```

```
## List of 9
##  $ statistic  : Named num 6.01
##   ..- attr(*, "names")= chr "t"
##  $ parameter  : Named num 3.45
##   ..- attr(*, "names")= chr "df"
##  $ p.value    : num 0.00613
##  $ conf.int   : atomic [1:2] 16.1 47.3
##   ..- attr(*, "conf.level")= num 0.95
##  $ estimate   : Named num [1:2] 102 70
##   ..- attr(*, "names")= chr [1:2] "mean in group A" "mean in group B"
##  $ null.value : Named num 0
##   ..- attr(*, "names")= chr "difference in means"
##  $ alternative: chr "two.sided"
##  $ method     : chr "Welch Two Sample t-test"
##  $ data.name  : chr "score by group"
##  - attr(*, "class")= chr "htest"
```

```
T$statistic  # now we can see how to get at the parts of the output--here is the T-value
```

```
##        t
## 6.008328
```

```
T$p.value  # here is the associated P-value
```

```
## [1] 0.006133406
```

```
T$conf.int  # and here is the confidence interval for the difference between the means of the two sampl
```

```
## [1] 16.06225 47.27109
## attr(,"conf.level")
## [1] 0.95
```

```
attr(T$conf.int, "conf.level")  # here we extract the confidence level of the interval
```

```
## [1] 0.95
```

## Project 1: A Simple Analysis of Probe Intensity Data

### Functions Used

- `abline()` # draw a straight line given a slope and intercept
- `boxplot()` # draw a box and whisker plot
- `cutree()` # cut a dendrogram produce by `hclust()` to divide the leaves into groups
- `dist()` # create a distance matrix for the rows of a matrix–this matrix is the input for `hclust()`
- `factor()` # create a new factor object
- `hclust()` # create hierarchical clusters given a distance matrix
- `head()` # see the first few lines of an object
- `hist()` # compute and plot a histogram
- `layout()` # set the layout for the graphics window
- `levels()` # see or set the levels of a factor
- `plot()` # plot an object using an object-specific method
- `rownames()` # get the vector of row names for a matrix or data.frame
- `sum()` # get the sum of the values in an object
- `t()` # generate the transpose of a matrix
- `vector()` # create a new vector

### The Data

Let's put together what we've learned to perform a simple, and yes, somewhat superficial analysis of some probe intensity data. We'll take you through a much more detailed analysis tomorrow. We'll analyze `geneData` which is a little expression matrix with 26 samples and 500 probe features. We're going to generate a hypothesis to test by looking for groupings among the 26 samples. To do that we will cluster the samples on the basis of the similarities in the intensities they give over the 500 probes.

To begin with, we will use `dist()` to generate a distance matrix `d` which is the result of an all against-all-comparison between the rows of a matrix. This matrix will then be used to perform hierarchical clustering on the rows of our matrix. However, since we want to cluster the samples, which are in the columns, we will have to feed `dist()` the transpose of `geneData`. To do that we will use the transpose function, `t()`.

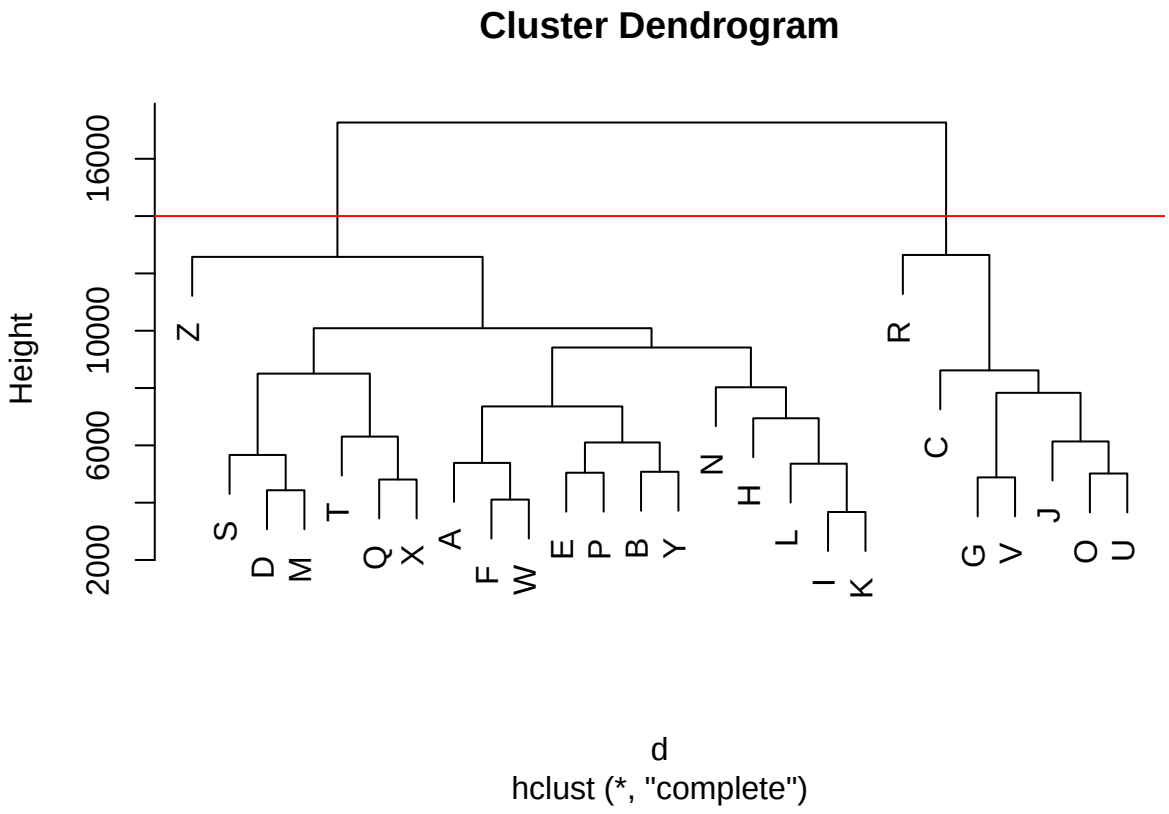
### Generation of a Distance Matrix for the Samples

