Computer Exercise 1

Introduction to Bioinformatics (MVE510)
Autumn, 2024

Introduction

Welcome to the first computer exercise in the course Introduction to Bioinformatics. This exercise aims to introduce you to R, a programming language focused on statistical analysis of data. We will familiarize ourselves with the R computing environment and learn how to use R to process, analyze, and visualize data. The computer exercise can be solved using either standard R or RStudio. In contrast to standard R, RStudio offers a more complete programming environment and is therefore recommended as a starting point. R and RStudio are installed on the computer systems. If you want to install R/RStudio on your own computer, you can download them for free from https://www.r-project.org/ and https://www.rstudio.com/ respectively.

There is a wide range of documentation available for R. The developers of R provide a comprehensive introductory text about the basics which often can serve as a good reference manual: https://cran.r-project.org/manuals.html. R also offers built-in help functions that provide detailed information about all functions and commands. When encountering problems with a specific R function, this is typically the first place to look for help. Furthermore, the 'Base R cheat sheet' offers an overview of the most fundamental R commands: https://iqss.github.io/dss-workshops/R/Rintro/base-r-cheat-sheet.pdf. In addition, the notes from Lecture 1 contain some basic recommendations for getting started with R, including some of the pitfalls encountered when moving from MATLAB to R. Thus, if you are getting stuck, there are several sources where you can search for help. You can, of course, also ask the course assistants for help. The course assistants will be present at all the scheduled computer exercises.

The exercise should be done in groups of a maximum of two people. This first introductory exercise does not require a written report, but you need to be examined by one of the assistants when you are done. You should then be prepared to show any code that you have written and answer questions about the exercises. If you choose to do this computer exercise outside of the scheduled hours, you need to provide all the code you have written to be examined.

Finally, be aware that knowledge of R will be necessary for the upcoming exercises so you should take this opportunity to get familiar with the programming language.

Part 1: Some R fundamentals

Once R has been started, you will be greeted with a console. This is the place where you will issue any commands to R. To quit R, you run the function **q**, i.e.

```
> q()
```

Note that you *always* need to supply parentheses when executing functions in R. Simply typing only 'q' will not work and will instead show how the function **q** is defined.

R has an extensive help function which you can access either by typing,

> help(q)

or, by adding a question mark before the function, i.e.

> ?q

If you have problems with a function, it is always good to read the help page. The help pages are often (but sadly not always) highly detailed with information on how the function should be used.

Furthermore, R has a large number of packages that can be loaded to increase functionality. A package is loaded into R by the **library** command. For example,

> library(MASS)

loads the MASS package into R (a package expanding the available statistical functionality). Many of the install packages contain examples and tutorials called 'vignettes'. To get a list of the vignettes available in your R installation, type

> browseVignettes()

There is a large repository of packages called 'The Comprehensive R Network' or simply CRAN. These packages can easily be installed into R, either through the commands in the 'Packages' menu or by using the **install.packages()** command. In this computer exercise, we will, however, not need to install any additional R-packages but it can be important for future references.

Part 2: R as a calculator

R can, similarly to MATLAB, be used as an advanced calculator. Any mathematical expression you write into the R console will be executed immediately and the results printed out. Try to enter

> 4+7
> 4-7
> 4*7
> 4/7
> 4^7

R uses standard operator precedence meaning that e.g. multiplications are executed before additions. For example,

> 4+7*3
> (4+7)*3

will generate different results. For a full list of the different operators and their priorities, please see **help(Syntax)**. Note that you can also use the help command to get information about an operator, but here quotation marks are necessary, i.e. **help('+')**.

R has many in-built standard mathematical functions, such as **sqrt, exp, log,** and **sin.** Open and read the help page for the function **exp.** Apply **sqrt**, **exp,** and **log** in any calculation of your choice. Calculate also

> log(0)
> log(-1)

What happened in the second case?

Part 3: Vectors and matrices

R can, similarly to MATLAB, work efficiently with vectors and matrices. To create a vector, use the command **c** (abbreviation for 'concatenate')

> x=c(1,2,3,2,3)

This creates a vector containing five elements stored in the variable **x**.

Important remark: In R, you can use both '=' and '<-' to assign a value to a variable. For example,

> x<-c(1,2,3,2,3)

will also result in a vector containing five elements stored in the variable **x**.

