FEDERAL UNIVERSITY OF RIO GRANDE DO SUL - UFRGS
NORTH COAST CAMPUS
CENTER FOR COASTAL, LIMNOLOGICAL AND MARINE STUDIES - CECLIMAR

Introduction to R statistical software

Prof. Matias do Nascimento Ritter (CECLIMAR/UFRGS)
Prof. Ng Haig They (CECLIMAR/UFRGS)
Prof. Eneas Konzen (CECLIMAR/UFRGS)

imbe, RS
July 2019

Center for Coastal, Limnological and Marine Studies (Ceclimar)
Av. Tramandaí, 976 Centro Imbé, RS, CEP 95625-000

The authors declare that there is no conflict of interest with respect to this publication.

The datasets used in this document are available for download from the repository **GITHUB:**

https://github.com/haigthey/Curso-Introducao-ao-software-estatistico-R

Questions, suggestions and criticisms:

Prof. Matias do Nascimento Ritter: e-mail: matias.ritter@ufrgs.br; (51) 3308-1276
Prof. Ng Haig They: e-mail: haigthey@ufrgs.br; (51) 3308-1273
Prof. Eneas Konzen: e-mail: erkonzen@gmail.com; (51) 3308-1273

OR is free software, but built from the work of many people. When using it, always cite it:

R Core Team (2019) R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL https:// www.R-project.org/.

And, always remember to quote the packages used too!!

SUMMARY

Introducing R and RStudio1

OR is at the same time a kind of language and a computational and graphic software. Two great advantages make the program very popular: the first is that it is open source software, free, free, of the GNU General Public License (GNU's Not Unix) type, that is, users can run the program as they wish, they can study how it works and adapt it to needs (free access to source codes). In addition, they can freely redistribute it and make improvements. The second is that it is highly extensible, that is, it can be used to perform any computational activity, as long as it is compatible with its capabilities. This is done by creating your own functions and packages, sets of codes/commands related to certain themes. The expression “R environment” is often used to refer to the computational space, that is, to its characteristic of being a coherent and fully planned system. The official page of Project R can be accessed through the address: https://www.r-project.org/.

OR is widely used as statistical software, but it performs many other tasks, such as bioinformatics analysis, including sequence alignment and database comparison, modeling, map production, and many others. If you can't find a package that meets a specific demand (which is very unlikely!), anyone can develop a function or package for it. Overall, R is an excellent tool for storing and manipulating data, performing calculations, performing statistical tests, exploratory analysis, and producing graphs.

During This one course we will use O R Studio (https://www.rstudio.com/products/rstudio/), which is an integrated development environment for R, a program that is based on R, but which has a friendlier and more functional interface, such as graph plotting, history, workspace management and editor windows. syntax-highlighted text that allows direct code (script) execution. And just like R, it's also open source software. RStudio features a free open version and several paid versions with extra features.

This course has the modest intention of being an introduction and facilitating the first contact with the tool, serving as a kick-off without exhausting the subject in any way. Throughout the booklet, several notes will always be placed after the hash symbol (#) or hashtag; all information from this symbol should not be typed in R, but if it is there is no problem, because R recognizes the symbol and ignores what is after it. OR is essentially a program for self-taught students, that is, you only learn by using the program and discovering ways to solve problems as they arise. The community that uses R is very large and very active and posts many solutions to problems on forums, blogs, social media and websites. Therefore, internet search tools will be your

[1] https://www.r-project.org/about.html

best teachers on this arduous but rewarding journey. Let's start?

Program download and update

R is downloaded through one of the CRAN (Comprehensive R Archive Network) mirrors. Many countries have these mirrors, including Brazil, which even has several of them.

To download, visit https://cran.r project.org/mirrors.html and select one of the mirrors. A page will appear where you can select a system compatible with the three operating systems: Windows, Linux and Mac (OS). On this page you can also find information about new versions, bug fixes, source codes, etc. Here you can also find a link to download packages (“Contributed extension packages”) in compressed format, which can later be installed directly without the need for internet.

One of the few disadvantages of R is that it does not have the possibility of updating the installed version, that is, with each new version released by the R development team, it must be installed from the beginning. The old version can be kept or removed without any conflict, and can even be used at the same time. Important: All packages from the old version must be installed in the new version if they are still needed.

RStudio can be downloaded from the RStudio Desktop Open Source page License - FREE: https://www.rstudio.com/products/rstudio/download/.

R and RStudio layout: functions and operators

One of the features of R that often intimidates new users is the graphical interface, which is done through command lines at the prompt (a window where you can type commands). However, one of the great advantages of using this feature is that the user has great control over what is being done, because with the exception of standard arguments embedded in functions, all command parameters must be specified by the user.

When the prompt is ready to receive a new command, a greater than sign (>) appears to the far left of the command line. Throughout the booklet all command lines will be indicated with this sign at the beginning. However, when copying commands to RStudio, do not copy this signal together, as the signal will be duplicated and R will report an error:

>

A command is composed of one or more functions with their respective

arguments. A function is a code that determines a computational algorithm, which may have one or more parameters to be determined or modified through the arguments, which are placed in parentheses right after the function. Some arguments are mandatory while others are optional; some mandatory have a default mode.

Example: log function

```
> log(10)
```

calculates the natural logarithm of the number 10. The number in parentheses is the only argument needed for this function.

```
 [1] 2.302585
```

This function can be modified by adding a second argument to indicate the base of the logarithm to be calculated. Let's say, for example, that you want the base 10. This is indicated by putting a comma and the value of the base you want right after the number:

```
> log(10,10)
 [1] 1

> log(100,10)
 [1] 2
```

If by any chance a (+) sign appears right after entering a command, it means that some argument or information was missing (often parentheses) and R is waiting for this to be informed. If you want to abort a started command, simply press ESC and then ENTER:

```
> log(10
+
+ )
 [1] 2.302585

> log(10
+
# press ESC
+ >
# press ENTER
>
```

OR also has basic mathematical operators that are very similar to those used in Excel® or similar: (+) for summation, (-) for subtraction, (/) for division and (*) for multiplication. They are often used within functions.

packages

The basic version of R is already automatically installed with 8 or more packages: base, compiler, datasets, graphics, grid, methods, parallel, splines, grDevices, stats, stats4, tcltk, tools, utils, and . This can be accessed via the command:

```
> subset(as.data.frame(installed.packages()),Priority %in% ("base"), select=c(Package, Priority))
```

Package	Priority
base	base
compiler	base
datasets	base
graphics	base
grDevices	base
grid	base
methods	base
parallel	base
splines	base
stats	base
stats4	base
tcltk	base
tools	base
utils	base

In RStudio the packages are installed via the drop down button in the top of the page: Tools ÿ Install packages ÿ [done directly]. can also be of package name from a command line:

```
> install.packages("vegan")
# example with Vegan package
```

Once installed, the package needs to be loaded, which is done via the library command:

```
> library(vegan)
# example with Vegan package
```

Every time R is opened after having logged out, the packages need to be loaded again. The installation of each package is done only once for a given version of R, but the loading must be done after each session opening. It is not necessary to load all packages, just what is used in that session.

scripts

Scripts are R files with sets of commands that can be executed directly in R or RStudio via shortcut keys. to your advantage

consists in the fact that the program executes each command line and already jumps to the next one, making it very fast to execute several command lines in sequence with just the keyboard shortcut keys. Without the script it is necessary to copy and paste in the command prompt all the lines one by one and press ENTER between all of them. Many changes to graphics, for example, require all the command lines to be re-executed to include the new change, so the scripts are very quick at this time.

To execute scripts in R, just place the keyboard cursor at any point on the command line to be executed and press CTRL + R. In RStudio the only difference is that it must be replaced by CTRL + ENTER.

A script can be created in either R or RStudio. In R through the File ÿ New Script drop down button; in RStudio via the drop down button “File ÿ
New File ÿ R Script”. A file with a .r extension will be generated that will be saved in the working directory and that can be opened for viewing in any text editor (Notepad or Notepad++, for example). If you already have a common notepad file with an R routine and you want to turn it into a script, just copy its contents into a New R Script and name it as you wish.

Examples: R demos

OR has a function called demo that shows examples of some packages. The examples are generally quite elaborate and include various functions combined, for example for producing complex graphics and advanced functions.

```
> demo()
# a list of available demos will appear. Choose one.

> demo(graphics)
# demo example of the graphics function from the graphics package .
Several windows with graphics and the list of
commands at the prompt. To change graphics windows, press ENTER.
```

Citation of R and its packages

To quote R, use the following command:

```
> citation()

# information about how R should be cited in publications will appear. To find the R version go
to the HELP top drop down button and select About RStudio (in RStudio) or Help ÿ About (in R).
```

To cite R in publications use:

R Core Team (2018). A: The language and environment for
statistical computing. R Foundation for Statistical
Computing, Vienna, Austria. URL https://www.R-project.org/.

The BibTeX entry for LaTeX users is

```
@Manual{,
title = {R: A Language and Environment for Statistical
Computing},
author = {{R Core Team}},
organization = {R Foundation for Statistical Computing},
address = {Vienna, Austria},
year = {2018},
url = {https://www.R-project.org/},
}
```

We have invested a lot of time and effort in creating R,
please cite it when using it for data analysis. See also
'citation("pkgname")' for
citing R packages.

When making a citation and reference, put R and the version. Example:

Citation:
Statistical analyzes were performed in the program R v. 3.5.1 (R Core Team, 2018).

Reference:

R Core Team (2018). A: THE language and environment for statistical computing .
R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

To quote the packages also use the citation function, putting between
parentheses the package name in quotes:

> citation("vegan") # example with the *Vegan* package .

Jari Oksanen, F. Guillaume Blanchet, Michael Friendly, Roeland Kindt, Pierre Legendre,
Dan McGlinn, Peter R.
Minchin, RB O'Hara, Gavin L. Simpson, Peter Solymos, M.

Henry H. Stevens, Eduard Szoecs and Helene Wagner (2018). vegan: Community Ecology Package. R package version 2.5-1. https://CRAN.R-project.org/package=vegan

OR as calculator

OR performs basic arithmetic functions with the operators already mentioned. Examples:

```
> 2 + 2
 [1] 4

> 2 - 2
 [1] 0

> 2 * 2
 [1] 4

> 2 / 2
 [1] 1

> log(64.8)
 [1] 2

> log(9+1,10)
 [1] 1
# calculate the base 10 log of the number 9 + 1

> sqrt(100)
 [1] 10
#sqrt = square root (square root)

> (9+3)/4
 [1] 3

> 10^2
 [1] 100
# powers are indicated after the caret sign

> 100^-2
 [1] 1e-04
# e-04 is a notation for 10-4
```

help menus

RStudio has a facility that is to open the help menus inside a window of the program itself. A help menu presents the function and what it does (), its main arguments (), notes about), references and even application examples. It is a very useful function (arguments descriptiondetails

to find out how a function works. To get a help menu for a function, type (?) followed by the function name:

```
> ?sum
#example with the sum function
```

Or alternatively,

```
> help(sum)
```

Look at the sum function help menu and reproduce the examples.

Creating objects in R

OR essentially operates with so-called objects. To create an object in R, you must assign a name to it, followed by an arrow or equal symbol (<- or =) pointing to the object and indicating that all content to the right of the arrow must be inserted into it. In the example below we will create an object called “a” and insert the number 5 inside it:

```
> to <- 5
> to
# typing “a” and pressing ENTER, R will display the contents of object “a”. [1] 5
```

The types of data that are inserted into objects are divided into the following classes:

i) character (categorical or qualitative)
ii) numeric (numeric)
iii) integer (integer)
iv) logical (true/false) (logical, true/false)
v) complex (complexes, related to the imaginary number “i”) - will not be covered here

Examples:

```
#character
> a <- "I love R"
> to
 [1] "I love R"

> CECLIMAR <- "Imbé"
> CECLIMA
 [1] "Imbe"
```

#categorical data must always be enclosed in quotes to avoid being confused with objects.
The quotation marks mean "read as is".

#numeric
> A <- 67.7
> A
[1] 67.7

> x <- 1
> and <- 2.5

integer
> z <- as.integer(5.4)
> z
[1] 5

> z <- as.integer(9.8)
> z
[1] 9
the as.integer function returns the integer value of the decimal number. Note that it is not a rounding function!
For this use round(9.87, 1), for example. The first round argument is the number to be rounded and the second is the desired number of decimal places.

logical
> m <- x>y
> n <- x<y
> m
[1] FALSE
the result indicates that the content of m (x>y) is false.

To query an object's class, use the class() function:

> class(m)
[1] "logical"

> class(A)
[1] "numeric"

A few important things about object names:

• OR is case sensitive (it is different from an “A” object. case sensitive). Then an object "a"

• If you name a second object with a name already used by another, R will overwrite the contents of the second object over the first.
• Never leave spaces between object names in R! If you want to nominate one

object of “R object”, put some symbol (_) / (.) between the names. Ex.: R_object or R_object. For categorical data, it's okay to leave spaces. _____

```
> R_object <- "R"
> R_object
 [1] R
```

• it is not possible to use a single number to name objects. Numbers accompanied by letters, however, can be used:

```
> Omega3 <- 50
> Omega3
 [1] 50
```

• Avoid using accents, cedillas, apostrophes, quotation marks, and other punctuation characters in object names in R. They are often very confusing and often cause error messages.

Objects, in turn, can be divided into the following classes:

i) to climb (to climb)
ii) vector (vector)
iii) array (arrangement)
iv) list (lists)
v) data frame
vi) matrix (matrix)

i) **scalar:** it is formed of a single numeric value, integer or decimal. It's already gone used in several previous examples.

Example in R:

```
> z <- 4
> z
 [1] 4
```

ii) **vector:** it is a set of numerical values or characters (letters, words). It has a single dimension and can only contain data from a single class.

Example in R:

```
> a <- c(1:100)
# vector “a”, numeric with 100 elements (numbers from 1 to 100).
```

```
> b <- c("Semester 1, "Semester 2", "Semester 3",
```

“Semester 4”) # name vector with 4 elements

iii) **array:** it is a multidimensional vector. It is built from the function array.

```
> vector1 <- c(5,9,3)
> vector2 <- c(10,11,12,13,14,15)
> array <- array(c(vector1,vector2),dim = c(3,3))
```

the c function stands for “concatenate” and will produce a vector by combining the two vectors in parentheses. The dim argument stands for “dimensions” and specifies the number of rows and columns dim=c(rows, columns).

```
> arrangement
     [,1] [,2] [,3]
[1,]    5   10   13
[2,]    9   11   14
[3,]    3   12   15
```

iv) **lists:** set of vectors, dataframes or matrices. They don't have to be the same length or the same class. It's the way most functions return results.

Example in R:

```
> n <- c(5, 6, 8) > s <- c("ab",
"bc", "cd", "de", "ef") > b <- c(FALSE, TRUE, TRUE, FALSE,
FALSE) > x <- list(n, s, b, 3) # x contains copies of n, s, b
```

v) **dataframe:** similar to the matrix, but more generic, accepting numeric and character vectors in the same set. It is the most commonly used in the natural sciences, as we usually have numerical observations associated with
categorical variables.

Example in R:

```
> n <- 1000
> a <- rnorm(n, 9, 1) # Creates a dataset with 1000 elements, with mean 9 and standard
deviation of 1.

> b <- c(rep(1, 0.4*n),rep(2, 0.6*n))

> data <- data.frame(group=b, grade1=sample(b), var1=a, var2=a+rnorm(100, 5, 2),
var3=sqrt(sample(1:n)))

> head(data)
# the head function displays the first 6 lines by default.
To show more lines, specify the n argument:
```

```
> head (data, n=6L) # default
> head (data, n=8L) # specifies the first 8 lines
```

To see the last few lines, use the tail() function, which works in a similar way.

vi) **matrix:** is: a set of vectors in rows and/or columns. all vectors must be of the same class (numeric or character, for example).

Example in R:

```
> matrix1 <- cbind(seq(1, 11, 2), rep(4, 6), 1:6)
> matrix1 # see how the matrix turned out
# the cbind (column bind) function “glues” the columns side by side. The rbind (row
bind) function pastes lines one below the other.
```

Removing and Listing Objects in R

To remove an object, use the rm() (remove) function, followed by the object's name:

```
> rm(a)
# remove object “a”.
```

Note: sometimes when inserting different data frames, but containing the same variable names, R may not know which variable it is referring to, if the attach() function is used (see below in Importing data). In this case it may be necessary to remove objects or even entire data frames so that there is no confusion in the use of variables.

To list all objects that were created in R, use the ls (list) function, followed by empty parentheses:

```
> ls()
```

data import

There are numerous ways to import data into R. Here are the most common methods, which you will probably come across using R from now on.

In R it is possible to define a working directory through the “setwd()” function. This directory, in effect, means that you are telling R both the location where your import files will be, and the folder where it will save your exported files.

With this logic, for example, you can save your datasets in your working directory, in ".csv" datasheet format, for example; that is, separated by commas. In this case, it is very simple to import a dataset into R, using the following commands:

```
> setwd("C:/Users/So-and-so/R files")
# Path of the folder with the files

> mydata <- read.csv("mydata.csv", sep=";")
> head(mydata)
```

*If your computer is configured for the US decimal system, it is likely that the command can be simplified to just:

```
> mydata <- read.csv("mydata.csv")
```

Similarly, we could still use the following command instead:

```
> read.table("mydata.csv", header=TRUE, sep=",")
# or sep=";"
```

However, there are dozens of other ways to import data into R. Another simpler method is to use the "read.table" command. In this case, you select, in the Excel® file, the data that you would like to insert in R and copy them (CTRL + C), leaving this data on the clipboard. In this case, you can add them to R with the following commands:

```
> mydata <- read.table("clipboard", header=FALSE)

# The argument header = FALSE indicates that the data does not contain a header.
Otherwise, used header=TRUE. If the first column contains the row names, include the
row.names=1 argument.

# more generic mode; data stored as "dataframe", which allows more operability. Can be used to
import data directly from an Excel® table.
```

Text files (.txt) can also be used and easily inserted into R, as shown below:

```
> mydata <- read.table("mydata.txt", header = T, sep ="\t")

# Note that the "sep=\t" argument may vary depending on how the data was entered in the
text file.
```

“t” indicates that the data is tab separated.

To import vectors, use the scan() function. In this case, the vector must be copied to the clipboard. First, type the scan function with empty parentheses and press ENTER. Then the data must be copied to the clipboard and the command CTRL + V given in R. It will list all the elements of the vector. To indicate that the pasting is finished, press ENTER.

Enter the following vector in Excel (can be in row or column):

4
5
6
7
8

```
> r <- scan()
# hit CTRL + C on vector data in Excel
1:
# will appear the number 1 followed by a colon indicating that R is waiting for paste. Press
CTRL + V

1:4
2:5
3: 6
4:7
5:8
6:

# R will indicate that 5 items have been read and it is waiting to add more. If there is
no more data to be pasted, press ENTER.

Read 5 items

> r
 [1] 4 5 6 7 8
# check the result of object r.
```

To import character arrays, use the what=”character” argument:

I
master
O
R

```
> r <- scan(what=”character”)
1: I
2: love
```

```
3: the
4: R
5:
Read 4 items

> r
 [1] "I" "love" "the" "R"
```

Some users also use the attach() function, which makes variables (columns) accessible just by typing their name on the command line. However, if we are working with two different dataframes that have the same variable names, this trick can cause problems. In this case, it may be necessary to detach one of them or in some cases even remove one of them.

```
> attach(mydata)
> detach(mydata)

*
  if there is missing data in the vector or matrix, put “NA” in place

4
5
6
AT
8

> n <- scan()
> n
 [1] 4 5 6 NA 8

> is.na(n)
 [1] FALSE FALSE FALSE TRUE FALSE
# function that returns a logical vector indicating whether there are NAs in the data.

sum(n)

> sum(n, na.rm=TRUE)# operations with vectors or other objects containing NAs need to
be specified that these are ignored.
```

Manipulation of vectors, data frames and matrices

Once we have entered the data in R, the next step is the correct manipulation of this data. To exemplify some commands, let's use a dataset that comes with the basic R package. Use the following commands:

```
> date(trees)
# This function lists all datasets
```

available. This dataset, “trees”, presents some data on circumference *(girth),* height *(height)* and volume of some black cherry trees in the USA.

If you use the head command, R will illustrate the first six lines of data:

```
> head(trees)

  Girth Height Volume
1 8.3                 70 10.3
2 8.6                 65 10.3
3 8.8 4 10.5          63 10.2
5 10.7 6              72 16.4
10.8                  81 18.8
                      83 19.7
```

Other commands that are valuable for verifying that the data has been correctly imported, or for knowing the specifics of the data we are working on, are listed below:

```
> str(trees) # Will briefly display the internal structure of an R object


 'data.frame': 31 obs. of 3 variables:
  $ Girth : num 8.3 8.6 8.8 10.5 10.7 10.8 11 11 11.1
11.2 ...
  $ Height: num 70 65 63 72 81 83 66 75 80 75 ...
  $ Volume: num 10.3 10.3 10.2 16.4 18.8 19.7 15.6 18.2
22.6 19.9 ...

> dim(trees)# number of rows and columns
 [1] 31 3

> summary(trees)# this function shows minimum, maximum, mean, median, first and
third quartiles.

      Girth Height Min.                                  Volume
            : 8.30 Min.                :63 Min. 1st          :10.20
  Qu.:11.05 1st Qu.:72 1st Qu.:19.40 Median :12.90 Median :76 Median
  :24.20 Mean :13.25 Mean :76 Mean :30.17 3rd Qu.:15.25 3rd Qu.:80
  3rd Qu.:37.30 Max.

            :20.60 Max. :77.00          :87 Max.
```

The R software allows the manipulation of different types of objects. For many types, however, the first contact will be due to specific and one-off demands.

vectors

To replace one or more values in an array, use the following command:

```
> (r <- c(4, 5, 6, 7, 7, 8))
 [1] 4 5 6 7 7 8
```

```
> r[3]<- 10 # this command indicates that the 3rd element of the vector must be replaced
by the data to the right of the arrow.
```

```
> r
 [1]       4 5 10 7                8
```

```
> r[c(3,4)] <- c(10,20) # to replace more than one value
>r
 [1]       4 5 10 20 8
```

To sort ascending:

```
> sort(r)
 [1] 4 5 8 10 20
```

To sort in descending order, just add the argument decreasing=TRUE:

```
> sort(r,decreasing=TRUE)
 [1] 20 10 8 5 4
```

To check the position (rank) of each value:

```
> order(r)
 [1] 1 2 5 3 4
```

Data frames and matrices

To exercise the most basic manipulations on a dataset, let's again using the dataset called “trees”:

```
> date(trees)
```

```
# This dataset, “trees”, presents some data on circumference (girth), height (height) and volume
of some black cherry trees in the USA.
```

Now let's select (extract) just some parts of our trees dataset using [ ] square brackets. The use of square brackets works with this logic: [lines, columns], where lines are written you specify the desired lines, in most cases each line indicates a sample unit. Where is it written columns,

you can specify the columns (attributes) you want to select. See below:

```
> trees[, 1] # extract the first column and all rows

 [1] 8.3 8.6 8.8 10.5 10.7 10.8 11.0 11.0 11.1 11.2 11.3 11.4 11.4 11.7 12.0 12.9 12.9 13.3
13.7 13.8 14.0 14.2 14.5 16.0 16.3 7 16.3

 [28] 17.9 18.0 18.0 20.6

> trees[, 2] # extract second column and all rows
 [1] 70 65 63 72 81 83 66 75 80 75 79 76 76 69 75 74 85 86 71 64 78 80 74 72 77 81 82 80
80 80 87

> trees[1, ] # extract the first row and all columns
  Girth Height Volume
1 8.3                 70 10.3

> trees[3, 3] # extract third row and third column, 1 value

 [1] 10.2

> trees[1, 3] # extract value from first row and third column

 [1] 10.3

> trees[c(1:5),c(2, 3)] # extract only rows 1 to 5 and columns 2 and 3

  Height Volume
1 70 10.3
two       65 10.3
3         63 10.2
          72 16.4
4 5       81 18.8
```

An efficient way to access variables (columns) by name is to use the dollar sign command $. In this case, we do not recommend using the attach() function, as mentioned above. Usage is basically as follows: mydata$variable
(mydata specifies the dataset and variable the selected variable I want to extract). For example, to extract data from just the circumference() from the “trees” dataset, use the following commandgirth

```
> girth <- trees$Girth

> girth # to see the values of just the circumference
 [1] 8.3 8.6 8.8 10.5 10.7 10.8 11.0 11.0 11.1 11.2 11.3 11.4 11.4 11.7 12.0 12.9 12.9
13.3 13.7 13.8 14.0
14.2 14.5 16.0 16.3 17.3 17.5
 [28] 17.9 18.0 18.0 20.6
```

This way of selecting variables is very practical and will be useful at different times in this booklet. As an example, observe the use of some useful functions:

```
> mean(trees$Girth)# mean value of the circumference of trees

[1] 13.24839

> mean(trees$Height) # average height value [1] 76
```

To advance a little further in data manipulation, we will now use another dataset, available for the course, called “insects” (files “insects.csv”).

These data represent numerical counts (abundance) of insects that were collected from traps at 101 locations in North Dakota, South Dakota, Wyoming and Montana (all in the US). Each site was rated as “”. Each site was also classified as "forest" or "protected" or "grassland" or "unprotected" by state agencies. For each species, we also have information on adult body size and origin ("native" or "invasive").

As a reminder, you can use the following command to import this data into R:

```
> mydata <- read.csv("Insects.csv", sep=",")
> head(mydata) # what the data should look like
```

	habitat	region	website	status	sp.1	sp.2	sp.3	sp.4	sp.5	sp.6	sp.7	sp.8	sp.9	sp.10	sp.11	sp.12	sp.13	sp.14	sp.15	sp.16	sp.17	sp.18
1	1 grassland	WY	unprotected	6	0	1	6	4	0	0	10	0	0	two	0	1	4	two	0	10	0	two
2	grassland	3	3	ND	unprotected	3	4	3	5	3	3	3	3	3	3	5	3	3	3	3	3	3
grassland	3	WY	unprotected	4	9	3	3	4	9	3	3	3	3	6	3	3	5	4	3	3	4	
4 grassland	3	two	two	ND	unprotected	16	two	two	two	two	two	two	two	two	two	two	two	two	3	two	two	
5	5 grassland	3	6	ND	unprotected	3	4	3	3	3	3	3	3	3	3	6	3	3	3	3	3	
6	grassland	3	ND	unprotected	3	3	5	3	16	3	3	3	3	3	3	5	3	3	3	3	17	

	sp.19	sp.20	sp.21	sp.22	sp.23	sp.24	sp.25	sp.26	sp.27	sp.28	sp.29	sp.30
1	0	two	0	0	0	0	0	3	0 0	0	0	
two	4	3	0	0	0	0	0	0	0	0	0	0
3	6	3	0	0	0	0	0	0	1	0	0	0
4	two	3	0	0	0	0	0	1	0	1	0	0
5	3	3	0	0	0	0	0	5	0	two	0	0
6	3	3	0	0	0	0	0	1	0	0	3	0

As we work with large data sets, it is important, in

a first moment, to know this data in terms of structure in R. What commands could we use? See below:

```
> dim(mydata) # 101 rows and 34 columns

[1] 101 34

> str(mydata)

'data.frame': 101 obs. of 34 variables:
        $ site : int 1 2 3 4 5 6 7 8 9 10 ...
        $ habitat: Factor w/ 2 levels "forest","grassland": 2 2 2 2 2 2 2 2 2 2
                                                ...
        $ region : Factor w/ 4 levels "MT","ND","SD",...: 4 2
4 2 2 2 4 4 2 2                 ...
        $ status : Factor w/ 2 levels "protected","unprotected":
2 2 2 2 2 2 2 2 2 2 $sp.1 (...)                                                 ...
                                : int 0 3 3 2 3 3 0 0 1 4                       ...
```

Note that we are working with an object of the “dataframe” type, as we have more than one type of data collected. These data are both categorical and numerical. In this case, an array-type object would not be suitable, since it is premised on only one type of data category, as mentioned above.