```
d = dist(t(geneData))  # transpose geneData so the columns (samples) become the rows and compute the di
```

We next generate the clustering from `d` and plot the resulting cluster object, `h`. The dendrogram clearly shows two major groups that we can partition by drawing a red line at 14,000.

### Performing the Clustering

```
h = hclust(d)  # perform the hierarchical clustering and put the result in `h`

plot(h)  # plot the dendrogram

abline(h = 14000, col = "red")  # add a red line to the plot we've already made to show how to split th
```




### Deriving Groups from Our Cluster Tree

We can use use `cutree()` to cut the dendrogram at 14,000 and generate the two groups that we will then compare.

```
groups = cutree(h, h = 14000)  # cut the dedrogram and generate a 26 position vector giving group membe

groups  # here are our groups
```

```
## A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
## 1 1 2 1 1 1 2 1 1 2 1 1 1 1 2 1 1 2 1 1 2 2 1 1 1 1
```

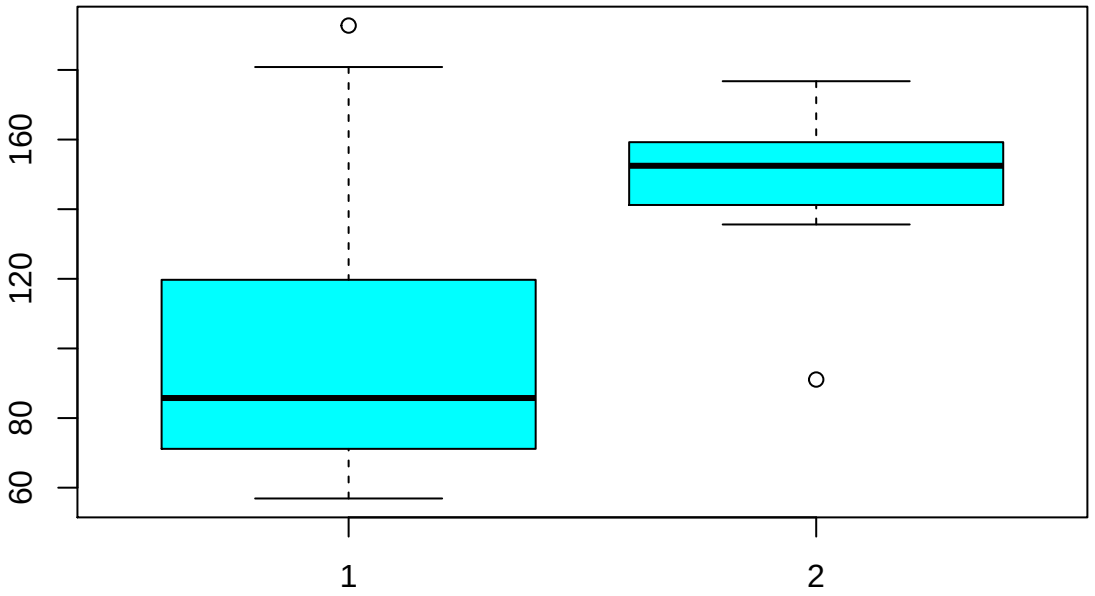
```
groups = factor(groups)  # make the group vector a factor (not strictly required but good practice)

levels(groups)  # check the levels--there should be only 2
```

```
## [1] "1" "2"
```

### Graphically Comparing Groups

Now we can explore a bit. Let's generate a quick box plot for probe 1 and give it some color.

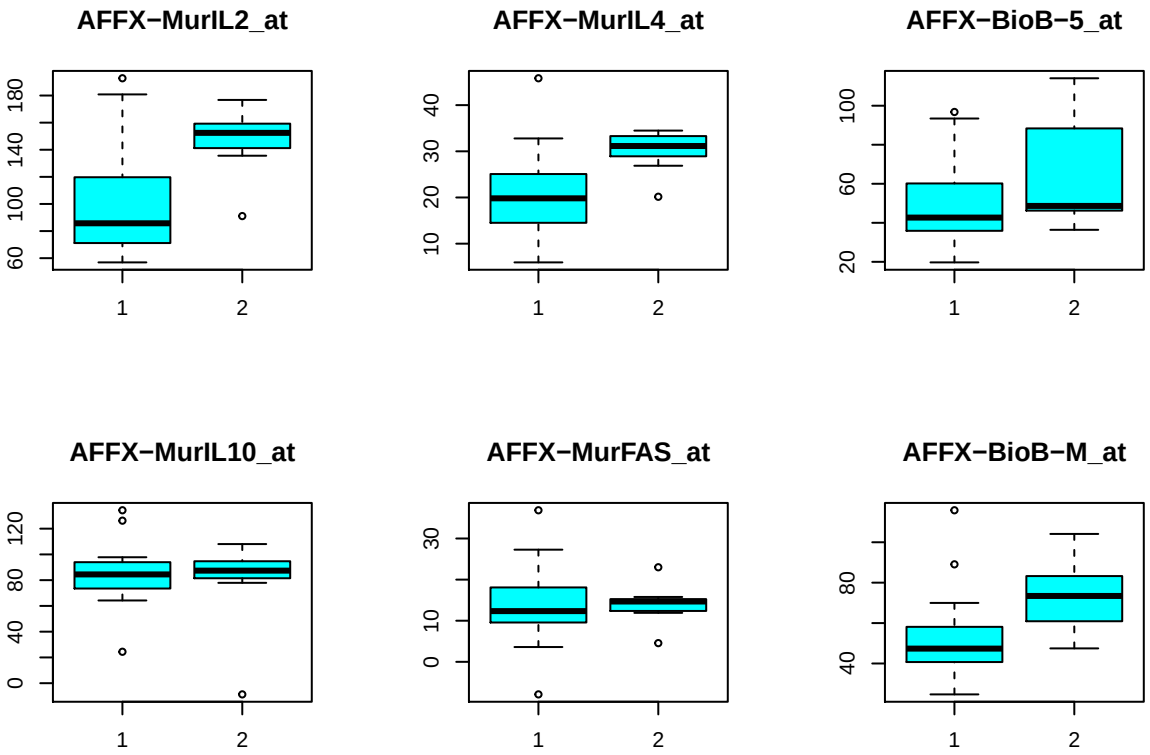
```
boxplot(geneData[1, ] ~ groups, col = "cyan")  # make a boxplot for the probe 1 intensities, grouped by
```




The box plot for probe 1 shows a distinct difference in intensity between the two groups of arrays.

We can plot a few more box plots using `layout()`.