Vector operations are done element-wise in R. Try to calculate

> x-1
> x*2
> x^2
>exp(x)

Vectors can be indexed using brackets ('[]'). To get an element from a vector, type

> x[2]

R has several built-in functions to easily create vectors with numbers. One way is to use ':' which takes two integers and creates a sequence between them. For example, try

> 1:10
> 5:15

This can be used for efficient indexing. If a range of integers is provided when indexing a vector, R will return all the values. For example, try

> x[2:4]
> x[c(1,3)]

You can, of course, also store a sequence in a variable and use it to index a vector. Try, for example,

> y=2:4
> x[y]

Elements of a vector can also be extracted using a logical vector. Try

> y=x>2
> x[y]

This can be written more compactly as

> x[x>2]

Sequences can also be created by the **seq** and **rep** functions. Use **seq** to create a vector between 1 and 100 with 10,000 equally spaced values. After that, use **rep** to create a vector with 100 elements, where each element is equal to 42.

Matrices can be created with the **matrix** command, which takes three parameters: a vector with the values in the matrix, the number of rows, and the number of columns. For example,

> z=matrix(1:12, 4,3)

creates a 4×3 matrix with the values 1 to 12. Note that the values are filled in column-wise, where 1 to 4 end up in the first column, 5 to 8 in the second column, and so on. This can be changed by setting 'byrow=TRUE', i.e.

> z=matrix(1:12, 4,3, byrow=TRUE)

Matrices are indexed similarly to vectors but using two dimensions. For example,

> z[3:4,2:3]

creates a 2×2 matrix based on rows 3 to 4 and columns 2 to 3 of **z**.

The function **apply** is very useful when making computations with matrices. **apply** takes a matrix and 'applies' a function (of your choice) to all its rows or columns. For example,

> apply(z, 1, sum)

calculates the sum of each row in **z**. Changing the second argument from 1 to 2 makes **apply** work with the columns instead. Use **apply** and **mean** to calculate the mean value over all the columns of **z.**

Part 4: Writing scripts

The most efficient way to work with R is to write scripts. A script is a set of R commands that are consecutively executed when loaded into R. An R script can be created in the editors built-in into R and RStudio. It is also possible to use any other text editor, such as Notepad++.

Once a script has been created it can be loaded and executed in R by using the **source** function, i.e.

> source("myscript.R")

Note that you first need to point R to the directory where you stored **myscript.R** (by using 'Change dir' under 'File' in R or 'Change working directory' under 'Session' in RStudio). When executing a script, the result for every single command will not be printed out. If you want to print out a specific variable you need instead to use the **print** function.

Write the following lines in an R script, save it into a file, and execute it using the **source** function.

```
# This is my first script
x=1.5
y=exp(x)
print(y)
```

The first line of this script starts with '#' indicating that it is a comment and therefore disregarded by R. This is useful to introduce comments anywhere into your R scripts.

From now on, it is highly recommended that you work with scripts in R. This will save you a lot of time when the code you write becomes more complex

Part 5: Working with data frames

Data frames are a versatile object used to organize data. Data frames are organized in a matrix-like structure but can, in contrast to standard matrices, contain data of multiple types, such as both numbers and strings. This is very convenient for many forms of data which may consist of more than only numbers. In this part of the exercise, we will use one of the many example datasets provided by R (use **data()** to list all available example datasets). We will first use the **mammals** dataset. To load the **mammals** dataset, type

```
> library(MASS)
> data(mammals)
```

The first line is required since the **mammals** dataset is a part of the **MASS** R-package. This creates a variable, called **mammals** containing the data. Use **ls()** to check that the data was loaded. Use **help** to read the information about the **mammals** dataset. The **mammals** variable is organized in a **data.frame** data type. This can be seen by using the class function, i.e.

> class(mammals)

The **mammals** data object consists of 62 rows and 2 columns, which can be seen by using the **dim** command. Furthermore, type

> head(mammals)

to view the 6 first rows of the **mammals** data frame. Note that this data object also has row names describing which animal the values correspond to. In R, both rows and columns in matrices and data frame can be given names. These can be extracted and set with the **rownames** and **colnames** functions, i.e.

> rownames(mammals)
> colnames(mammals)

To calculate the mean weight of a mammal, we can use the **mean** function. To calculate the mean value for the body weight in column 1, type

> mean(mammals[,1])

Similarly, we can calculate the mean value for the brain weight in column 2 by

> mean(mammals[,2])

Now, use **median, var** and **sqrt** to calculate the median, variance and standard deviation of the body and brain weight. Note that the body weight is provided in kg and the brain weight in g (as specified by the help page).