Once we've entered this data into R, we can start training some manipulations. Initially, we are going to sum the values of columns or rows using the functions colSums to sum columns and rowSums to sum rows. But, we will add only those columns corresponding to the insect species. See how important it is for us to know the data we are working with.

The columns representing the species range from column 5 to column 34. So let's use the square brackets to select this range:

```
> rowSums(mydata[                 ,    5:34])

 [1] 51 66 84 59 72 95 29 15 41 94 27 74 55 45 55 55 62 37 117 63 102 80 51 36 45 66 179 18
62 7 85 52 93


   [34] 52 85 77 21 92 59 39 375 172 98 44 294 54
27 57 55 29 26 22 62 43 19 131 34 36 28 24
58 40 27 15 23 38
   [67] 73 49 22 17 26 25 38 44 51 37 19 46 62
45 49 18 23 27 41 64 24 30 51 43 66 107 14 24 59 26 27 40 53

[100] 62              0
```

If each line represents a trap, the sum of the partial abundance of each species per observation results in the total abundance of individuals per trap!

Similarly, we can sum each column, and we will have the total abundance of each insect species in the 101 traps:

```
> colSums(mydata[ , 5:34])

sp.1 sp.2 sp.3 sp.4 sp.5 sp.6 sp.7 sp.8 sp.9 sp.10 sp.11 sp.12 sp.13 sp.14 sp.15 sp.16 sp.
17 sp.18 sp.19 sp.20 sp.21 sp.22 sp.23 225 194 239 259 169 212 182 168 196


158 280 305 328 349 228 263 215 228 256 187 0 sp.24 sp.25 sp.26 sp.27 sp.28 sp.29 sp.30 0 256
230 166 208 228 1

0
```

I believe that the great advantage and usefulness of understanding data manipulation in R has now become evident. It is useful to note that the argument of any function can be any other argument, from another function to data.
And in the case of data, they can be manipulated, selected, etc.

In this context, another interesting measure (at least in the context of the purpose of the course) that requires data manipulation is to try to extract from the data the average abundance of insects, for example, between protected and unprotected areas. How could this be performed in R?

To do so, we will use the taaply() function. The function's logic is to calculate some metric from numerical data between groups (categories), as can be seen below:

```
> tapply(rowSums(mydata[ , 5:34]), mydata$status, mean)


protected unprotected 39.55000
        60.97531
```

Alternatively, we could have previously created an object to allocate sum of rows (total abundance):

```
> abund <- rowSums(mydata[ , 5:34])

> abound
 [1] 51 66 84 59 72 95 29 15 41 94 27 74 55 45 55 55 62 37 117 63 102 80 51 36 45 66 179 18
62 [34] 52 85 77 21 92 59 39 379 8 172 55 29 26 22 62 43 19 131 34 36 28 24 58
                    7 85 52 93
```

```
40 27 15 23 38
 [67] 73 49 22 17 26 25 38 44 51 37 19 46 62 45 49 18 23 27 41 64 24 30 51 43 66 107 14

24 59 26 27 40 53
 [100] 62                0
```

In this case, the new command would be more simplified, although in two steps:

```
> tapply(abund, mydata$status, mean)
protected unprotected 39.55000
        60.97531
```

What if instead of creating a new object with the data on the total insect abundance, we added a new permanent column to the data with these values? How could we accomplish this in R? That's what we'll see next:

Let's use the apply() function (note that the similarity with the previously used function tapply() is not causal). The function has the following arguments:

apply(x, INDEX, fun); where x represents a data set; INDEX, which normally explains whether we want to apply a function to rows or columns.
The value 1 marks rows and the value 2, columns. And finally, the function we want to apply, such as sum (sum()), average (mean()) etc. Furthermore, since we can create functions in R – as we will see at the end of this document – we could therefore use a specific function that meets our specific demands. But, see below, what the command would look like for our example:

```
> mydata$abund <-apply(mydata[ , 5:34], 1, sum)
> mydata$abund

 [1] 51 66 84 59 72 95 29 15 41 94 27 74 55 45 55 55 62 37 117 63 102 80 51 36 45 66 179 18
62 [34] 52 85 77 21 92 59 39 379 8 172 55 29 26 22 62 43 19 131 34 36 28 24
                    7 85 52 93


58 40 27 15 23 38
 [67] 73 49 22 17 26 25 38 44 51 37 19 46 62 45 49 18 23 27 41 64 24 30 51 43 66 107 14

24 59 26 27 40 53
 [100] 62                0
```

Notice that using the practical $, R was told to create a new column called “abund”, which will be the result of the sum of the rows of columns 5 to 34 (those columns with the abundance values of each insect species per trap) . This part will still be very useful in the graphics part, as we will see later.

What if we want to add the species abundance total (sum of rows) abundance line to the dataframe? One way to build this, in two steps, is as follows: first we will create a vector containing the total abundance of the species and then we will ask for this line to be added at position 102 (since the data has 101 lines) and between the columns 5 to 34:

```
> sp.abund <-apply(mydata[ , 5:34], 2, sum)
> sp.abund

> mydata[102,5:34]<-sp.abund # add a newline with
the vector sp.abund
> tail(mydata)
# check the results
```

What if we want to create a new empty line (no information)? Remember that R does not accept blank values, that is, we must fill in with NAs. Let's do the same steps and create a 103rd row with this vector of NAs:

```
> vet.na <-rep(“NA”,30)
> vet.na <-rep("NA",30)
> vet.na
 [1] "NA" "NA" "NA" "NA" "NA" "NA" "NA" "NA" "NA" "NA" "NA"
"NA" "NA" "NA"
 [15] "NA" "NA" "NA" "NA" "NA" "NA" "NA" "NA" "NA" "NA" "NA"
"NA" "NA" "NA"
 [29] "NA" "NA"
> mydata[103.5:34]<-vet.na
> tail(mydata) # check how it turned out
```

To remove this last line, let's overwrite the “mydata” file excluding this last line:

```
> mydata<-mydata[-103,]
```

Another important point is the selection of more specific data. For example, in the data set on screen, the selection of traps in which the abundance was only greater than 20 individuals. How could we do this in R?

To do this, we are going to use the which() function. This function, in a generic way, provides results from a logical object. This function is very important for selecting data and will also be very useful in the graph construction part. In our case, the command below will select only rows with abundance values greater than 20 (for this new dataframe, the name “data_abund_20” was assigned, in reference to abundance values per trap greater than 20):

```
> data_abund_20 <- mydata[which(mydata$abund>20), ]
```

```
> head(data_abund_20)# observe how the data was
```

In addition to numerical logical arguments, it is possible to use the which() function to select character (nominal) objects. For example, how could we select only the dataset collected in protected areas? Let's use our new dataset, data_abund_20 as a base.

```
> data_abund_20_prot <-
mydata[which(mydata$status=="protected"), ]
```

The above command uses a very similar logic to the previous command, as you can see. This is an advantage of R. Once the logic behind each function is understood, the applicability of the function is also naturally expanded.

In this logic, instead of selecting a portion of the data based on some logical object, we could exclude portions that are not of our interest, as shown below.

```
> data_abund_20_prot <- mydata[-
c(which(mydata$abund<=20),
which(mydata$status=="unprotected")), ]
# note the (-) sign before the c. This minus sign indicates to remove the specified data.
```

With this single command, we performed the previous steps, excluding abundance values less than 20 and deleting observations in unprotected areas.
Always test the output of the command:

```
> data_abund_20_prot
```

Note: If you want to combine multiple criteria to filter the data, use the function c and multiple which combined. Example:

```
> data_abund_20_prot_MD_ND <-
mydata[which(mydata$abund>=20 &
mydata$status=="protected" & mydata$region %in% c("MT","ND")), ]

#data were selected that have an abundance >20 at the same time, either in
unprotected areas and only in MT and ND states.
```

Generating sequences, random numbers and repetitions

This topic presents some commands that can be useful in several more complex ones in R; moments during the formulation of a script or artifacts that

can be useful for graphing and creating functions, as we will see throughout this text. But, they can also be useful in the construction of graphs, our next topic.

Simply, to generate a sequence in R, we use a very simple command:

```
< 1:10 # generates a sequence of numbers from 1 to 10
 [1] 1 2 3 4 5 6 7 8                                    9 10
```

A string can be added to a new object:

```
< to <- 1:100
< to # The result will be a sequence of numbers from 1 to
100
```

Another way to create a sequence, but with the possibility of specifying the intervals, is to use the seq() function. The generic arguments of this function are as follows:

seq(from = 1, to = 100, by = 4), sequence from one to 100, in intervals of 4

```
> seq(1, 200, 10) # sequence
from 1 to 200 in intervals of 10
 [1]          1 11 21 31 41 51 61 71 81 91 101 111 121
131 141 151 161 171 181 191
```

Another interesting command is the one that generates repetitions of numbers. For this, the rep() function is suitable:

To repeat the number four 20 times, we use the following command:

```
< rep(5, 20)
 [1] 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
```

Here's a small variation artifact in this command:

```
< rep(1:10, each=2) # see the result!
```

To combine repetitions, use the concatenate(c) function:

```
< c(rep(1, 5),rep(2,5))
```

Other commands that will be very useful are functions that create random datasets. Let's start with the runif() function, which generates random data with uniform distribution:

> runif(n, min=0, max=1) # generates a uniform distribution with n values, with maximum and minimum values.

For example, let's generate a dataset with 100 values, ranging from 20 (minimum) to 60 (maximum):

```
> runif(100, 20, 60)
 [1] 59.17655 44.77081 28.51730 52.52110 37.88434 20.97518 52.07002 53.89839 25.38896
23.45999 46.15411 24.18540 48.74681 40.83567 50.60774 31.91596 44.13625 (...)
```

Remember that you can always allocate this data string in a new object, as shown below:

```
> temp <- runif(100, 20, 60)
> temp # Check the result. However, as they are randomly generated numbers, they
may differ minimally from the first time we run the command.
```

Another way of generating random data is to vary the type of distribution. For example, we can generate random data with normal distribution using the rnorm function, which has the following generic arguments:

> rnorm(n, mean=0, sd=1) # Generates a random sequence with “n” elements, with mean equal to zero and standard deviation (sd: standard deviation) equal to one.

Now, let's run an example:

```
> rnorm(200, 5, 3) # create 200 values with mean 5 and standard deviation 3
```

In the next step we will work with graphs and we will be able to observe variations in the distributions visually. But, for the moment, use the help of the ?Distributions function to learn about other ways to generate random data with different distributions.

Yet another way to extract random samples using the sample() function. It is used to extract random samples and works like this:

> sample(x, size=1, replace = FALSE) # where x is the dataset from which the samples will be taken, *size* is the size of the sample (n), and *replace* is the argument indicating whether the sample will be executed with replacement (TRUE) or without replacement (FALSE).

Let's look at some examples:

```
> sample(1:20, 5) # extracts 5 numbers with values between 1 and 20.
```

Note that the replace argument of the sample() function was not specified.
In this case, R defaults to considering that the sample is without replacement (replace = F). Let's see how this argument works:

```
> sample(1:20, 25) # error, sample larger than the set of values; we ask for 25 random
numbers, but our dataset only has 20 elements (ranging from 1 to 20). In order to be able to
perform this function correctly, R needs to replicate some numbers. To do so, we need to add
the argument “replace=T”, as
```

Follow:

```
> sample(1:20, 25, replace=TRUE) # now yes! OR also understands if you use only replace=T
OR replace=F.
```

We can add another argument inside this function, such as the rep() function, seen earlier. In this case, we can add inside the sample() function a repetition of this extraction “n” times, as shown below:

```
> sample(1:20, 5, rep=100) # 100 repetitions. Run twice and notice the differences!!
```

* for sampling or generating random numbers, it may be interesting to use a seed, that is, a seed that will always determine the same sampling or random number. This can be useful when it is pertinent to reproduce the result:

```
> set.seed(123) # sets seed = 123. the set.seed command must always be repeated
immediately before any sampling or random number generation, that is, the command does
not permanently establish that the seed is always the same .
```

```
> sample(1:5,5)
 [1] 2 4 5 3 1
> set.seed(123)
> sample(1:5,5)
 [1] 2 4 5 3 1 # repeat the above two commands a few times and compare the results.

# repeat now with different seeds
```

Creating graphics in R

As scientists, graphs are, in general, one of the main results we aim for in addition to statistical analysis itself. Graphs add a visual component to the numbers, making the results easier to interpret. OR presents many possibilities for us to edit the graphics, making them very personified. Within this universe, the basic components will be discussed here, regarding the manipulation of commands that are useful tools in the construction of these graphs.

The basic function that generates a graph in R is plot(). The plot has as basic arguments the set of our data that we want to plot; normally the generic command starts with plot(x, y). Let's return that dataset called trees to make our first plot in R.

```
> date(trees)

?head(trees) #let's remember the data information
```

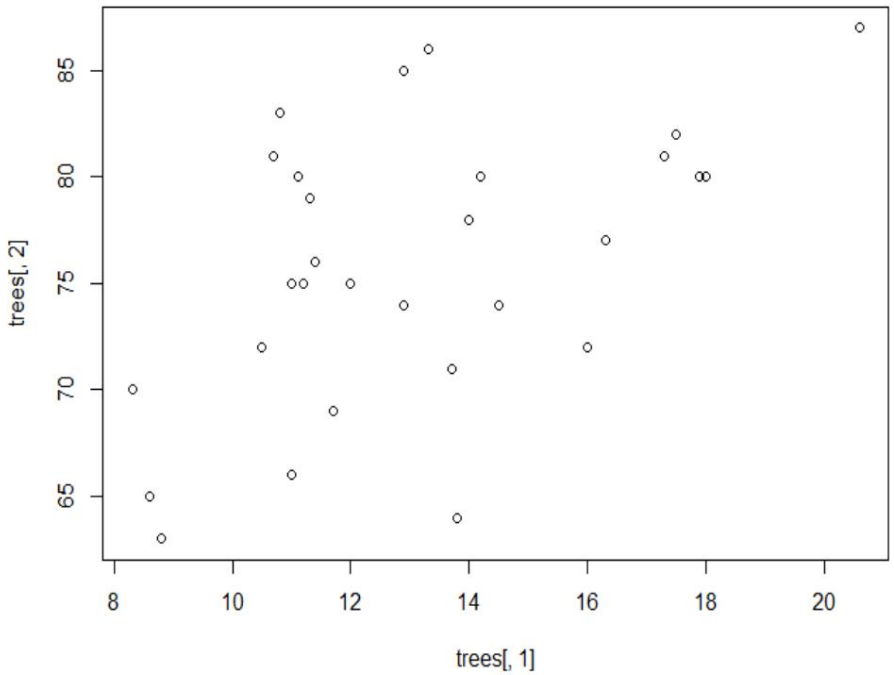
Well, we can, in a very practical way, create a graph called xy dispersion, where we will visually analyze how a variable y (dependent variable) behaves as a function of a variable x (independent variable). Without thinking too much about ecological issues, let's see how height () behaves in relation to circumference (height girth):

```
> plot(trees$Girth, trees$Height) #
```

We can also use the following command alternatively:

```
> plot(trees[, 1], trees[, 2])
```

The output of this last command is as follows:

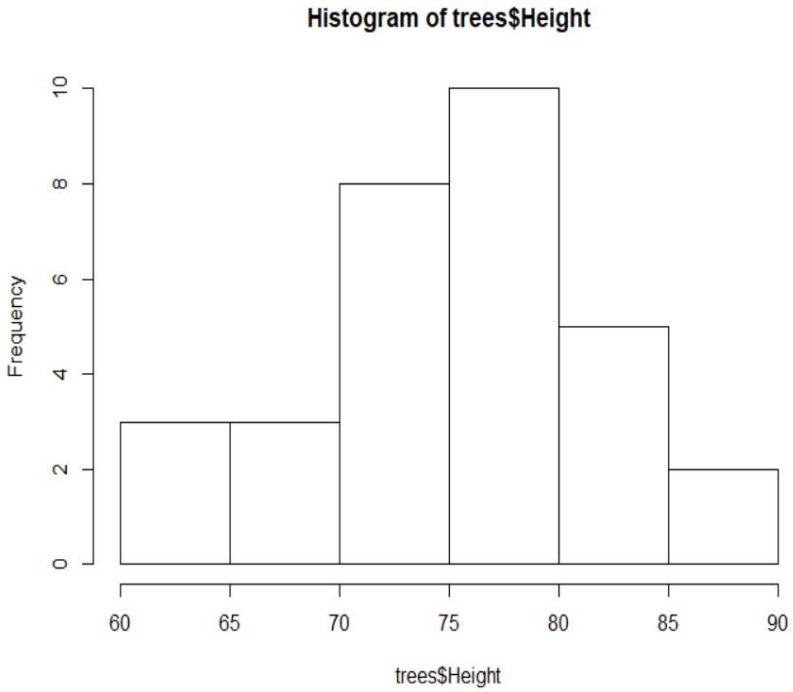


This basic command naturally does not generate a visually beautiful and publishable graph, but it is an important first step. Some visual components will be explored here. Notice how the correct manipulation of data is an important argument within the plot() function, like the use of $ and [ ].

Another commonly used chart type is the histogram. The function to create one is hist(). It basically lacks one argument, which is the dataset. Still using the trees dataset, let's see how the height of trees is distributed in classes:

```
> hist(trees$Height)
```

The graph should look like the one below:



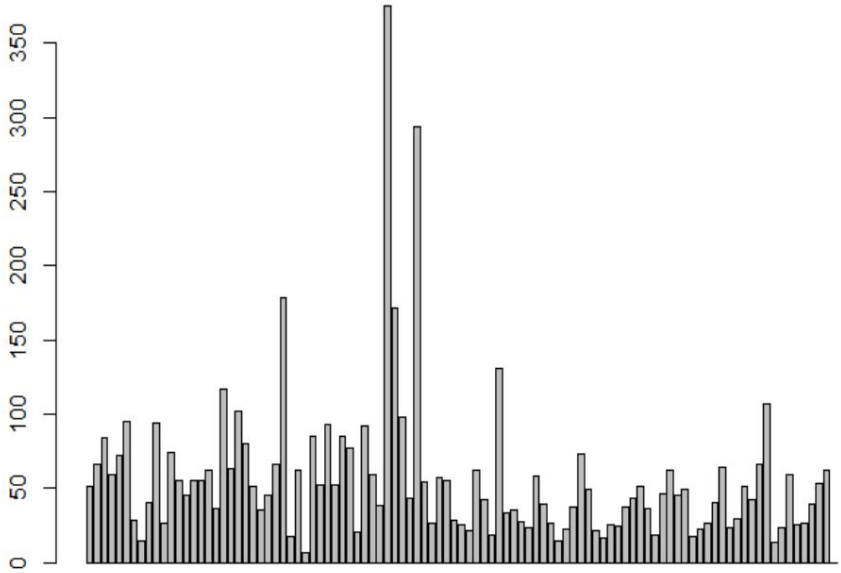
Another very common type of chart is the bar chart. The function that allows us to create this type of plot is barplot(). As an example, let's go back to using that dataset of insects collected in 101 traps, which was made available during the course:

> head(mydata) #let's remember the variables of this dataset

Previously, we added a column with the abundance of insects in each trap. Let's use this data to build a bar graph.

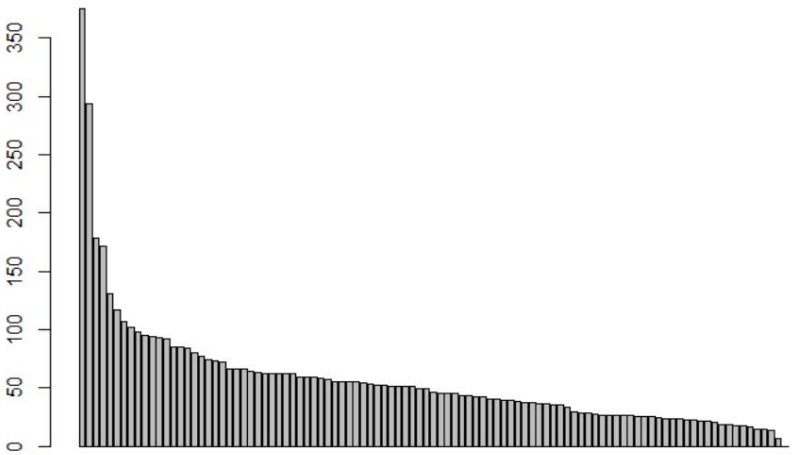
> barplot(mydata$abund)

The output is as follows:



An interesting argument in barplot, is to arrange the bars in some logical order, like ascending or descending. The sort() command sorts the data in order of increasing abundance. The rev() command reverses the order. See the command below and its result:

```
> barplot(rev(sort(mydata$abund)))
```



You can try some variations of this last command, and see the graphical results. Try for example:

```
> barplot(sort(mydata$abund))
```

Notice what the rev() argument did earlier!

In the next steps of the handout we will present some arguments to make the graphs more logical and publishable. For the moment, let's move on to the next type of plot, also commonly used, the boxplot. The command to generate this type of plot is, of course, boxplot().

This function has the following construction as its basic argument:

> boxplot(x, ...)

But, it can also present variations, and present the following arguments:

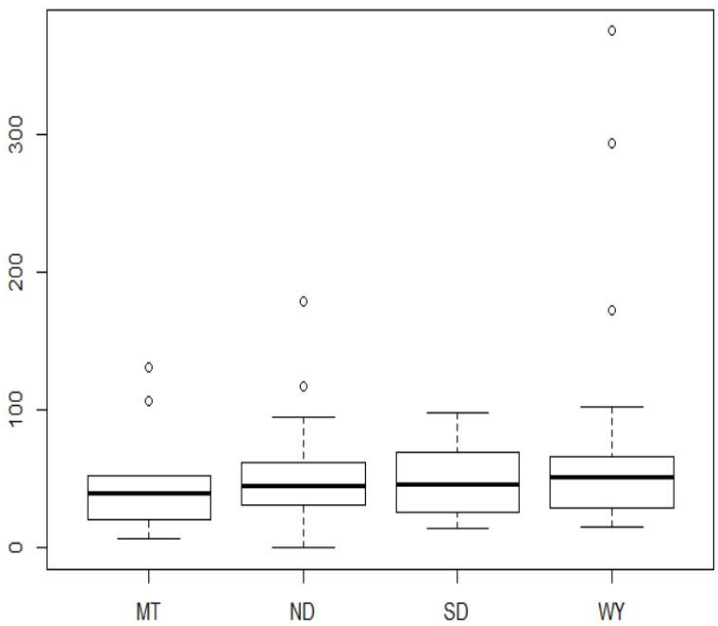
> boxplot(formula, ...)

In this case, the formula argument might be something like y ~ grp, where y is a numeric vector of data values to be divided into groups according to the clustering variable grp (usually a factor). the tilde symbol (~) means “modeled by” (see more details in the topic Univariate Statistics)

Let's continue using the insect dataset. Let's build a boxplot that will analyze how the abundance (numerical vector of values) varied across the collection sites (cluster variable, one factor):

> boxplot(mydata$abund ~ mydata$region)

The resulting graph will look like this:



Once the most common types of graphs have been elucidated in terms of basic functions and arguments, the next topics will focus on how to change the appearance of graphs.

Dashboard charts

A very interesting graphic resource concerns the use of space. In general, the diagramming of graphs is an important aspect, since the spaces for figures in articles, conclusion works, dissertations and theses, among others, are limited. One of the ways to take advantage of this space is to make a single figure with several graphs. The screen that R generates (and the R Studio graphics panel) by default is one graphic per panel. To put four figures in a single panel, we need to inform R, through a command. The most commonly used command is the following: par(mfrow = c(x,y)), where x is the argument that specifies the number of rows and y the number of columns in the panel:

In this case, to leave the four most common chart types shown here in a single panel, we need to divide the panel area into 2 rows and 2 columns, which will generate four areas in a single panel.

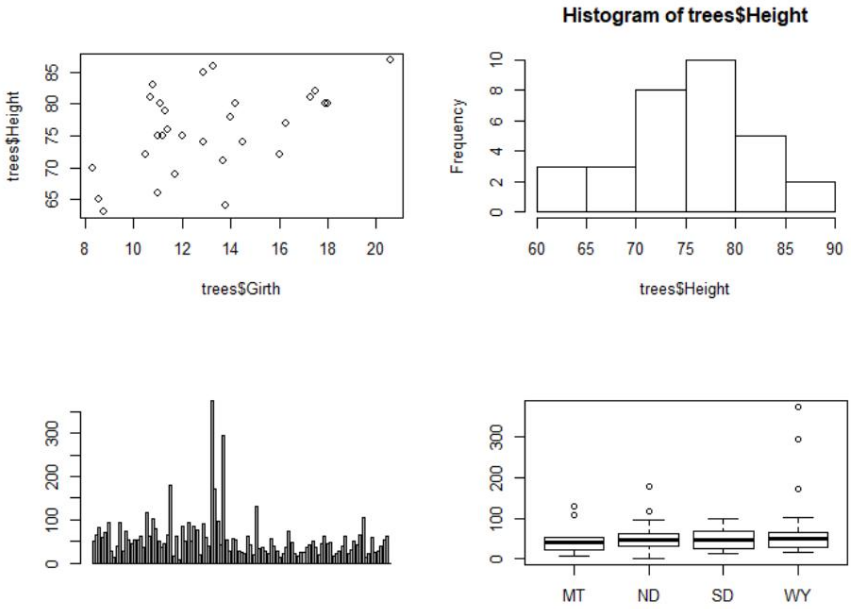
The argument, therefore, inside the par() function, can be as follows:

```
> par(mfrow = c(2, 2))
```

Now, we can enter the commands referring to the four types of charts that we illustrated above:

```
> plot(trees$Girth, trees$Height)
> hist(trees$Height)
> barplot(mydata$abund)
> boxplot(mydata$abund ~ mydata$region)
```

The result will be like this below:



An important point is that all the next graphs that R generates from commands will be in 4 panels. Try running one of the four chart types above again and you will see this. To avoid this small detail, it is recommended to add a command to inform R that the next graphs will be in single panel or any other type of division that is of interest.

To return to having only one chart per panel, use the following command:

```
> par(mfrow = c(1, 1))
```

To test its correct functionality, test by running any of the four types of charts that we learned above, such as:

```
> hist(trees$Height)
```

Now that we've learned this little trick, let's explore other arguments within the functions that produce graphs, for the purpose of improving the appearance of those four basic types of graphs.