```
layout(matrix(c(1, 2, 3, 4, 5, 6), 2, 3))  # plot 6 graphs on one page


# loop through probes 1 to 6, boxplot, and give a title based on the probe
# names (rownames)

for (j in 1:6) {
    boxplot(geneData[j, ] ~ groups, main = rownames(geneData)[j], col = "cyan")
}
```



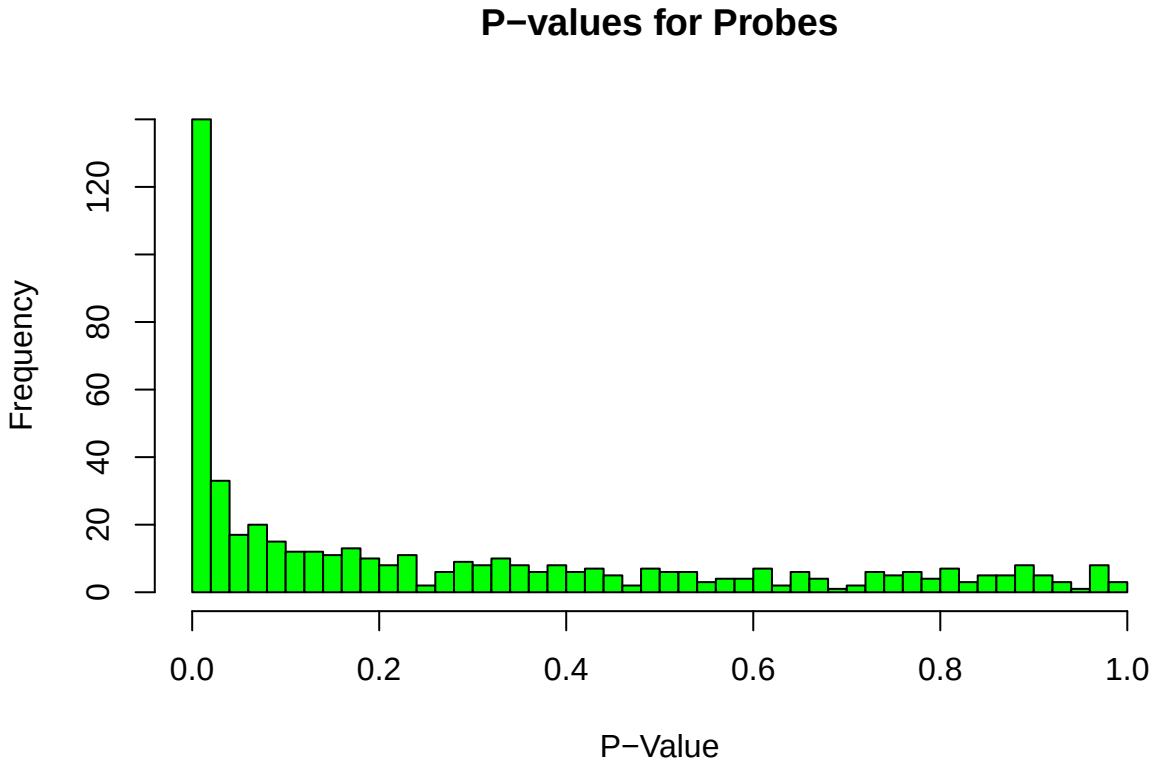

We can also run a T-test on the two groups for each of the 500 probes. To do so we need a loop and we need to extract the P-value from each `t.test()` return object we get.

### Performing T-tests on the Groups

```
T = vector()  # initialize an empty vector

# loop from 1 to the end of rownames (all the probes), run the T-test and
# take only the P-value from the results

for (j in 1:length(rownames(geneData))) {
    T[j] = t.test(geneData[j, ] ~ groups)$p.value
}
```

. . . and plot the results as a histogram of P-values.

```
# T is a vector and we can easily make a histogram of its values to see
# where we stand

hist(T, breaks = 50, main = "P-values for Probes", xlab = "P-Value", col = "green")
```




Now to find the significant P-values. Testing for $P <= .05$ yields a logical array in which the `TRUE` values correspond to probes with significant P-values. Summing over the array (remember, `TRUE` evaluates to `1` while `FALSE` evaluates to `0`) gives us the count of significant probes.

