

Seminar III: R/Bioconductor

Leonardo Collado Torres

`lcollado@lcg.unam.mx`

Bachelor in Genomic Sciences

`www.lcg.unam.mx/~lcollado/`

August - December, 2009

Reviewing how to use R

Welcome

Basic R intro

Finding help

R objects and structures

Reading files into R

Basic R plots

Reviewing how to use R

Flow control

Exercises

And so it begins

- ▶ First 32hr Bioconductor only course at LCG
- ▶ BioC2009 as an inspiration source
- ▶ All the material in English and Spanish
- ▶ Classes in English: Bioc and OCW
- ▶ Assistants: Alejandro, José and Víctor
- ▶ Official course page:
<http://www.lcg.unam.mx/~lcollado/B/>
- ▶ Remember to ask for help through the forum

Course syllabus

- ▶ Objectives
- ▶ Project: search for Bioconductor papers.
- ▶ A Sample Class
- ▶ Evaluation
- ▶ *Tentative* class calendar

Course info

- ▶ The course is meant as a Bioconductor overview.
- ▶ Several Bioconductor experts looked at the syllabus and gave us pointers!
- ▶ The calendar is directly linked to *Bioinformatics and Statistics / course*. Biostrings case.
- ▶ Trying to get an expert to visit us :)

Video recording

- ▶ This course is LCG's pilot for a complete OpenCourseWare course.
- ▶ All classes will be recorded: thanks to the UATI group!
- ▶ So, **English** at all times
- ▶ One week lag

R background

- ▶ R is an open-source implementation of the S language: Becker, Wilks and Chambers. S-PLUS is a private one.
- ▶ Created by Ross Ihaka and Robert Gentleman¹
- ▶ It's an interpreted language and *lives* on the interpretation moment.
- ▶ Useful as a programming environment: plots, statistics, packages such as the biological (genomic) ones from Bioconductor.
- ▶ Six month release cycle: stable and devel versions.
- ▶ R is multi-plataform: Windows, Linux/Unix and Mac.
- ▶ R Core and the Comprehensive R Archive Network (CRAN)
<http://cran.r-project.org>

¹He also created the Bioconductor project

Installing R

- ▶ For **Windows** and **Mac**, basically download the base binary from CRAN, double click on it and follow the instructions.
 - ▶ Windows stable and Mac stable releases.
- ▶ For **Linux/Unix**, it will depend on the flavor you have. Say you have Ubuntu, then you need to follow [these instructions](#) to get the latest stable version as `sudo apt-get install r-base` is generally not updated to the latest version.
- ▶ For this course you'll need the R devel version which currently is named 2.10.0devel and Bioconductor release 2.5.
 - ▶ Installed on Monteban (Windows) and will soon be installed on the Solaris servers.

A basic R session

- ▶ We highly recommend you to use **Emacs** or **XEmacs** for your R work. At the very least use a text editor and copy paste your commands².
- ▶ Either type R on your terminal or double click on the R icon. Basic info shows up.
- ▶ You can simply use R as a calculator, so type in some commands :)

```
> 2 + 3 * 5
```

```
[1] 17
```

```
> 2^3
```

```
[1] 8
```

A basic R session

```
> 6/3
```

```
[1] 2
```

```
> sqrt(pi)
```

```
[1] 1.772454
```

```
> exp(log(5))
```

```
[1] 5
```

- ▶ You can insert comments into your code by using the `#` symbol.
- ▶ Quit by using the `q` or `quit` function.

```
> q("no")
```

²In windows you can use the R GUI script editor and run commands by using CTRL + R.

Workspace and history

Sometimes you need to interrupt your work, so saving your R objects, history and/or session is useful.

- ▶ You can **save** and **load** objects by specifying the objects, path and file name into a **.Rda** file.

```
> save(object1, object2, file = file.path("folder",  
+     "file.Rda"))  
> load(file = file.path("folder",  
+     "file.Rda"))
```

- ▶ To view your recent commands use the **history** function. You can save and load your history using **savehistory** and **loadhistory**.

Workspace and history

```
> history()
> savehistory(file = file.path("folder",
+   "file.Rhistory"))
> loadhistory(file = file.path("folder",
+   "file.Rhistory"))
```

- ▶ You can save your session into a .Rdata file by specifying so when quitting or by using the `save.image` function and use `load` to reload it.

```
> q(save = "yes")
> save.image(file = file.path("folder",
+   "file.Rdata"))
> load(file = file.path("folder",
+   "file.Rdata"))
```

Workspace and history

- ▶ While working, you might need to change your working directory or view what's in there. Functions such as `getwd`, `setwd`, `list.files()` and `dir()` will be most helpful.

