

# CS 5890: Installing and Using R and BioConductor

For gene expression analysis, we can use a variety of tools in R. These notes are to help students install R on their own computer (or CD or flash drive), and then run a few simple commands.

## Installing R

Run the executable file available at the following website:

<http://cran.r-project.org/bin/windows/base/R-2.3.1-win32.exe>

Follow the installation wizard that pops up. Note that you will probably choose to follow the English directions and accept the installation agreement. It's probably best to select the default "User Installation" rather than customizing. If you want, you can put R on the start menu and create a desktop icon. This will all take just a few minutes.

Note that you can install R on a CD (probably best to use a rewritable CD with nothing else on it) or a flash drive. To do this, install R on a campus computer's desktop (or somewhere else where you can get to the installed files), and don't worry about creating a start menu folder or desktop icon. Once this is done, access the `bin\Rgui.exe` file in the created folder. You might want to create a shortcut from this file to somewhere else in the created folder you can easily access. Then copy the entire created folder to your CD (or flash drive). This will allow you to run R on any machine that can read your disk. Note that some older machines might not be able to read rewritable CD's, but that shouldn't be a problem in any student computer lab on campus.

## Running R: A Simple Example

You can run R for Windows by either double-clicking on the desktop icon or the shortcut you created to access the `bin\Rgui.exe` file in the folder created during installation. This will open up an interactive environment where we will be using many features of R. Use the "Introduction to R" .pdf file available through manuals portion of the help menu in R as a good reference for the basic syntax of common commands in R. The search facility through CRAN (<http://cran.r-project.org>) is also very useful.

Consider the following simple example to look at a few commands in R. Suppose we have midterm and final exam scores for five students. Let  $x$  be the midterm score and  $y$  be the final score:

Student	1	2	3	4	5
Midterm	50	66	72	75	92
Final	65	65	89	79	97

To define these in R, use the following commands:

```
id <- c(1,2,3,4,5)
x <- c(50,66,72,75,92)
y <- c(65,65,89,79,97)
```

This will create three “vectors” named `id`, `x`, and `y`. Think of `a <- b` as being an arrow assigning the right-hand side (`b`) value to the left-hand side name (`a`). Note that we can type these commands directly into R, or type them in a text editor (like Notepad) where we can save them for later use. If we have commands in a text file, we can either copy and paste them into R, or we can reference them using the `source` command (more on that later).

Suppose instead that the data were in some external file that we wanted to read in. For example, suppose the data are in the comma-delimited file found in `a:\datafiles\scores.csv`, with three columns labeled `student`, `midterm`, and `final`. Then the following commands would do the same as the commands given above:

```
dataset <- read.table("a:\\datafiles\\scores.csv",sep="," ,header=T)
id <- dataset$student
x <- dataset$midterm
y <- dataset$final
```

Note that the `$` is like a pointer to the variable inside the data set.

Look at a simple graphical representation of these data using a scatterplot by using the `plot` command:

```
plot(x,y,main='Scores',xlab='Midterm',ylab='Final',pch=16)
?plot
```

Note that the `main`, `xlab`, `ylab`, and `pch` are options to affect the appearance of the plot, and the `?` will give a pop-up window with a description of the plot command and its options. (As an aside, the graphical abilities of R are one of its great strengths. You can create very nice graphics in a variety of formats; type `?Devices` for more.)

We could add points or lines to the plot by doing the following:

```
points(x=c(60,60),y=c(70,75),pch=1,col='red')
abline(v=70,col='blue',lty=2)
```

Note that `col` and `lty` are options to affect the appearance of the plot. The `v` is for a vertical reference line; use `h` for horizontal.

We will use the square brackets `[ ]` to subset objects in R. For example, suppose we wanted to print out the final scores for all students who scored more than 70 on the midterm.

```
t <- x > 70
# Then t is a vector of TRUE/FALSE
y[t]
## Or, do it by hand:
y; y[c(3:5)]
```

Note that the semicolon ; allows us to put multiple commands on a single line, while the pound sign # comments out a line.

It is straightforward to “program” in R using such approaches as `for` loops and `if..then` statements. For example, suppose that we wanted to define a new variable called `high` that is 1 if the student scored the highest final score and 0 otherwise. There are several ways to do this:

```
## Approach 1
high <- rep(0,5)
# So high is a vector of 0's repeated 5 times
for(i in 1:5)
  {
    if(y[i]==max(y))
      {
        high[i] <- 1
      }
  }
```

```
## Approach 2
high <- rep(0,5)
t <- y==max(y)
high[t] <- 1
```

```
## Approach 3
high <- rep(0,5)
high[which.max(y)] <- 1
```

Note that in general, explicit `for` loops and `if..then` statements are relatively inefficient (time-wise). Things will usually work fastest if we can do them with straightforward matrix manipulations.