```
head(T < 0.05)  # see a bit of the logical array we are about to use
```

```
## [1]  TRUE FALSE  TRUE FALSE FALSE  TRUE
```

```
sum(T < 0.05)  # get the sum, which is the number of TRUE values
```

```
## [1] 181
```

## Project 2: A MA-Plot in 4 Steps

### Functions Used

- `abline()` # draw a straight line given a slope and intercept
- `abs()` # get the absolute value of a number
- `apply()` # apply a function to the rows or columns of a matrix or data.frame
- `ceiling()` # round a number up to the next integer
- `colnames()` # get the columns names of a matrix or data.frame
- `dim()` # get the dimensions of a matrix or data.frame
- `function()` # create a function, defining its input parameters
- `head()` # see the first lines of an object
- `hist()` # compute and plot a histogram for the values in an object
- `log2()` # get the base 2 logarithm of a number
- `max()` # get the maximum of the values in an object

- `median()` # get the median of the values in an object
- `plot()` # initialize the graphics window plot a set of points–previous graphics are erased
- `round()` # round a number to an integer
- `sum()` # get the sum of the values of an object
- `sum()` # sum over the values in an object
- `topo.colors` # generate a color array of as many levels as we like using the topographical color scheme

To further consolidate what we've learned, let's create a function to generate a MA-Plot for the sample microarray data. We'll do it in several stages. Once again, we'll use `genedata`.

As a reminder, the data set *geneData* is a matrix of 500 probe intensity values for 26 microarrays. The array names are the letters A-Z, while the probe names are Affymetrix ids. We can verify this easily enough:

```
dim(geneData)  # get the dimensions of geneData
```

```
## [1] 500  26
```

```
head(geneData, 2)  # see two lines of it
```

```
##                        A        B        C        D       E       F
## AFFX-MurIL2_at  192.742  85.7533 176.7570 135.5750 64.4939 76.3569
## AFFX-MurIL10_at  97.137 126.1960  77.9216  93.3713 24.3986 85.5088
##                        G       H       I        J       K       L       M
## AFFX-MurIL2_at  160.5050 65.9631 56.9039 135.6080 63.4432 78.2126 83.0943
## AFFX-MurIL10_at  98.9086 81.6932 97.8015  90.4838 70.5733 94.5418 75.3455
##                        N       O       P        Q       R       S       T
## AFFX-MurIL2_at  89.3372 91.0615 95.9377 179.8450 152.467 180.834 85.4146
## AFFX-MurIL10_at 68.5827 87.4050 84.4581  87.6806 108.032 134.263 91.4031
##                         U        V       W        X       Y        Z
## AFFX-MurIL2_at  157.98900 146.8000 93.8829 103.8550 64.4340 175.6150
## AFFX-MurIL10_at  -8.68811  85.0212 79.2998  71.6552 64.2369  78.7068
```

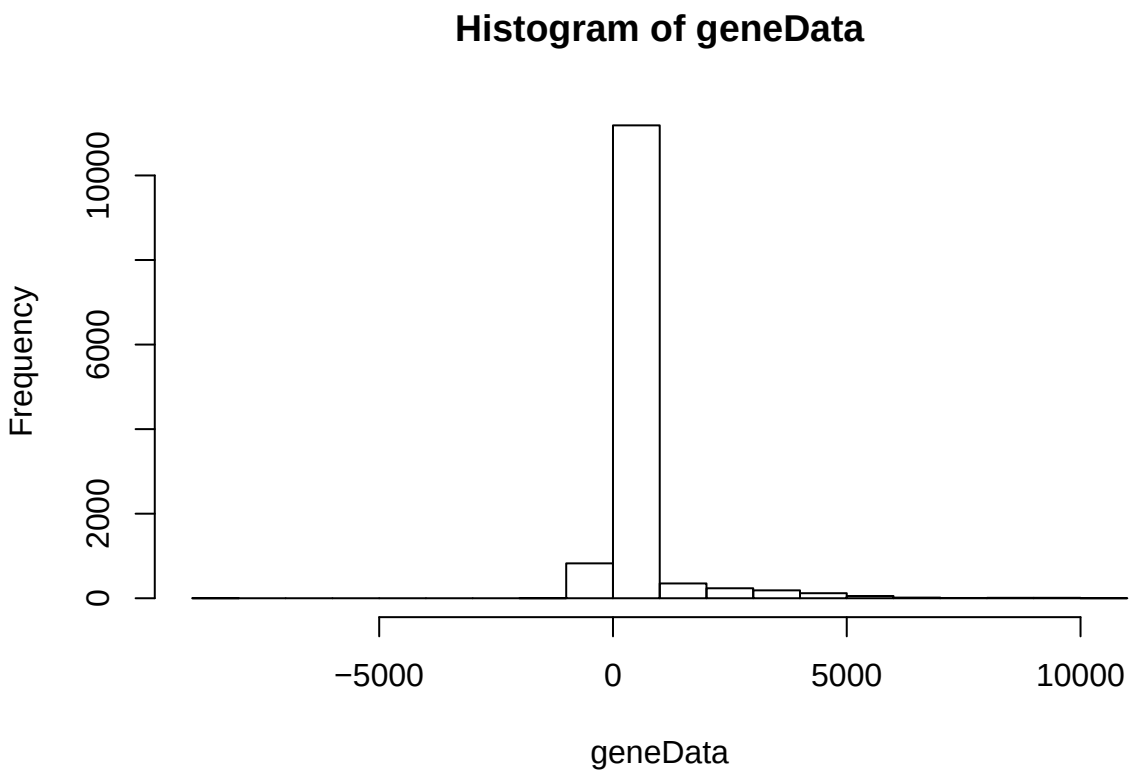
```
colnames(geneData)  # get the sample names (they are the names of the columns)
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q"
## [18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

```
head(rownames(geneData))  # look at some of the probe names (they are the names of the rows)
```

```
## [1] "AFFX-MurIL2_at"  "AFFX-MurIL10_at" "AFFX-MurIL4_at"  "AFFX-MurFAS_at"
## [5] "AFFX-BioB-5_at"  "AFFX-BioB-M_at"
```

```
hist(geneData)  # let's see how the intensitites look before proceding
```




There are a relatively small number of negative intensities showing up in the histogram. For purposes of this demonstration we will simply eliminate them as they will generate errors latter on.

```
sum(geneData <= 0)  # we are targeting 827 data points of 13,000 (500 x 26)
```