R help

There are a lot of ways to get help in R. I mention some below.

- ▶ The most basic help function is simply, `help`. I generally use its shortcut: `?`

```
> help(quit)  
> `?`(q)
```
- ▶ Another great help tool is to start the help browser by using `help.start`. During the same session, the help pages will open in your browser.

```
> help.start()
```
- ▶ I also use the `apropos` and `args` quite frequently. The first one lists all the functions whose name includes your query and the second one lists the arguments of a function.

```
> apropos("history")
```

R help

```
[1] "history"      "loadhistory"
```

```
[3] "savehistory"
```

```
> args(savehistory)
```

```
function (file = ".Rhistory")
```

```
NULL
```

- ▶ If you want to search on the R web site, you can use **RSiteSearch**. For example:

```
> RSiteSearch("help")
```
- ▶ For a specific package, you can also view some basic information using the following syntax. Try it out with the package stats.

```
> library(help = packagename)
```


R help

- ▶ Another excellent tool is to use the R mailing list
<https://stat.ethz.ch/mailman/listinfo/r-help>
- ▶ Spend some time reading the *posting guide*. Using the function `sessionInfo` is very important here.

```
> sessionInfo()
```

```
R version 2.10.0 Under development (unstable) (2009-07-
i686-pc-linux-gnu
```

```
locale:
```

```
[1] LC_CTYPE=en_US.UTF-8
[2] LC_NUMERIC=C
[3] LC_TIME=en_US.UTF-8
[4] LC_COLLATE=en_US.UTF-8
```

R help

```
[5] LC_MONETARY=C
[6] LC_MESSAGES=en_US.UTF-8
[7] LC_PAPER=en_US.UTF-8
[8] LC_NAME=C
[9] LC_ADDRESS=C
[10] LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8
[12] LC_IDENTIFICATION=C
```

attached base packages:

```
[1] stats      graphics  grDevices
[4] utils      datasets  methods
[7] base
```

Objects

- ▶ Everything in R is an object and they can be named with numbers, letters, period and underscore³.
- ▶ Assigning a value to a variable⁴, is done with the `<-` operator or alternatively with `=`. However, a best practice is to use `=` only inside functions and argument definitions.
- ▶ Any object has a *class* such as `integer` and can have *attributes* which you can attach and manipulate by using the `attr` function. To view them use the `attributes` function.

```
> x <- 1:10  
> names(x) <- letters[1:10]  
> attributes(x)
```

Objects

```
$names
```

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

```
[10] "j"
```

- ▶ As for the functions, they can have different *methods* and R supports two object oriented-programming systems OOP (S3 and S4) but we won't get into them.

³It can't start with the last two

⁴Which creates an object

Vectors

- ▶ It's the most basic data structure in R. You can create one by using the **most** used R function... **c**

```
> x <- c("hola", seq(0, 25, by = 5),  
+       TRUE)  
> x
```

```
[1] "hola" "0"      "5"      "10"     "15"  
[6] "20"   "25"     "TRUE"
```

- ▶ What is the class of the object `x`?
- ▶ *Atomic vectors* contain all values of the same type such as integers, doubles, logicals or character strings.

Vectors

```
> y <- c(NA, sample(rep(c(TRUE, FALSE),  
+      10), 4))
```

```
> y
```

```
[1]    NA  TRUE  TRUE  TRUE FALSE
```

- Is y an atomic vector?

A curious parenthesis

- ▶ Type⁵ the following code:

```
> a <- sqrt(2)
```

```
> a * a == 2
```

```
> a * a - 2
```

- ▶ What do you notice?

⁵The R code is available on the official course website

Factors

- ▶ They are useful for when you have data that can be categorized. For example, kids, adults and elderly people.

```
> f <- sample(c("kid", "adult", "elderly"),  
+           10, replace = T)  
> f <- factor(f)  
> f
```

```
[1] elderly kid      kid      adult  
[5] elderly adult    kid      kid  
[9] adult  adult
```

```
Levels: adult elderly kid
```

- ▶ You can also create ordered factors by using the **ordered** function.