Changing the appearance of graphics

Let's start working with that xy scatter plot, our first type of graphical analysis explored. The command was as follows:

```
> plot(trees$Girth, trees$Height)
```

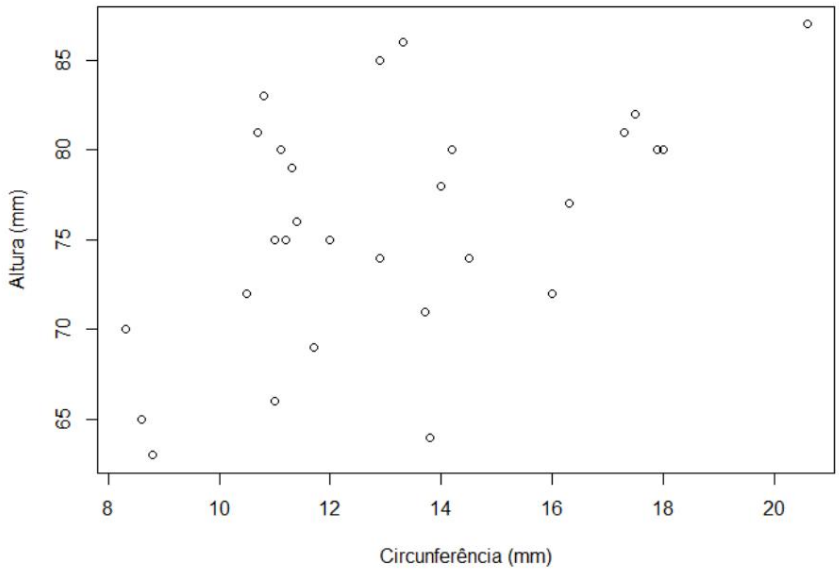
A graphic, like a figure, has the precept of self-explanation, that is, there is a need to contain minimal information so that the reader understands the purpose of that image or that graphic. In the case of the graph, specifically, the correct assignment of the x and y axes is essential, for example.

Almost all of these appearance tweaks are arguments inside the functions of the graphs, in this case, inside the plot() function, as we will see below.

The command in which we specify the x-axis title is xlab, as well as for the y-axis, it's ylab. Our command, so in this case, could look like this:

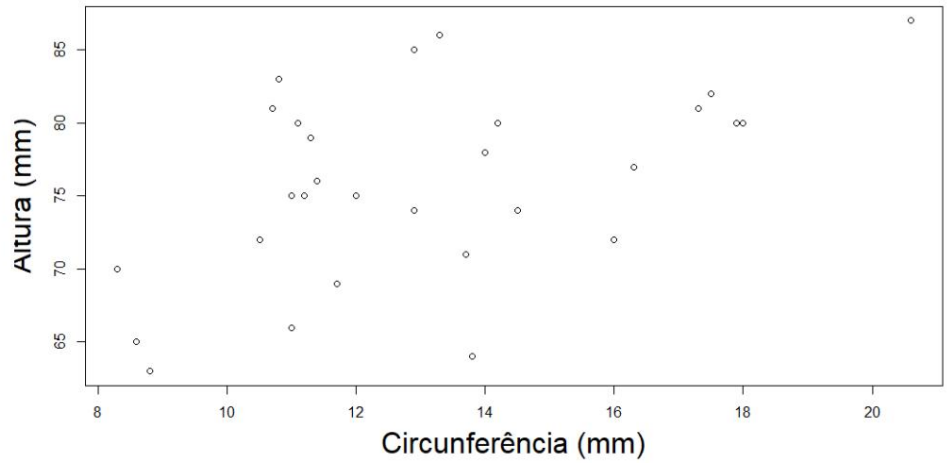
```
> plot(trees$Girth, trees$Height,xlab="Circumference (mm)",ylab="Height (mm)")
```

The result would be this:



To change the font size, use the cex.lab argument:

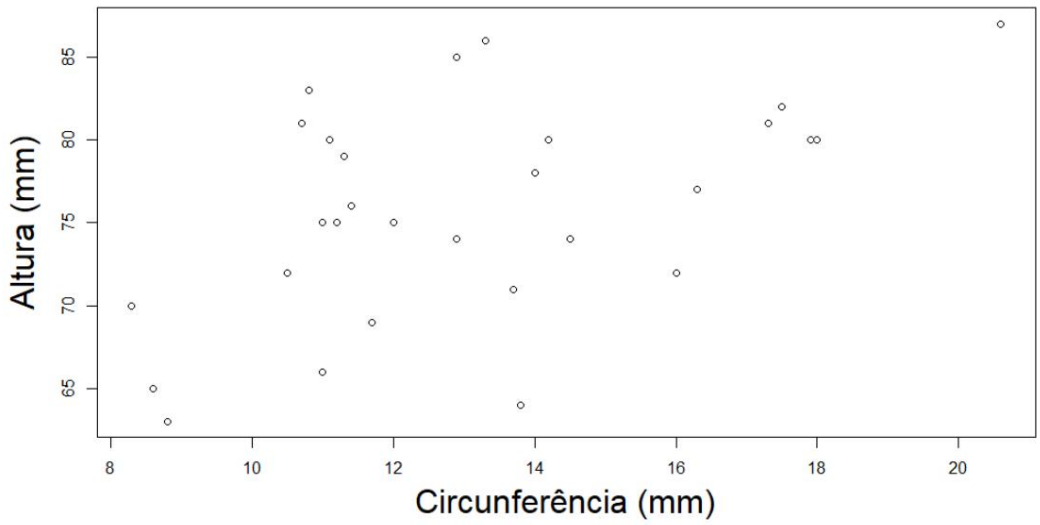
```
> plot(trees$Girth, trees$Height,xlab="Circumference (mm)",ylab="Height
 (mm)",cex.lab=2)
```



Note that increasing the font size of the labels left little margin. To adjust it, use the command par(mar=c(5,4,4,2)). This is the default for c(bottom, left, top, right) margins. let's change only the left margin:

> par(mar=c(5,6,4,2))
> plot(trees$Girth, trees$Height,xlab="Circumference (mm)",ylab="Height (mm)",cex.lab=2)

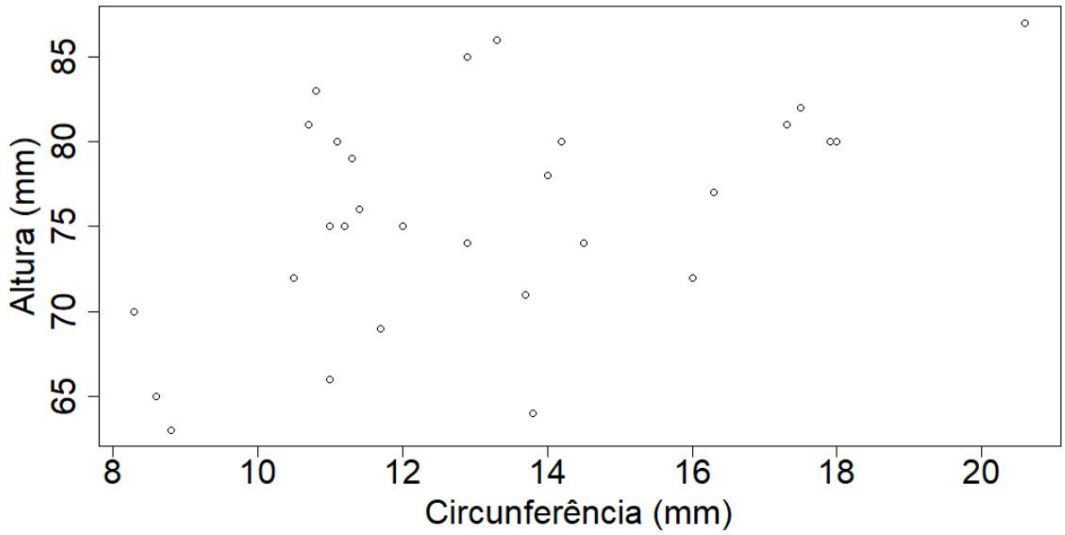
*an important note: the par(mar()) command permanently changes the plot area. To change, you must use the command again with other specifications.



To change the text size of the axes, use the cex.axis command.

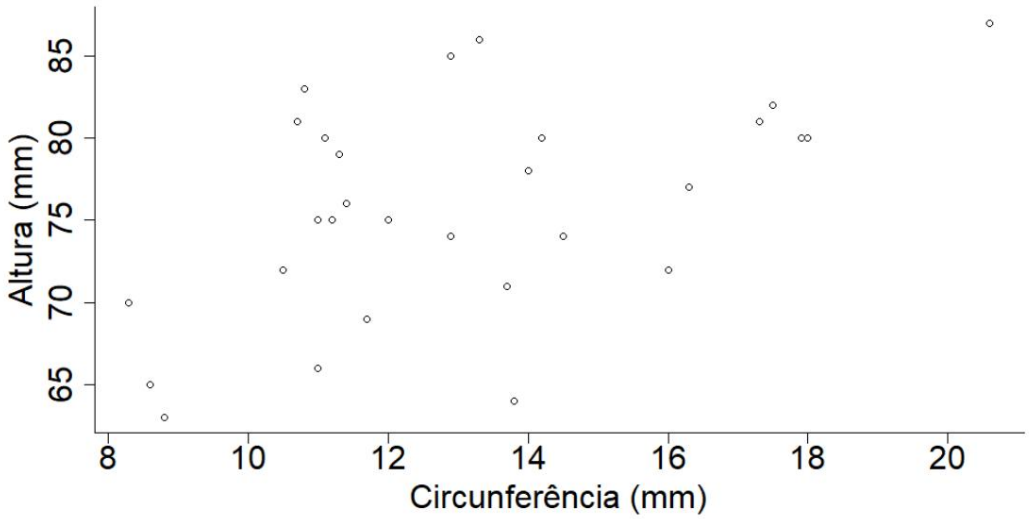
> plot(trees$Girth, trees$Height,xlab="Circumference

(mm)",ylab="Height (mm)",cex.lab=2,cex.axis=2)



To remove the box around the graphic and keep only the bottom and left, use the argument bty=”l”.

> plot(trees$Girth, trees$Height,xlab="Circumference (mm)",ylab="Height (mm)",cex.lab=2,cex.axis=2,bty=”l”)



*bty=”o” produces the same default graph and bty=”n” leaves just the axes, with none of the box edges.

There are a number of other arguments, however, that we could use. Among all, another interesting one is the pch. This argument concerns the dot symbol.
The default is these open circles, but there are other symbols we could use.

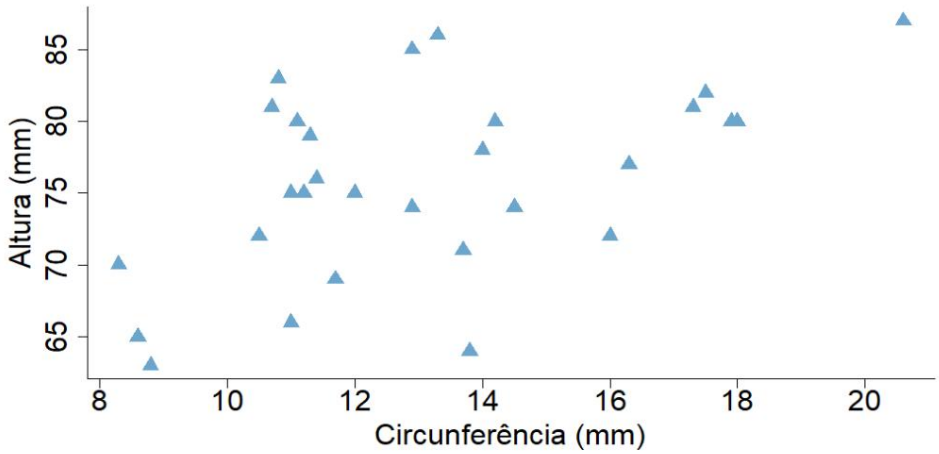
To visualize the possibilities of the pch argument, let's use the R help (See more details in Quick Graphical Parameters https://www.statmethods.net/advgraphs/parameters.html)

```
> ?pch
# below the most important information at the moment
```

So let's use this argument, in conjunction with another important argument, the one that adds colors, col(). First, google the PDF booklet with the default R colors, looking for “RColors”. A possible command with these two arguments could be something like this:

```
> plot(trees$Girth, trees$Height,xlab="Circumference
 (mm)",ylab="Height(mm)",cex.lab=2,cex.axis=2,bty="l",cex=
2,pch=17, col="skyblue3")
#the cex argument determines the size of the points
```

The new chart layout will look like this:



Similarly, we can add title to the axes and color our histogram, for example:

```
> hist(trees$Height, col=”red3”, xlab=”Height (mm)”, ylab=”Frequency”)

# Notice how the new histogram layout looks
```

The argument that adds a default main title in the graphics is main.
We can add this argument to modify the main title that the hist function automatically inserts into the chart, as follows:

```
> hist(trees$Height, col="red3", xlab="Height (mm)",
ylab="Frequency", main="")
```

Another important aspect in the histogram is the division of classes and the limits of the axis that represent our data. To modify this, there is an argument of the hist function called breaks. To generate these intervals, we will use a function already presented, the numerical sequences, the seq() function.

We will try. First, it is important to know the numerical limits of the data we are exploring:

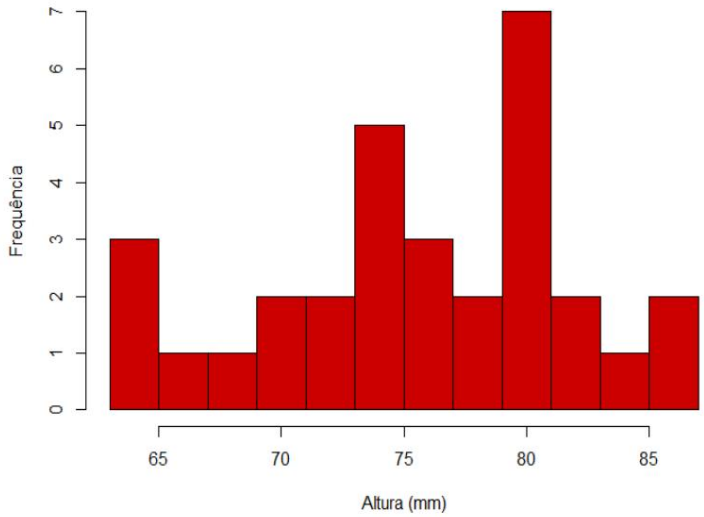
```
> max(trees$Height)
 [1] 87

> min(trees$Height)
 [1] 63
```

Remembering, the seq() function has three arguments, which are the limits and the division of the intervals, as we can see below:

```
> seq(63, 87, 1)
```

Incorporating this as an argument of breaks, inside the hist function, we have the following command and result:

```
> hist(trees$Height, breaks=seq(63, 87, 2), col="red3", xlab="Height (mm)",
ylab="Frequency", main=" ")
```



Remember that topic in which we discussed how to create random numbers, data with a defined mean and standard deviation, with the rnorm() function, for example? Try exploring variations in the chart layout. Below an example:

```
> simul <- rnorm(500, 10, 3) # dataset with
n=500, mean equal to 10 and standard deviation equal to 3
```

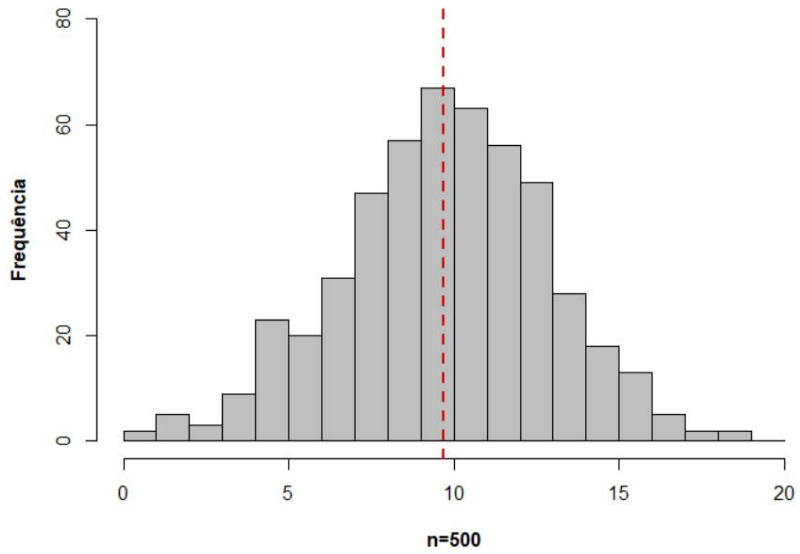
A simpler graph would be just the command

```
> hist(simul) # see the result
```

Now, try to make the graph more elaborate, such as adding a dotted line representing the mean (which will be close to 10!). Let's go in parts:

```
> hist(simul, breaks=seq(0, 20, 1), col="gray75", xlab="n=500",
ylab="Frequency", main="", ylim=c(0, 80), font.lab=2)

> abline(v=mean(simul), col="red3", lty=2, lwd=2)
```

The graphic design will be as follows:



Note that new arguments have been added inside the hist() function, such as font.lab=, which is related to the font style of the chart axes titles (italics, bold, underlined, etc.).

A vertical red line was also added to indicate the mean value of the data. For this, the abline() function and its respective arguments were used,

as you can see in the previous command.

The abline() function can also be interesting to add a linear regression model to XY (or scatter) graphs. To illustrate this, let's resume using tree data.

To create a graph that analyzed a possible relationship between circumference and height of trees in the Northern Hemisphere. The last command we used to observe this possible relationship, with minor layout changes, was the following:

```
> plot(trees$Girth, trees$Height,xlab="Circumference
 (mm)",ylab="Height(mm)",cex.lab=2,cex.axis=2,bty="l",cex=2,pch
=17, col="skyblue3")
```

The lm() function is responsible for creating a linear regression model between dependent (y) and independent (x) variables.

It has the following basic arguments as logic:

```
> lm(y ~ x) # how y responds to x
```

In the proposed example, the command would be something like:

```
> lm(trees$Height ~ trees$Girth)
```

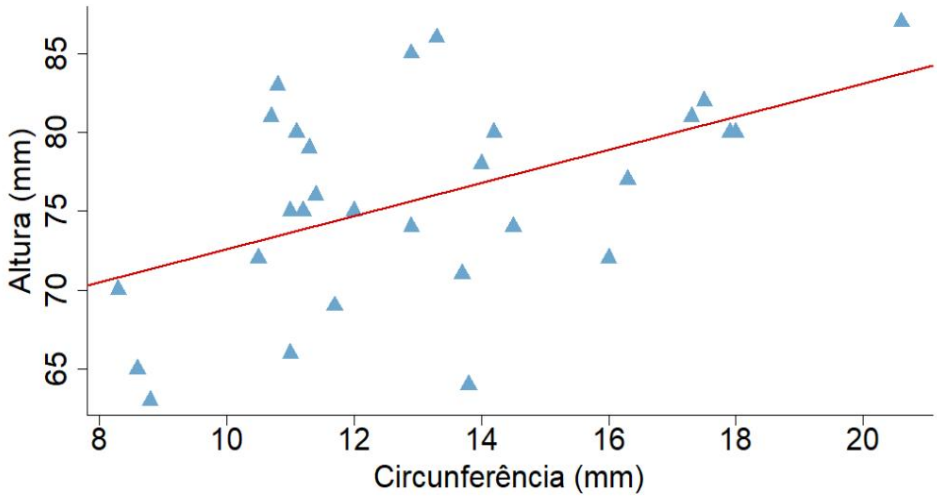
But, it is always recommended to allocate these results to a new object, which here we will call “a”:

```
< a <- lm(trees$Height ~ trees$Girth)
```

Now, we can add the graphical result of this linear regression:

```
< abline(a, col=”red3”, lwd=2)
```

The result will be approximately this:



Advanced graphics changes:

Insert text: function text(x, y, ”my text”). x and y are the positions in plot where the text should be added:

< text(18, 65, ”I love R”, cex=2, col=”red”) # the cex and col arguments are used to change the text size and color, respectively.

Text can be placed directly also via mouse click with the locator() function.

< text(locator(), ”I love R”, cex=2, col=”red”)

R will wait for the position to be specified.
Left-click where the text should be placed and then press ESC. The text will be placed centrally to where it was clicked.

The locator() function can also be used separately to find the coordinates of specific points within the plot.

< locator() # right click on all the points you want to find the coordinates. Then press ESC and the coordinates will appear at the command prompt. This can be useful for figuring out the coordinate to put in the text() function.

Change the appearance of axes independently: some axes modifications are not possible directly by specifying arguments through the function

plot(). In these cases, it is necessary to plot the graph hiding the axes and add them later one by one in a custom way:

```
> plot(trees$Girth,
trees$Height,axes=F,bty="l",cex=2,pch=17,
col="skyblue3",ylab="",xlab="",bty="l" ,xlim=c(8,20),ylim=c
 (60.90))
# axes=F or FALSE indicates that axes should not be plotted. xlab="" and ylab=""
indicate that nothing should be written in the axis labels.


> axis(1,cex.axis=2) # 1 indicates the x axis

> axis(2,cex.axis=2,las=2) # 2 indicates the y-axis; the las argument rotates the
axis labels

> mtext("Circumference (mm)",side=1,cex=2)
# notice that the axis label is on top of the axis.
Adjust the text position with line (it also works for the axis position, in the axis function):


> mtext("Circumference (mm)",side=1,cex=2,line=3)

> mtext("Height(mm)",side=2,cex=2,line=3.5)
```

Note that the plot generates a “gap” between the axes, which makes the graph not so elegant. To remove it, we must add an argument next to the par function, xaxs and yaxs. Then specify the appropriate limits of x and y, remembering to change them in the placement of the axes as well:

```
> par(mar=c(5,8,4,4),xaxs="i",yaxs="i")
> plot(trees$Girth, trees$Height, axes=F, bty="l",cex=2,pch=17,
col="skyblue3", ylab="",xlab="",bty="l" ,xlim=c(8,20),ylim=c(60,90))

> axis(1,cex.axis=2,xlim=c(8,20))
> axis(2,cex.axis=2,las=2,ylim=c(60,90)
> mtext("Circumference (mm)",side=1,cex=2,line=3)
> mtext("Height(mm)",side=2,cex=2,line=3.5)
```

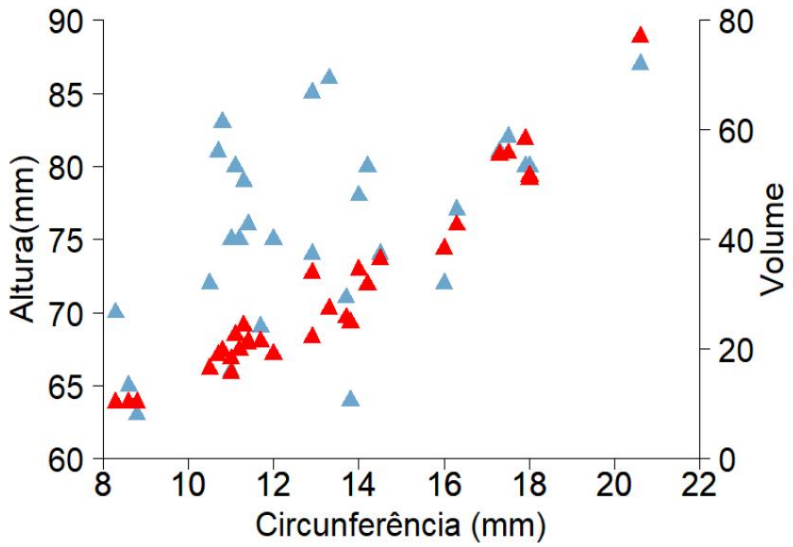
IMPORTANT: Whenever the axes are added in this way, care must be taken to verify that the amplitudes of x and y are the same. Otherwise the plots will be done wrongly at different scales:

```
> plot(trees$Girth,
trees$Height,axes=F,bty="l",cex=2,pch=17,
col="skyblue3",ylab="",xlab="",xlim=c(0 ,20),ylim=c(0.85))
> axis(1,cex.axis=2,xlim=c(0,20))
> axis(2,cex.axis=2,ylim=c(0.85))
```

```
> mtext("Circumference (mm)",side=1,cex=2,line=3)
> mtext("Height(mm)",side=2,cex=2,line=3.5)
```

To plot second axis or more: Each time the plot function is used, the plot area is cleared and replaced by the new command. To activate subscription (multilayer plotting), use the par(new=T) function.

```
> par(mar=c(5,6,4,6),xaxs="i",yaxs="i")
> plot(trees$Girth, trees$Height, axes=F, bty="l", cex=2, pch=17, col="skyblue3",
ylab="",xlab="", xlim=c(8 .22), ylim=c(60.90))

]>axis(1,cex.axis=2,xlim=c(8,22))
> axis(2,cex.axis=2,las=2,ylim=c(60,90))
> mtext("Circumference (mm)",side=1,cex=2,line=3)
> mtext("Height(mm)",side=2,cex=2,line=3.5)
> par(new=T)
> plot(trees$Girth, trees$Volume, axes=F, bty="l", cex=2, pch=17,col="red", ylab="", xlab="",
bty="l" , xlim=c(8.22), ylim=c(0.80))

> axis(4, cex.axis=2,las=2,ylim=c(0.80))
> mtext("Volume",side=4,cex=2,line=3.5)
```



Special characters: below are some examples of how to place characters specials in texts (can be applied to axis titles as well):

```
> text(locator(),expression(paste("Bacterial Biomass"," ","(x"," ","10"," ",mu,"g"," ""C"," ",
mL^-1,")")), cex=2)
> text(locator(),expression("PO"[4]^-3),cex=2)
> text(locator(),expression("NO"[3]^-""),cex=2)
> text(locator(),expression("NO"[2]^-""),cex=2,)
> text(locator(),expression(paste(bold(Chl)," ",bold(italic(a)))),cex=2)
```

Some useful additional functions:

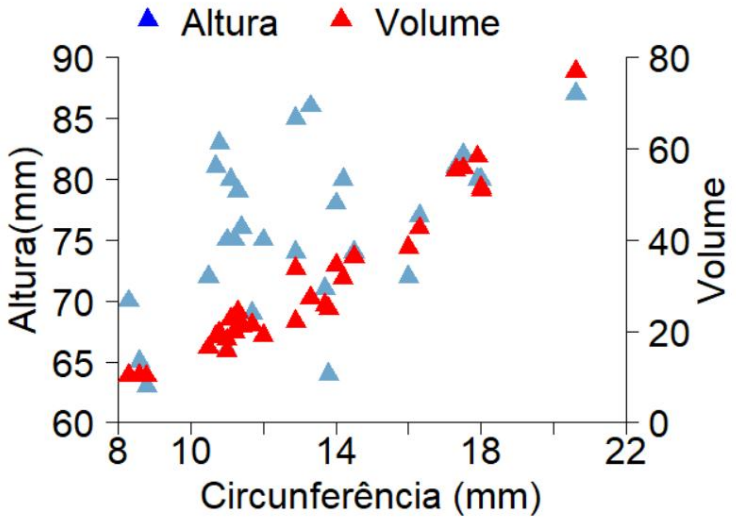
There is an argument inside the par function that allows data, text or legends, etc. to be plotted. outside the plot area. This is xpd=TRUE. It can be added together with the initial pair that determines the general characteristics of the plotting area of the graph or it can be used in a standalone command line:

```
> par(xpd=TRUE)
#Example 1
> par(mar=c(5,6,4,6),xaxs="i",yaxs="i", xpd=TRUE)
```

Example 2: let's say you want to place the caption on top of the figure. In this case we must use the par(xpd=TRUE) before the legend command:

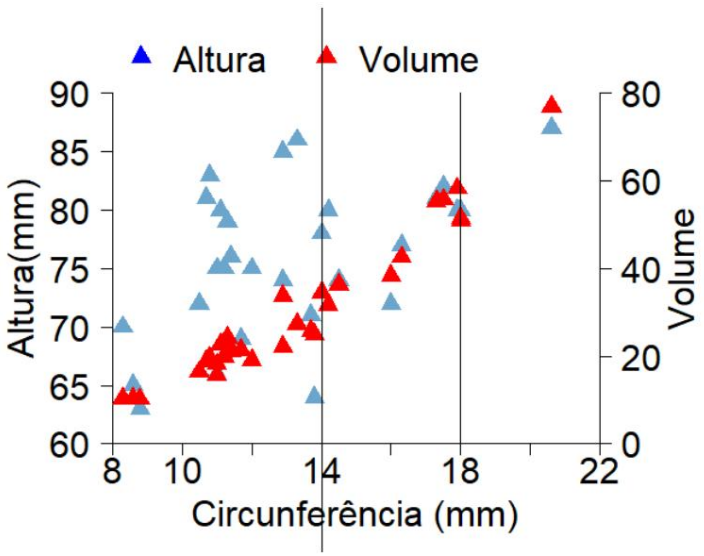
```
>plot(trees$Girth, trees$Height, axes=F, bty="l", cex=2,pch=17, col="skyblue3",
ylab="",xlab="", xlim=c(8 .22), ylim=c(60.90))

>axis(1,cex.axis=2,xlim=c(8,22))
>axis(2,cex.axis=2,las=2,ylim=c(60,90))
>mtext("Circumference (mm)",side=1,cex=2,line=3)
>mtext("Height(mm)",side=2,cex=2,line=3.5)
>par(new=T)
>plot(trees$Girth, trees$Volume, axes=F, bty="l", cex=2, pch=17,col="red", ylab="", xlab="",
bty="l" , xlim=c(8.22), ylim=c(0.80))

>axis(4, cex.axis=2,las=2,ylim=c(0.80))
>mtext("Volume",side=4,cex=2,line=3.5)
>par(xpd=TRUE)
>legend(locator(),c("Height","Volume"),
pch=17,col=c("blue","red"),cex=2,bty="n", horiz=T)
```

test placing at the very top of the graph. The horiz=T argument plots the legend in horizontal mode:



Some functions, such as abline, are circumscribed to the plot area. if you use abline after using par(xpd=TRUE), it will create a straight line that will pierce this area:

```
>abline(v=14)
> par(xpd=F)
> abline(v=18) # reverting xpd to “False” returns plots to the default plot area:
```



Finally, we can still change the ranges of the scales of the x and y axes. Remember that we changed the limits (the amplitude) of the axes through the xlim function and ylim. Sometimes R returns with some not very elegant solution (too long or too short intervals). We can also customize these intervals according to our preference. To do this, we just change the argument

xlim and ylim inside axis by at=seq(minimum, maximum, by=desired range). In the example below we show only one of the command lines of the last graph in the line where we ask for the x axis to be plotted. In this case,
we want the limit to go from 8 to 22, with a range of 4:

```
> axis(1,cex.axis=2,at=seq(8,22,by=4))
```

Lastly, if you are uncomfortable with scientific notation, you can change the R setting so that numbers with less than 999 digits are displayed exactly (999 was a random number - it could be any other number).

```
> options(scipen = 999)
> options(scipen = 0) # to return to default
```

Export of graphics

One of the great advantages of R is the possibility of making custom graphics, as we demonstrated with a few examples above. Another excellent advantage is that we can export graphics from R (or RStudio) in formats and layout ready for publication; either in PDF, JPG or TIFF format. We can choose the resolution, the size of the figure, among other almost infinite possibilities.