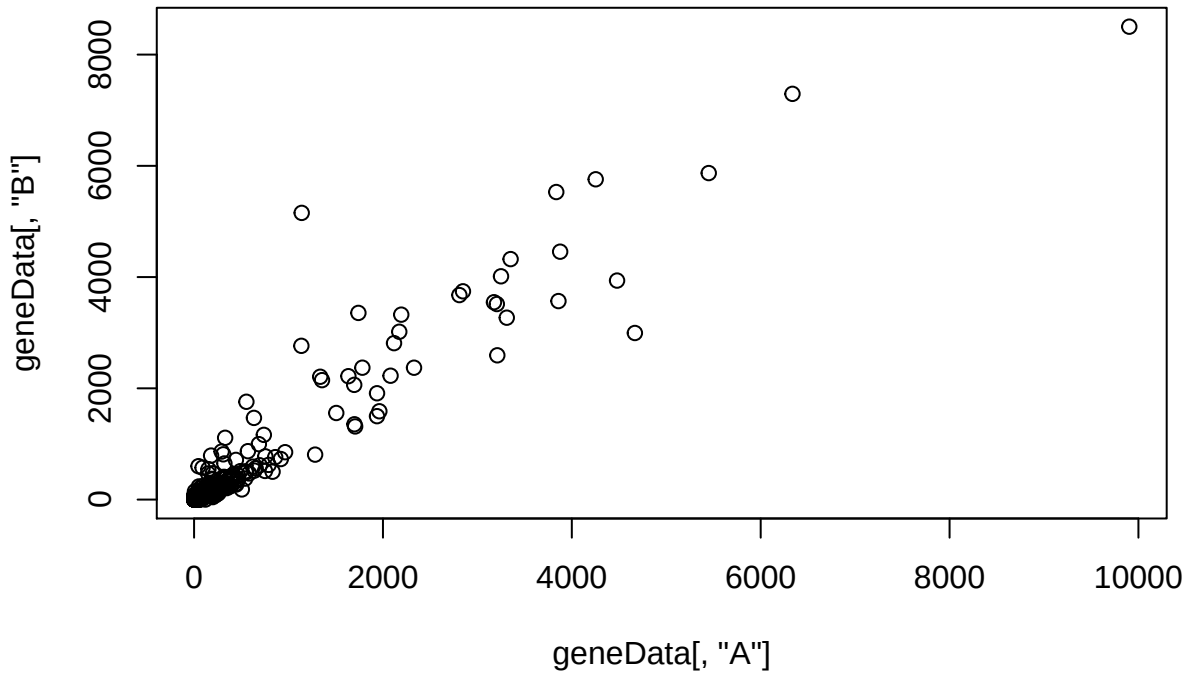
```
## [1] 827
```

```
geneData[geneData <= 0] = 1  # this does it--note that we can address a matrix with a single subscript
```

### Step 1: An X-Y Plot for Two Arrays

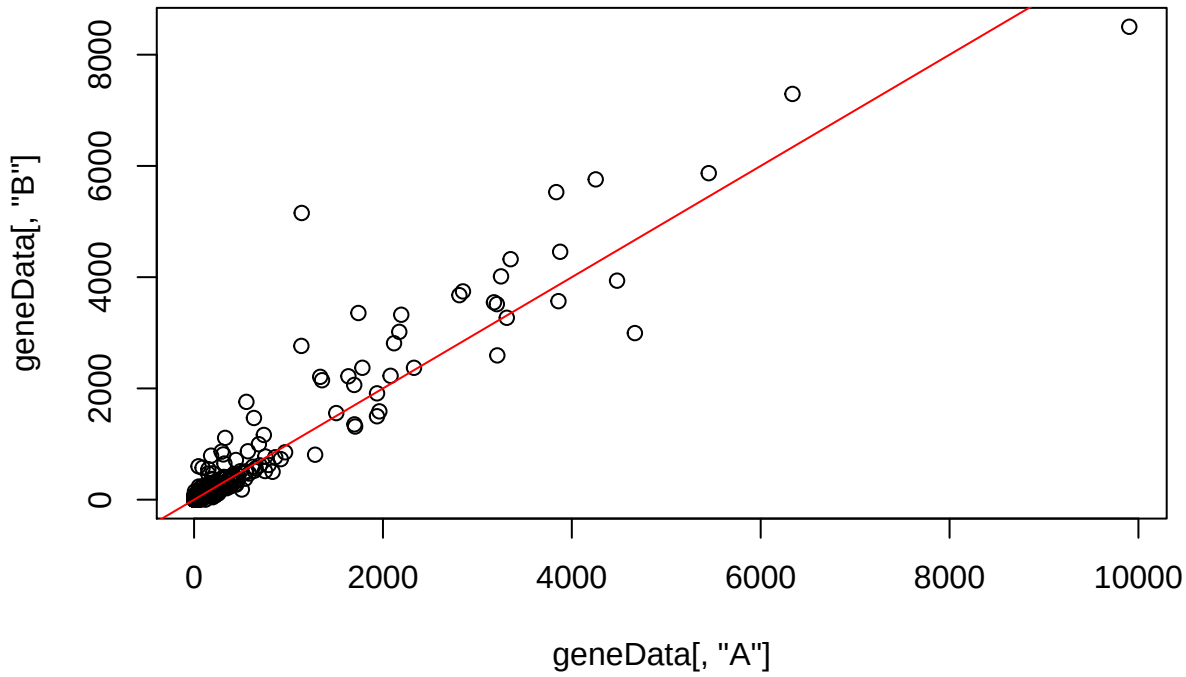
A MA-Plot is simply the plot of the corresponding the values of one data set plotted against corresponding values of another; then rotated by 45 degrees and scaled to make deviations from linearity more apparent. So to begin with, let's just plot the values of one array against those of another:

```
plot(geneData[, "A"], geneData[, "B"])  # plot the 500 probe intensities of sample 'A' against those of
```




Now, let's add a red line with a slope of 1 and intercept of 0 as a reference:

```
plot(geneData[, "A"], geneData[, "B"])  # identical to what we did before

abline(a = 0, b = 1, col = "red")  # add the red line
```

