# LGBIO2010 – Bioinformatics

# A very brief introduction to R

## 1 Introduction

During this course you will learn how to automate computational biology tasks, by running existing programs and sometimes by adapting them. To do so, we recommend you to get familiar with the software environment provided with the R programming language.

The purpose of this document is to introduce you some of the basics of the R programming language, to give you some practice of work in the R software environment and to give you pointers to further informations.

After going through this tutorial, you should be able to write a small R program, load it and execute it to perform a simple analysis task. Through this document you will be led iteratively to the creation of a R program. The incremental steps proposed hereunder are just a suggested flow. At each step, it is recommended to reuse the code of previous steps as much as needed.

## 2 A bit of reading to start from

R is a free software environment for statistical computing and graphics[1]. It compiles and runs on a wide variety of Unix platforms, Windows and MacOS. It is available on the INGI student machines and you are welcome to install it as well on your own computer. R offers a powerful *programming language*, with loads of pre-existing *packages*. R also comes with an interpreter to execute interactively R *functions* from a R *terminal*. R also includes lots of *graphical tools* to visualize data, models and results.

The first chapter of the R tutorial[2] contains only 6 pages that **must** be read first. Delighted and brave as you are, we strongly recommend to pursue your reading till ending chapter 3 on page 15. The rest of this tutorial could be used as a quick reference guide. Many complementary sources of information are available here:

`http://stackoverflow.com/tags/r/info`

and clear answers to loads of concrete questions (not limited to R) can be found at

`http://stackoverflow.com/questions/`

[1]Check `www.r-project.org`

[2]Check `cran.r-project.org/doc/manuals/R-intro.pdf`

## 3 Your first R session

We will guide you below through the use of `R` from a simple *terminal*, that is a simple window where you type in `R` commands and get results.

An alternative approach would be to install and to use `RStudio` which is a powerful IDE[3] offering many additional functionalities: an R source editor, integrated help, package management tools, and many more... You can get a sense of the RStudio environment by watching the short R video tutorial available from Moodle (look at `A short R (and R studio) tutorial`). The `R` terminal mentioned below is just one specific window of RStudio, known as the *Console*. We let you discover the roles of the other RStudio windows by yourself, if you decide to use this IDE.

We stick now to the minimal view of a unique window. You should check first that R is installed in your computing environment. In a Unix/Linux context, launching it amounts to type `R` in a terminal. You should then get something like the following lines. The last line (starting with `>`) is the R *prompt* used to enter commands[4].

```
R version 3.4.3 (2017-11-30) -- "Kite-Eating Tree"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-redhat-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

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.

>
```

Let us first define a vector `x` made of the 10 digits:

```
> x <- 0:9
```

We can check what `x` has been assigned to[5],

```
> x
 [1] 0 1 2 3 4 5 6 7 8 9
```

Note that we could have used `=` instead of `<-` as an assignment operator[6].

```
> x = 0:9
> x
 [1] 0 1 2 3 4 5 6 7 8 9
```

[3]Integrated Development Environment.

[4]We let you translate our discourse to a MacOS or Windows context if needed.

[5]Read the documentation to figure out what `[1]` refers to.

[6]The subtle difference between both operators is beyond the scope of this introduction.

We can now multiply each element of **x** by **2**, and check the result:

```
> x <- x*2
> x
 [1]  0  2  4  6  8 10 12 14 16 18
```

An alternative way of defining a vector is through the **seq** function.

```
> y <- seq(from=1,to=20,by=2)
> y
[1]  1  3  5  7  9 11 13 15 17 19
```

From your careful reading of the R documentation, you surely recall that typing **?** in a R terminal, followed by the name of a function, returns the usage of this function. We suggest the following example.

```
> ?seq
```

We can now multiply **x** and **y** component-wise and see the result

```
> x * y
[1]   0   6  20  42  72 110 156 210 272 342
```

Beyond what you define yourself, such as **x** and **y** above or your own R functions (as you will learn soon), your R environment comes with several *installed packages* already containing various functions or datasets. To get the list of those packages, type the following.

```
> library()
```

**Some** of those packages are loaded by default in your current working environment, also known as your R session. The following command returns the list of currently *attached packages*.

```
> sessionInfo()
```

To get a sense of what a specific package, say **stats**, is doing, you simply pass the package name as an argument to the help command:

```
> help(stats)
```

If you need a function from an **installed** package **not yet attached** to your session, you should load it first. Here is how to do it for the **ape** package:

```
> library(ape)
```

This package will be available till the end of your R session or till you detach it.

```
> detach(package:ape)
```

The loading of a package should work as described above, provided this package is already part of your R installation in your system. If a package is not already installed, you will likely get an error while trying to load it.

```
> library(ape)
Error in library(ape) : there is no package called 'ape'
```

The details of package installation are beyond the scope of this brief introduction and might depend on your specific OS. Nevertheless, the basic command that you could try is **install.packages()** and pass it the name (between quotes) of the package to install as argument. For example,

```
> install.packages("ape")
```

You could then be prompted to select a CRAN Mirror site. Select `Belgium`, which is close enough, and then click `OK`. If everything has gone smoothly[7], you should now be able to load this package in your R session.