Here, we will focus on the two most commonly used formats for exporting figures, the PDF format and the TIFF format. Both are vectors and, in our understanding, store more graphical information.

Let's use the creation of the last chart as an example. The commands are again entered below:

```
> par(mar=c(5,6,4,6),xaxs="i",yaxs="i")
> plot(trees$Girth, trees$Height, axes=F, bty="l", cex=2, pch=17, col="skyblue3",
ylab="",xlab="", xlim=c(8 .22), ylim=c(60.90))

]>axis(1,cex.axis=2,xlim=c(8,22))
> axis(2,cex.axis=2,las=2,ylim=c(60,90))
> mtext("Circumference (mm)",side=1,cex=2,line=3)
> mtext("Height(mm)",side=2,cex=2,line=3.5)
> par(new=T)
> plot(trees$Girth, trees$Volume, axes=F, bty="l", cex=2, pch=17,col="red", ylab="", xlab="",
bty="l" , xlim=c(8.22), ylim=c(0.80))

> axis(4, cex.axis=2,las=2,ylim=c(0.80))
> mtext("Volume",side=4,cex=2,line=3.5)
```

To export a graphic in PDF, we will use the pdf() function. See more details of this function, using the R help function (?pdf).
The basic command can be as follows:

```
> pdf("Graph 1__Test.pdf", width=7, height=5)
# The first argument is the name you want to give the PDF file to be exported. And, here in this example, the size of the graph. In this function, the units are in inches (US system). 1 inch represents 2.54 cm.
```

The logic is that the first command is the one above, then all those commands from the graph, and finally, a specific command for it to export this file, signaling that you will no longer add arguments to the graph:

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

The entire command is below. But where will R export this PDF file to?

To the working directory you defined at the beginning of this document! Try using the getwd() command that R will point to the location, if you don't remember.

```
> pdf("Graph 1__Test.pdf", width=7, height=5)
# Exporting a graphic in PDF. Note the generic name given to the file

> op <- par(mar=c(5,6,4,6),xaxs="i",yaxs="i")
# Another way to configure the chart margins

> plot(trees$Girth, trees$Height, axes=F, bty="l", cex=2, pch=17, col="skyblue3", ylab="",xlab="", xlim=c(8 .22), ylim=c(60.90))

> axis(1, cex.axis=2, xlim=c(8,22))
> axis(2, cex.axis=2, las=2, ylim=c(60,90))
> mtext("Circumference (mm)",side=1,cex=2,line=3, font=2)

> mtext("Height(mm)",side=2,cex=2,line=3.5, font=2)
> par(new=T)
> plot(trees$Girth, trees$Volume, axes=F, bty="l", cex=2, pch=17, col="red", ylab="", xlab="", bty="l" , xlim=c(8.22), ylim=c(0.80))

> axis(4, cex.axis=2, las=2, ylim=c(0.80))
> mtext("Volume", side=4, cex=2, line=3.5, font=2)
> par(op)
# Reset previously defined margin parameters > dev.off()
```

How was the graph?

In a very similar way, we can export this same graphic in TIFF format. Just change the first command to:

```
> tiff("Graph 1__Teste.tiff", height=12, width=17, units='cm', compression="lzw", res=300)

# Here we change the units to cm, set the most common type of file comprehension and the resolution, 300
```

dpi. See more arguments of this function with the help of R.

Insert all other graphic commands. Then use the command below to signal to R that you want to export the graph:

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

OR will export a picture in TIFF format to the defined working directory. Find the picture and see how it turned out. The same goes for other formats, such as JPG, for example.

Univariate statistics: comparing two or more means, assumptions, tests of association between variables

Comparison between two averages

t-test

Used for continuous data associated with two categories (groups). It can be performed using the t.test function. For this, we will use the Insects.csv dataset (mydata) and test whether the total insect abundance differs between protected and unprotected areas. Let's check first how many observations (N) there are in each category:

```
> summary(mydata$status)
    protected unprotected 81
                       20
```

As N differs between groups, let's randomly sample 20 elements within the unprotected category. To ensure reproducibility
data, let's also use the set.seed function:

```
> set.seed(200)
>t.test(mydata[which(mydata$status=="protected"),35]
, sample(mydata[which(mydata$status=="unprotected"), 35],20))


Welch Two Sample t-test
data: mydata[which(mydata$status == "protected"), 35] and sample(mydata[which(mydata$status
== "unprotected"), 35], mydata[which(mydata$status == "protected" ), 35] and 20)


t = -2.116, df = 29.83, p-value = 0.0428
alternative hypothesis: true difference in means is not equal to 0

95 percent confidence interval:
  -33.9029775 -0.5970225
```

sample estimates:
mean of x mean of y 39.55 56.80

*t-test can be used in a one-tailed or two-tailed manner. The default is two-tailed. To change, add the argument alternative=(“greater”,”less”,”two.sided”).

Mann-Whitney-Wilcoxon test (non-parametric version of the t-test):

```
> set.seed(200)
>wilcox.test(mydata[which(mydata$status=="protected"
),35],sample(mydata[which(mydata$status=="unprotecte
d"), 35], 20))

Wilcoxon rank sum test with continuity correction
data: mydata[which(mydata$status == "protected"), 34] and sample(mydata[which(mydata$status
== "unprotected"), 35], mydata[which(mydata$status == "protected" ), 35] and 20)

W = 141, p-value = 0.1133
alternative hypothesis: true location shift is not equal to 0

# note that the non-parametric test did not show a significant difference. This is
because this test is less robust than the parametric test and requires greater differences
between groups to indicate a significant result.
```

Comparison between > 2 means

ANOVA (Analysis of Variance)

Analysis of variance tests whether there is a difference between at least two means in a set of 2 or more means (categories). To perform this analysis, we will use the lm() function. In this function it is necessary to specify a factor. Let's create a dummy factor called Factor 1:

linear model

```
> Factor1<-c(rep(1,5),rep(2,5),rep(3,5))
> Factor 1
  [1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
> Factor1<-factor(Factor1) # this function indicates that the factor object data should be read as
categorical - a factor with 3 levels. Check out:

> Factor 1
  [1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
Levels: 1 2 3
```

Now let's generate a vector with 15 observations:

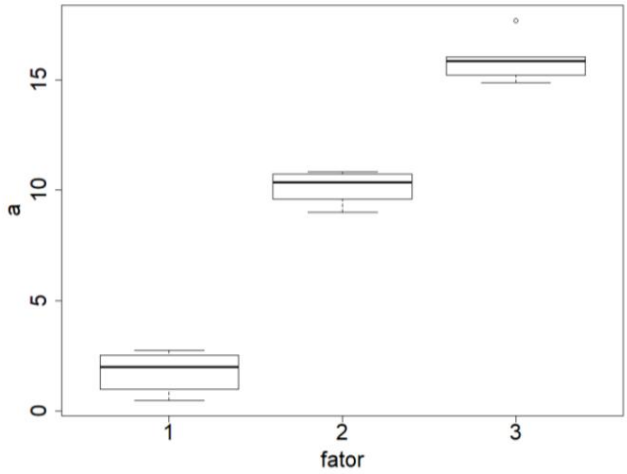
```
> set.seed(176)
> a <- c(rnorm(5, mean=2, sd=1),rnorm(5, mean=10, sd=1),rnorm(5, mean=15, sd=1))

# note that random numbers with the same standard deviation but different means were generated
```

Now the ANOVA:

```
> model1 <- lm(a~Factor1)
> summary(model1)

Call:
lm(formula = a ~ Factor1)

Residuals:
     Min       1Q Average       3Q      Max
-1.6842 -0.8113 -0.1642 0.8405 1.5556

Coefficients:
                    Estimate Std. Error t value Pr(>|t|) 0.7655 -6.415 2.29e-05 ***
 (Intercept) -4.9105 Factor1 7.0889
                                                0.3543 20,006 3.78e-11 ***
---
Meaning codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1


Residual standard error: 1,121 on 13 degrees of freedom
Multiple R-squared: 0.9685, Adjusted R-squared: 0.9661 F-statistic: 400.2 on 1 and 13 DF, p-value:
3.784e-11
```

Let's inspect the graph:

```
> boxplot(a~Factor1,cex.axis=2,ylab="a",xlab="Factor 1",cex.lab=2)

# test by redoing vector a with different standard deviations and see what happens.
```



Kruskal-Wallis test (nonparametric version of ANOVA):

```
> model2 <- kruskal.test(a~Factor1)
> model2

Kruskal-Wallis rank sum test

date: a by Factor1
Kruskal-Wallis chi-squared = 12.5, df = 2, p-value = 0.00193
```

Finding the differences (Tukey's (parametric) and Mann's post-hoc test Whitney-Wilcoxon (nonparametric))

Tukey's test requires an aov-formatted object:

```
> b <- aov(a~Factor1)
> TukeyHSD(b)
  Tukey multiple comparisons of means
        95% family-wise confidence level

Fit: aov(formula = a ~ Factor1)

$`Factor1`
                    diff lwr upr p adj
2-1 8.347145 6.728374 9.965916 0.0e+00
3-1 14.177751 12.558980 15.796522 0.0e+00
3-2 5.830607 4.211836 7.449378 1.5e-06
```

For the Mann-Whitney test, the function is pairwise.wilcox.test:

```
> pairwise.wilcox.test(a,Factor1,p.adjust.method = "bonferroni")

Pairwise comparisons using Wilcoxon rank sum test
```

date: a and Factor1

1 2 2 0.024 -

3 0.024 0.024

P value adjustment method: bonferroni
look in ?p.adjust for more methods of correcting p-values

2-way ANOVA

In 2-way ANOVA there are 2 factors and all combinations of levels of the two factors must be present. For this we will modify Factor 1 and create a second factor (Factor 2):

```
> Factor1 <- c(rep("High",6),rep("Low",6))
> Factor1 <- factor(Factor1)
> Factor 1
 [1] High High High High High High Low Low Low
Low Low Low
Level: High Low

> Factor2 <- rep(c("small","large"),6)
> Factor2 <- factor(Factor2)
> Factor2
 [1] little big little big little big little big little big [11] little big


Level: big small
5
# note the combination of levels:
> cbind(Factor1, Factor2)
> set.seed(1234)
> a <- c(rnorm(6,mean=2,sd=1),rnorm(6,mean=10,sd=1))

> model3 <- lm(a~Factor1 + Factor2 + Factor1:Factor2)
> library(car)# load the car package
> anova(model3)

Analysis of Variance Table

Response: a
                                Df Sum Sq Mean Sq F value                 Pr(>F) 1
Factor 1                          170,286 170,286 171,9035 1,089e-06 ***
Factor 2                          1 0.602 0.602 0.6076 0.4581
Factor1:Factor2 1 0.092 0.092 0.0928                                     0.7685
```

residuals 8 7,925 0,991

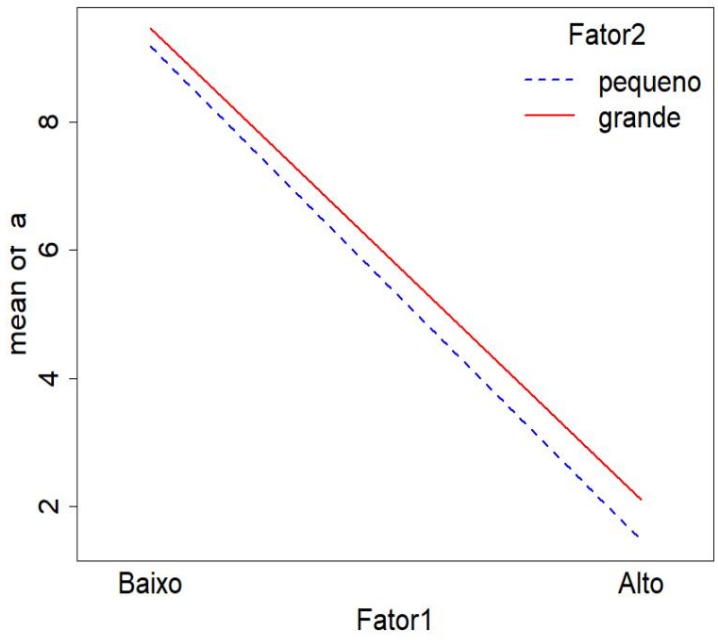
Meaning codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

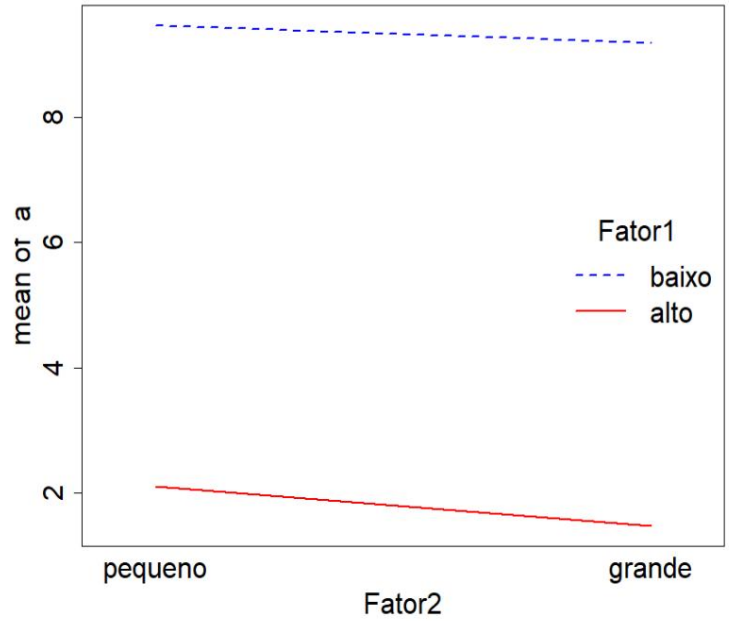
Interaction graphics

```
>Factor1 <- ordered(Factor1,levels=c("Low","High"))
>interaction.plot(Factor1,Factor2,a,cex.axis=2,cex.lab=2,lwd
=2,col=c("blue","red"),lty=c(2,1),legend="FALSE")
>legend("bottomright",c("small","large"),bty="n",lty=c(
2,1),lwd=2,col=c("blue","red"), title="Factor2",cex=2)

>Factor2<-ordered(Factor2,levels=c("small","large"))
>interaction.plot(Factor2,Factor1,a,cex.axis=2,cex.lab=2,lwd
=2,col=c("blue","red"),legend="FALSE")
>legend("right",c("low","high"),bty="n",lty=c(2,1),lwd=2
,col=c("blue","red"), title="Factor1",cex=2)
```

The levels of categorical variables are by default placed in alphabetical order. The ordered function places the levels in the order that is most logical or convenient:

plot the two interaction graphs:





We will now test the ANOVA assumptions, the homogeneity of variances of the groups (homoscedasticity, also with the car package) and the normality of the residuals:

```
> leveneTest(a, Factor1)
Levene's Test for Homogeneity of Variance (center = median)

      Df F value Pr(>F)
group 1 0.0593 0.8125
      10

> leveneTest(a, Factor2)
Levene's Test for Homogeneity of Variance (center = median)

      Df F value Pr(>F)
group 1 2.5083 0.1443
      10

> model1 <- lm(a~Factor1) >
model1$residuals >
shapiro.test(model1$residuals)# Shapiro-Wilk normality test.


 Shapiro-Wilk normality test
date: model1$residuals
W = 0.96142, p-value = 0.8039

> model2 <- lm(a~Factor2)
> model2$residuals
> shapiro.test(model2$residuals)
```

```
Shapiro-Wilk normality test
date: model2$residuals
W = 0.79249, p-value = 0.007707
```

The two factors showed homogeneity of variances, as expected.
Regarding the residuals, Factor 1 showed homogeneity of variances between the groups, which did not occur with Factor 2. Try to perform some transformation of the variable a to adjust the normality of the residuals.

Association tests between variables

One of the ways to test the association between two variables, that is, if when there is an increase in one, there is an increase or decrease in the other, but no cause and effect relationship is assumed between them, is the correlation. Calculating a correlation index in R is very easy with the color function:

```
> to <- 1:10
> b <- 16:25
> color(a,b)
   [1] 1 # perfect correlation. Visualize using the plot function.

> b[3] < -10 +including some variability in the
  Dice
> color(a, b)
 [1] 0.8408916
# note that the correlation was lower. View again
  using the plot function
```

Let's now extract the probability value associated with the correlation index. Note that the default is Pearson correlation. To change the index, use the method=(”pearson, ”spearman”, “kendall”) argument.

```
> color.test(a, b)

Pearson's product-moment correlation
date: a and b
t = 4.3947, df = 8, p-value = 0.002303
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
  0.4489729 0.9614700
sample estimates:
            color
0.8408916

> cor.test(a,b,method=”spearman”)

Spearman's rank correlation rho
date: a and b
```

S = 6, p-value < 2.2e-16
alternative hypothesis: true rho is not equal to 0
sample estimates:
rho
0.9636364

The correlation is always bivariate, but we can apply the test to a matrix or data frame, and then a correlation matrix with pairwise indices is calculated.
For this we will load the dataset “Abioticos.txt”:

> Abioticos<-read.table("Abioticos.txt", header=TRUE, row.names=1)

> color(Abiotics)

	Temp	salt	OD	No	P
Temp	1.00000000	0.71346946	-0.6434480	-0.3126558	0.07378633
Salt	0.71346946	1.00000000	-0.9422687	-0.4248663	-0.09584155
OD	-0.64344804	-0.94226867	1.0000000	0.4727692	0.11588103
No	-0.31265582	-0.42486629	0.4727692	1.0000000	0.44166642
P	0.07378633	-0.09584155	0.1158810	0.4416664	1.00000000

Now calculate the p-values. For this we will need the Hmisc package.

> correlation<-rcorr(as.matrix(Abioticos),type="pearson")
> correlation
>write.table(correlacao$P,"clipboard",sep="\t")# exporting data to excel. This command copies the table of p-values to the clipboard. Open an excel sheet and press CTRL+V.

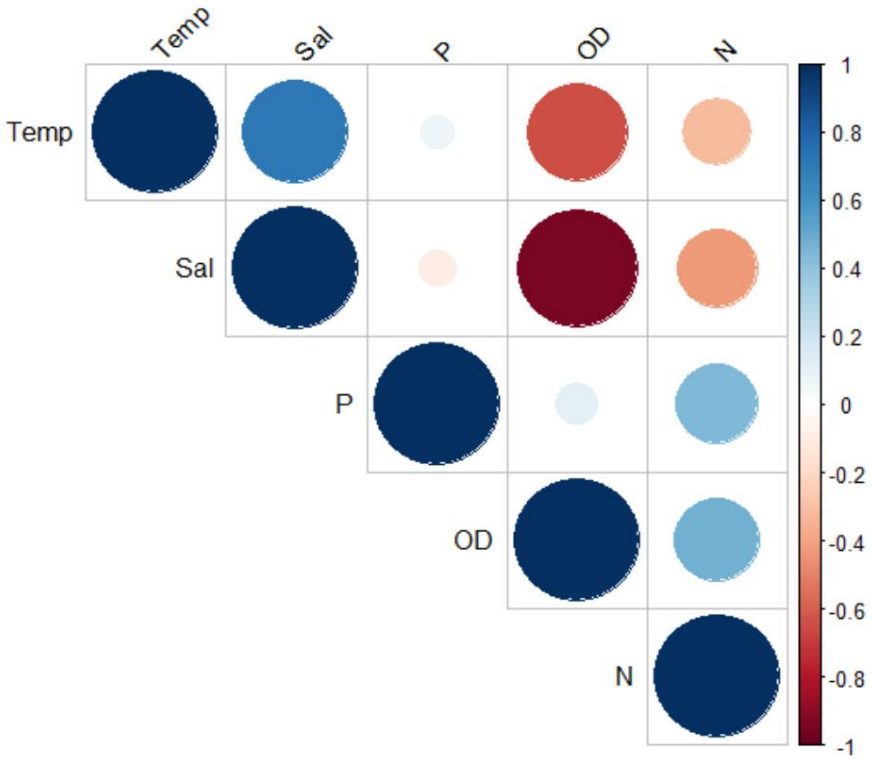
> write.table(correlacao$r,"clipboard",sep="\t")# performs the same function, but with the values of r

Another interesting tool for visualizing correlation matrices is offered by the corrplot command of the eponymous package:

> library(corrplot)# if you haven't already installed it, install it before loading it.

> Abioticos.cor<-cor(Abioticos)
> corrplot(Abioticos.cor, type = "upper", order = "hclust", tl.col = "black", tl.srt = 45)

the type=”upper” arguments keep only the upper diagonal, tl.col=”black” adds the black letters (default is red) and tl.srt=45 controls the angle of the texts.



~~linear regression~~

One of the ways to measure the association between two variables when interference is assumed is simple linear regression. For this we will continue using the “Abioticos” dataset, but using the attach function so that we can use the variables directly:

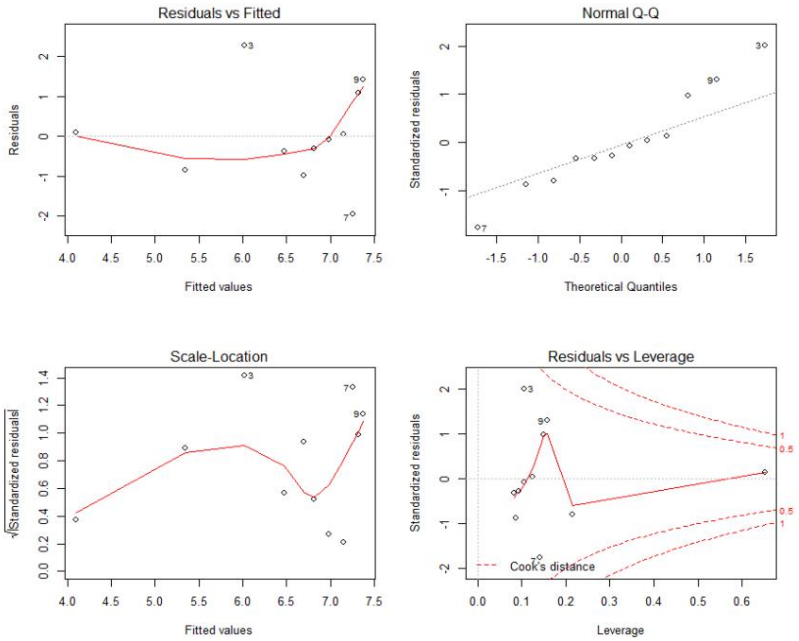
```
> attach(Abiotics)
> model1<-lm(OD~Temp)# remember that the tilde means “modeled by”. In this case, y~x
(response variable y modeled by the explanatory variable x).

> summary(model1) #2

Call:
lm(formula = OD ~ Temp)
Residuals:
      Min       1Q Average       3Q      Max
-1.9623 -0.4894 -0.1953 0.3451 2.2799 Coefficients:

                    Estimate Std. Error t value Pr(>|t|)
(Intercept) 21.6042 Temp -0.5646      5.6928 3.795 0.00351 **
                                      0.2124 -2.658 0.02398 *
---
```

[sic] To better understand the difference between multiple and adjusted r2 , see the page: http://thestatsgeek.com/2013/10/28/r-squared-and-adjusted-r-squared/

Meaning codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1,198 on 10 degrees of freedom
Multiple R-squared: 0.414, Adjusted R-squared: 0.3554 F-statistic: 7.066 on 1 and 10 DF, p-value: 0.02398

> plot(Temp,OD) # always in (x,y) order. Note that the variables are in a different order than lm().
> abline (model1)
> par(mfrow=c(2,2))
> plot (model1)# this command will plot windows in which it will be possible to perform the regression diagnosis3:



The first plot (residuals vs. fitted) indicates whether there is any non-linear trend in the data; ideally, the points are distributed randomly and symmetrically around the horizontal dashed line 0, with no tendency to increase or decrease with the adjusted values.