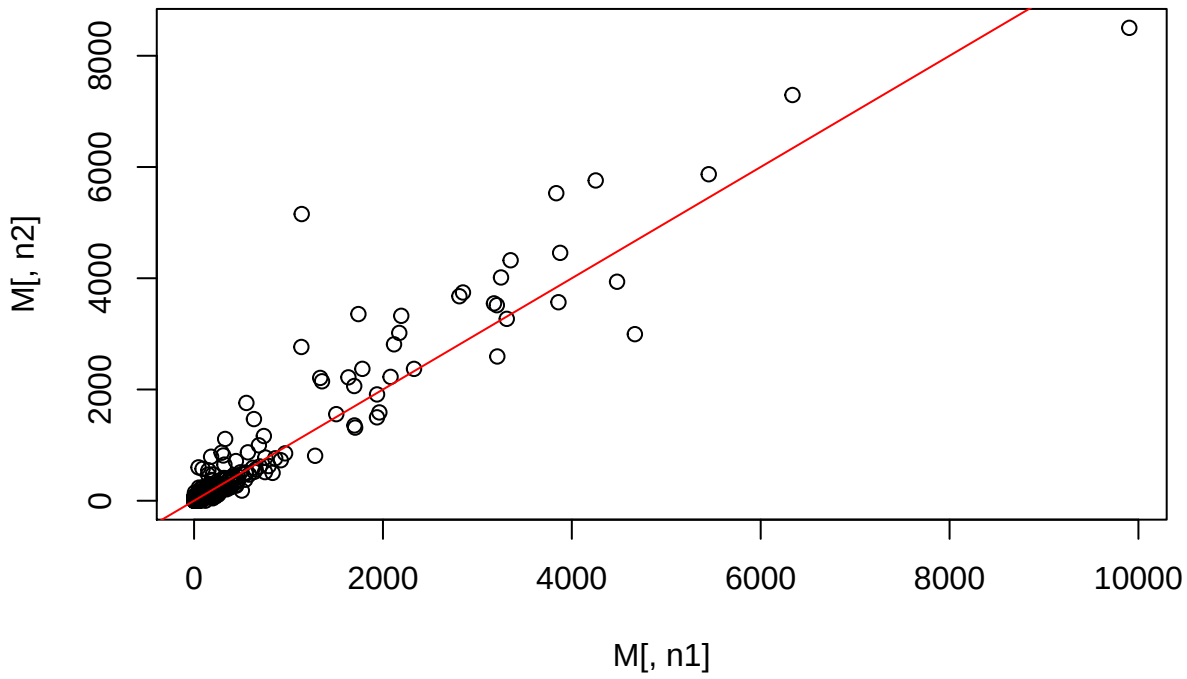



**Step 2: Package the X-Y Plot as a Function**

Before we go further, let's turn what we have into a function. To do this, we need to configure the function so that it accepts a matrix of probe intensities, `M`, as its first argument and a pair of indices, `n1, n2`, to indicate

which arrays to compare.

```
# the function takes a matrix 'M' as the first parameter and n1, n2 as the
# indexes of the arrays to plot

maplot = function(M, n1, n2) {
    # these parameters are passed to the function and we use them by name

    plot(M[, n1], M[, n2])  # we plot the n1-th column of M against the n2-th column

    abline(a = 0, b = 1, col = "red")  # and draw our line

}

# we can call the function like this

maplot(geneData, 1, 2)  # geneData becomes M, '1' becomes n1, and '2' becomes n2
```