[7]Obviously, your computer must be connected to Internet to install packages from a CRAN mirror.

```
> library(ape)
```

If you do need a package not already installed and the above procedure does not work for you, please refer to your system administrator.

Before ending your first R session, you can check the list of objects currently available in your environment.

```
> ls()
[1] "x" "y"
```

Whenever you finish your R session through the `q()` command, you are offered the possibility to save your workspace.

```
> q()
Save workspace image? [y/n/c]: y
```

If you answer **y** (yes) , the **x** and **y** objects will be already defined, and assigned to the same values, the next time you launch a R session in the same directory[8]. Whenever you start a new R session, you should then see:

```
...

[Previously saved workspace restored]

>
```

Note however that the additional packages you loaded explicitly during a R session need to be reloaded whenever you start a new session[9]. Writing your own R programs will help you to automate this step. Note also that saving your environment from one R session to the next may be convenient but at times confusing if you forgot that you had already defined some objects in a previous session. Listing the available objects through `ls()` is useful in this regard. In contrast, you can also make sure that you clean things up. Here is a convenient function that you can write and call to do so.

```
# Clean up all symbols/functions defined in the Global Environment
> clean <- function() {
    env = .GlobalEnv
    rm(list=ls(envir=env),envir=env)
}
> clean()
```

[8]The environment is saved in the `.RData` file in the current directory.

[9]This is done through the use of the `library()` function. You do not need to reinstall packages every time, simply load them.

# 4 A simple genome analysis task

A fundamental task in bioinformatics is to analyze DNA sequences. To do so, one should be able to automate the access to such sequences from within R. Suppose for instance that you would like to analyze the complete mitochondrial DNA of *Homo Sapiens*.

## Accession Method 1

The most convenient way to download a sequence is to query a database with a proper accession number. The Genbank accession number of this sequence is `NC_012920`. To load the sequence, we can use the **`read.GenBank()`** function from the **`ape`** package[10].

```
> library(ape)
> list <- read.GenBank("NC_012920", as.character=TRUE)
```

The **`read.GenBank()`** function is actually returning a list of sequence(s) since one can pass a vector of accession numbers as argument. Here, we are interested in a single sequence and we should thus get the first element of the **`list`**[11]:

```
> seq <- list[[1]]
```

## Accession Method 2

Sometimes, we do not know the accession number or we have already accessed the sequence from a WEB server and saved it in a file in the FASTA format. Let us assume, this file[12] is called `Human_mtDNA.fasta` in the current working directory. We can then load it as follows.

```
> library(ape)
```

(the above command is only needed if this package has not yet been loaded)

```
> mat <- read.dna("Human_mtDNA.fasta", format="fasta", as.character=TRUE)
```

The **`read.dna`** function returns a matrix of sequence(s). Here, we are interested in a single sequence and this matrix contains a single row. It can thus be transformed into a vector of characters as follows:

```
> seq <- as.vector(mat)
```

[10]This package is primarily intended to build and manipulate phylogenetic (= evolution) trees but it includes several functions to access nucleotide sequences. You may also search for alternative R packages to automate this access.

[11]A careful reader might notice that **`seq`** and **`list`** are also the names of existing R *functions*. We use them here as the names of R *variables*. This should not be too confusing since actual function calls would require to pass some arguments between parentheses: **`seq(...)`** or **`list(...)`**.

[12]This file is accessible on the Moodle page associated to the first project.

## Accession Method 3

You might actually wonder how we got the above FASTA file to make it available on the Moodle website. The sequence of operations is as follows.

1. Go to https://www.ncbi.nlm.nih.gov/ and search for *Homo sapiens mitochondrion, complete genome*
2. Under the *Genomes* category, click on `Nucleotide`.
3. Under the *Source databases* left menu, click on `RefSeq`, the first match is the circular DNA we are interested in.
4. Click on this first match and you will get many details about this sequence. By then, you should be on the page: https://www.ncbi.nlm.nih.gov/nuccore/NC_012920.1
5. Click on `FASTA`, you should arrive to https://www.ncbi.nlm.nih.gov/nuccore/NC_012920.1?report=fasta
6. On the upper right of this last page, click `Send to` and choose `File` as destination. Finally, click on `Create File` and the `sequence.fasta` file should be downloaded on your computer.

We note that this third accession method is actually fairly manual and error prone. Hence we recommend to use primarily the first accession method to fully automate the download of sequences from within the R environment.

