PDCB BioC for HTS topic
Reviewing R

LCG Leonardo Collado Torres
lcollado@wintergenomics.com – lcollado@ibt.unam.mx

August 18th, 2010

Install 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.
- We'll be using the current R version: 2.11.1 though at some point we will use R 2.12.1

Open R

Options:

- Type R on the terminal window
- Open emacs and then use `alt+X` followed by `R`

To quit R type:

```
> q()
```

and choose `no`

Get Help

- On a package:

```
> help(package = "pkgName")
```

- On a function, for example q:

```
> `?`(q)
> args(q)
```

- Find functions:

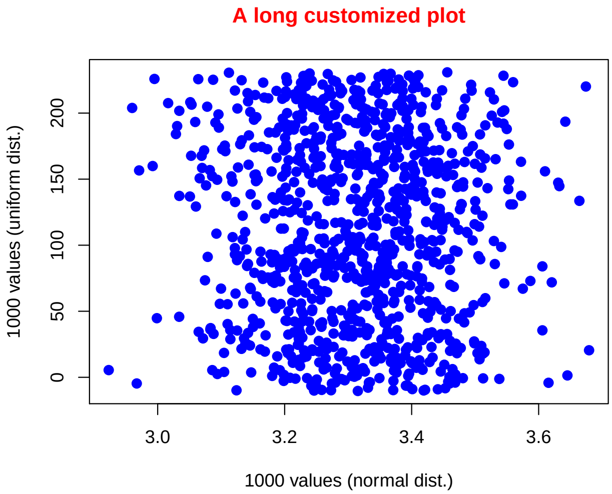
```
> apropos("session")

[1] "sessionData"
[2] "sessionInfo"
[3] "setSessionTimeLimit"
```

Use the .R files

- Whenever you see R code, instead of typing it yourself you should copy paste it from the `.R` file into your R session.
- This will save you a lot of time!

```
> plot(rnorm(1000, mean = 3.3128,
+     sd = 0.123), runif(1000, -10.74,
+     231), type = "p", lwd = 2,
+     col = "blue", main = "A long customized plot",
+     col.main = "red", pch = 19,
+     xlab = "1000 values (normal dist.)",
+     ylab = "1000 values (uniform dist.)")
```

Use the .R files



A basic R session

- I highly recommend using Emacs or XEmacs for your R work. At the very least use a text editor and copy paste your commands[1].
- 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.

[1]In 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"))
```

- 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.

Objects

- Everything in `R` is an object and they can be named with numbers, letters, period and underscore[2].
- Assigning a value to a variable[3], 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"
```

[2]It can't start with the last two

[3]Which 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 FALSE FALSE
```

- Is `y` an atomic vector?

A curious parenthesis

- Type[4] the following code:

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

- What do you notice?

[4] The 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 adult   kid     kid
 [5] elderly elderly elderly elderly
 [9] elderly 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 = 23,
+      x = c(TRUE, FALSE, NA))
> x

$name
[1] "Leonardo"

$age
[1] 23

$x
[1]  TRUE FALSE    NA
```

Lists

```
> names(x)

[1] "name" "age"  "x"

> x$age

[1] 23

> 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 funcion 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
```

Reading files into R

- 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 might work for me, but my file path is different from yours.[5] 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)
```

[5]We use the file.path function to be plataform 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 :)

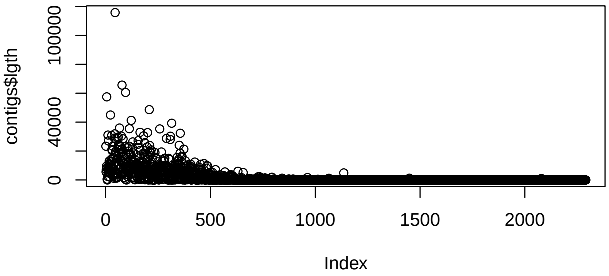
```
> class(contigs)
> object.size(contigs)
> names(contigs)
> head(contigs)
> tail(contigs)
> dim(contigs)
> summary(contigs$lgth)
```

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

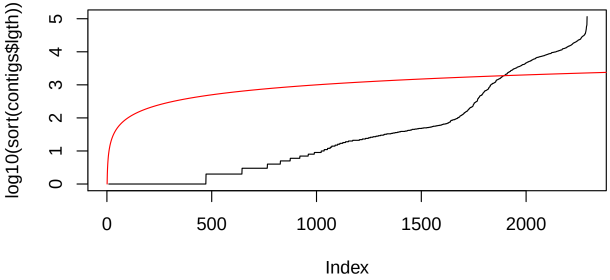
Plot

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



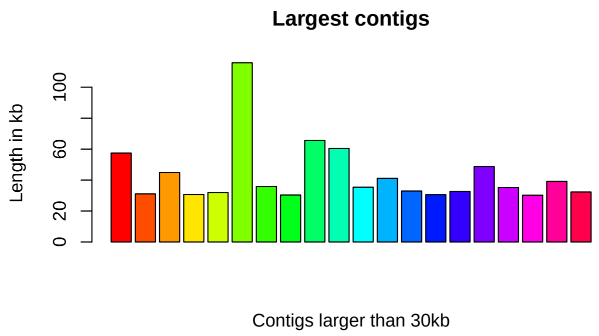
Lines

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



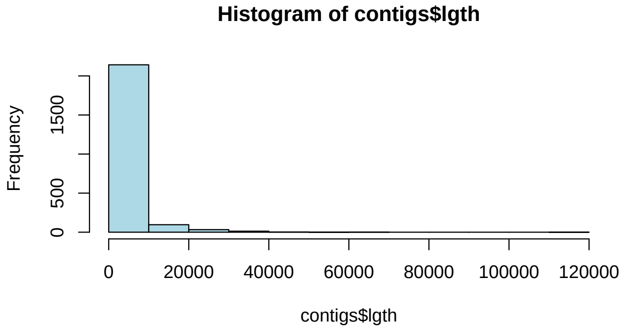
Barplot