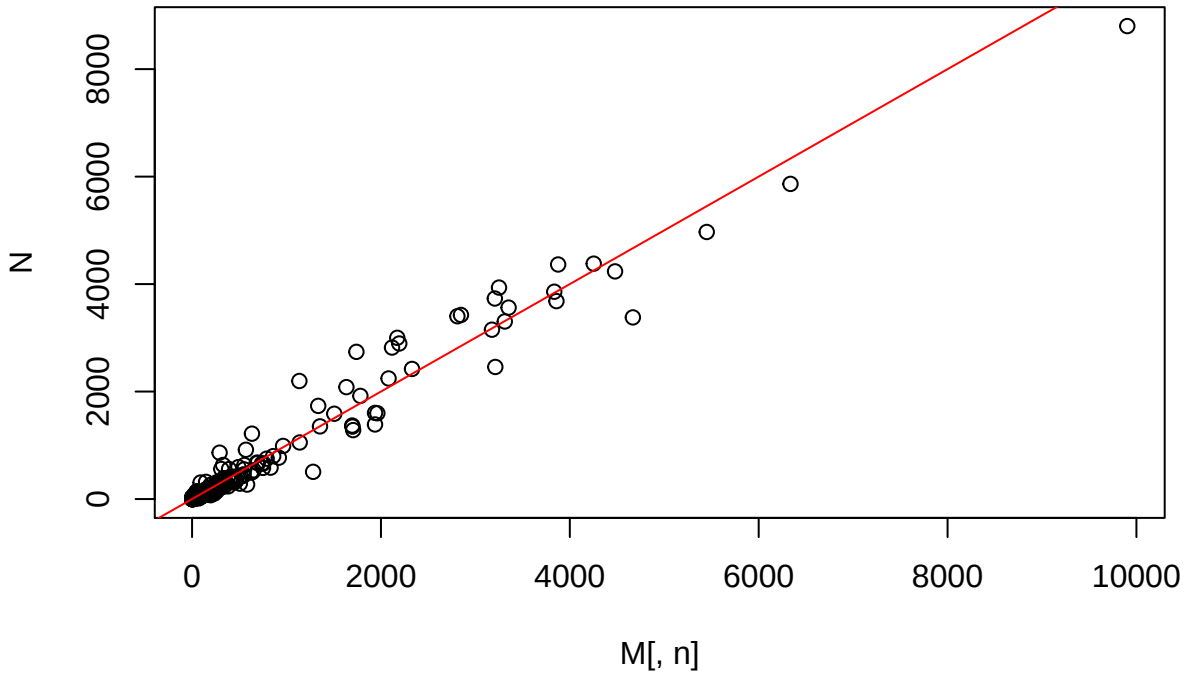



**Step 3: Create a Median Array for Comparison**

We have 26 arrays in `geneData` but we don't want to look at MA-plots for each of these pairs. Generally, we want to compare the intensities of each array to a standard such as the median intensities over all the arrays. Sound hard? Not a bit of it. This is a one-liner using `apply()`.

```
# since we are now comparing one array to the median we need only the matrix
# `M` and one array index `n`

maplot = function(M, n) {
    # n1 and n2 go; we need only a single sample name, 'n', to compare to the
    # median array
```

```
    # N is the new median array--to get it, we simply apply the median function
    # to the rows of geneData

    N = apply(M, 1, median)

    # now, use this new array in place of the second parameter in plot

    plot(M[, n], N)

    abline(a = 0, b = 1, col = "red")

}

# we can call the new function with:

maplot(geneData, 1)
```




**Step 4: Rotate and Scale the Plot**

Finally, we can re-scale and in the process rotate the plot 45 degrees. The X-axis will be the mean log2 intensity of the corresponding probe intensities for the sample and median array. The Y-axis will be the log2 ratio, or fold change for the sample intensities between the two. We can also label the plot with the letter of the array that is being compared to the median array. Our red line now becomes horizontal so we change its parameters from (a=0, b=1) to (h=0).

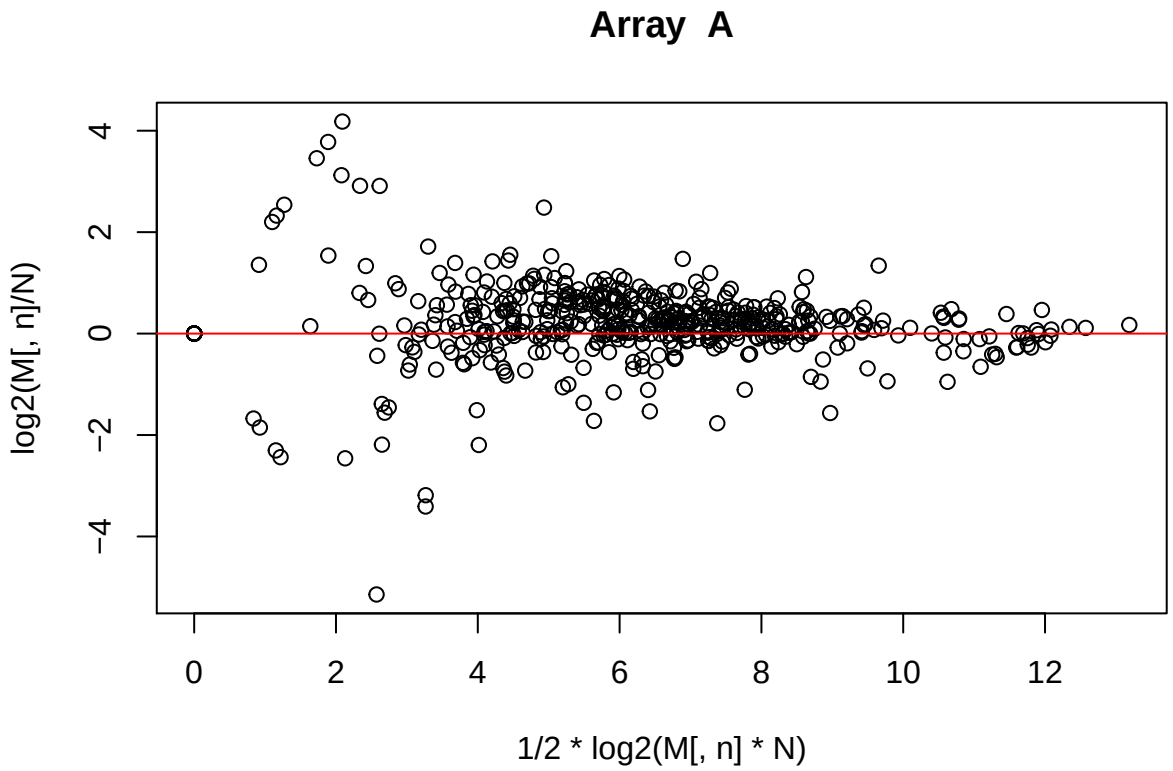
X-axis:

$$1/2 * log2(sample.array\ i\ * median.array\ i\ )$$

Y-axis:

$$log2(sample.array\ i\ /median.array\ i\ )$$

```
maplot = function(M, n) {
    # 'M' is the expression matrix, 'n' is the sample to plot against the median

    N = apply(M, 1, median)  # generate 'N', the median array

    plot(1/2 * log2(M[, n] * N), log2(M[, n]/N), main = paste("Array ", LETTERS[n]))  # plot and includ

    abline(h = 0, col = "red")  # draw a horizontal red line for reference

}

# we can now call the new function

maplot(geneData, 1)
```