Lists

- ▶ It's a vector-like object that can hold different types of data including other R objects.

```
> x <- list(name = "Leonardo", age = 22,  
+          x = c(TRUE, FALSE, NA))  
> x
```

```
$name
```

```
[1] "Leonardo"
```

```
$age
```

```
[1] 22
```

```
$x
```

```
[1] TRUE FALSE NA
```

Lists

```
> names(x)
[1] "name" "age"  "x"

> x$age
[1] 22

> x[[3]]
[1] TRUE FALSE NA

> y <- "name"
> x[[y]]
[1] "Leonardo"
```

Data frames and matrices

- ▶ You can define a *matrix* by using the `matrix` function or by changing the dimensions of a vector with `dim`. All the values have to be of the same type.

```
> x <- 1:4  
> dim(x) <- c(2, 2)  
> x[, 2]  
  
[1] 3 4
```

- ▶ *Data frames* are rectangular just like matrices but every column (variable) can hold different types of data.

```
> students <- data.frame(age = 18:21,  
+   height = 170:173, passed = c(TRUE,  
+   FALSE, TRUE, TRUE))  
> students
```

Data frames and matrices

	age	height	passed
1	18	170	TRUE
2	19	171	FALSE
3	20	172	TRUE
4	21	173	TRUE

Basis

- ▶ The two basic functions for reading files into R are `scan` and `read.table`. For example, `read.csv` is analog to a type of `read.table`. Check their help files for more details.
- ▶ Lets read the `stats.txt` file which contains information on several contigs.

```
> contigs <- read.table(file = file.path("../../data",  
+   "stats.txt"), header = T)
```

- ▶ The above line works fine for me, but my file path is different from yours.⁶ We can solve this simply by reading the file from the web :)

```
> contigs <- read.table(file = file.path("http://www.lcg.unam.mx/~lcollado/  
+   "stats.txt"), header = T)
```

⁶We use the `file.path` function to be platform independent

Exploring your object

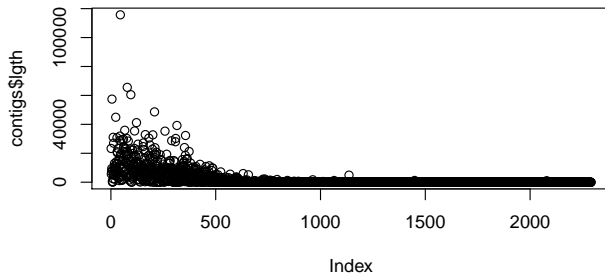
- ▶ Once we have read a file, there are some functions which can help us explore our new object.
- ▶ Try them out :)
 - > `class(contigs)`
 - > `object.size(contigs)`
 - > `names(contigs)`
 - > `head(contigs)`
 - > `tail(contigs)`
 - > `dim(contigs)`
 - > `summary(contigs$lgth)`

Basis

- ▶ R is quite strong for plotting data fast.
- ▶ Some plotting functions start a new graphic while others plot on top of a previous graph.
- ▶ Most arguments are passed as ... You can learn more about graphical parameters with `?par`
- ▶ <http://www.harding.edu/fmccown/R/> is quite useful for beginner tips.
- ▶ Plots are a *crucial* part of doing **Exploratory Data Analysis**

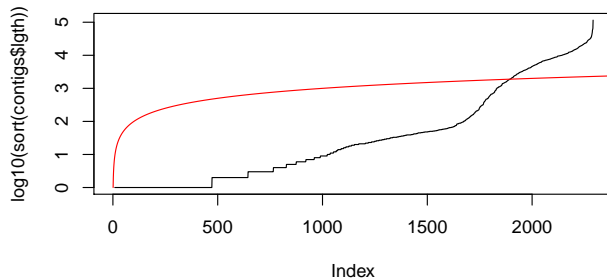
Plot

```
> plot(contigs$lgth)
```



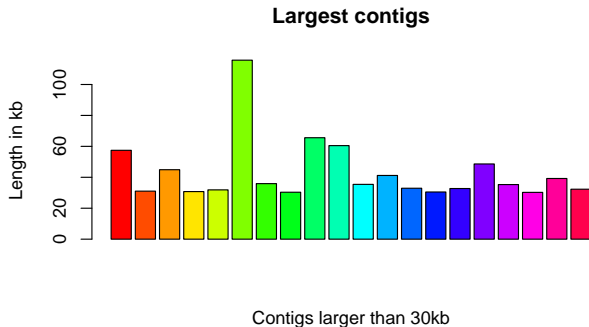
Lines

```
> plot(log10(sort(contigs$lgth)),  
+      type = "l")  
> lines(log10(1:length(contigs$lgth)^2),  
+      col = "red")
```