We can also write functions to do specific things. Here is a simple function (called `sq.med`) that takes a vector and returns the square of the median:

```
sq.med <- function(v)
  {
    med <- median(v)
    med.squared <- med^2
    return(med.squared)
  }
# Call this function:
sq.med(x)
```

Note that we could include other features in this function, such as a check that the argument `v` is in fact a vector.

To quit R, type `q()`, and for now, click ‘No’ when asked if you want to save the workspace image. (Equivalently, type `q('no')`.) If you’ve done a lot of work that would take a long time to run again, and you will want to pick up where you left off in another R session, you can save the workspace image, either by saying ‘Yes’ to this option, or by an explicit command in R (either way, these `.RData` files can be quite large):

```
save(file="a:\\worksinprogress\\historyname.RData")
# Then read this back in later using:
load(file="a:\\worksinprogress\\historyname.RData")
```

## Bioconductor Packages

There are many “automatic” functions that come with R; however, most of what we’ll use in this class will require specialized functions dealing with gene expression data. These functions are included in separate “packages” that must be downloaded and installed.

Once you have installed R, you will need to get the packages necessary to do the analyses for this course. You can get them one at a time as needed during the course. (This is why you’d want to use a rewritable CD if you’re going that route for installing R.) For example, suppose you know you need the R package called `affy` (either because you know what it does or because you tried to do something in R and R stopped and told you it needed this package). There are many ways to get packages, such as installing them using the drop-down menus in R. However, you can use a pre-written function to get most packages:

```
Sys.putenv("http_proxy"="http://proxy.usu.edu:80/")
Sys.getenv("http_proxy")
# These first two lines may be necessary on-campus to access
# online resources.
source("http://www.stat.usu.edu/~jrstevens/get.packages.R")
need.pkgs <- c("affy") # or c("affy","vsn"), for example
get.packages(pkgs=need.pkgs)
```

Note that the `source` command will go to the referenced file and run the code there in R. Running this the first time may take a minute or two because R will also go install other “dependent” packages.

Once a package has been downloaded and installed, it will only be used in an R session if you explicitly request that it be loaded (to save on memory space). For example, if we want to look at an image of a particular microarray, we would first need to load the `affy` package (using the `library` function) and then call the necessary functions. To see which *CEL* files are available in a directory:

```
library(affy)
cels <- list.celfiles("C:\\folder")
cels
```

To read in particular *CEL* file(s):

```
data.0 <- ReadAffy(filenamees="C:\\folder\\celfile.CEL")
data.1 <- ReadAffy(filenamees=c("C:\\folder\\cel1.CEL",
                              "C:\\folder\\cel2.CEL"))
data.2 <- ReadAffy(celfile.path="C:\\folder")
length(data.2)
```

This creates an *AffyBatch* object in R. Its length tells how many arrays are in it.

To select a specific array from the *AffyBatch* object:

```
array.1 <- data.1[,1]
```

To look at the array image (with a title as given):

```
image(array.1,main='mouse array image')
```

To look at gene names (probe set names):

```
gn <- geneNames(data.1)
gn[1] # Name of first gene
pn <- probeNames(data.1)
pn[1:13] # Names of first 13 probes
```

But how does R know which genes (or probesets) are on this array? We need a “decoder ring.” Probe information is available in *CDF* files. R will automatically download and install the necessary packages when it encounters a *CEL* file. (Alternatively, you can download and install the packages yourself, by hand.) You can also create your own *CDF* file for custom arrays.

To look at intensities for specific probe set(s) and to put them together in one matrix:

```
pm.gn1 <- pm(data.1,gn[1])
mm.gn3 <- mm(data.1,gn[3])
mat <- cbind(pm.gn1,mm.gn3)
colnames(mat) <- c('pm1.gn1','pm2.gn1','mm1.gn3','mm2.gn3')
mat
```

A few graphical summaries of these data (four arrays here) provides motivation for normalization:

```
library(RColorBrewer)
# May need first: get.packages(pkgs="RColorBrewer")
par(mfrow=c(2,2))
cols <- brewer.pal(4, "Set3")
boxplot(data.2,col=cols,names=1:4)
hist(data.2, col=cols, xlab="Log2 intensities", lwd=2,lty=c(1,1,2,1))
legend(12,0.2,1:4,col=cols,lwd=2,lty=c(1,1,2,1))
```