Next, we aim to identify the mammal that has the largest brain compared to its body weight. Calculate the brain-to-body weight ratio by

> ratio=(mammals[,2]/1000)/mammals[,1]

This will, since R performs element-wise operations, calculate the ratio between the brain weight and body weight of each mammal. This ratio can easily be added to the mammal data frame by using the **cbind** function (stands for 'column bind'), i.e.

> mammals2=cbind(mammals, ratio)

The new column will get the name of the variable but we can change that with

> colnames(mammals2)=c("Body", "Brain", "Brain/Body-ratio")

Finally, we would like to sort the data frame so the mammals with the highest brain-to-body ratio are on top. We will do this using the **order** function, which takes a vector and provides the order on how it should be reorganized so it is sorted in an increasing or decreasing order. Type

> mammals.order=order(mammals2[,3], decreasing=TRUE)

The 'decreasing=TRUE' tells the function that we want the highest value at the top. Read the help for **order** and take a look at the **mammal.order** variable that we created. Do you understand what **order** is doing? The sorted data frame can then be created by

> mammals2.sorted=mammals2[mammals.order,]

Which mammal has the highest brain-to-body weight ratio? Which one has the lowest?

Part 6: Using statistical functions and tests

R is a statistical programming language and has thus a large collection of statistical tools which we will now start to explore. To do this, we will continue our investigation of the weight of animals, but this time focus on cats. Load the **cats** dataset and use the help function to read about its contents. Use the **head** and **summary** functions to describe the dataset. Make sure that you understand the output of **summary**.

Use the **mean**, **median**, and **var** to calculate the mean value, the median, the variance, and the standard deviation of the body weight of a cat. After that, use **cor** to see if the body weight correlates with the heart weight.

The **cats** dataset consists of observations from 47 female and 97 male cats, which is indicated in the first column of the data frame. To create a data frame only containing the female cats, we will first write

> female=cats[,1]== "F"

This uses the '==' operator to compare the first column of cats to the character 'F'. Rows that are equal to 'F' will result in TRUE, while all other rows will result in FALSE. Look at the content of **female** and make sure that you understand how it was constructed. We can then use **female** to take out the rows corresponding to female cats and save them in a new data frame by

> cats.female=cats[female,]

Examine **cats.female** and make sure it only contains data from female cats. Then, repeat the procedure for the male cats and store the result in **cats.male**.

Calculate the mean, median, variance, and standard deviation for the body weights of the male and female cats. Are they similar?

To analyze if there is a significant difference in the body weight of a female and male cat we will use a statistical test in the form of a t-test. As you may recall from your statistics course, the t-test for testing two groups (here female and male cats) comes in two different variants: one assuming equal variance and one assuming different variances. We will therefore first perform a test to assess if the variability of the body weight is different between the female and male cats. One test for this purpose is the F-test, which compares the variances from the groups to see if they differ significantly. In R, an F-test for assessing variances from two groups of observations can be done using the **var.test** function. To use the **var.test** on the female and male cat body weight, write,

> var.test(cats.female[,2], cats.male[,2])

Interpret the results. Is the difference in variance significant?

The t-test in R is performed using the **t.test** function. Test the difference in body weight between female and male cats. Use a test with equal or unequal variances based on your findings from the variance test. Look at the help of the **t.test** function to see how this is specified. What type

of test is used by default? Is the difference in body weight significant? How large is the difference?

An alternative to the t-test is the non-parametric Wilcoxon-Mann-Whitney test (WMW test), which is implemented in the function **wilcox.test.** In contrast to the t-test, the WMW test does not rely on a normal distribution and can therefore be more robust in many situations. Use the WMW test to assess if there is a difference in body weight between female and male cats.

Part 7: Plotting and visualization of data

R has a powerful set of functions to plot and visualize data. In this part of the exercise, we will explore these functions by continuing to explore the **cats** dataset, so make sure that it is loaded (using the **data** function). To create a plot of the cats' body weight against their heart weight, type

> plot(cats[,2], cats[,3])

This plot is called a scatter plot and visualizes each cat's body weight (x-axis) and heart weight (y-axis). The appearance of almost any plot in R can be changed with a set of common parameters. For example, to make the circles read, type

> plot(cats[,2], cats[,3], col="red")

Use the **colors** function to print out all colors available in R by default. Furthermore, the plotted character can be created by adding the **pch** parameter (stands for 'plot character'), i.e.

