DJILLALI LIABES UNIVERSITY OF SIDI BEL ABBES
FACULTY OF EXACT SCIENCES
DEPARTMENT OF COMPUTER SCIENCES

Module : Aide à la décision
1ST YEAR OF MASTER'S DEGEREE IN
NETWORKS,INFORMATION SYSTEMS & SECURITY(RSSI)
2021/2022

Introduction to R
TP-01

Students:
HADJAZI M.Hisham
AMOUR Wassim Malik
Group: 01/RSSI

Instructors:
Pr. YOUSFATE Abderrahmane
Dr. BENBEKRITI Soumia

A paper submitted in fulfilment of the requirements for the
Aide à la décision TP-01

March 1, 2022

Contents

Chapter 1

Introduction to R

1.1 Introduction

R is a very powerful, popular environment for statistical computing and graphics, freely available for Mac OS X, Windows, and UNIX systems. Knowing how to use R is an extremely useful skill.[1]

Just to confuse you, R refers to two things. There is R, the programming language, and R, the piece of software that you use to run programs written in R. Fortunately, most of the time it should be clear from the context which R is being referred to. R (the language) was created in the early 1990s by Ross Ihaka and Robert Gentleman, then both working at the University of Auckland. It is based upon the S language that was developed at Bell Laboratories in the 1970s, primarily by John Chambers. R (the software) is a GNU project, reflecting its status as important free and open source software. Both the language and the software are now developed by a group of (currently) 20 people known as the R Core Team.[2]

The fact that R's history dates back to the 1970s is important, because it has evolved over the decades, rather than having been designed from scratch (contrast this with, for example, Microsoft's .NET Framework, which has a much more "created"1 feel). As with life-forms, the process of evolution has led to some quirks and inconsistencies. The upside of the more free-form nature of R (and the free license in particular) is that if you don't like how something in R is done, you can write a package to make it do things the way that you want. Many people have already done that, and the common question now is not "Can I do this in R?" but "Which of the three implementations should I use?

R is an interpreted language (sometimes called a scripting language), which means that your code doesn't need to be compiled before you run it. It is a high-level language in that you don't have access to the inner workings of the computer you are running your code on; everything is pitched toward helping you analyze data.[2]

R supports a mixture of programming paradigms. At its core, it is an imperative language (you write a script that does one calculation after another), but it also supports object-oriented programming (data and functions are combined inside classes) and functional programming (functions are first-class objects; you treat them like any other variable, and you can call them recursively). This mix of programming styles means that R code can bear a lot of similarity to several other languages. The curly braces mean that you can write imperative code that looks like C (but

the vectorized nature of R means that you have fewer loops). If you use reference classes, then you can write object-oriented code that looks a bit like C# or Java. The functional programming constructs are Lisp-inspired (the variable-scoping rules are taken from the Lisp dialect, Scheme), but there are fewer brackets.[2] All this is a roundabout way of saying that R follows the Perl ethos:
"There is more than one way to do it."

Larry Wall

1.1.1 Installing R

If you are using a Linux machine, then it is likely that your package manager will have R available, though possibly not the latest version. For everyone else, to install R you must first go to http://www.r-project.org. Don't be deceived by the slightly archaic website;2 it doesn't reflect on the quality of R. Click the link that says "download R" in the "Getting Started" pane at the bottom of the page.[2]

Once you've chosen a mirror close to you, choose a link in the "Download and Install R" pane at the top of the page that's appropriate to your operating system. After that there are one or two OS-specific clicks that you need to make to get to the download. If you are a Windows user who doesn't like clicking, there is a cheeky shortcut to the setup file at http://<CRAN MIRROR>/bin/windows/base/release.htm.[2]

1.1.2 Choosing an IDE

If you use R under Windows or Mac OS X, then a graphical user interface (GUI) is available to you. This consists of a command-line interpreter, facilities for displaying plots and help pages, and a basic text editor. It is perfectly possible to use R in this way, but for serious coding you'll at least want to use a more powerful text editor. There are countless text editors for programmers; if you already have a favorite, then take a look to see if you can get syntax highlighting of R code for it.[2]

If you aren't already wedded to a particular editor, then I suggest that you'll get the best experience of R by using an integrated development environment (IDE). Using an IDE rather than a separate text editor gives you the benefit of only using one piece of software rather than two. You get all the facilities of the stock R GUI, but with a better editor, and in some cases things like integrated version control.[2]