```
> 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

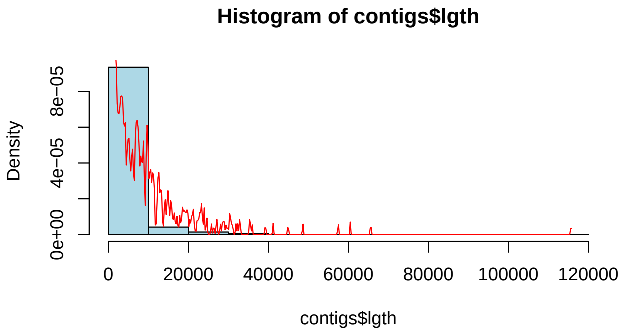
```
> hist(contigs$lgth, col = "lightblue")
```



Plotting the density

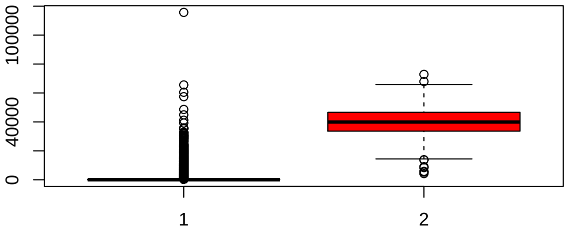
```
> hist(contigs$lgth, col = "lightblue",
+      prob = T)
> lines(density(contigs$lgth), col = "red")
```

Plotting the density



Graphical view of the summary

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



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()
```

Flow control: 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!

Flow control: two options

```
> x <- 1
> repeat {
+     x <- x + 2
+     print(x)
+     if (x > 10)
+         break
+ }
[1] 3
[1] 5
[1] 7
[1] 9
[1] 11
```

A third option

- 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)`

A third option

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

Basic function creation

- 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.[6]

```
> x <- 5
> y <- function(x) rnorm(x)
> y(2)

[1] 1.2872938 0.4782721

> x
```

Basic function creation

```
[1] 5
```

- Now lets create a function with more than one operation:

```
> z <- function(a = 1, b = 3, c = 2) {
+     res <- a * b/c
+     if (res > 2)
+         return(0)
+     else return(1)
+ }
> z()

[1] 1

> z(1, 1, 1)

[1] 1
```

Basic function creation

```
> z(2, 3, 2)
[1] 0
```

[6]For more curious users, look for guides on environments

apply: 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]  1.88159210 -0.06584425 -1.46010022
 [4] -0.26163413  1.98855126  1.21219218
 [7]  0.62494290  2.38272053  1.98251592
[10]  2.36858136
```

apply: 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` It's the list version
  - `sapply` Simplest one to use (lists and vectors)

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

apply: a neat family

```
               [,1]      [,2]       [,3]
0%    -2.06910692 0.0459839 0.09124978
25%   -0.75307113 0.2587952 0.44579931
50%   -0.09670833 0.4673580 1.08635723
75%    0.54405495 0.7071656 1.93126399
100%   2.32897122 0.9970738 6.44270491
```

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

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

```
         1          3
-0.2546954  0.3583813
```

- eapply For environments (which we might see later on)

apply: a neat family

- mapply Multivariate version of sapply

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

[[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

Review Exercises

- Why does the following expression show a warning? This is part of what rule?

  ```
  > c(2, 3) + c(4, 5, 7)
  ```

- For all the prime numbers between 1 and 10, calculate its square root. What is the sum, median and mean?
- Read the following file[7] into R: `ftp://ftp.ebi.ac.uk/pub/databases/genome_reviews/gr2species_phage.txt` and make the following plots. Check whether using a log10 scale on the *y* axis helps.
  1. Sort the genome sizes (column 2) and plot them in a line with increasing values.
  2. Plot a histogram with a density line for the same data.

Review Exercises

3. Plot a boxplot for the differences between contigous sorted genomes. Meaning, 2nd smallest - smallest, 3rd smallest - 2nd smallest, etc.[8]
4. Make a barplot showing the 10 biggest genomes. Include the names[9] on the *x* axis and every bar has to have a different color and/or density.[10]

- What is the mean genome size for every type of replicon (column 4)? You have an atomic vector and a factor so use . . .

[7] Look for the useful function for this case

[8] You might want to use `apropos` searching for diff. . .

[9] They have to be readable

[10] The `which` function might be useful

Finally. . .

- Please check the file *Short-refcard.pdf* as it will be quite helpful :)

Session Information

```
> sessionInfo()

R version 2.12.0 Under development (unstable) (2010-07-13 r52517)
Platform: x86_64-pc-mingw32/x64 (64-bit)

locale:
[1] LC_COLLATE=English_United States.1252
[2] LC_CTYPE=English_United States.1252
[3] LC_MONETARY=English_United States.1252
[4] LC_NUMERIC=C
[5] LC_TIME=English_United States.1252

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