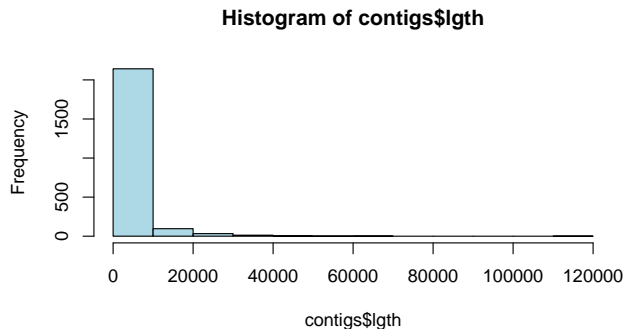
Barplot

```
> barplot(contigs$lgth[contigs$lgth >  
+ 30000]/1000, col = rainbow(length(contigs$lgth[contigs$lgth >  
+ 30000])), xlab = "Contigs larger than 30kb",  
+ ylab = "Length in kb", main = "Largest contigs")
```



Basic histogram

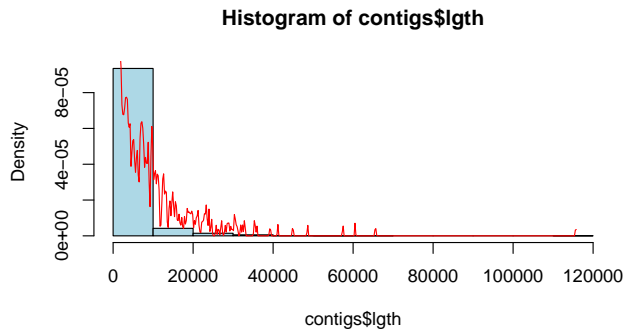
```
> hist(contigs$lgth, col = "lightblue")
```



Plotting the density

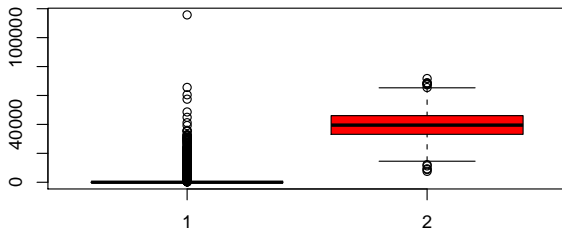
```
> hist(contigs$lgth, col = "lightblue",  
+      prob = T)  
> lines(density(contigs$lgth), col = "red")
```

Plotting the density



Graphical view of the summary

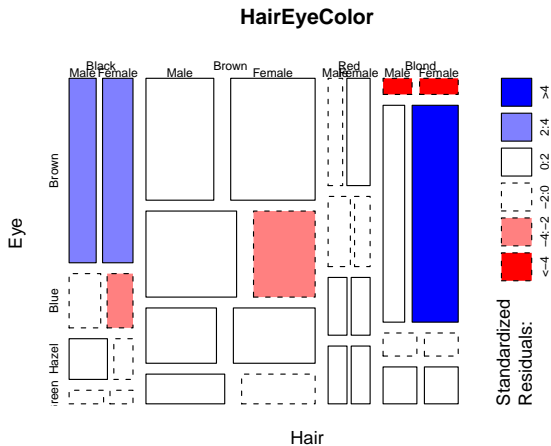
```
> boxplot(contigs$lgth, rnorm(1000,  
+ 40000, 10000), col = c("lightblue",  
+ "red"))
```



Great for table with 3 dims

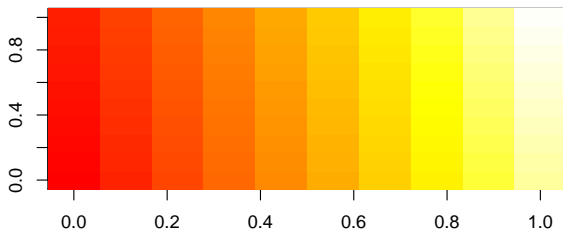
```
> mosaicplot(HairEyeColor, shade = TRUE)
```

Great for table with 3 dims



Helps visualize your matrix

```
> x <- matrix(1:100, 10, 10, byrow = T)  
> image(x, col = heat.colors(100))
```



Exporting images

- ▶ You can always export your images into PDF or PNG files.

```
> pdf(file = "file.pdf", onefile = T)
> plot("some data")
> dev.off()
> png(file = "image.png")
> plot("some data")
> dev.off()
```

Two options

- ▶ **While** is quite easy to use: `while (cond) expr`

```
> x <- NULL  
> while (length(x) < 10) {  
+   x <- c(x, runif(1))  
+ }
```

- ▶ What is the length of the x object? Now lets use **repeat** with **break**.
- ▶ With **while** and **repeat** be careful to avoid **infinite loops**!