The following sections introduce five popular choices, but this is by no means an exhaustive list (a few additional suggestions follow). It is worth trying several IDEs; a development environment is a piece of software that you could be spending thousands of hours using, so it's worth taking the time to find one that you like. A few additional suggestions follow this selection.[2]

1. RStudio (MAC, Linux, Windows)

2. RKWard (Linux)

3. Tinn-R (Windows)

4. Emacs + ESS (MAC, Linux, Windows)

5. Eclipse/Architect (MAC, Linux, Windows)

1.2 Classes

All variables in R have a class, which tells you what kinds of variables they are. For example, most numbers have class numeric (see the next section for the other types), and logical values have class logical. Actually, being picky about it, vectors of numbers are numeric and vectors of logical values are logical, since R has no scalar types. The "smallest" data type in R is a vector.[2]

You can find out what the class of a variable is using class(my variable):

```
class(c(TRUE, FALSE))
```

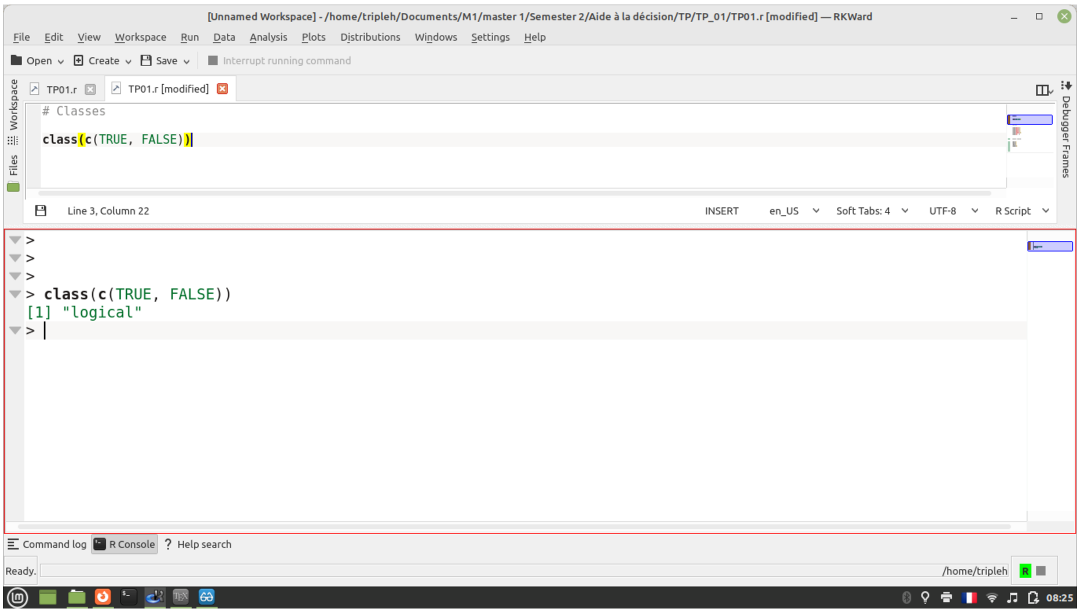

FIGURE 1.1: Classes 1

It's worth being aware that as well as a class, all variables also have an internal storage type (accessed via typeof), a mode (see mode), and a storage mode (storage.mode). All the variables that we created in the previous chapter were numbers, but R contains three different classes of numeric variable: numeric for floating point values; integer for, ahem, integers; and complex for complex numbers. We can tell which is which by examining the class of the variable:

```
class(sqrt(1:10))

class(3 + 1i)
 #"i" creates imaginary components of complex numbers

```

```r
class(1)
 #although this is a whole number, it has class numeric

class(1L)
 #add a suffix of "L" to make the number an integer

class(0.5:4.5)
 #the colon operator returns a value that is numeric...

class(1:5)
 #unless all its values are whole numbers
```