> plot(cats[,2], cats[,3], pch=2)

will plot triangles. Try other numbers to see what other characters are available. You can change the labels of the axis and the title of the plot,

> plot(cats[,2], cats[,3], pch=2, main="Cat body-heart weight", xlab="Body weight", ylab="Heart weight")

It is also possible to alter the size of the plotted character, axis, and labels using the **cex**, **cex.main**, **cex.axis**, and **cex.lab** arguments. **cex** stands for character expansion and decides how much the size of different parts of the plot should increase or decrease. A value of >1 means an increase in the size while a value of <1 decreases the size. For example,

> plot(cats[,2], cats[,3], pch=2, main="Cat body-heart weight", xlab="Body weight", ylab="Heart weight", cex=1.5, cex.main=1.5)

increase the size of the plotted triangles and the heading of the plot by 50%.

Furthermore, the parameters **xlim** and **ylim** can be used to set the range of the x-axis and the y-axis. For example,

plot(cats[,2], cats[,3], pch=2, main="Cat body-heart weight", xlab="Body weight", ylab="Heart weight", cex=1.5, cex.main=1.5, xlim=c(2, 2.5), ylim=c(6, 15))

restricts the plots between 2 and 2.5 on the x-axis and 6 and 15 on the y-axis.

Use the **hist** function to create a histogram of the cats' body weights. Use the documentation of **hist** to find out how to increase the number of bars in the histogram.

Use the **barplot** function to create a bar plot of the weight of the heart of the 10 first cats.

When plotting, R uses 'devices' which decide the form of the created plot. If no device has been opened, R will automatically create a window in R. It is also possible to specify a device and thereby direct the results to an image or PDF file. This can be done using the functions **bitmap** and **pdf**. For example,

```
> pdf(file="cats.pdf", width=8, height=8)
> plot(cats[,2], cats[,3], pch=2, main="Cat body-heart weight", xlab="Body weight", ylab="Heart weight", cex=1.5, cex.main=1.5)
> dev.off()
```

creates a PDF file with the scatter plot of the cat weights. This file will be saved in the directory R/RStudio has been pointed to. The final line, **dev.off**, closes the last opened device. This is necessary to make 'cats.pdf' a valid PDF file. If you want to close all opened devices you can use the function **graphics.off.** Using devices is very convenient if you want to create figures in scripts, but do not want to save them manually.

The function **layout** can be used to draw multiple subplots within a single plot. After a device has been opened, the **layout** takes a matrix with numbers from 1 to the number of subplots. The subplots will then be plotted, starting from 1 to the maximum number. For example,

```
> windows()
> layout(matrix(1:2, nrow=1, ncol=2))
> hist(cats.female[,2], breaks=20, xlim=c(2,4), ylim=c(0,12))
> hist(cats.male[,2], breaks=20, xlim=c(2,4), ylim=c(0,12))
```

creates a plot with two subplots. They are located on a 1×2 grid and will be plotted from the left to the right.

Part 8: Writing functions

In R, you can easily write your own functions. This is especially powerful when you have more complex code containing multiple R commands that you want to apply multiple times. A function is created using the **function** command, e.g.

```
myfunc=function(x,y){
z=x+y
return(z)
}
```

This creates a function called 'myfunc'. The function takes two arguments, 'x' and 'y'. The variable 'z' is a local variable and will only be present within the function. The **return** function

specifies what 'myfunc' should return. Note that you need to use curly brackets ('{' and '}') to specify the start and stop of the function.

Write a function that given a number *n* returns the sum of the first *n* integers.

Write a function that, given two vectors *x* and *y* with *n* elements each, calculates the Euclidean distance defined as,

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}.$$

Write a function that, given a number *n*, returns the first *n* first Fibonacci numbers. The nth Fibonacci number a_n is defined according to the formula

$$a_n = a_{n-1} + a_{n-2}$$

with $a_0 = 0$ and $a_1 = 1$. To solve this exercise you need to use for-loops, which are structures used for iteration over a variable. R uses the following syntax for loops,

```
# For loop in R
for (i in 1:100){
  print(i)
}
```

This will loop 100 times where the variable *i* starts at 1 and ends at 100. Note that the parentheses around 'i in 1:100' and the curly brackets ({}) around the code are required.

Done!

You are now done with your first introduction to R and have hopefully gained some insight into how R works. We will return to R in the upcoming computer exercises where we will use it to analyze and interpret complex data to solve biological problems.