The second plot() shows whether the residuals are normally distributed; ideally, all points should be on the line. Normal QQ

The third plot () shows the distribution of residuals along the amplitude of the predictor variable, that is, homoscedasticity. Ideally, the residues should be evenly distributed around the red line, which should be horizontal. Care must be taken with the “cone” effect (residue amplitude increasing or decreasing). Scale-Location

[3] https://data.library.virginia.edu/diagnostic-plots/

The last plot() identifies the points that are influencing the regression (outliers) to extreme points. Care should be taken with points that fall above the upper and lower right corners (outside the outermost dotted line), determined by the Cook distance.

residuals vs Leverage

Based on the plots above, what do you think? Repeat analysis and diagnosis of regression performing data transformations (eg log).

It is important to note that these diagnostic plots are only visual and serve for a global view of the regression assumptions. Some analyzes with few points may not be effective in reliably determining whether the analysis meets the assumptions or not. To have a little more objectivity, let's perform a test of normality of residuals, the Shapiro-Wilk test:

```
> model1$residuals # model residuals
> shapiro.test(model1$residuals)

Shapiro-Wilk normality test
date: model1$residuals
W = 0.95442, p-value = 0.7023
# there is a high probability that the residuals have the same distribution as a
theoretical normal distribution (p = 0.7023), so it is concluded that the distribution of
the residuals is normal.
```

This test can be applied to a variable as well, to see if the original data obeys a normal distribution:

```
> shapiro.test(Temp)
Shapiro-Wilk normality test
date: temp
W = 0.82945, p-value = 0.02066

> shapiro.test(OD)
Shapiro-Wilk normality test
date: OD
W = 0.95821, p-value = 0.758
```

Let's now see how a multiple regression is performed. This type of analysis is performed when there is a response variable (Y) and several explanatory variables (X1, X2, X3…). The commands are very similar to a simple linear regression, the only difference being that you must place all explanatory variables with a (+) sign between them:

```
> mult1<-lm(OD~Temp+Sal+N+P)
> summary(mult1)

Call:
```

lm(formula = OD ~ Temp + Salt + N + P)
Residuals:
Min 1Q Average 3Q Max
-1.17928 -0.13788 0.08296 0.24712 0.63636
Coefficients:
Estimate Std. Error t value Pr(>|t|)
(Intercept) 9.140105 3.669674 2.491 0.04155 * 0.059603 0.156669 0.380 0.71490 Temp
Salt -0.197072 0.037882 -5,202 0.00125 **

No 0.002432 0.003624 0.671 0.52364
P -0.002879 0.016181 -0.178 0.86383

Meaning codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.6015 on 7 degrees of freedom
Multiple R-squared: 0.8965, Adjusted R-squared: 0.8374 F-statistic: 15.16 on 4 and 7 DF, p-value: 0.001474

Note that the salinity (Salt) variable is the only significant one. Let's repeat the analysis, suppressing the non-significant variables one by one, starting with the least significant (P). This method of selecting variables that will compose the linear model is called
stepwise backwards :

> mult1<-lm(OD~Temp+Sal+N)
> summary(mult1)

Call:
lm(formula = OD ~ Temp + Salt + N)
Residuals:
Min 1Q Average 3Q Max
-1.18172 -0.13435 0.06138 0.27564 0.60366
Coefficients:
Estimate Std. Error t value Pr(>|t|)
(Intercept) 9.259208 3.382690 2.737 0.02556 * 0.053050 0.142764 0.372 0.71985 Salt Temp

-0.196555 0.035410 -5,551 0.00054 ***
No 0.002137 0.003021 0.707 0.49936

Meaning codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.5639 on 8 degrees of freedom
Multiple R-squared: 0.8961, Adjusted R-squared: 0.8571 F-statistic: 22.99 on 3 and 8 DF, p-value: 0.000275

Salinity remained significant, but two non-significant variables remained (
temp and N). Let's remove the least significant first (temp):

```
> mult1 <- lm(OD~Sal+N)
> summary(mult1)

Call:
lm(formula = OD ~ Salt + N)

Residuals:
      Min 1Q Average 3Q Max
-1.16162 -0.17680 0.09292 0.28536 0.58464
Coefficients:
                    Estimate Std. Error t value Pr(>|t|)
 (Intercept) 10.480918 0.756635 13.852 2.25e-07 ***
salt -0.187670 0.024836 -7.556 3.48e-05 ***
No 0.002120 0.002872 0.738 0.479
---
Meaning codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1
' ' 1
Residual standard error: 0.5363 on 9 degrees of freedom
Multiple R-squared: 0.8943, Adjusted R-squared: 0.8708 F-statistic: 38.06 on 2 and 9 DF, p-value:
4.063e-05
```

The salinity became even more significant, but there was still nitrogen, which is not significant. Finally, we will remove this variable, resulting in a simple linear regression, which has already been performed in the previous examples.

Let's perform a second example with the dataset trees:

```
> mult2 <- lm(Volume~Girth+Height,data=trees) # Note that the command
data=trees is needed to indicate where the variables are because attach(trees) was not
used.


Call:
lm(formula = Volume ~ Girth + Height, data = trees)
Residuals:
      Min 1Q Average 3Q Max
-6.4065 -2.6493 -0.2876 2.2003 8.4847 Coefficients:

                    Estimate Std. Error t value Pr(>|t|) 8.6382 -6.713 2.75e-07 ***
 (Intercept) -57.9877 Girth 4.7082
                                        0.2643 17.816 < 2e-16 ***
height 0.3393 0.1302 2.607 0.0145 *
---
Meaning codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1
' ' 1
Residual standard error: 3,882 on 28 degrees of freedom
Multiple R-squared: 0.948, Adjusted R-squared: 0.9442 F-statistic: 255 on 2 and 28 DF, p-value: <
2.2e-16
```

Note in this case that the variables (girth and height) are significant to determine the volume of trees. Let's include an interaction term between them to verify that this is also significant:

```
> mult3 <- lm(Volume~Girth+Height+Girth:Height,data=trees)
# another way to include the interaction term is through mult3 <- lm(Volume~Girth*Height,data=trees)

Call:
lm(formula = Volume ~ Girth + Height + Girth:Height, data = trees)

Residuals:
     Min       1Q Average      3Q     Max
-6.5821 -1.0673 0.3026 1.5641 4.6649 Coefficients:

                    Estimate Std. Error t value Pr(>|t|)
 (Intercept) 69.39632 23.83575 2.911 0.00713 ** Girth -5.85585 1.92134 -3,048 0.00511 **

Height -1.29708 0.30984 -4,186 0.00027 ***
Girth:Height 0.13465 0.02438 5.524 7.48e-06 ***
---
Meaning codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2,709 on 27 degrees of freedom
Multiple R-squared: 0.9756, Adjusted R-squared: 0.9728 F-statistic: 359.3 on 3 and 27 DF, p-value: <
2.2e-16
```

Note that the interaction term is very significant, but there is a large increase in model complexity. By the principle of parsimony, it is only justified to maintain a more complex model if it results in significant increments in explanation. Let's test if there is a significant difference between the models:

```
> anova(mult2, mult3)

Analysis of Variance Table

Model 1: Volume ~ Gyrth + Height
Model 2: Volume ~ Girth + Height + Girth:Height
  Res.Df RSS Df Sum of Sq F 28 421.92 27 198.08 1 Pr(>F)
1
two 223.84 30,512 7,484e-06 ***
---
Meaning codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1
' ' 1
```

The two models are significantly different, therefore, it is justified to adopt the more complex model containing the interaction.

Multivariate Statistics: Cluster Analysis and Sorting Methods

While uni and bivariate statistics analyze how one or several independent variables (x) influence a single dependent variable (y), in multivariate we can also have different response variables (several Y). In these cases, statistics usually work on ranking in 2 or 3 dimensions the variables with greater variance (as in the case of principal components analysis; PCA). This reduction to two-dimensionality normally entails loss of information; but, on the other hand, it simplifies both n-dimensionality and the correct interpretation of graphs.

Among a range of possibilities, due to the introductory proposal of this document, we will focus on only three types of multivariate analysis: cluster, principal components analysis and non-metric multidimensional ordering. The focus, however, will be on the R commands and the generated graphical product; and not on the logical merit of statistics, which we will leave for another opportunity.

Cluster analysis (cluster)

One of the simplest exploratory multivariate analyzes is cluster analysis, which creates a dendrogram of relationships between sample units, based on similarity or dissimilarity (distance) between them. There are numerous measures of similarity and dissimilarity (ecological indices), such as Jaccard, Bray-Curtis, Euclidean distance, Hellinger distance, etc. The discussion about selection criteria for the best index is beyond the objectives of this booklet; here we will present a demonstration of an application, always remembering that the user has great freedom to adjust all the ideal parameters for his case.

We are going to work with a new dataset: () analysis of planktonic bacteria in the fingerprinting Lagoa dos Patos estuary (RS) over a year (June 2010 to May 2011). During the period to a La Niña event, a phenomenon that drastically affects the salinity of the estuary. Since the composition of bacteria is strongly affected by salinity, the hypothesis is that the composition of the bacterial community will vary over time. To access the database, open the “Cluster_TTGE_Barra” worksheet and copy the data to the clipboard, loading in R:

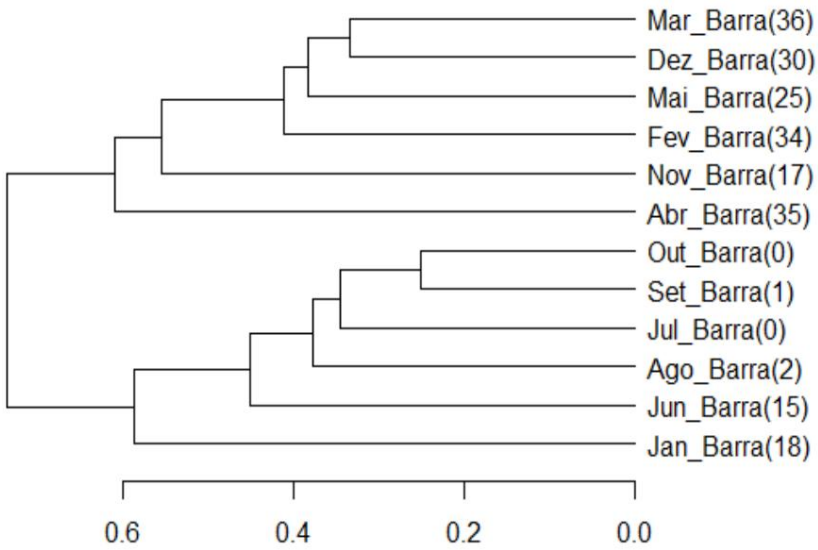
Temporal Temperature Gradient Gel 01 to May 2011 during the transition from an El Niño

```
> TTGE <- read.table("clipboard",row.names=1,header=F)
> TTGE
# observe the structure of the data. Note that the data are presence/absence for each bacterial
Operating Taxonomic Unit (OUT)

> TTGE.frame<-as.data.frame(TTGE)
# load the “vegan” package
```

> TTGE.frame.bray<-vegdist(TTGE.frame,"bray") # vegdist calculates the matrix of distances between sample units using the Bray-Curtis distance. Look at the structure of the distance matrix

> TTGE.frame.bray >TTGE.frame.bray.clust<- hclust(TTGE.frame.bray,method="average") # hclust selects the grouping method, in this case “average” is the UPGMA (Unweighted Pair Group Method with Arithmetic Mean). > par(mar=c(5,5,5,10))

> TTGE.dendrogram<-as.dendrogram(TTGE.frame.bray.clust)
> plot(TTGE.dendrogram, horiz=T)
see vegdist commands for more distance matrix options and hclust for more grouping method options.



Does the hypothesis seem consistent? Note that two large groups were formed, one with samples from June to October 2010 and January 2011 and another with samples from November to December 2010 and February to May 2011. With the exception of the January sample, the pattern seems quite plausible. , as the samples from the first period have lower salinity (0-18), while the second group has higher salinity (17-36), with little overlap.

However, we can improve the chart layout a bit:

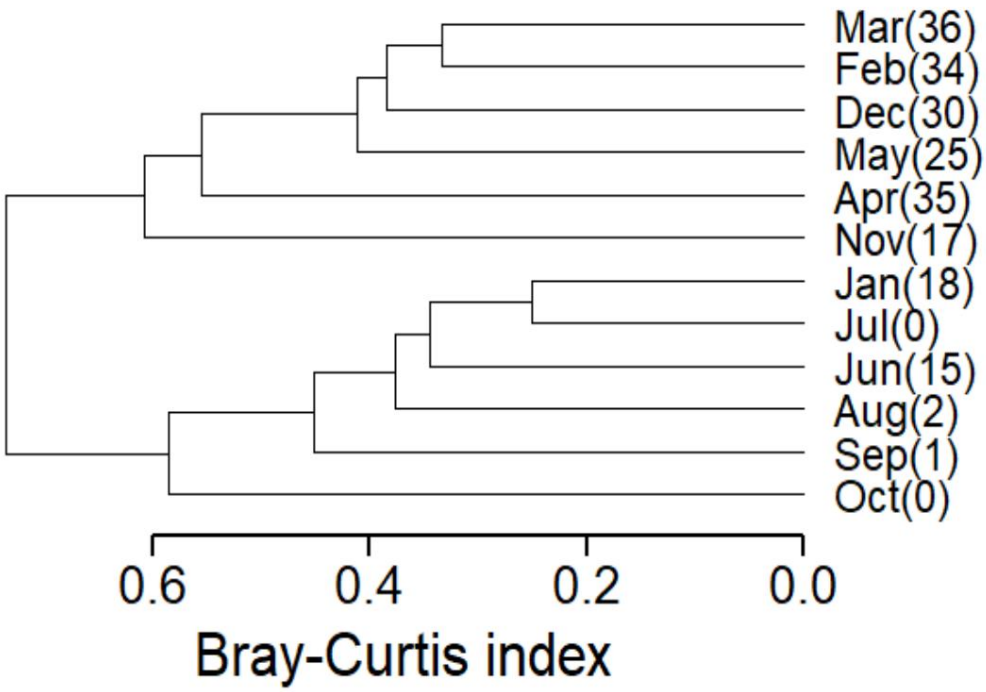
> labels<-c(" "," "," "," "," "," "," "," "," "," "," "," "," ")

> TTGE.frame <- as.data.frame(TTGE,row.names=labels)
> TTGE.frame.bray <- vegdist(TTGE.frame,"bray")
> TTGE.frame.bray.clust <- hclust(TTGE.frame.bray, method="ave>rage")

> TTGE.dendrogram<-as.dendrogram(TTGE.frame.bray.clust)

```
> TTGE.labels <-
c("Oct(0)","Sep(1)","Aug(2)","Jun(15)","Jul(0)","Jan(18)",
"Nov(17)","Apr(35)","May(25)","Dec(30)","Feb(34)","Mar(36)
")
> par(mar=c(5,5,5,6))
> plot(TTGE.dendrogram,axes=F,lwd=3,horiz=T)
> axis(1,lwd=2,cex.axis=1.75)
> axis(4,at=1:12,lab=TTGE.labels,
cex.axis=1.5,tick=F,line=-1,las=2)
> mtext("Bray-Curtis index",side=1,line=3,cex=2)
```

#used a trick to remove the original labels from the data to customize them with the TTGE.labels command.
We suppress the axes in the plot command by putting axes=F and ask the cluster to be plotted horizontally and not vertically through horiz=T (horiz=V is the default). The axes were later placed one by one with the axis commands and the x-axis label was added through the mtext command indicating the similarity index used.

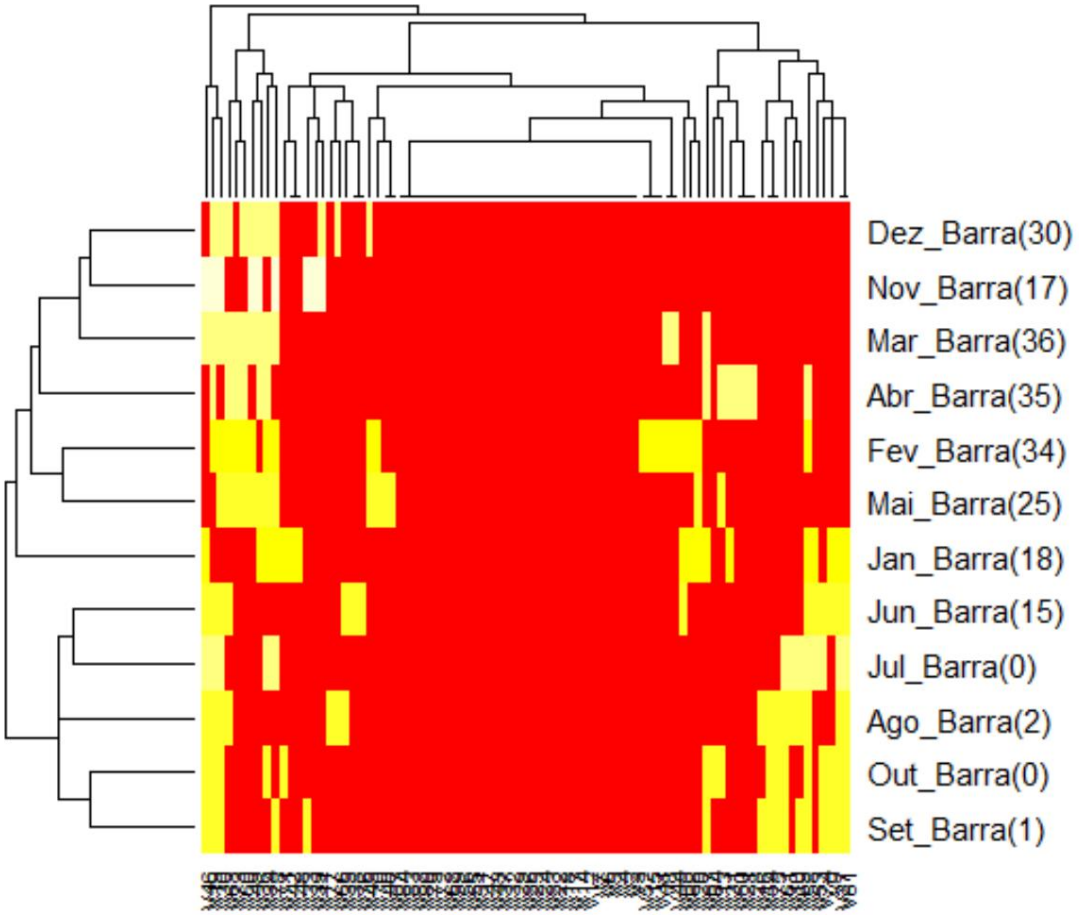


Another interesting cluster analysis feature is the heatmap, which plots two combined dendrograms, one by sampling units (right) and one by variables (top), producing a relative scale with colors () indicating the mean of the lines (units samples) and in the columns (variables in the center. false color image

For this we are going to use the data frame and force it to be read as a matrix through

from the as.matrix command. A series of parameters can be controlled through arguments inside the heatmap function (for example the order in which they are presented), but here we will only indicate the Bray-Curtis distance:

```
> heatmap(as.matrix(TTGE.frame,dist="bray") )
```



Principal component analysis

Principal Component Analysis (PCA) is one of the simplest and most straightforward ways of ordering variables in a two-dimensional space. The primary use of PCA is to reduce the dimensionality of multivariate data. In other words, we use PCA to create a few key variables that are likely to characterize the entire range of data. The basic precept is that when transposing the variables to two arbitrary axes, they will always be orthogonal to each other, that is, they are not correlated with each other (independent). The axes are ordered according to the variance (inertia) of the data: axis 1 has most of the data variance, while axis 2 has the second and so on.

We will now present one of the ways to make a PCA in R, as well as plot the result. As an example, we will use a dataset from the work of MacLaren et al. (2017) on the disparity of the mandibles of

herbivorous dinosaurs.

First, as in all cases, we are going to upload the data to R. We are going to follow the standard command, although there are several ways to do this step. At this point you can already make the decision about which commands you prefer to insert data in R.

```
> myd <- read.csv("MacLaren.csv", na.strings="?") # note the “?” instead of ";"
```

As the data is not known to most readers, it is always important to do this first analysis that we call: knowing the structure of the data. In R, these are some of the commands, which were previously presented:

```
> str(myd)
> dim(myd)
> head(myd)
> summary(myd)
```

In this dataset, we have morphometric measurements of the jaw of several species of herbivorous dinosaurs. In total, we have 167 observations. As mentioned, the primary objective of PCA is to reduce these variables (morphometric measures) within a two-dimensional space, ordering according to the greatest variance of these measures. In PCA, which uses Euclidean distance, it is important that all measurements are in the same unit, as is the case in this example.

The first step is to put in a new dataframe only the columns with the observations of the morphological characters of the jaws of the dinosaurs. An example:

```
> data <- myd[, 4:14] # 15 morphometric variables
> group <- factor(myd$Higher.Taxonomy) #groups of
dinosaurs
```

To reduce the high variation of the data, before performing the PCA, we use a method of transforming the data:

```
> pca <- decostand(na.omit(data), method = "standardize") # note that we exclude data
that is marked as NAs

# see more in the help of the decostand function of the “vegan” package.
```

There are several ways to make a PCA in R. Here we present one of these possibilities. You will notice that although PCA is a simple and straightforward analysis, some statistical knowledge is needed to extract the

correct results in R. The first command already requires this:

```
> r2pca <- princomp(x=pca, cor=F, scores=T)
# This is a command for a PCA based on a covariance matrix (the most common). Therefore,
the argument “cor=” was marked as “F” (False). “cor=T” would return a PCA based on a
correlation matrix.
```

PCA returns eigenvectors (more related to direction) and eigenvalues (related to variance). As we are illustrating PCA, we need to extract these results (see the importance of data manipulation in R):

```
> r2vectors <- r2pca$loadings # eigenvectors
> r2values <- r2pca$sdev^2 # eigenvalues
> r2scores <- r2pca$scores # axes scores
```

We can also see the axes of the PCA:

```
> head(r2pca$scores)
```

There is a command to center the axes around zero:

```
> PCscores <- scale(r2scores)
```

Now, let's transform the scores of the eigenvalues of the first three axes into percentage and, at the same time, use a different way of naming the axes:

```
> xlab=paste("PC1
 [",100*round(r2values[1]/sum(r2values),3),"%]", sep="")
> ylab=paste("PC2
 [",100*round(r2values[2]/sum(r2values),3),"%]",sep="")
> zlab=paste("PC3
 [",100*round(r2values[3]/sum(r2values),3),"%]",sep="")
```

Let's, before plotting the PCA, assign a vector with the names of the dinosaur groups:

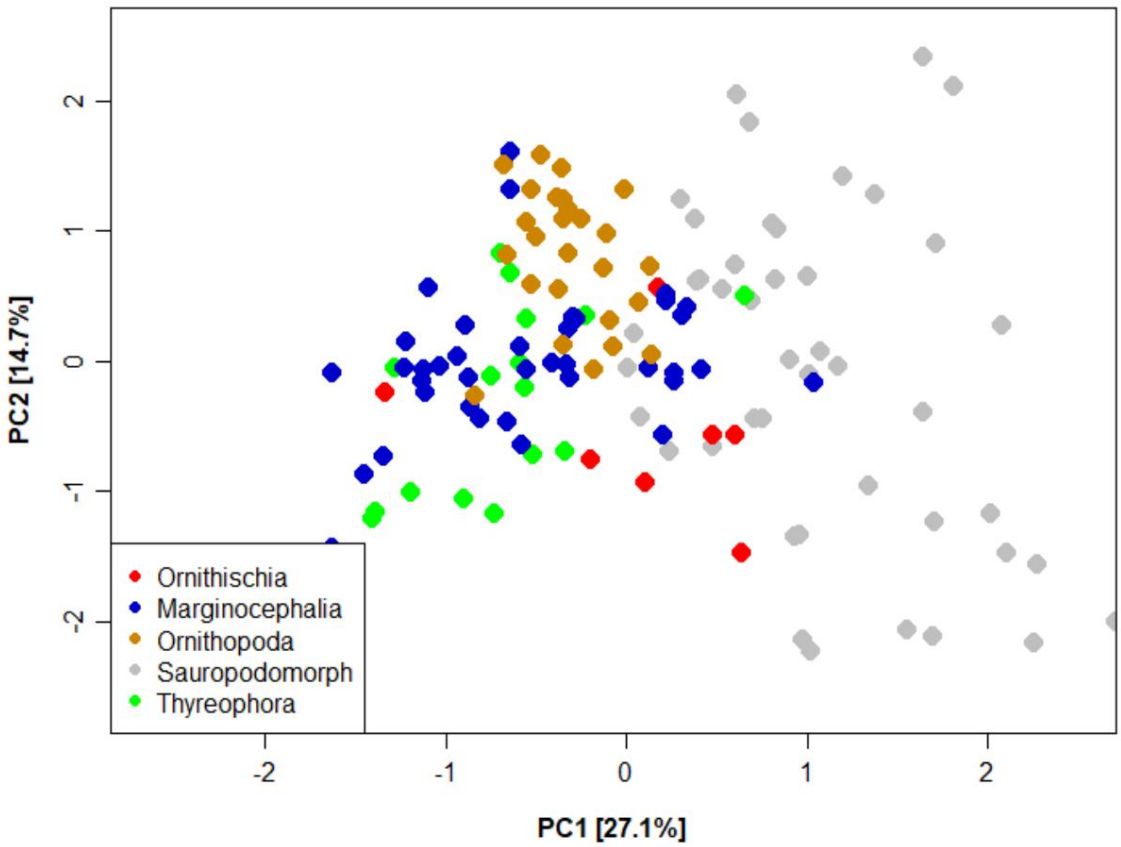
```
> name_groups <- c("Ornithischia", "Marginocephalia",
 "Ornithopoda", "Sauropodomorph", "Thyreophora")
```

Now, let's plot the PCA:

```
> op <- par(mfcol=c(1,1), mar=c(4,4,2,1), mgp=c(2.5, 0.75, 0))

> plot(PCscores[,1],PCscores[,2], xlab=xlab, ylab=ylab, cex=1.6, pch=16, col=c("red",
"blue3", "orange3", "grey75" , "green")[group], ylim=c(min(PCscores[,1]),
```

```
max(PCscores[,1])), font.lab=2)
> legend("bottomleft", name_groups, col=c("red", "blue3", "orange3", "grey75", "green"),
text.col = "black", pch=16, cex=1.0, xpd=T)

> op(pair)
```

The graphical result will be this:



In a very summarized way, notice that axis 1, the one that has most of the variation in the data, represents 27% of the variance, followed by axis 2 (14.7%).
As an exercise, they try to plot the 1 (x) axis versus the 3 (y) axis. How was the result?

Those who have seen a PCA before may be surprised by the shape of the chart.
A traditional PCA could be plotted with the command below, followed by the graphical result:

```
> biplot(r2pca) # check the result
```

However, there are still other ways to plot a PCA and other ways to perform this analysis in R. As we are systematically emphasizing, here some operations are presented only as a first introduction to the language and basic commands, without detailing the idiosyncrasies of each analysis. , as well as possible derivations. Please, for more advanced topics see others

documents and bibliographies listed in the references.

Non-metric multidimensional scaling

Non-metric multidimensional scaling (MDS, also NMDS and NMS) is a sorting technique that differs from almost all other sorting methods (like the one we saw earlier). In most sorting methods, many axes are calculated, but only a few are displayed, due to graphical limitations. In contrast, in NMDS, a small number of axes are explicitly chosen before parsing (note this in the arguments of the function that runs NMDS below). According to most other sorting methods they are analytical and therefore result in a unique solution for a given set of data. In contrast, NMDS is a numerical technique that iteratively seeks a solution and the computation is terminated when an acceptable solution is found, or after a number of trials, which is also made explicit a priori (also through a specific argument, in the case of R). As a result, NMDS is not a one-size-fits-all solution. A subsequent analysis of the same dataset following the same methodology will likely result in a slightly different ordering. This may be more evident in datasets with a smaller number of observations.