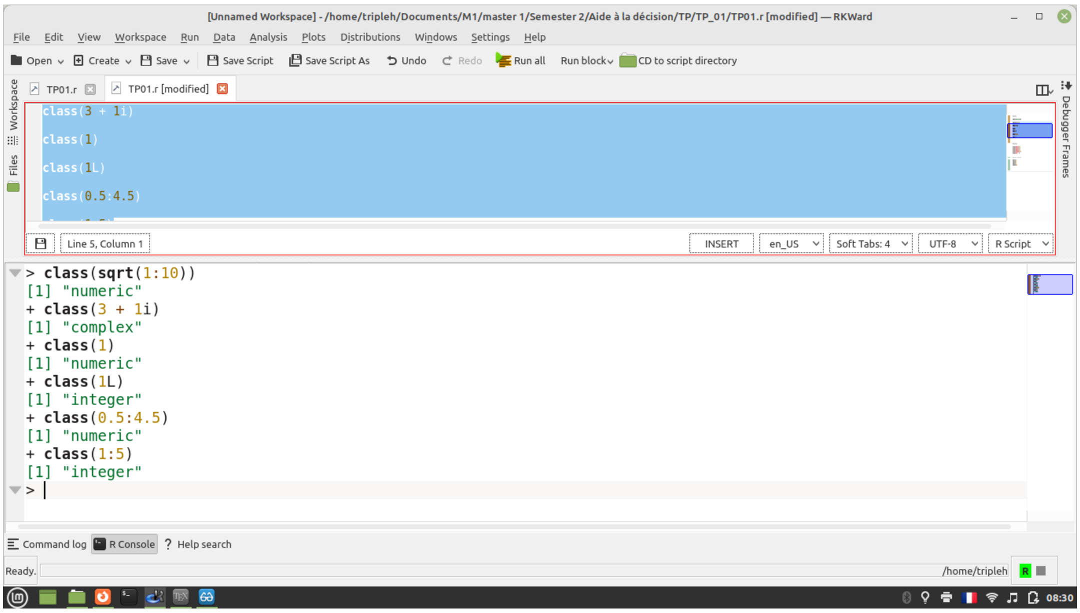

FIGURE 1.2: Classes 2

Note that as of the time of writing, all floating point numbers are 32-bit numbers ("double precision"), even when installed on a 64-bit operating system, and 16-bit ("single precision") numbers don't exist.

In addition to the three numeric classes and the logical class that we've seen already, there are three more classes of vectors: character for storing text, factors for storing categorical data, and the rarer raw for storing binary data. In this next example, we create a character vector using the c operator, just like we did for numeric vectors. The class of a character vector is character:

```r
class(c("she", "sells", "seashells", "on", "the", "sea", "shore"))
```

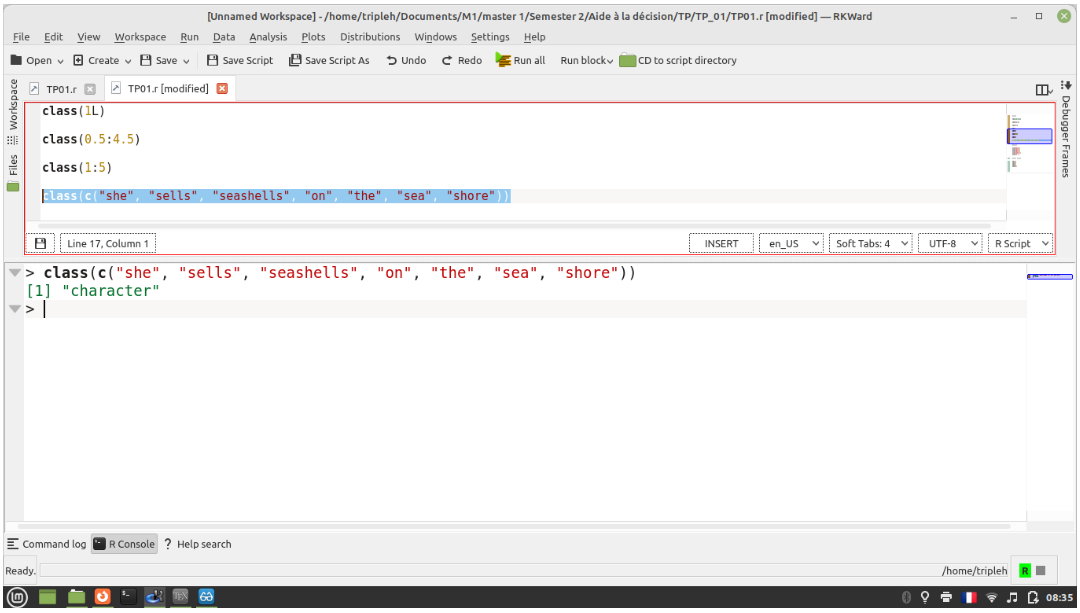

FIGURE 1.3: Classes 3

1.3 Vectors

The vector function creates a vector of a specified type and length. Each of the values in the result is zero, FALSE, or an empty string, or whatever the equivalent of "nothing" is:

```r
vector("numeric", 5)
vector("complex", 5)
vector("logical", 5)
vector("character", 5)
vector("list", 5)
```