Two options

```
> x <- 1
> repeat {
+   x <- x + 2
+   print(x)
+   if (x > 10)
+     break
+ }

[1] 3
[1] 5
[1] 7
[1] 9
[1] 11
```

An alternative

- ▶ The most widely used form of iteration is the **for** cycle: `for (var in seq) expr`

```
> for (i in seq_len(3)) print(i)
```

```
[1] 1
```

```
[1] 2
```

```
[1] 3
```

```
> for (i in letters[4:6]) print(i)
```

```
[1] "d"
```

```
[1] "e"
```

```
[1] "f"
```

- ▶ Using `seq_len` is recommended versus using `1:length(object)`

An alternative

- ▶ As you might want to use conditionals `if`, `ifelse` and `switch` could be of your interest.

Basis

- ▶ Its quite easy to write your own R functions using **function**.
- ▶ While it can take several arguments as input, it only returns **one** object which can be a vector.
- ▶ The object returned is either the last one to be evaluated or the one specified with **return**.
- ▶ Say you use an argument `x` inside a function, this one will not be related to a variable `x` outside the function.⁷

```
> x <- 5  
> y <- function(x) rnorm(x)  
> y(2)  
[1] 0.7703938 -0.3087408  
> x  
[1] 5
```

⁷For more curious users, look for guides on environments

A neat family

- ▶ Their main utility is to *apply* a function to all the elements of an object. Say all the columns of a matrix.
- ▶ In most cases, the return value is simplified and in others its an argument.
- ▶ Its easier for someone to understand a code with **apply** functions than for loops.

```
> mat <- matrix(rnorm(100), 10, 10)
```

```
> apply(mat, 1, sum)
```

```
[1] 1.7292725 1.9177798 6.8277027
```

```
[4] 2.0040560 -2.6896243 0.6864544
```

```
[7] -2.4880562 -1.2662979 -2.4098190
```

```
[10] -1.9685294
```


A neat family

- ▶ Keep in mind that some R functions are way faster than using `apply`, such as `rowMeans`.

```
> apply(mat, 1, sum) == rowSums(mat)

[1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE
[8] TRUE TRUE TRUE
```

- ▶ Some packages implement new `apply` functions, but here are the common ones:
 - ▶ `apply` Useful for matrices and data.frames
 - ▶ `lapply` Its the list version
 - ▶ `sapply` Simplest one to use (lists and vectors)

```
> x <- list(rnorm(100), runif(100),
+          rlnorm(100))
> sapply(x, quantile)
```

A neat family

	[,1]	[,2]
0%	-1.84188040	0.0006243007
25%	-0.60503591	0.2211666518
50%	0.06995335	0.4113912598
75%	0.69469081	0.7329286771
100%	2.20350408	0.9813923477

	[,3]
0%	0.05173019
25%	0.60655543
50%	1.09605235
75%	2.32080322
100%	22.31834357

- `tapply` Uses a vector and a factor, great for grouped data

A neat family

```
> x <- data.frame(info = rnorm(10),  
+   group = as.factor(sample(1:3,  
+   10, replace = T)))  
> tapply(x$info, x$group, mean)
```

```
      1      2      3  
0.9285140 -0.6002328 -0.3655565
```

- ▶ `eapply` For environments and the curious ones
- ▶ `mapply` Multivariate version of `sapply`

```
> mapply(rep, 1:4, 4:1)
```

A neat family

```
[[1]]
```

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

```
[[2]]
```

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

```
[[3]]
```

```
[1] 3 3
```

```
[[4]]
```

```
[1] 4
```

- ▶ `rapply` Recursive version of `lapply`
- ▶ You might find this site useful: [advanced_function_r.htm](#)

or homework :P

- ▶ Please go to the official course site and complete the first exercise file.
- ▶ Homework specifications are available on the Course Syllabus.
 - ▶ For this homework only hand in a portable .R file with comments. Next week we'll learn about Sweave and *vignette* files.