Furthermore, NMDS produces neither an eigenvector nor an eigenvalue like analysis in principal components analysis (PCA); or as in correspondence analysis (CA) which sorts the data in such a way that axis 1 explains most of the variance, while axis 2 explains the second largest variance, and so on. As a result, the NMDS sort can be rotated, flipped, or centered in any desired configuration.

NMDS routinely finds the simplest possible result in terms of sorting. Furthermore, it is possible to use any association index (Bray-Curtis, Euclideana, Manhattan, etc.). The NMDS does not generate a probabilistic value (such as a value of), but only a value of STRESS, coming from the residual sum of squares of the difference between the real variation and the simplified model. The smaller the value of STRESS, the closer is the relationship between the real distances and the differences in the NMDS. Let's try to visualize some of this in R, and take the opportunity to learn the commands in this type of analysis. To do this, let's go back to the insect data (object called “mydata”) to illustrate how to perform an nMDS ordering analysis, as well as the resulting graph. Let's try to explore different types of data manipulation first, as a way of reviewing some commands. First of all, let's remember which variables this data had:

```
> head(mydata)
> dim(mydata)
 [1] 101 35
```

Basically, there are columns of environmental data (categories) and variables related to insect species (abundance). We can separate these columns
into two subsets, one representing only the species and the other the other information:

```
> sp <- mydata[, 5:34]
> var_amb <- mydata[, 1:4]
```

Still, arbitrarily, we could exclude some rows and columns that
present few observations or few species, respectively:

```
> n <- 10
> o <- 2
```

We can then exclude this dataset (this must be done for both subsets):

```
> X <- which(colSums(sp>0)<=o)
> Y <- which(rowSums(sp)<=n)

> Data1 <- sp[-c(Y),-c(X)]
> groups <- var_amb[-c(Y), ]
```

Let's check with how many lines (observations) and with how many variables our data remained after these deletions. How many were excluded?

```
> dim(mydata) - dim(Data1)
```

Often, before the analysis itself, it can be interesting to transform our data. Here follows a modification called a double transformation. To learn more read the help of the deconstand() command.

```
> Data2 <- decostand(Data1, "max") #divide all values by the maximum value found
in a row
> NMDSdata <- decostand(Data2, "total") #divide all values by the maximum value found in
the columns
```

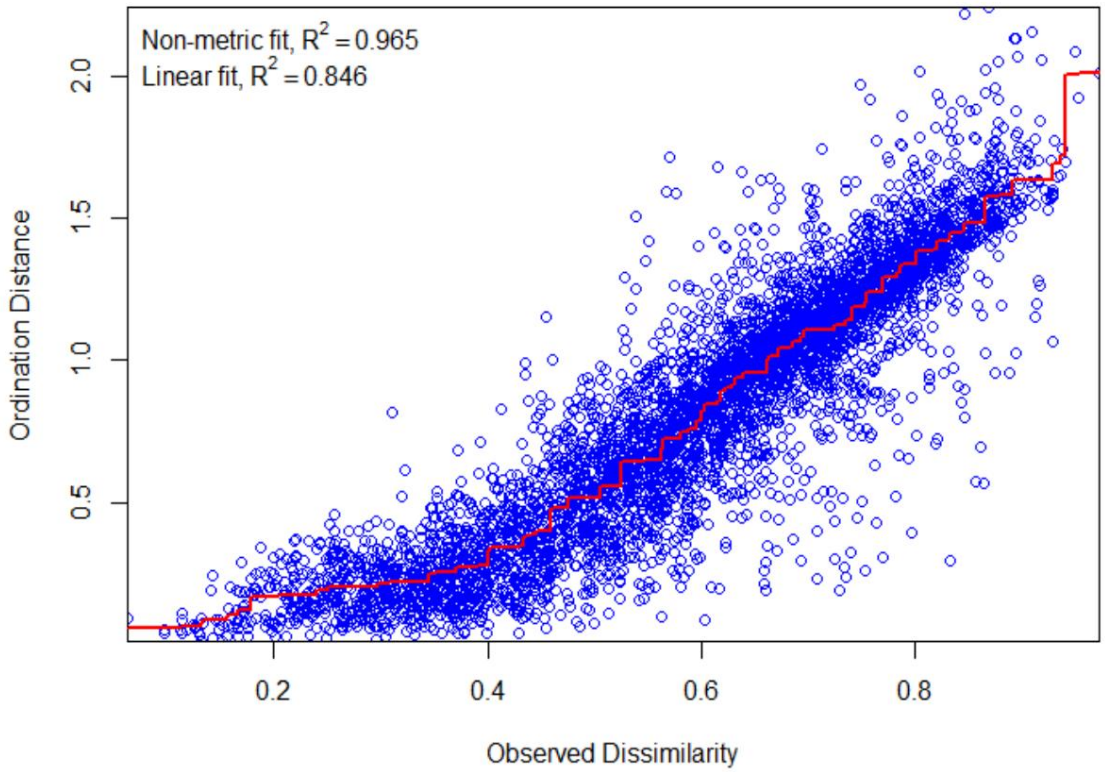
The execution of an nMDS is variable as well. Particularly, we prefer to do it in two steps. Run a first analysis and then again on the best results of the previous one. This is not, however, very common, but a particularity that varies from scientist to scientist.

```
> initial_nMDS1 <- metaMDS(NMDSdata, distance="bray", k=2, trymax=100)

> nMDS1 <- metaMDS(NMDSdata, previous.best = initial_nMDS1, k=2,
trymax=100)
# As we previously informed, it is possible to tell R as many dimensions as we want (k=), as well
as
```

how many times it will look for the simplest result (trymax=). Theoretically, the more dimensions (axes) we order, the lower the value of STRESS. Remembering that only two axes will be illustrated. Therefore, make it clear to the reader how many axes were arbitrarily chosen.

Before plotting the final NMDS result, let's have a basic idea of how the STRESS value is calculated, in graphic terms. Use the command below and observe the relationship between the observed data and those from the sort:

> scarplott(nMDS1)



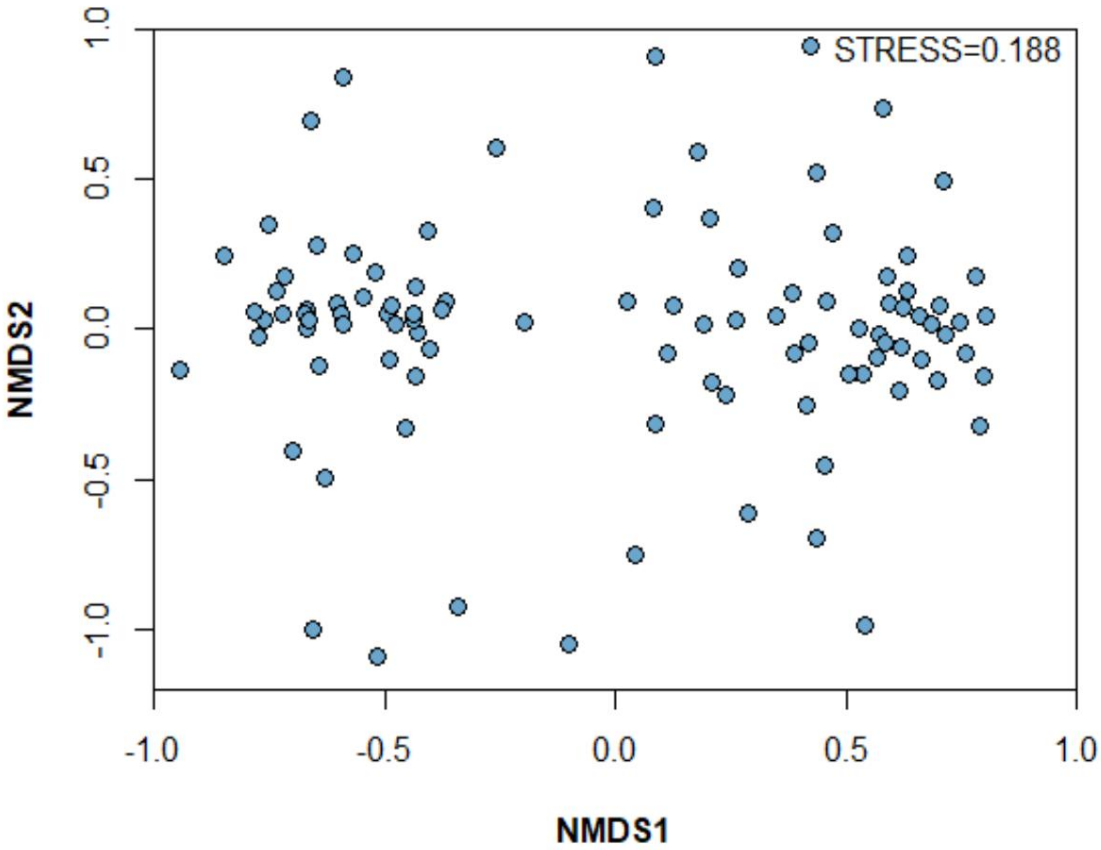
Try arbitrarily increasing both the number of dimensions (k) and number of iterations (trymax) and observe how the stressplot can vary, reflecting the NMDS STRESS value.

Now, finally, we can plot the graphical result of nMDS. Here's an example of how to do this:

```
> op <- par(mar=c(4,4,1,1))
> plot(nMDS1$points[,1], nMDS1$points[,2], font.lab=2, pch=21, bg="skyblue3",
xlab='NMDS1', ylab='NMDS2', cex=1.6 , main="", xlim=c(-1, 1), ylim=c(-1.2, 1))
```

```
mtext(side=3, line=-1, adj=0.98,
paste('STRESS=',round(nMDS1$stress, 3),sep=''), cex=1.0)
> par(op)
```

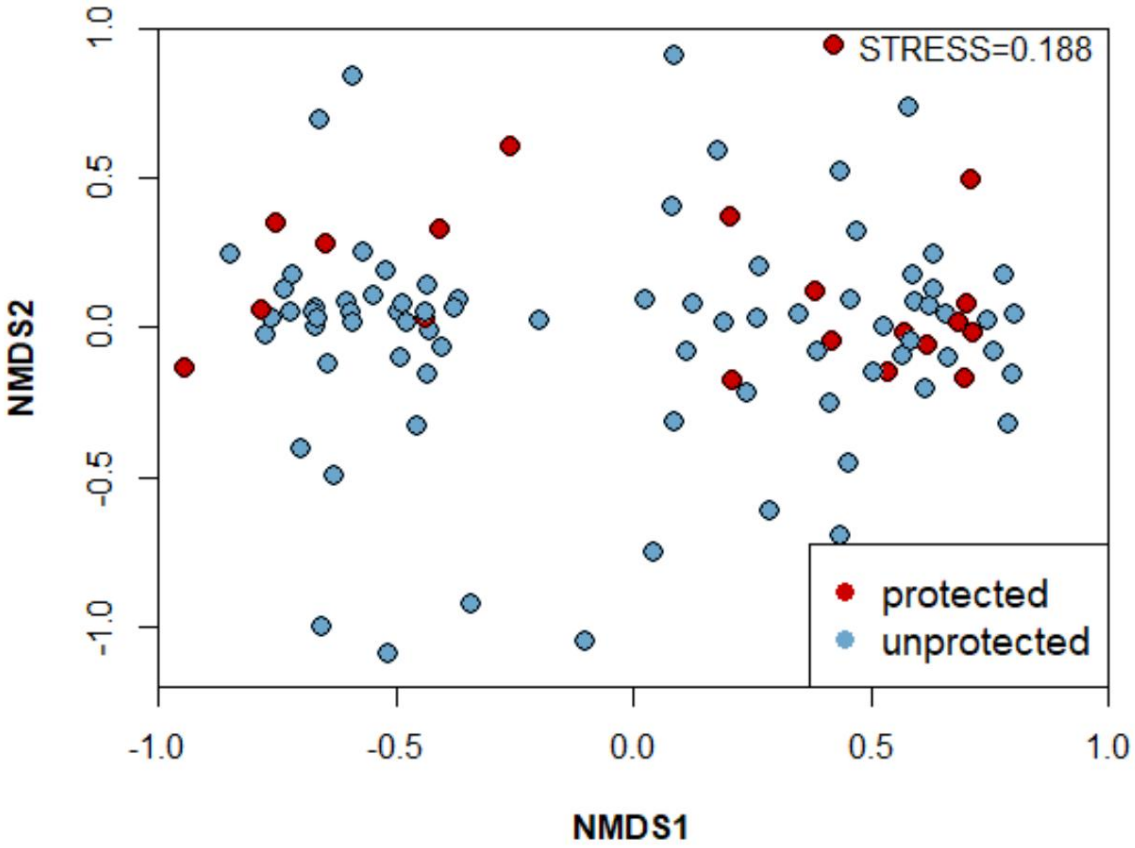
The graphical result will be approximately the following:



An interesting feature in ranking charts is to color each observation according to some factor of interest. Here, we could color in relation to the status of collection sites in terms of protected and unprotected areas. This is easily possible by manipulating arguments that have already been entered in this graph, as we will see next. Tip: R always puts vectors of characters in alphabetical order.

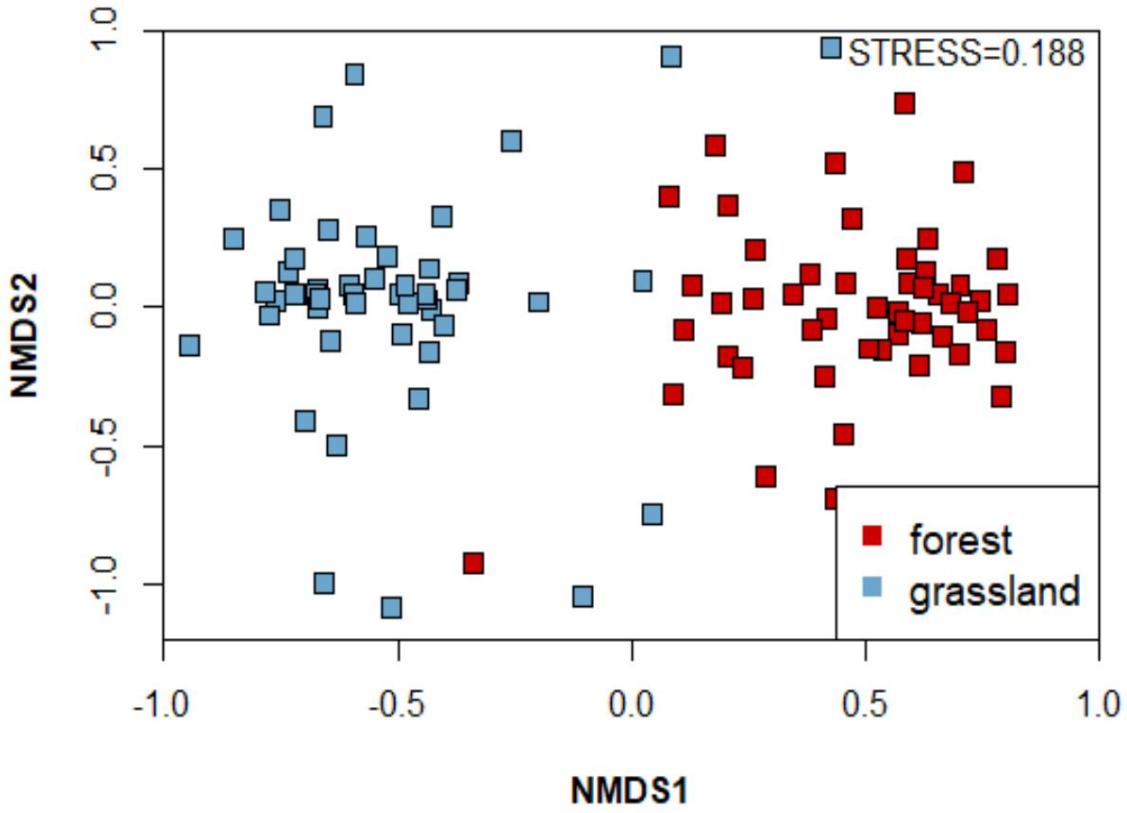
```
> op <- par(mar=c(4,4,1,1))
plot(nMDS1$points[,1], nMDS1$points[,2], font.lab=2, pch=21, bg=c("red3",
"skyblue3")[groups$status], xlab='NMDS1 ', ylab='NMDS2', cex=1.4, main="",
xlim=c(-1, 1), ylim=c(-1.2, 1))

> mtext(side=3, line=-1, adj=0.98,
paste('STRESS=',round(nMDS1$stress, 3),sep=''), cex=1.0)
> legend("bottomright", c("protected", "unprotected"), col=c('red3', "skyblue3"),

        text.col = "black", pch=16, cex=1.2, xpd=T)
```

Notice that we've added a subtitles command. The end result will be probably this one:



Now, we can try with other factors, such as habitat(). forestgrassland
Let's add some other possible arguments to increment the graph. See commands below.

```
> op <- par(mar=c(4,4,1,1))
> plot(nMDS1$points[,1], nMDS1$points[,2], font.lab=2, pch=22, bg=c("red3",
"skyblue3")[groups$habitat], xlab=' NMDS1', ylab='NMDS2', cex=1.4, main="",
xlim=c(-1, 1), ylim=c(-1, 1))

> mtext(side=3, line=-1, adj=0.98,
paste('STRESS=',round(nMDS1$stress, 3),sep=''), cex=1.0)
> legend("bottomright", c("forest", "grassland"), col=c('red3', "skyblue3"),
text.col = "black", pch=17, cex=1.2, xpd=T )
```



It is possible to see that the species seem to be distributed in a relatively standardized way between the two habitats (as opposed to the fact that the area is protected or not; previous graph). To know if these groups are different, we need to perform a multivariate statistical test to verify if the null hypothesis is true or false. A widely used and valid test in this case is the Permutation Analysis of Variance (PERMANOVA). The function that PERMAVA performs is adonis(), and is part of the “vegan” package. See the commands below:

```
> permanova <- adonis(NMDSdata ~ groups$habitat, permutations=999,
distance='bray')
> permanova >          # check the results
str(permanova) # note how the results are organized
```

An important point in R is to extract from an analysis only the information that is important at that moment, which we can still add to the graph.
For example, to extract the value of F and the value of P from PERMANOVA, we could use the following commands:

```
> obs.F <- permanova$aov.tab$F.Model[1] # value of F
> p.value <- permanova$aov.tab$Pr[1] # p-value
```

With this, we can add some new arguments to the graph. See the output of these commands:

```
> op <- par(mar=c(4,4,2,1))
> plot(nMDS1$points[,1], nMDS1$points[,2], font.lab=2, pch=22, bg=c("red3",
"skyblue3")[groups$habitat],
xlab='NMDS1', ylab='NMDS2', cex=1.4, main="", xlim=c(-1, 1), ylim=c(-1.2, 1))

> ordihull(nMDS1,
groups=groups$habitat,draw="polygon",col=c("grey90", "grey75")
[groups$habitat],label=F)
> mtext(side=3, line=-1, adj=0.98,
paste('STRESS=',round(nMDS1$stress, 3),sep=''), cex=1.0)
> mtext(side=3, line=0.2, adj=0.5, paste('PERMANOVA, p=', round(p.value, 3), ',
F=',round(obs.F, 3), sep= ''), cex=1.0)

> legend("bottomright", c("forest", "grassland"),col=c('red3',
"skyblue3"),
    text.col = "black", pch=17, cex=1.2, xpd=T)
> par(op)
```

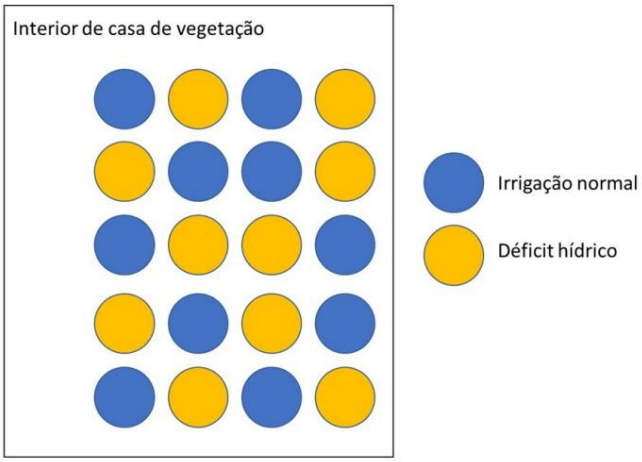
What did you find?

Analysis of biological experiments constituted according to designs with increasing levels of complexity

Analysis of DIC and DBC experiments: the difference in the block factor

Before departing for the analysis in R *per se* on this topic, we will briefly review two experimental designs that are very common in biological experiments. Whether an experiment in which the experimental units (pots, individual plants, individual birds or fish, Petri dishes containing culture medium for fungal or bacterial growth, test tubes, glass flasks, etc.) are uniform and homogeneous, there is no procedure that allows a distinction of these units in categories by similarities. In this case, the variation to be entered is the experimental treatment that will be imposed.

Let's consider a brief example to understand this design and how its analysis can be performed in R. Suppose you submit 20 seedlings of the tree species popularly known as timbaúva *(Enterolobium contortisiliquum),* produced in pots from the seeds of the same tree, the two treatments: 10 receive normal irrigation (at the substrate's field capacity) and 10 have their irrigation suspended for a significant period, aiming to induce water deficit stress. The pots are randomly distributed in a greenhouse, each one receiving one of the treatments, as shown in the figure below.



This experiment has the configuration of a *completely randomized design* (CRD): it has 10 repetitions of each treatment; repetitions are randomly distributed. The experiment consists of two treatments.

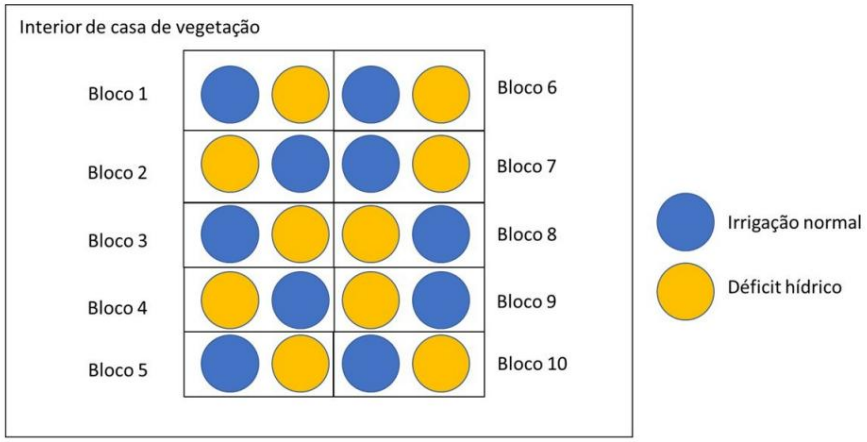
If we measure the variable height of the seedlings *(H)* after the application of the treatments and submit it to an analysis of variance, we can do the following in
R (similar procedure to that seen earlier in this booklet):

> anova = aov(h ~ treatment)

However, it is common for biological experiments to present more complex designs, especially when they are larger (larger number of experimental units) and include many treatments. This complexity is even greater when they are carried out under field conditions *(in situ* or *ex situ).*

To illustrate, let's consider the same experiment as before and insert a restriction on the drawing of experimental units. This time, we will consider that the interior of a greenhouse presents variations in the intensity of incident light radiation. One side of the house often receives more light than the other, which is more pronounced as latitude increases in both hemispheres of the planet. Under the above conditions, plants on one side could receive more light than those on the other, which often influences their growth and can interfere with the statistical evaluation of the effect of each treatment.

To circumvent this heterogeneity of the environment, we can group the experimental units in blocks. Each block will consist of an experimental unit of each treatment, that is, a vessel with normal irrigation and one with suspension of irrigation. In this way, 10 blocks can be constituted and distributed along the greenhouse, as follows.



In the figure, the experimental units remained distributed in the same way, however, we introduced the block creation constraint. Each block contains the two treatments represented once. The blocks can be arranged linearly next to each other, but they can also be distributed in other ways, according to the space of the experimental environment. It may be that block 1 has a different height growth than the seedlings in block 10, simply because of the difference in position in the greenhouse. This restriction allows us to identify the magnitude of the effect of environmental heterogeneity on the measured variable (local control). This is a simplified example of a randomized block design *(RBD)* experiment: the experimental units are grouped into groups called blocks; each block contains one unit of each treatment in the experiment; the treatments are randomly selected within each block and the distribution of the blocks is also randomized, although their numbering is sequential (for the purpose of organizing the experiment).

In R, we can analyze the data of variable *H* by implementing the block variation factor in the analysis model:

> anova = aov(h ~ block + treatment)

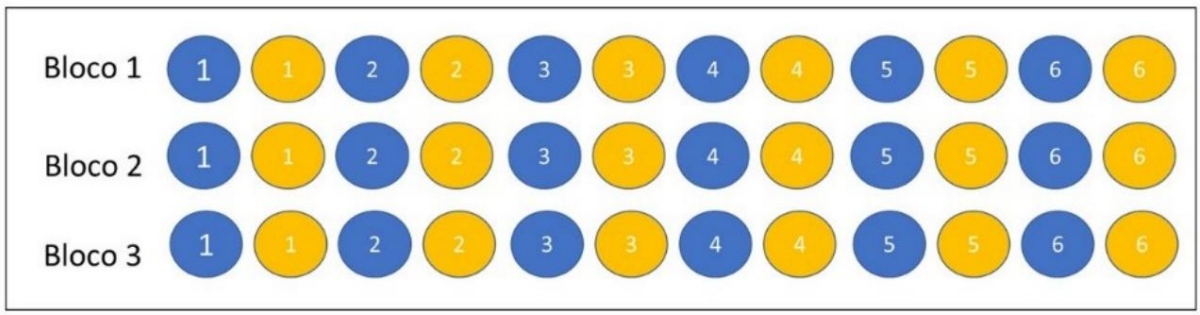
If the block factor presents statistical significance (according to the most usual criterion of $P < 0.05$), the division of the experiment into blocks was adequate, due to the environmental heterogeneity. However, block differences should not cause block and treatment interaction, as this can increase experimental error and reduce experiment accuracy (Storck et al., 2000). The verification of the interaction between blocks and treatments can be done through the model:

> anova = aov(h ~ block + treatment + blocks* treatment)

If there is no block effect (considering the statistical criterion), the same experiment could be analyzed as a completely randomized design (according to the model previously proposed for analysis in R). This is justified because the introduction of one more variation factor decreases the error degree of freedom, increasing its numerical value.

Analysis of a DBC experiment with more than one factor

We will consider data from a hypothetical experiment carried out with common bean *(Phaseolus vulgaris),* constituted in randomized blocks, using treatments similar to the previous ones (irrigation regimes). However, this new experiment introduces another level of complexity. In addition to the treatments related to the irrigation regime, six different bean genotypes were compared. We call each type of treatment a factor. Factor 1 is the genotype, with six levels (six genotypes) within this factor. Factor 2 is the irrigation regime, with two levels (normal irrigation and irrigation suspension). The experiment was composed in three replications (three blocks) and evaluated in a greenhouse. This is the case of a bifactorial experiment. Two-factor experiments involve two categories of treatments (two factors). Within each factor, the levels (number of sub-treatments of each factor) are constituted. In our case, the following scheme illustrates the experiment with three blocks accommodating the two factors (genotypes represented by numbers and irrigation regimes by colors). In the representation, however, the experimental units were not randomly selected. In a real situation, it is necessary to randomize all units of each factor and each level within each block.