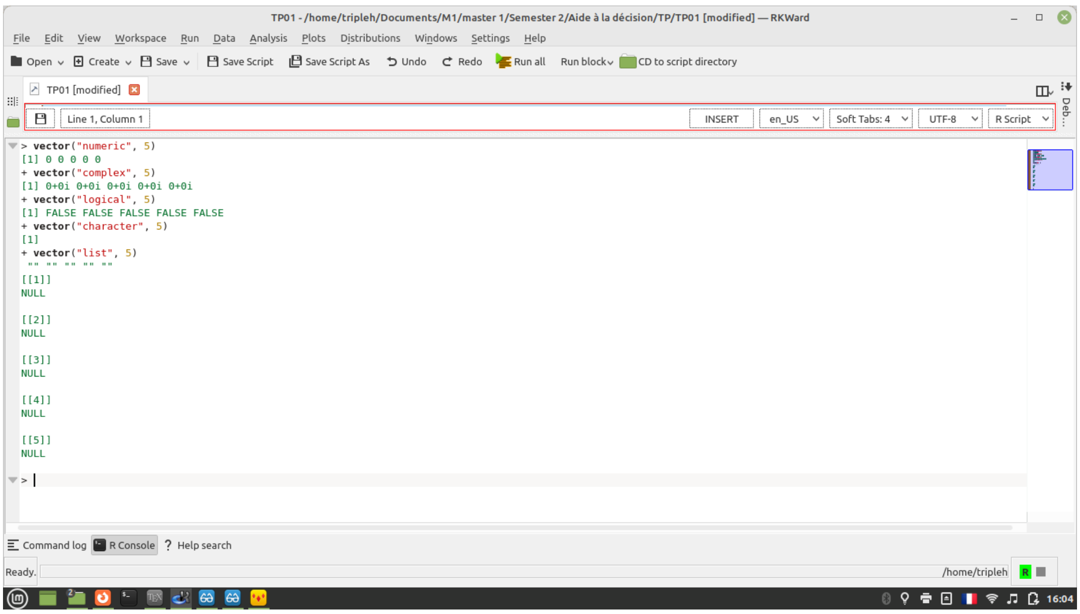

FIGURE 1.4: Vectors

In that last example, NULL is a special "empty" value (not to be confused with NA, which indicates a missing data point). For convenience, wrapper functions exist for each type to save you typing when creating vectors in this way. The following commands are equivalent to the previous ones:

```
numeric(5)
complex(5)
logical(5)
character(5)
```