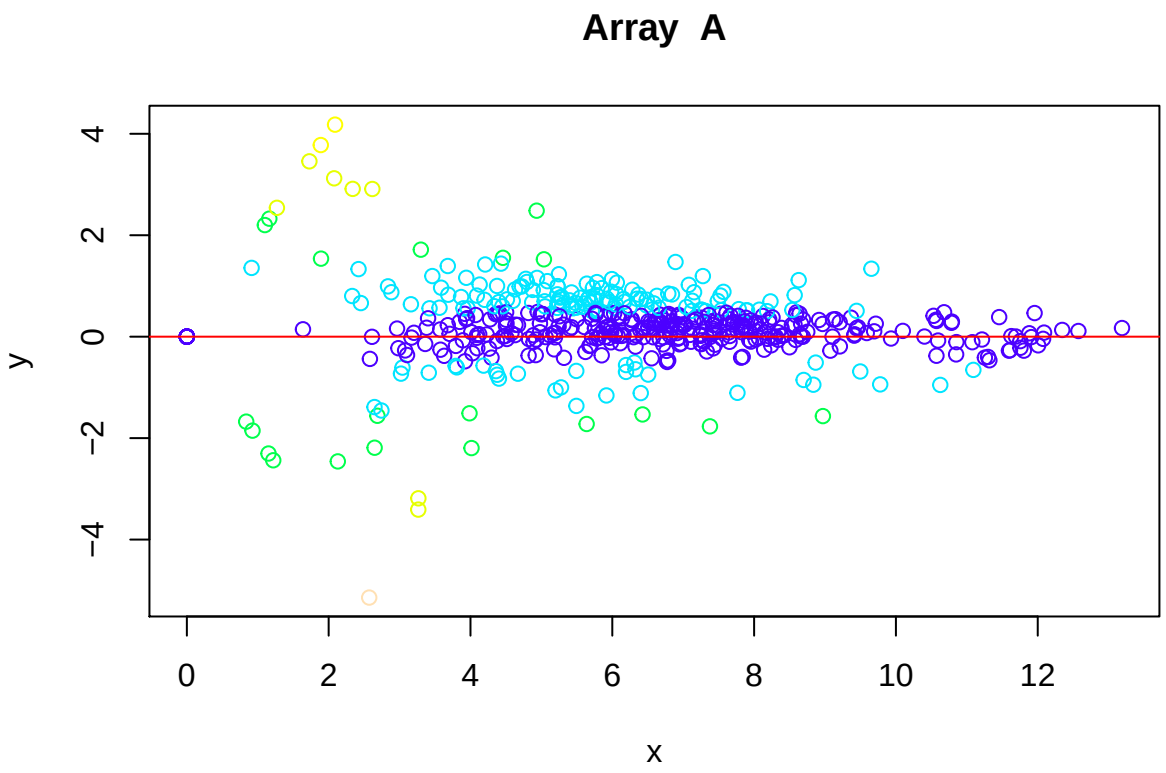

Can we color the points?

**Step 4+: Adding Color**

Of course. The technique is to create an array of the same length as the array of points to plot and assign a color to each position in this array. We can do that by first creating a color array with as many levels of color as we like–here we ask for enough to accommodate the largest absolute value in the y-array. We then set the color for each point in the y-array by rounding its absolute value and using this as the subscript into our color array to get the right color.

```
maplot = function(M, n) {
    # 'M' is the expression matrix, 'n' is the sample to plot against the median
```

```
    N = apply(M, 1, median)  # generate 'N', the median array

    x = 1/2 * log2(M[, n] * N)

    y = log2(M[, n]/N)

    color = topo.colors(ceiling(max(abs(y))))[round(abs(y)) + 1]

    plot(x, y, col = color, main = paste("Array ", LETTERS[n]))  # plot and include the sample in the t

    abline(h = 0, col = "red")  # draw a horizontal red line for reference

}

# we can now call the new function

maplot(geneData, 1)
```




## Appendix: Function Index by Frequency

Here is the tally of the functions appearing in the "Functions Used" sections. The tallies heavily reflect the content of the course but also partially reflect the essential nature of some functions. In any case, the list and the tallies may be of interest.

```
# read in the table of counts and set colnames

F = read.table("~/Rcourse/R-intro.counts", col.names = c("fun", "counts"))
```

```r
F[order(-F$count, F$fun), ]  # order the entries in F
```

```
##              fun counts
## 23        head()     10
## 53           c()      8
## 3   data.frame()      7
## 41         sum()      7
## 17       class()      5
## 65  attributes()      4
## 19      length()      4
## 6        names()      4
## 10      abline()      3
## 16    colnames()      3
## 51         dim()      3
## 85          ls()      3
## 83      matrix()      3
## 25         max()      3
## 73        plot()      3
## 2     rownames()      3
## 35         str()      3
## 77        tail()      3
## 74         abs()      2
## 21       apply()      2
## 9       boxplot()      2
## 52         cat()      2
## 76     colSums()      2
## 61        data()      2
## 13    function()      2
## 24        hist()      2
## 70       is.na()      2
## 28      layout()      2
## 14      levels()      2
## 67        mean()      2
## 37      median()      2
## 22  save.image()      2
## 4       search()      2
## 63      t.test()      2
## 81       anyNa()      1
## 78   as.matrix()      1
## 62  as.numeric()      1
## 49        attr()      1
## 82     barplot()      1
## 34       cbind()      1
## 26     ceiling()      1
## 60    colMeans()      1
## 27      cutree()      1
## 45     dev.off()      1
## 15    dimnames()      1
## 48        dist()      1
## 66        edit()      1
## 36       emacs()      1
## 32     example()      1
## 40      factor()      1
```

```
## 72         fix()      1
## 64      hclust()      1
## 56        help()      1
## 43     history()      1
## 1      library()      1
## 39        list()      1
## 54          lm()      1
## 50        load()      1
## 11 loadhistory()      1
## 18        log2()      1
## 80         min()      1
## 75        mode()      1
## 30       nchar()      1
## 84       paste()      1
## 7       paste0()      1
## 46        pico()      1
## 55         png()      1
## 29      points()      1
## 42        quit()      1
## 47       rbind()      1
## 68  read.table()      1
## 59      return()      1
## 79          rm()      1
## 38       rnorm()      1
## 20       round()      1
## 31     rowSums()      1
## 33        save()      1
## 57 savehistory()      1
## 58          sd()      1
## 8         sink()      1
## 44      source()      1
## 71     summary()      1
## 12           t()      1
## 69      vector()      1
## 5  write.table()      1
```