## Analyzing the sequence

Let us check how long is this sequence.

```
> length(seq)
[1] 16569
```

We observe that this sequence is 16,569 base pairs long. We can check the first ten letters (of the forward strand):

```
> seq[1:10]
 [1] "g" "a" "t" "c" "a" "c" "a" "g" "g" "t"
```

We would like now to count how many times a specific nucleotide appears in the sequence. One convenient approach would be to define an R function[13]. This function takes a sequence **`s`** and a specific character **`c`** as arguments. It uses the built-in **`which`** function to produce a vector of specific indices **`i`** for which **`s[i]`** is equal to the character **`c`**. The number of such matching positions is obtained as the length of this vector:

```
> count <- function (s, c) {
  length(which(s == c))
}
```

We can now look at how many **`"a"`** or **`"t"`** are found in our sequence of interest:

```
> count(seq, "a")
[1] 5124
> count(seq, "t")
[1] 4094
```

[13]This common analysis task is already implemented in existing packages. We use it here as an illustrative example to define our own R function.

## 4.1 Saving your work in a R source file

As you are unlikely to want to type again and again similar instructions in a R session, we recommend you to use a text editor[14] to store your functions (or more generally any R code) in a R source file. Call it, for example, **DNAnalysis.R**. Once it will be edited and saved, you can load it in your R session as follows.

```
> source("DNAnalysis.R")
```

## 4.2 Using graphical functions

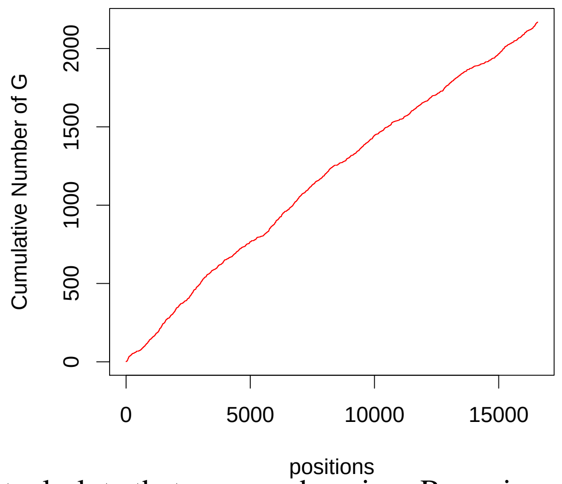
An important aspect of any sequence analysis is to get a sense of the data you manipulate. Graphical functions can be very useful in this regard.

Suppose you would like to analyze the frequency of a specific nucleotide, say **"g"** in a long sequence. One way to approach this problem is to compute how many **"g"** are seen from the beginning of the sequence till the current position **i**, while making **i** vary from 1 till the final position. A short R code would do the job:

```
> positions <- 1:length(seq)
> cumul <- cumsum(seq == "g")
```

It is now easy to plot graphically the **cumul** vector in function of the **positions** in the sequence. We decide to make such a plot with a red line and a specific label for the Y-dimension.

```
> plot (cumul ~ positions,type="l",col="red",ylab="Cumulative Number of G")
```



Note that the actual plots that you produce in a R session can be easily saved in a file to be included in another document. A convenient way to do so is to call the **pdf** function with a specific filename as first argument. For example:

```
> pdf("MyAnalysis.pdf", pointsize=18)
```

Next, you call whatever graphical functions you want to use

```
> plot (...)
```

Finally, you finish the writing to the file and close it

```
> dev.off()
```

[14]Some text editors, like *emacs*, have a convenient statistical mode to edit R programs and to execute them jointly. An interesting alternative is RStudio available at www.rstudio.com. This IDE includes a specific window where you can directly edit some R code and, for instance, execute it interactively line by line.

## 4.3 Do it yourself

You are invited to extend and to adapt the above examples to check your initial understanding of the R environment. Here is a list of possible questions you could consider.

1. Repeat the above analysis by producing a plot of how many `"a"` or `"t"` you find along the Human mitochondrial DNA.

2. Produce a plot with two curves, respectively for the cumulative number of `"a"` in blue and the cumulative number of `"t"` in red, along the positions in the sequence.

3. Download an alternative sequence, for instance, the *Lactococcus lactis subsp. lactis Il1403* (Accession number `NC_002662`) and perform a similar analysis.

4. An alternative analysis of the frequency of a particular nucleotide in a sequence would consider a sliding window through the sequence and report the observed frequency within the window for each possible position of this sliding window. Write your own R function to solve this problem. Consider 3 arguments to pass to your function: the sequence to analyze, the nucleotide of interest, the size of the sliding window.

5. Execute your R function on a sequence of interest and produce a plot of the observed frequencies for a particular nucleotide. Study how the plot evolves while varying the sliding window size.

- Feel free to interact with the teaching assistant of this course if you need help during this tutorial session.
- Throughout the course, you are primarily invited to post your questions on the **student forum** on Moodle. Students are very welcome to *answer questions* from other students on this forum. We will moderate or complement them if needed.