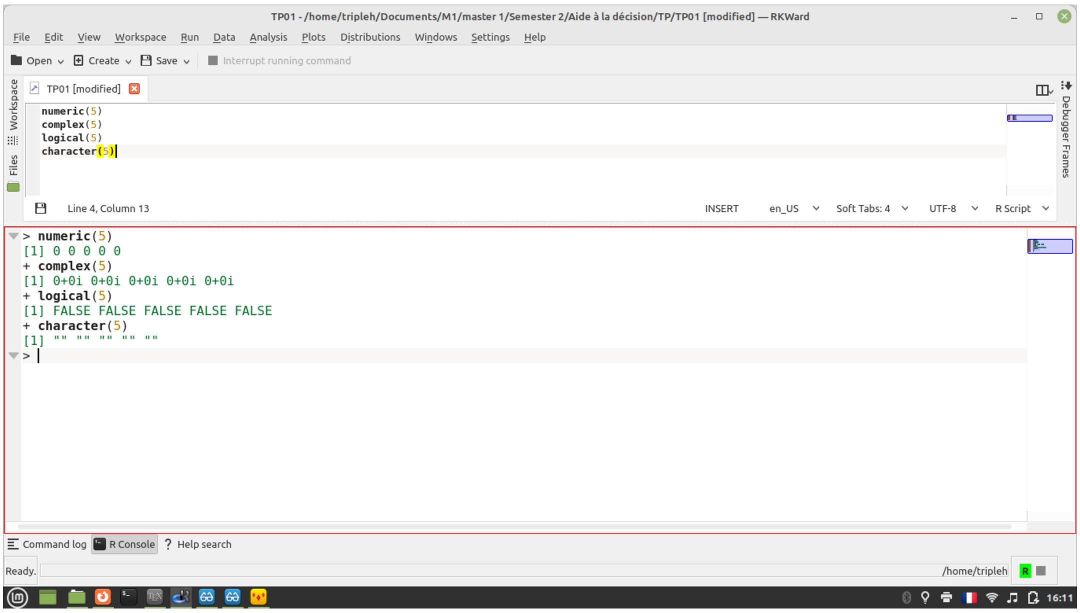

FIGURE 1.5: Vectors 2

Command	What it does
`c(1,1,0,2.7,3.1)`	creates the vector $(1, 1, 0, 2.7, 3.1)$
`1:100`	creates the vector $(1, 2, \ldots, 100)$
`(1:100)^3`	creates the vector $(1^3, 2^3, \ldots, 100^3)$
`rep(0,50)`	creates the vector $(0, 0, \ldots, 0)$ of length 50
`seq(0,99,3)`	creates the vector $(0, 3, 6, 9, \ldots, 99)$
`v[5]`	5th entry of vector v (index starts at 1)
`v[-5]`	all but the 5th entry of v
`v[c(3,1,4)]`	3rd, 1st, 4th entries of vector v
`v[v>2]`	*entries* of v that exceed 2
`which(v>2)`	*indices* of v such that entry exceeds 2
`which(v==7)`	*indices* of v such that entry equals 7
`min(v)`	minimum of v
`max(v)`	maximum of v
`which.max(v)`	indices where $\max(v)$ is achieved
`sum(v)`	sum of the entries in v
`cumsum(v)`	cumulative sums of the entries in v
`prod(v)`	product of the entries in v
`rank(v)`	ranks of the entries in v
`length(v)`	length of vector v
`sort(v)`	sorts vector v (in increasing order)
`unique(v)`	lists each element of v once, without duplicates
`tabulate(v)`	tallies how many times each element of v occurs
`table(v)`	same as `tabulate(v)`, except in table format
`c(v,w)`	concatenates vectors v and w
`union(v,w)`	union of v and w as sets
`intersect(v,w)`	intersection of v and w as sets
`v+w`	adds v and w entrywise (recycling if needed)
`v*w`	multiplies v and w entrywise (recycling if needed)

FIGURE 1.6: Vectors 3

1.4 Arrays

To create an array, you call the array function, passing in a vector of values and a vector of dimensions. Optionally, you can also provide names for each dimension:

```
(three_d_array <- array(
1:24,
dim = c(4, 3, 2),
dimnames = list(
c("one", "two", "three", "four"),
```

```
c("ein", "zwei", "drei"),
c("un", "deux")
)
))
```

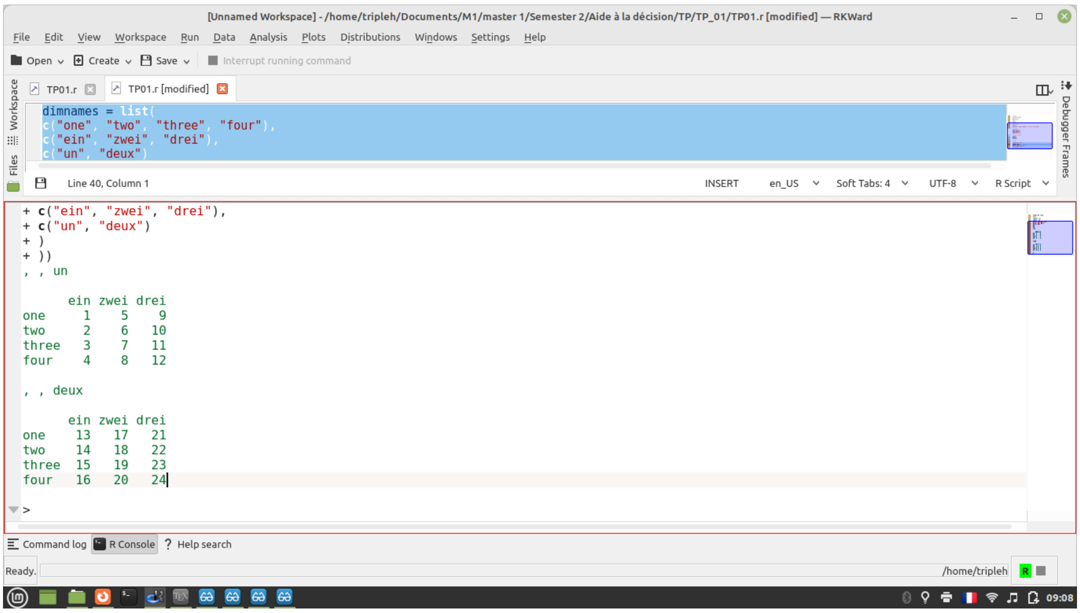

FIGURE 1.7: Arrays

1.5 Matrices

The syntax for creating matrices is similar, but rather than passing a dim argument, you specify the number of rows or the number of columns:

```
(a_matrix <-
 matrix(
1:12,
nrow = 4,
 #ncol = 3 works the same
dimnames =
 list(
c("one",
 "two", "three", "four"),
c("ein",
 "zwei", "drei")
)
))
```