Let us now analyze hypothetical data for the two-factor experiment described.
First, let's read the data in R, inserting it into a new object called data. The decimal operator of the set has already been defined as the period (.) in the data file itself, and it is not necessary to indicate it when reading the table.

```
> data = read.table(file="exp_gen_seca.txt", sep="", header=T)

> data
```

genotype	block	treatment	DPV3	1	1		PR	H	MST	MSR
			1	control	18	70.1	145.2	2.1	0.819	
two	1		1	dry	20	82.3	120.1	1.7	0.663	
3	two		1	control	19	72.5	150.5	2.2	0.858	
4	two		1	dry	20	77.6	115.2	1.6	0.624	
5	3		1	control	18	74.9	140.1	2.3	0.897	
6	3		1	dry	22	86.5	98.3	1.4	0.546	
7	1		2	control	17	56.8	110.5	2.4	0.936	
8	1		two	dry	19	52.1	65.2	1.8	0.702	
9	two		2	control	18	62.4	114.7	2.5	0.975	
10	two		two	dry	19	47.9	51.5	1.7	0.663	
11	3		2	control	17	66.3	138.3	2.6	1,014	
12	3		2	dry	20	57.8	77.4	1.6	0.624	
13	1		3	control	21	94.5	75.5	1.8	0.702	
14	1		3	dry	24	102.4	69.6	1.5	0.585	
15	two		3	control	22	97.5	80.1	1.5	0.585	
16	two		3	dry	25	118.3	75.4	1.5	0.585	
17	3		3	control	22	92.1	92.3	1.6	0.624	
18	3		3	dry	22	123.5	90.1	1.4	0.546	
19	1		4	control	16	56.8	115.3	2.1	0.819	
20	1		4	dry	22	57.4	49.8	1.7	0.663	
21	two		4	control	15	65.3	128.1	3.1	1,209	
22	two		4	dry	21	62.1	39.4	1.8	0.702	
23	3		4	control	17	72.9	136.6	3.2	1,248	
24	3		4	dry	20	67.3	65.5	1.6	0.624	
25	1		5	control	20	39.8	87.5	2.5	0.975	
26	1		5	dry	25	65.6	86.1	1.4	0.546	
27	two		5	control	21	37.4	89.4	2.6	1,014	
28	two		5	dry	26	78.3	88.3	1.4	0.546	
29	3		5	control	22	32.5	102.1	2.5	0.975	
30	3		5	dry	23	79.8	90.3	1.3	0.507	
31	1		6	control	23	82.1	117.5	2.4	0.936	
32	1		6	dry	25	100.1	56.3	1.3	0.507	
33	two		6	control	25	80.3	125.5	2.5	0.975	
34	two		6	dry	27	102.4	49.4	0.9	0.351	
35	3		6	control	24	66.5	135.3	2.5	0.975	
36	3		6	dry	27	105.6	38.5	1.5	0.585	

Next, let's read the variables of the set:

```
> attach(data)
> names(data)
 [1] "block"                 "genotype" "treatment" "DPV3"
"PR"                  "H"                  "MST"                  "MSR"
```

The following variables were hypothetically measured: *DPV3* – number of days to reach the V3 vegetative stage, which consists of the complete expansion of the first trifoliate of the plant; *PR:* root depth; H: plant height; *MST:* total dry mass; *MSR:* root dry mass.

In this dataset, the factors are: block, genotype and treatment. The treatment factor refers to irrigation regimes, although genotypes are also a treatment factor. It was just a way of identifying the factors in the table.

In R, it is essential to verify that the factors are, in fact, factors, and that the numeric variables are effectively being interpreted as numbers. This verification can be done as follows.

```
> is.factor(block)
 [1] FALSE
```

Block verification indicated that this is not a factor. So it is necessary convert it to factor:

```
> as.factor(block)
   [1] 1 1 2 2 3 3 1 1 2 2 3 3 1 1 2 2 3 3 1 1 2 2 3 3 1 1
2 2 3 3 1 1 2 2 3 3
Levels: 1 2 3
```

We proceed by checking the other factors:

```
> is.factor(genotype)
 [1] FALSE
> as.factor(genotype)
   [1] 1 1 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3 3 4 4 4 4 4 4 5 5
5 5 5 5 6 6 6 6 6 6
Levels: 1 2 3 4 5 6
> is.factor(treatment)
 [1] TRUE
```

Next, the numerical variables are verified:

```
> is.numeric(DPV3)
 [1] TRUE
```

Statistical model adequacy tests for ANOVA

Before the analysis of variance, the necessary tests are carried out to meet the basic assumptions of statistical models: normality of residuals and homogeneity of variances. Normality can be done using the Shapiro Wilk test for each variable in the table.

```
> shapiro.test(DPV3)

        Shapiro-Wilk normality test

date: DPV3
W = 0.97449, p-value = 0.5601

> shapiro.test(PR)
```

```
        Shapiro-Wilk normality test

date: PR
W = 0.98316, p-value = 0.8461


> shapiro.test(H)

        Shapiro-Wilk normality test

date: HW =
0.96595, p-value = 0.3255


> shapiro.test(MST)

        Shapiro-Wilk normality test

date: MST W =
0.93257, p-value = 0.02997


> shapiro.test(MSR)

        Shapiro-Wilk normality test

date: MSR
W = 0.93257, p-value = 0.02997
```

The analysis suggests that the *MST* and *MSR* variables are not normal, as $P < 0.05$.

The homogeneity of variances can be verified, among other tests, by the Bartlett method. For example, in the case of the variable *MST,* the test showed no homogeneity of variances in the factors treatments and genotypes:

```
> bartlett.test(MST ~ treatment)

        Bartlett test of homogeneity of variances

data: MST by Bartlett's K-squared
treatment = 7.4516, df = 1, p-value = 0.006338

> bartlett.test(MST ~ genotype)

        Bartlett test of homogeneity of variances

data: MST by Bartlett's K-squared
genotype = 11.732, df = 5, p-value = 0.03865
```

```
> bartlett.test(MST ~ block)

        Bartlett test of homogeneity of variances

date: MST by block
Bartlett's K-squared = 2.4318, df = 2, p-value = 0.2964
```

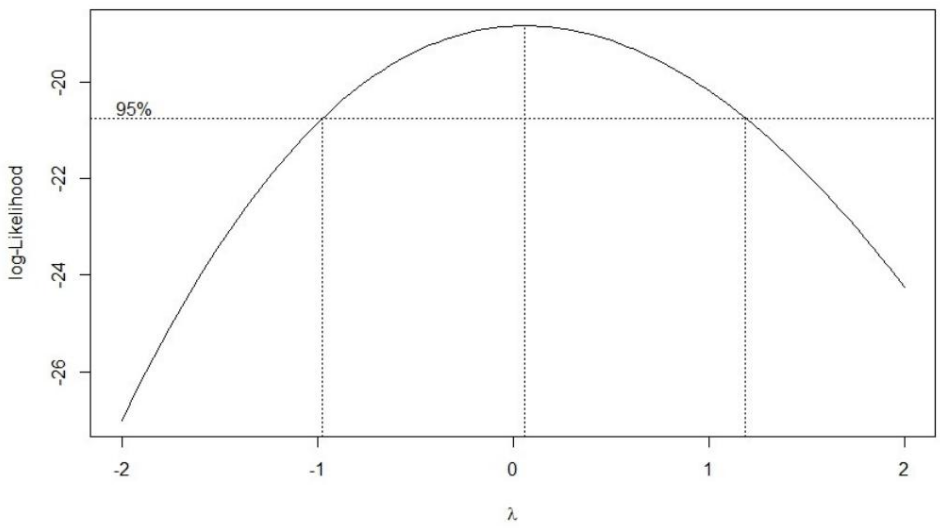
The test results suggest that there is no normality of residuals and also heterogeneity of variances in the variables *MST* and *MSR.* In this way, one can proceed with the transformation of the data for analysis of variance. One of the methods for data transformation is boxcox (Box & Cox, 1964). The test aims to stabilize or reduce existing variability and normalize residuals. The transformation is performed according to the following mathematical criteria:

$$( ) = \begin{cases} \dfrac{\ddot{y}^{\lambda} - 1}{\lambda}, & \ddot{y} \neq 0 \\ \log \ddot{y}, & \lambda = 0 \end{cases}$$

In R, the boxcox transformation can be performed using the MASS package (Venables & Ripley, 2002; Ripley et al., 2013). The boxcox() command generates a graph involving the data set to be transformed. In this graph, the maximum value of the axis will have its correspondent as the value .

```
> require(MASS)
> boxcox(MST ~ genotype)
```

By typing this last command line, a graph is generated to determine the value of .



Now, you have to type locator(), and click with the left mouse button on the top of the graph, pressing ESC next. you should get something as:

```
> locator()
$`x`
 [1] 0.06246118

$y
 [1] -18.85288
```

The value of is the value of which you will use to apply to the previous equation. You can modify the identification of your variables, for example, adding a t after it, as a way of differentiation. In the case of the *MST variable,* we will create the *MSTt* transformed variable and apply the transformation equation. Since it is nonzero, we apply the equation as follows:

```
> MSTt = ((MST ^ 0.06246118)-1)/0.06246118
```

All values of the *MST* variable will be transformed by the above expression, creating the *MSTt variable.* Now, the *MSTt* variable can be subjected to normality and homogeneity tests.

```
> bartlett.test(MSTt ~ genotype)

        Bartlett test of homogeneity of variances

date: MSTt by genotype
Bartlett's K-squared = 10.771, df = 5, p-value = 0.05611

> shapiro.test(MSTt)

        Shapiro-Wilk normality test

date: MSTt
W = 0.95137, p-value = 0.1155
```

The tests suggest that the transformed data present normality of residuals and homogeneity of variances (by the criterion of $P < 0.05$).

After verifying the basic criteria of normality and homogeneity of variances, with the necessary transformations, the analysis of variance of all variables can be performed. If any variable, even transformed, does not meet the assumptions, non-parametric tests will be conducted, as seen above.

Realization of ANOVA

In the case of the proposed example, the ANOVA can be performed in a simplified way by the following model, considering the variable *MSTt.*

```
> an = aov(MSTt ~ block + treatment + genotype + treatment*genotype)
```

In the model, each factor is analyzed separately (block, treatment, genotype), however, the interaction between treatments and genotypes is also considered. The objective is to understand if each factor has different effects, but also to verify if one factor influences (interacts) the other. Data analysis returns the following results:

```
> summary(an)
                          Df Sum Sq Mean Sq F value Pr(>F) 1 0.000004 0.000004
block                       0.036 0.8511
treatment                   1 0.007408 0.007408 60.002 9.72e-09 ***
genotype                    1 0.000033 0.000033 0.269 0.6075
treatment:genotype 1 0.000767 0.000767 6.213 0.0182 * Residuals 31 0.003828 0.000123
---
Meaning codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1'                                ' 1
```

From the analysis, there are differences between irrigation regimes, but not between genotypes. However, there is interference of some genotype(s) with the irrigation treatments. The table suggests the need to break down the analysis to see which treatments differ.

Mean comparison tests

We will make comparisons of means between treatments, using methods such as Tukey, Duncan and Scott-Knott. For this, the laercio package can be used (Silva, 2015).

```
 > require(laercio)

> LTukey(an)

  TUKEY TEST TO COMPARE MEANS

  Confidence level: 0.95 Dependent
  variable: MSTt
  Variation Coefficient: 1.068897 %

Independent variable: block
    Means Factors
    3              1.04017391891725 to
```

1 1.03931499428973 to
two 1.03915627178815 to

Independent variable: treatment
Means Factors
control 1.05389374415566 a 1.02520304584109
dry b

Independent variable: genotype
Means Factors
4 1.04948363284198 a
two 1.04624904441631 ab
5 1.03971729985892 ab
1 1.03934087552518 ab
6 1.03493703689112 ab
3 1.02756248045675 b

> LDuncan(an)

DUNCAN TEST TO COMPARE MEANS

Confidence Level: 0.95 Dependent
Variable: MSTt
Variation Coefficient: 1.068897 %

Independent Variable: block
Means Factors
3 1.04017391891725 to
1 1.03931499428973 ab
two 1.03915627178815 b

Independent Variable: treatment
Means Factors
control 1.05389374415566 a 1.02520304584109
dry b

Independent Variable: genotype
Means Factors
4 1.04948363284198 to
2 1.04624904441631 to
5 1.03971729985892 ab
1 1.03934087552518 ab
6 1.03493703689112 ab

```
      3                1.02756248045675 b

> LScottKnott(an, "genotype")

  SCOTT-KNOTT ORIGINAL TEST

  Confidence Level: 0.95 Independent
  variable: genotype

    FACTORS          MEANS
1             4 1.049484 A
two           2 1.046249 A
3             5 1.039717 A
4             1 1.039341 A
5             6 1.034937 A
6             3 1.027562 A

> LScottKnott(an, "treatment")

  SCOTT-KNOTT ORIGINAL TEST

  Confidence Level: 0.95
  Independent variable: treatment

      FACTORS          MEANS
1 control 1.053894 A
two           dry 1.025203 B
```

The three tests show some variations, especially regarding genotypes. There are some differences between the genotypes that may not have been verified by the initial analysis of variance (ANOVA). An important detail of the analysis performed concerns the use of transformed data. Effectively, we used the transformed values to conduct statistical analyses. However, the original values should be placed in the table of your report, article or monograph. Alongside the original values, information from the statistical analysis (significance levels represented by asterisks; or different letters obtained by means of comparison tests) can be used with the transformed data. You will never use transformed values in your final table.

Analysis of bifactorial experiments in the ExpDes package

A more specific way of analyzing data from bifactorial experiments can be performed in the ExpDes package (Ferreira et al., 2014). Through simplified command lines, the package allows the performance of ANOVA followed by tests of multiple comparisons, presenting the consequences of each factor in the comparisons. It is noteworthy that the package was developed, however, to perform

analysis of experiments in a completely randomized design, randomized blocks and Latin square. In addition, it allows the analysis of double and triple factorial experiments, split plots in time, among other applications.

Let's use the package to do the ANOVA and the ramifications of the factors and their interactions, considering our example of the common bean genotypes under the two irrigation regimes.

```
> require(ExpDes)
```

In the model, we use fat2 to refer to a double factorial (bifactorial), where the factors are genotype and treatment. In addition, the experiment is analyzed in block (rbd) and the variable to be analyzed is DPV3, as follows.

```
> fat2.rbd(genotype, treatment, block, DPV3)
```

This simple command line produces the series of analyzes presented in sequence, unfolding all the factors, their levels and their possible interactions.

```
> fat2.rbd(genotype, treatment, block, DPV3)
```

Legend:
FACTOR 1: F1
FACTOR 2: F2

Analysis of Variance Table

	DF	SS	MS	Fc	Pr>Fc
Block	2	2.67	1.333	1.294	0.29420
F1	5	233.67	46,733	45,359	0.00000
F2	1	75.11	75,111	72,902	0.00000
F1*F2	5	10.89	2.178	2.114	0.10189
Residuals	22	22.67	1,030		
Total	35	345.00			

CV = 4.8%

Shapiro-Wilk normality test
p-value: 0.2779497 According to Shapiro-
Wilk normality test at 5% of significance, residuals can be considered normal.

No significant interaction: analyzing the simple effect

F1

Tukey's test

Groups	Treatments	Means
The	6	25.16667
B	5	22.83333
B	3	22.66667 19.5
ç	1	18.5 18.33333
ç	4	
ç	2	

F2

Tukey's test

Groups Treatments Means
The dry 22.61111 control
B 19.72222

In the case of the variable *DTV3,* no significant interactions were detected. Thus, the complementary tests of means (Tukey) were performed within each factor (comparing the levels), ie, the six genotypes within the genotype factor were compared, as well as the two irrigation regimes of the other factor.

Note that when interactions occur, they are broken down into detailing. See the ANOVA results, for example, for the *PR variable.*

> fat2.rbd(genotype, treatment, block, PR)

Legend:
FACTOR 1: F1
FACTOR 2: F2

Analysis of Variance Table

	DF	SS	MS	fc	Pr>Fc
Block	2	184.5	92.25	2,870	0.078057
F1	5	11526.6	2305.32	71,724	0.000000
F2	1	1685.1	1685.10	52,427	0.000000
F1*F2	5	2406.6	481.33	14.975	0.000002
Residuals	22	707.1	32.14		
Total	35	16509.9			

CV = 7.59%

Shapiro-Wilk normality test
p-value: 0.0999162 According to
Shapiro-Wilk normality test at 5% of significance, residuals can be considered normal.

Significant interaction: analyzing the interaction

Analyzing F1 inside of each level of F2

Analysis of Variance Table

	DF	SS	MS	Fc	Pr.Fc
Block	2	184.505	92.2525	2.8702	0.0781
F2	1	1685.102	1685.1025	52.4275	0
F1:F2 control	5	5510.732	1102.14633	34.2903	0
F1:F2 dry	5	8422.493	1684.49867	52.4087	0
residuals	22	707,115	32,14159		

Total 35 16509947

F1 inside of the level control of F2

Tukey's test

Groups Treatments Means

The	3	94.7
	6	76.3
b	1	72.5 65
bc bc	4	61.83333
ç	2	
d	5	36.56667

F1 inside of the dry level of F2

Tukey's test

Groups Treatments Means

The	3	114.7333 102.7
The	6	
B	1	82.13333
bc	5	74.56667
cd	4	62.26667 52.6
d	2	

Analyzing F2 inside of each level of F1

Analysis of Variance Table

	DF	SS MS Fc Pr.Fc
Block	2	184.50500 92.2525 2.8702 0.0781
F1	5	11526.59583 2305.31917 71.7239 0
F2:F1 1	1	139.20167 139.20167 4.3309 0.0493
F2:F1 2	1	127.88167 127.88167 3.9787 0.0586
F2:F1 3	1	602.00167 602.00167 18.7297 3e-04
F2:F1 4 F2:F1	1	11.20667 11.20667 0.3487 0.5609
5 1 2166.00000 F2:F1 6 1 1045.44000 Residuals 22		2166 67.3893 1045.44 0
707.11500 32.14159		32.5261 0

Total 35 16509.94750

F2 inside of the level 1 of F1

Tukey's test

Groups Treatments Means

The	two	82.13333 72.5
B	1	

F2 inside of the level 2 of F1

According to the F test, the means of this factor are statistical equal.

levels	Means
1	1 61.83333
2	2 52.60000

F2 inside of the level 3 of F1

Tukey's test

Groups	Treatments	Means
The	2 1	114.7333 94.7
B	2 1	

F2 inside of the level 4 of F1

According to the F test, the means of this factor are statistical equal.

	levels	Means
1	1	65,00000
2	2	62.26667

F2 inside of the level 5 of F1

Tukey's test

Groups	Treatments	Means
The		74.56667
B	2 1	36.56667

F2 inside of the level 6 of F1

Tukey's test

Groups	Treatments	Means
The		102.7
B	2 1	76.3

In this case, there was a significant interaction between the irrigation regime and genotype factors. Each interaction was broken down into all its levels, to check for possible differences by Tukey's test *(P* < 0.05).

In ExpDes, Ferreira et al. (2014) developed command lines simplified for double and triple factorials in different designs:

- for double factorial in a completely randomized design

> fat2.crd(factor1, factor2, resp)

- for double factorial in a randomized block design

> fat2.rbd(factor1, factor2, block, resp)

- for triple factorial in a completely randomized design

> fat3.crd(factor1, factor2,factor3,resp)

- for triple factorial in a randomized block design

> fat3.rbd(factor1, factor2, factor3, block, resp)

R applied to bioinformatics: fundamentals of sequence alignments and phylogenetic reconstruction

The use of packages in R has also been extensively applied to bioinformatics analyses. Bioinformatics is a branch of science that is dedicated to the application of informatics in the analysis of data from biological studies. In other words, computational and mathematical techniques are applied to the generation and management of biological information (Attwood et al., 2011).

Among the various applications of bioinformatics, it is worth highlighting two fundamental analyzes that a professional linked to the area of biological sciences must know: (i) the alignment of nucleotide and amino acid sequences; and (ii) phylogenetic reconstruction. Let's apply some R packages to perform nucleotide sequence alignment of a small set of nucleotide and amino acid sequences.

First, you should know that nucleotide and amino acid sequences need to conform to a standard format to be represented. One of the most common formats is FASTA, where the sequence is represented by an identifier, followed by the sequence itself. The beginning of each sequence is delimited by the > symbol. See an example of FASTA format for a nucleotide sequence of the region encoding a protein for the PvDREB6B gene *(Konzen*

et al., 2019).

> MK874722.1 Phaseolus vulgaris genotype UCD CANARIO 707 DREB6B (DREB6B) gene, complete cds

ATGGGAACCGCTATAGATATGTACAACAACACCAACATCGTACCGGATTTCATAGATCCGTATAGTGAGG
AGCTGATGAGAGCACTTGAGCCTTTTATGAAAACTGATTATTTCTCTGCCTCTTCTAATTCTGCATCTCC
CTCACGTCAATCACACCCTTCTTCTCTCATTTCATCTACTTCGTATTCCTCCCCCAACCAAATCAAACTA
AACCAACTCACCTCAGATCAAATCCTCCAGATTCAGGCCCAGGTCGGGATTCAACACCACGTGGGTCATT
CCAACCAACGTCAAAATCAAGCCCAGATGGGGCCAAAACGTGTCCCCATGAAACACGGTGGCGCGGCCGC
GAAAGCTGCGAAGCTGTACCGTGGGGTGCGGCAACGGCATTGGGGGAAGTGGGTCGCCGAGATCAGACTC
CAGAAGAACCGCACGCGCCTCTGGCTGGGAACATTCGACACCGGAGAGGAAGCGGCGCTGGCGTACGACA
ACGCAGCGTTTAAGCTCCGAGGCGAGTTCGCGCGCCTCAACTTCCCTCATCTGCGACACCAGGGAGCGTT
CGTATTCGGCGACTTTGGAGATTACAAGCCGCTACCTTCTTCCGTGGACTCCAAGCTCCAAGCTATTTGT
GAAAGCATGGGAAAACAAGAACAAAAAAAGTGTTGCTCCGTCGAAGACGTGAAGCCCGAGGTACACGCTG
CCGCTGAGCTGGCGCCGGTGGATTCTGACGTGGCGCAATCCAACGTTTGTCCCGAGTTGGACAATATTAA
GGTAGAAAACATGAACGAAACCCCAATGTTGTCATGGCTGTGTCTGGCGAATCTTCTTCGCCTGAATCG
GGTTTTACTTTCTTGGATTTGTCGGATTTCTCATATTCTAATTATGAGTGGGATGAAATAGAGAGTTTTG
GGTTGGAGAAGTACCCTTCGGTGGAGATTGATTGGGCGGCTATATGA

This sequence is deposited in the Genbank database. It's a sequence common bean for a specific genotype called UCD Canario 707.

Now, look at a FASTA sequence of amino acids:

> QCI62111.1 DREB6B [Phaseolus vulgaris]
MGTAIDMYNNTNIVPDFLDPYSEELMRALEPFMKTDYFSASSNSASPSRQSHPSSLISSTSYSSPNQIKL
NQLTSDQILQIQAQVGIQHHVGHSNHRLNQAQLGPKRVPMKHGGAAAKAAKLYRGVRQRHWGKWVAEIRL
QKNRTRLWLGTFDTGEEAALAYDNAAFKLRGEFARLNFPHLRHQGAFVFGDFGDYKPLPSSVDSKLQAIC
ESMGKQEQKKCCSVEDVKPEVDAAAELAPVESDVAQSNVCPELDNIKVENVNENPMLSWPVSGESSSPES
GFTFLDLSDFSYSNYEWDEIESFGLEKYPSVEIDWAAI

To perform alignments and build phylogenetic trees you can modify the sequence identification so that so many encodings do not appear. For example, the previous nucleotide sequence could be represented by:

> UCD Canario 707
ATGGGAACCGCTATAGATATGTACAACAACACCAACATCGTACCGGATTTCATAGATCCGTATAGTGAGG
AGCTGATGAGAGCACTTGAGCCTTTTATGAAAACTGATTATTTCTCTGCCTCTTCTAATTCTGCATCTCC
CTCACGTCAATCACACCCTTCTTCTCTCATTTCATCTACTTCGTATTCCTCCCCCAACCAAATCAAACTA
AACCAACTCACCTCAGATCAAATCCTCCAGATTCAGGCCCAGGTCGGGATTCAACACCACGTGGGTCATT
CCAACCAACGTCAAAATCAAGCCCAGATGGGGCCAAAACGTGTCCCCATGAAACACGGTGGCGCGGCCGC
GAAAGCTGCGAAGCTGTACCGTGGGGTGCGGCAACGGCATTGGGGGAAGTGGGTCGCCGAGATCAGACTC
CAGAAGAACCGCACGCGCCTCTGGCTGGGAACATTCGACACCGGAGAGGAAGCGGCGCTGGCGTACGACA
ACGCAGCGTTTAAGCTCCGAGGCGAGTTCGCGCGCCTCAACTTCCCTCATCTGCGACACCAGGGAGCGTT
CGTATTCGGCGACTTTGGAGATTACAAGCCGCTACCTTCTTCCGTGGACTCCAAGCTCCAAGCTATTTGT
GAAAGCATGGGAAAACAAGAACAAAAAAAAGTGTTGCTCCGTCGAAGACGTGAAGCCCGAGGTACACGCTG
CCGCTGAGCTGGCGCCGGTGGATTCTGACGTGGCGCAATCCAACGTTTGTCCCGAGTTGGACAATATTAA
GGTAGAAAACATGAACGAAACCCCAATGTTGTCATGGCTGTGTCTGGCGAATCTTCTTCGCCTGAATCG
GGTTTTACTTTCTTGGATTTGTCGGATTTCTCATATTCTAATTATGAGTGGGATGAAATAGAGAGTTTTG
GGTTGGAGAAGTACCCTTCGGTGGAGATTGATTGGGCGGCTATATGA

We can align a set of nucleotide sequences for several different genotypes that have had the same gene sequenced in order to compare the sequences and identify nucleotide variants (variants of one base along the sequence are called single nucleotide polymorphisms or SNPs).

To do so, we are going to use an R package called DECIPHER (Wright, 2016).
To install the package, you must use a different path than you normally use for packages in general:

```
> if (!requireNamespace("BiocManager", quietly = TRUE))
        install.packages("BiocManager")
> BiocManager::install("DECIPHER")
```

Let's open a file of nucleotide sequences saved in FASTA format to perform the alignment.