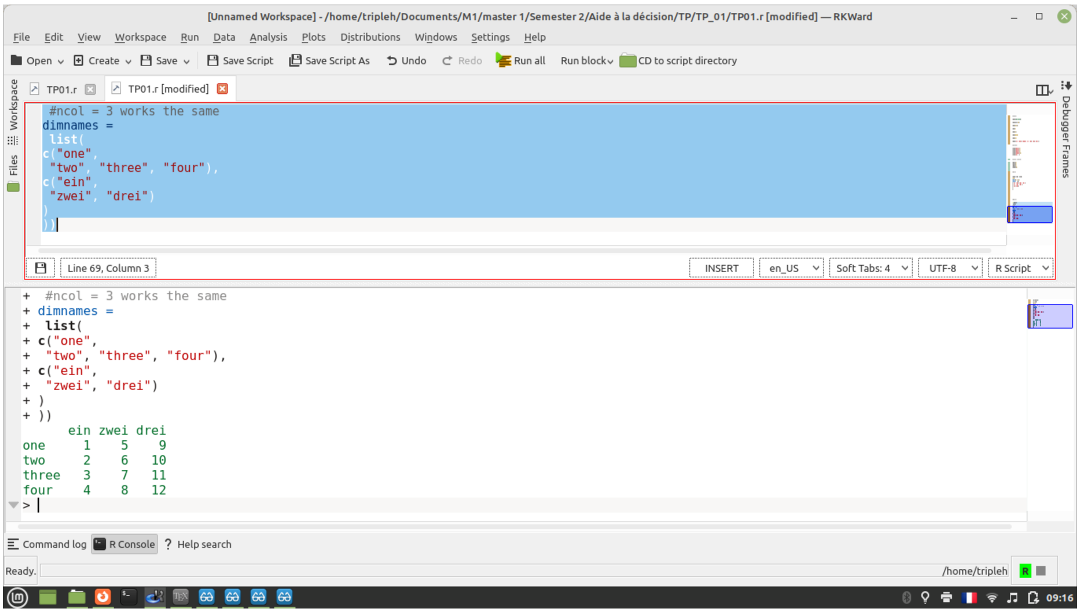

FIGURE 1.8: Matrices 1

This matrix could also be created using the array function. The following two dimensional array is identical to the matrix that we just created (it even has class matrix):

```
(two_d_array
 <- array(
1:12,
dim = c(4,
 3),
dimnames =
 list(
c("one",
 "two", "three", "four"),
c("ein",
 "zwei", "drei")
)
))
```

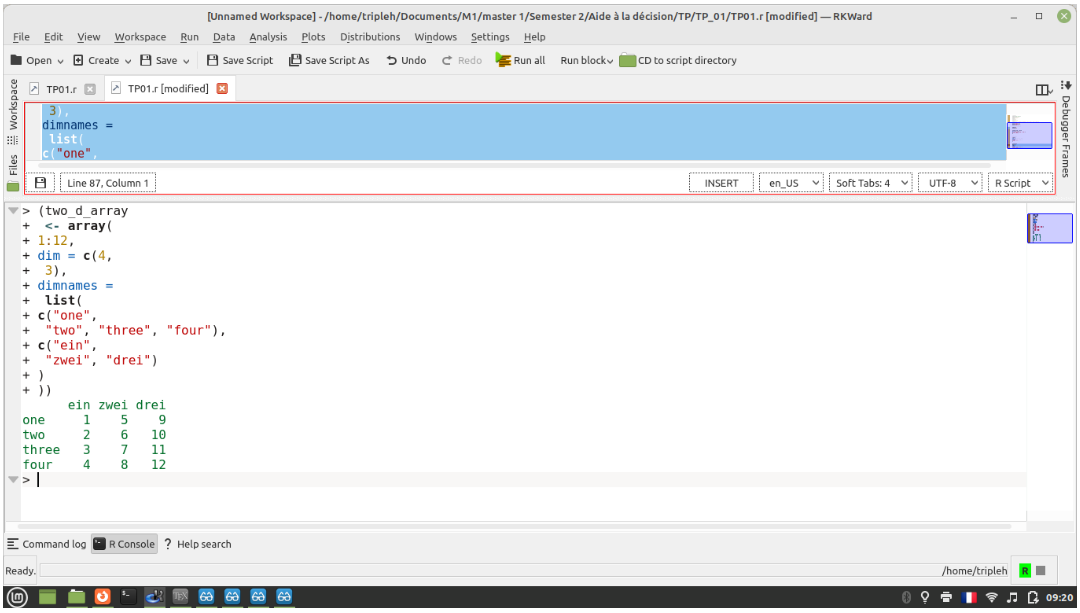

FIGURE 1.9: Matrices 2

When you create a matrix, the values that you passed in fill the matrix column-wise. It is also possible to fill the matrix row-wise by specifying the argument byrow = TRUE:

```
matrix(
1:12,
nrow = 4,
byrow = TRUE,
dimnames = list(
c("one", "two", "three", "four"),
c("ein", "zwei", "drei")
)
)
```

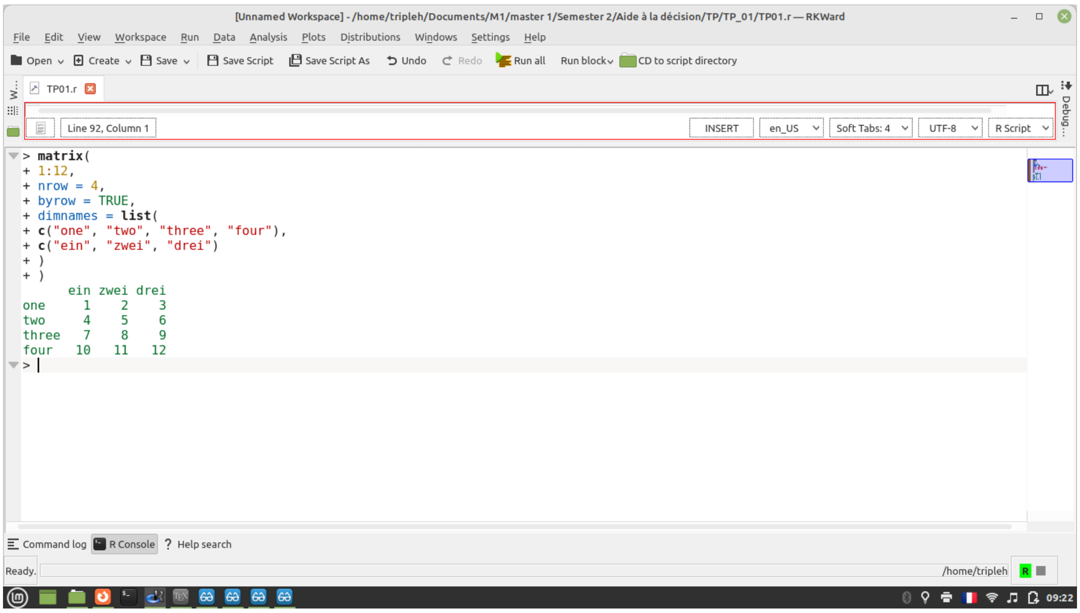

FIGURE 1.10: Matrices 3

Command	What it does
`matrix(c(1,3,5,7), nrow=2, ncol=2)`	creates the matrix $\begin{pmatrix} 1 & 5 \\ 3 & 7 \end{pmatrix}$
`dim(A)`	gives the dimensions of matrix A
`diag(A)`	extracts the diagonal of matrix A
`diag(c(1,7))`	creates the diagonal matrix $\begin{pmatrix} 1 & 0 \\ 0 & 7 \end{pmatrix}$
`rbind(u,v,w)`	binds vectors u, v, w into a matrix, as rows
`cbind(u,v,w)`	binds vectors u, v, w into a matrix, as columns
`t(A)`	transpose of matrix A
`A[2,3]`	row 2, column 3 entry of matrix A
`A[2,]`	row 2 of matrix A (as a vector)
`A[,3]`	column 3 of matrix A (as a vector)
`A[c(1,3),c(2,4)]`	submatrix of A, keeping rows $1, 3$ and columns $2, 4$
`rowSums(A)`	row sums of matrix A
`rowMeans(A)`	row averages of matrix A
`colSums(A)`	column averages of matrix A
`colMeans(A)`	column means of matrix A
`eigen(A)`	eigenvalues and eigenvectors of matrix A
`solve(A)`	A^{-1}
`solve(A,b)`	solves $A\mathbf{x} = \mathbf{b}$ for $\mathbf{x}$ (where $\mathbf{b}$ is a column vector)
`A %*% B`	matrix multiplication AB
`A %^% k`	matrix power A^k (using `expm` package)