To do this, open the DECIPHER package.

```
library(DECIPHER)
```

Now, specify the location of the file you want to insert (you can click on the file on your computer and specify the properties of its location):

```
fas <-          " C:\Users\Enéas\OneDrive\Documentos\1 - UFRGS
TEACHING\4 - Extension Projects\R Course"
```

However, if you leave only one \, R will not recognize it. You must double the forward slash or invert it to / (in which case you can only use one). In addition

In addition, include the filename in FA (fasta) format at the end:

```
> fas <-            " C:\\Users\\Eneas\\OneDrive\\Documents\\1 -
UFRGS DOCÊNCIA\\4 - Extension Projects\\Curso R\\sequencias.fa"
```

Now, let's read the strings from the file into an object:

```
> dna <- readDNAStringSet(fas)
```

Viewing with:

```
> dna # the unaligned sequences
```

```
A DNAStringSet instance of length 18
     width seq                                                                                names
 [1]   957 ATGGGAACCGCTATAGATATGTACA...GAGATTGATTGGGCGGCTATATGA
Phytozome(PvDREB6B)
 [2]   957 ATGGGAACCGCTATAGATATGTACA...GAGATTGATTGGGCGGCTATATGA BAT 477
 [3]   957 ATGGGAACCGCTATAGATATGTACA...GAGATTGATTGGGCGGCTATATGA BAT 93
 [4]   957 ATGGGAACCGCTATAGATATGTACA...GAGATTGATTGGGCGGCTATATGA CAL 143
 [5]   957 ATGGGAACCGCTATAGATATGTACA...GAGATTGATTGGGCGGCTATATGA G12873
 ...   ... ...
[14]   957 ATGGGAACCGCTATAGATATGTACA...GAGATTGATTGGGCGGCTATATGA SEA-5
[15]   957 ATGGGAACCGCTATAGATATGTACA...GAGATTGATTGGGCGGCTATATGA SXB405
[16]   957 ATGGGAACCGCTATAGATATGTACA...GAGATTGATTGGGCGGCTATATGA UCD-0801
[17]   957 ATGGGAACCGCTATAGATATGTACA...GAGATTGATTGGGCGGCTATATGA UCD-Canario
707
[18]   957 ATGGGAACCGCTATAGATATGTACA...GAGATTGATTGGGCGGCTATATGA ICA-Bunsi
```

There are different algorithms for sequence alignment, but it is not for this document to report details on such procedures. To align the DNA sequences using DECIPHER, proceed as follows:

```
> alignment <- AlignSeqs(dna)
```

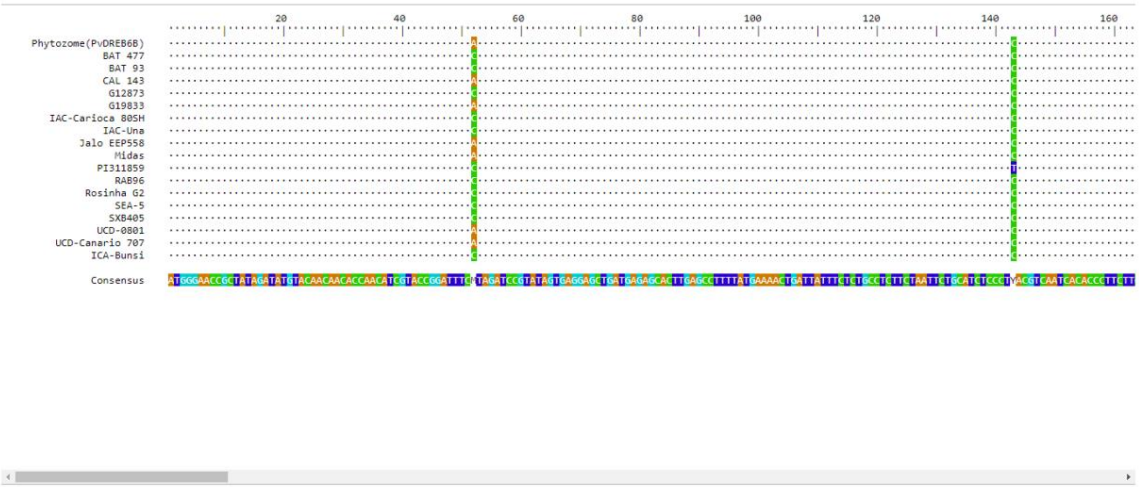
What should appear next is:

```
Determining distance matrix based on shared 9-mers:
    |========================================================== ============================| 100%

Time difference of 0.07 secs

Clustering into groups by similarity:
    |========================================================== ============================| 100%

Time difference of 0.01 secs

Aligning Sequences:
    |========================================================== ============================| 100%

Time difference of 0.23 secs

Iteration 1 of 2:

Determining distance matrix based on alignment: |
    ========================================== ====================================| 100%

Time difference of 0 secs

Reclustering into groups by similarity:
    |========================================================== ============================| 100%

Time difference of 0.01 secs

Realigning Sequences:
    |========================================================== ============================| 100%

Time difference of 0.01 secs

Alignment converged - skipping remaining iteration.
```

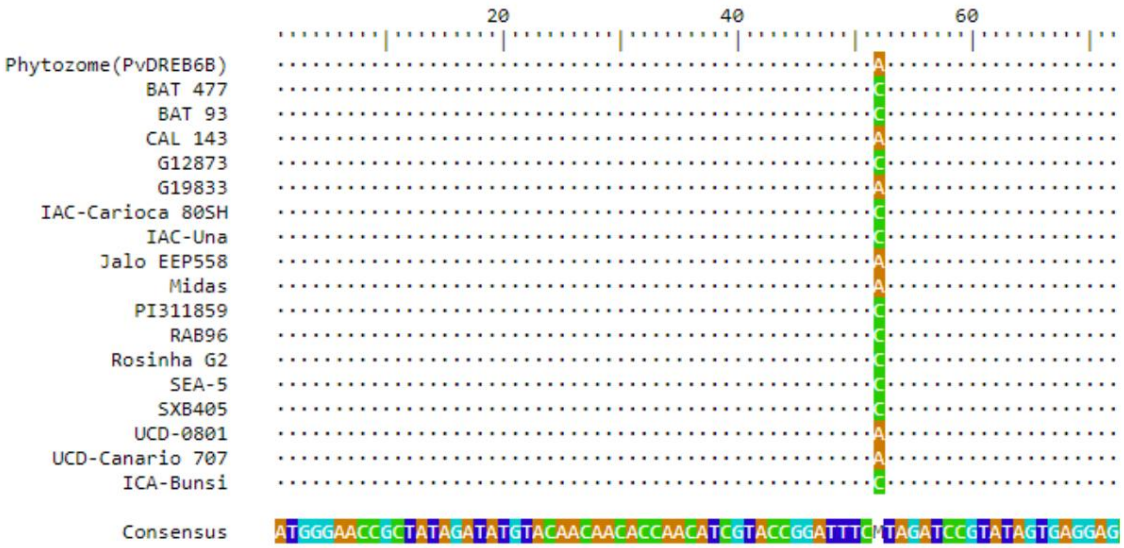
To view the alignment:

> BrowseSeqs(alignment, highligh=0)

The alignment will be shown in your internet browser:



Let's understand what this figure represents from a *zoom:*



In the figure you can visualize the 18 nucleotide sequences identified by the genotypes (from Phytozome to ICA-Bunsi). The top bar indicates the sequential number of bases that make up the sequence. "Consensus" is the consensus sequence, i.e. the sequence representing the most frequently occurring base among all sequences or an alternative base such as M, which represents the presence of A or C at the site considered. Most bases are represented by dots (.), which indicates that all sequences at that position have the same base. This can be easily identified by the consensus sequence. For example, the first base of all sequences is A, the second is T and the third is G. However, at base number 52 there is variation along the sequences, so these bases appear properly identified. That is, there is a one-nucleotide polymorphism (SNP) at this location. Some sequences have base A, while others have base C.

We can now build a phylogenetic tree from the sequences. Because they are DNA sequences from the same species, it is common to build trees using the nucleotide sequence itself to identify relationships between individuals, populations, etc. of this kind. But for analyzes involving different species (sometimes from phyla or other phylogenetically distant groups) it is more common to use amino acid sequences.

To build a phylogenetic tree, some crucial steps need to be followed. Detailing these steps is also far beyond the scope of this document. For those who wish, a simplified review of the main methods of phylogenetic inference can be consulted in the work of Yang & Rannala (2012).
It is worth noting, however, three sequential steps to build a tree

phylogenetics when it comes to a set of nucleotide or amino acid sequences:

1) Alignment of sequences;
2) Choosing a suitable nucleotide or amino acid substitution method: There are theoretical models that examine mutation rates between nitrogenous bases or between amino acids. Based on a set of sequences that you analyze, one of the models can be selected as the most suitable in a program and then used to build the phylogenetic tree;

3) Choice of the phylogenetic reconstruction method: it will depend on the study approach, the sequences that are being worked on (genetic or non-genetic regions, among others) and the phylogenetic proximity between the sequenced specimens (they are individuals of the same or close species or even very different). The most frequent methods of inference can be categorized into distance methods and discrete methods (parsimony, maximum likelihood and Bayesian inference). There are combinations between the methods as well. Often, maximum likelihood or *maximum likelihood* (ML) is used.

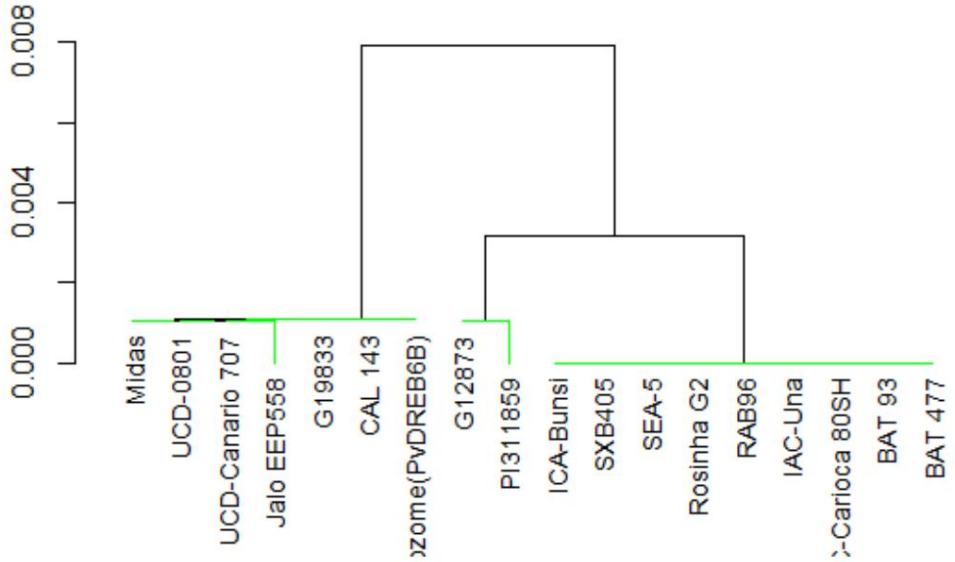
We will use the previous sequences to build a phylogenetic tree using a nucleotide substitution model called Jukes-Cantor (Jukes & Cantor, 1969). Let's start from the alignment generated earlier and build a distance matrix from nucleotide differences along the gene sequence:

```
> d <- DistanceMatrix(alignment, correction="Jukes-Cant
or", verbose=FALSE)
```

Now, let's build the phylogenetic tree based on maximum likelihood (ML) method:

```
> c <- IdClusters(d, method="ML", cutoff=.05, showPlot=TR
UE, myXStringSet=dna, verbose=FALSE)
```

The result is a tree computed from the distance matrix, using algorithms for the maximum likelihood reconstruction method.

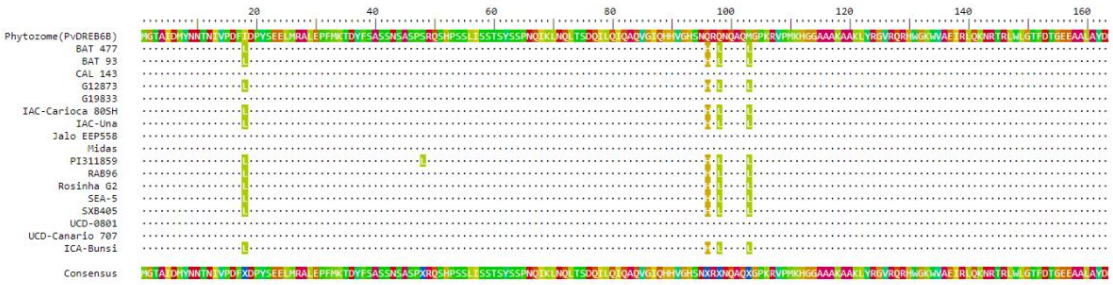


Knowing that the 18 sequences you worked with represent the coding region of a *DREB gene,* you can also translate these sequences, that is, convert nucleotides to amino acids (using the universal genetic code, in the case of our example with plants). Using DECIPHER you can perform the following steps:

> AA <- AlignTranslation(dna, type="AAStringSet") # align the translationDNA

> BrowseSeqs(AA, highlight=1) # view the alignment

In this alignment the first sequence appears, as it was highlighted on the command line (highlight = 1).



If you want to save the alignment, specify the location as shown:

> writeXStringSet(AA, file="C:\\Users\\Eneas\\OneDrive\\D
documents\\1 - UFRGS TEACHING\\3 - Research projects\\
Bioinformatics\\1001al.fas")

Several other analyzes can be performed with sequences such as those presented. Alignment and construction of phylogenetic trees, however, are essential to understand other steps. The beginning user in R will come across several new packages to perform these and other procedures. An extensive literature review is in order on the methods and packages that are most appropriate for the type of analysis that is needed.

Introduction to time series

Time series are usually derived from long-term information collection, such as long-term ecological programs (LELDs). Here, we will work with artificially created data, with the additional aim of enriching readers' R vocabulary.

In our example, we are going to create a dummy dataset that simulates the reading time (in minutes) per day of an undergraduate student over the course of the first year of the course.

```
> days <- 1:365 # numbers representing 365 days of the year
> reading <- days/365+2*runif(365,0,(1:365)/10)# creates 365 values representing minutes of reading
```

The series of commands below will work with loops, that is, functions that repeat an analysis along a specified series. Even though the commands seem more complicated, they are intuitive:

```
> result <-NULL # create an object to allocate values only after
```

```
> n <- length(days) # a way to not be limited to one value, but to the total number of values in an array
```

Here, below, comes the trickiest part of the script. Here, we will use the for() function, which has this basic structure:

```
> for(i in set of values){
    # commands that # will
    be repeated
                                                    }
```

For example:

```
> for(i in 1:5){
    print(letters[i])
}
 [1] "the"
 [1] "b"
 [1] "c"
```

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

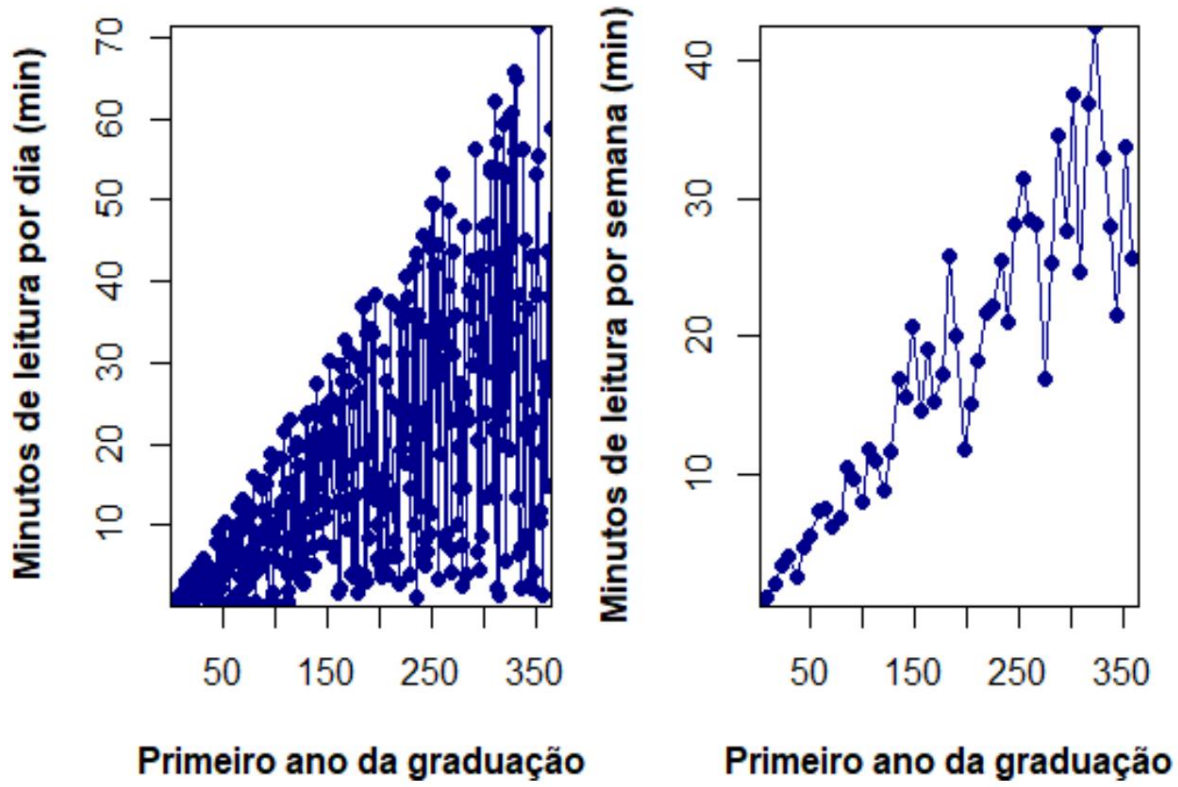
In our example, we will try to divide this reading time into 7 days.
(weekly series). For this, we can adapt the argument logic to:

```
> for (i in seq(1, n ,7)) {
# we set the loop to be every 7 days. The { defines the beginning of the arguments of a function,
as we will see next


weekly average <- cbind(i,mean(read[i:(i+6)]))
# here we are telling R to extract an average value every 7 days; note the sum detail (i + 6), which
adds up to 7 days of observations


result <- rbind(result, averageweekly)
}

> result # Look at the result
```

Since we have reading time values every 7 days, we can create a graph that represents this. Using graphs to analyze time series is always welcome. Let's create two charts in a single dashboard. The first graph will show the daily variation of the reading time, while the second will use the result of the weekly average. The commands could be the following:

```
> op <- par(mfrow = c(1, 2),
mar=c(4,4,3,1),xaxs="i",yaxs="i") #panel

#Graph 1 - By day
> plot(days, reading,pch=16, cex=0.95, col="dark blue",type="o",xlab="First year of
graduation", ylab="Reading minutes per day (min)", main=NULL, font.lab=2)


#Graph 2 - By weekly average
> plot(result[,1], result[,2],pch=16, cex=0.95, col="dark blue", type="o", xlab="First year of
graduation", ylab="Minutes of read per week (min)", main=NULL, font.lab=2)


> par(op)
```

The result for these commands will be the following graphic:



Creating your own function

One of the great advantages of R is the freedom of creation. Like a writer, or an artist, a data scientist can, depending on their prior knowledge, compose new roles that meet the specific demands of their research. This topic, a bit advanced within the scope of this document, is a first exposure to the power of R for beginning users.

The basic command to create a function is as follows:

> myfunction <- function(x) { }

Note that nothing was specified. Inside the braces, you can specify which type of calculation (or logical attributes) you want the function to perform with “x”, which can be a vector, dataframe, matrix, etc.

For example, we can make a function that calculates the value of “x” squared:

```
> X2 <- function(a) { a^2                                    }
> X2(10) [1]
 100
```

Another slightly less simple example is a function that calculates the final average of a set of grades for a subject and also assigns the final grade to each student. The final grades used here, for practical purposes, are those normally applied by UFRGS departments, with A (excellent) being assigned to final grades above 9.0, grade B (good), final grade between 7.5 and 8.9, grade C (regular), between 6.0 and 7.4, and final grades lower than 6, grade D (failed).

```
> final concept <- function(x) {y=mean(x)
 {ifelse(y >= 9.0, "A", ifelse(y >= 7.5 & y <= 8.9, "B",
                ifelse(y >= 6.0 & y <= 7.4, "C", "D")))
     }}

   # ifelse works with the following arguments (ifelse, condition, Yes, No). Note that the "yes"
   and "no" can again be the ifelse argument.
```

Let's use a dataframe with the partial scores of three evaluations out of 10 students of an undergraduate course (all data are fictitious):

```
> notes <- read.csv("notes201X.csv") #remember that you may need to add the
   argument sep=";".
```

Let's, for practicality, add a new column to the data with the final grades (averages of the partial grades), to check the concepts measured in the next command:

```
> grades$NF <- apply(grades[,2:4], 1, mean)
```

Now, let's use the function we created to add the concepts:

```
> grades$Final_Grade <- apply(grades[,2:4], 1, final grade)
```

```
> head(notes) # check if the NF corresponds to the assigned concept.
```

Furthermore, it would be useful to export this data to an excel spreadsheet and then send it to the students. There are very simple ways to do this, like the command below:

```
> write.csv(notes, "Final Notes__2019.csv")
# The file, "Notas__Finais__2019.csv", will be exported, as in the case of the figures, to the
folder that is the working directory of R. You can check it using the getwd() command.
```

Now let's demonstrate a more complete function that uses those previous reading time series data per day during the first year of graduation. The purpose of the function is simple; instead of using three command lines, let's put everything inside a single function. Note that the logic is the same as the commands used in time series analysis:

```
> timeseries <- function(x, y, timeseries, range) {

                    timedata <- NULL
                    vardata <- NULL
                    for (i in seq(1, timeseries-interval, interval)) {

                    timedata <- c(timedata, x[i])
                    vardata <- c(vardata, mean(y[i:(i + range
1)]))
                    result <- cbind(timedata, vardata) }
          return(result)                    }
```

Let's test the function by asking it to calculate an average reading every 30 days:

```
> test30 <- time series(days, reading, 365, 30)
> test30
# see the result
```

Now, we can plot the result, which is very similar to what we did previously with the time series.

```
> plot(test30, pch=16, cex=0.95, col="dark blue", type="o",xlab="First year of
graduation", ylab="Reading minutes per month (min)", main= "", font.lab=2)
```

Does it match what you expected?

Final considerations

As reported at the beginning of this document, the primary objective of this introduction is to present both the most common functions and arguments that can be very useful to a beginner in R. Far from exhausting any knowledge on the subject, many of the commands used here represent the style of the authors and, many times, there are other ways of performing the same operation. We ourselves share many differences in carrying out the same operations and here we adopt consensus on some points. This is a clear indication of the range of possibilities that working with R is, as well as fun.

OR lacks constant use and facing new challenges on this platform increases our experience, as well as learning a new language in addition to the mother tongue. It is possible to create an infinity of results in R. This just shows how much we can still learn. There are several types of users; some learn only basic operations that they use most in everyday life, as well as some who will do everything to make each line of script as elegant and practical as possible.

As well as this humble document, there are a number of other handouts that have a wealth of knowledge about this introductory part. Please don't be restricted to just this document. The help() and official R forums are also an inexhaustible source of knowledge. Use sparingly.

Last but not least, we would like to reinforce that this document will always be constantly being modified and updated. Always check the document version. Like any work, we are not error free and these can (and should!) be reported directly to the authors. Without further ado, we hope this is a first step for you to enter the challenging and fascinating world of R.

References

Adler, J. 2010. *R in a nutshell - a desktop quick reference.* O'Reilly. 609 p.

Attwood, TK; Gisel, A.; Eriksson, NE & Bongcam-Rudloff, E. 2011. Concepts, historical milestones and the central place of bioinformatics in modern biology: a European perspective. *Bioinformatics-Trends and Methodologies,* 1:1-31.

Batista, JLF, Prado, PI & Oliveira, AA (eds.) 2009. *Introduction to R - An Online Handout.* URL: http://ecologia.ib.usp.br/bie5782.

Borcard, D.; Gillet, F. & Legendre, P. 2018. *Numerical Ecology with* R. 2nd ed. new York, Springer-Verlag, 306 p.

Cox, TF & Cox, MAA 2001. *Multidimensional scaling.* 2nd ed. Chapman & Hall, 308 p.

Crawley, MJ 2005. *Statistics - an introduction using* R. John Wiley & Sons, 327 p.

Crawley, MJ 2012. *The R Book.* 2nd ed. Wiley, 1076 p.

Erthal, F. 2017. *Multivariate Statistics in Taphonomy (Actualistic). In:* RS Horodyski & F. Erthal (eds.) Taphonomy: Methods, Processes and Applications, CRV, p. 81-113.

Ferreira, EB; Cavalcanti, PP & Nogueira, DA 2014. ExpDes: an R package for ANOVA and experimental designs. *Applied Mathematics,* 5:2952-2958.

Gotelli, NJ & Ellison, AM 2013. *A primer of Ecological statistics.* 2nd ed. Sunderland, Sinauer, 614 p.

Jari Oksanen, F. Guillaume Blanchet, Michael Friendly, Roeland Kindt, Pierre Legendre, Dan McGlinn, Peter R. Minchin, RB O'Hara, Gavin L. Simpson, Peter Solymos, M. Henry H. Stevens, Eduard Szoecs and Helene Wagner. 2018. *vegan: Community Ecology Package.* https://CRAN.R-project.org/package=vegan

Konzen, ER; Recchia, GH; Cassieri, F.; Caldas, DGG; Berny Mier and Teran, JC; Gepts, P. & Tsai, SM 2019. DREB Genes from common bean *(Phaseolus vulgaris* L.) chow broad to specific abiotic stress responses and distinct levels of nucleotide diversity. *International Journal of Genomics,* 2019:9520642.

Landeiro, VL 2011. *Introduction to the use of the* R program. Available at: https://cran.r-project.org/doc/contrib/Landeiro-Introducao.pdf [Apostille].

Legendre, P. & Legendre, L. 2012. *Numerical Ecology.* 3rd ed. New York, Elsevier, 1006 p.

MacLaren, JA; Anderson, PSL; Barrett, PM & Rayfield, EJ 2017. Herbivorous dinosaur jaw disparity and its relationship to extrinsic evolutionary drivers. *Paleobiology,* 43:15-33.

Murrell, P. 2006. *R graphics.* Chapman & Hall./CRC. 303 p.

Owen, WJ 2019. *The R Guide.* Available at: https://cran.r project.org/doc/contrib/Owen-TheRGuide.pdf.

R Development Core Team. 2019. *An Introduction to* R. Available at: https:// cran.r-project.org/doc/manuals/r-release/R-intro.pdf

R Development Core Team. 2019. *R language definition.* Available at: https://cran.r-project.org/doc/manuals/r-release/R-lang.pdf.

R Development Core Team. 2019: *A: A language and environment for statistical computing.* R Foundation for Statistical Computing, Vienna, https://www.r project.org/.

Ripley, B.; Venables, B.; Bates, DM; Hornik, K.; Gebhardt, A.; Firth, D. & Ripley, MB 2013. *Package 'mass'.* Cran R.

Silva, LJ 2015. *Package 'laercio'.* Cran R.

Stork, L.; Garcia, DC; Lopes, SJ & Estefanel, V. 2000. *Vegetable experimentation.* Santa Maria: UFSM, 198 p.

Steven, HM 2009. *A Primer of Ecology with R* (Use R!) 1st ed. Springer, 388 p.

Venables, WN & Ripley, BD 2002. Random and mixed effects. In: *Modern applied statistics with* S, Springer p. 271-300.

Verzani, J. 2005. *Using R for introductory statistics.* Chapman & Hall/CRC. 402 p.

Verzani, J. 2019. *SimpleR – Using R for Introductory Statistics.* Available at: https://cran.r-project.org/doc/contrib/Verzani-SimpleR.pdf.

Wright, ES 2016. Using DECIPHER v2.0 to analyze big biological sequence data in R. *R Journal,* 8:352-359.

Yang, Z. & Rannala, B. 2012. Molecular phylogenetics: principles and practice. *Nature Reviews Genetics,* 13:303-314.