FIGURE 1.11: Matrices 4

1.6 Lists

A list is, loosely speaking, a vector where each element can be of a different type. Lists are created with the list function, and specifying the contents works much like the c function that we've seen already. You simply list the contents, with each argument separated by a comma. List elements can be any variable type—vectors, matrices, even functions:

```
(a_list <- list(
c(1, 1, 2, 5, 14, 42),
 #See http://oeis.org/A000108
month.abb,
matrix(c(3, -8, 1, -3), nrow = 2),
asin
))
```

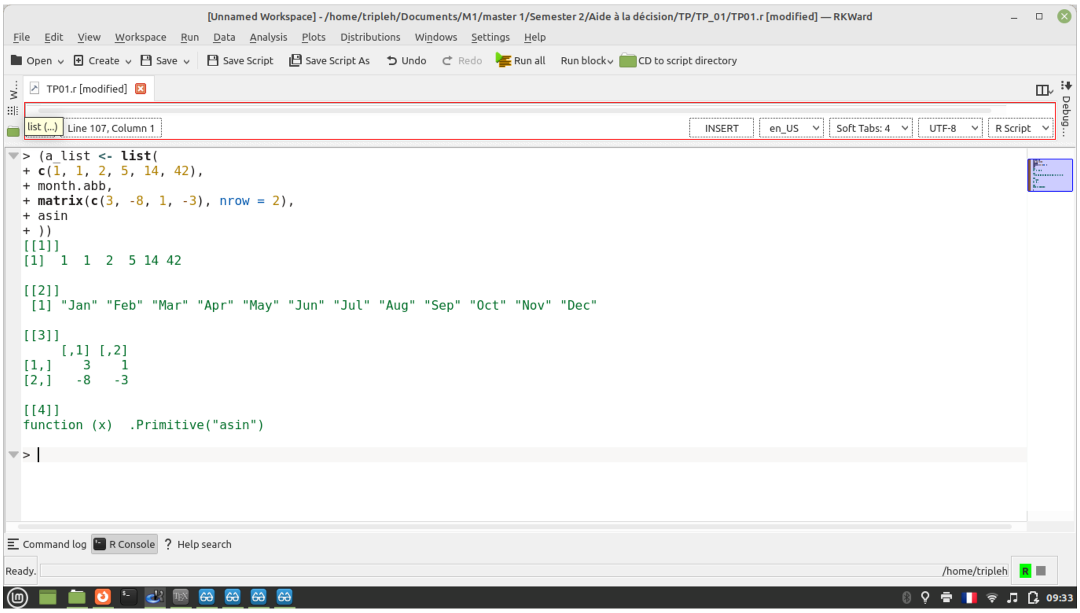

FIGURE 1.12: Lists 1

In theory, you can keep nesting lists forever. In practice, current versions of R will throw an error once you start nesting your lists tens of thousands of levels deep (the exact number is machine specific). Luckily, this shouldn't be a problem for you, since real world code where nesting is deeper than three or four levels is extremely rare.

1.7 Tables

Tables are often essential for organzing and summarizing your data, especially with categorical variables. When creating a table in R, it considers your table as a specifc type of object (called "table") which is very similar to a data frame. Though this may seem strange since datasets are stored as data frames, this means working with tables will be very easy since we have covered data frames in detail over the previous tutorials. In this chapter, we will discuss how to create various types of tables, and how to use various statistical methods to analyze tabular data. Throughout the chapter, the AOSI dataset will be used.[**donovan_6_nodate**]

A contingency table is a tabulation of counts and/or percentages for one or more variables. In R, these tables can be created using table() along with some of its variations. To use table(), simply add in the variables you want to tabulate separated by a comma. Note that table() does not have a data= argument like many other functions do (e.g., ggplot2 functions), so you much reference the variable using dataset variable. Some examples are shown below. By default, missing values are excluded from the counts; if you want a count for these missing values you must specify the argument useNA="ifany" or useNA="always".[**donovan_6_nodate**]

1.7.1 Create a table from scratch

```
tab <- matrix(c(7, 5, 14, 19, 3, 2, 17, 6, 12), ncol=3, byrow=TRUE)
colnames(tab) <- c('colName1','colName2','colName3')
rownames(tab) <- c('rowName1','rowName2','rowName3')
tab <- as.table(tab)
tab
```

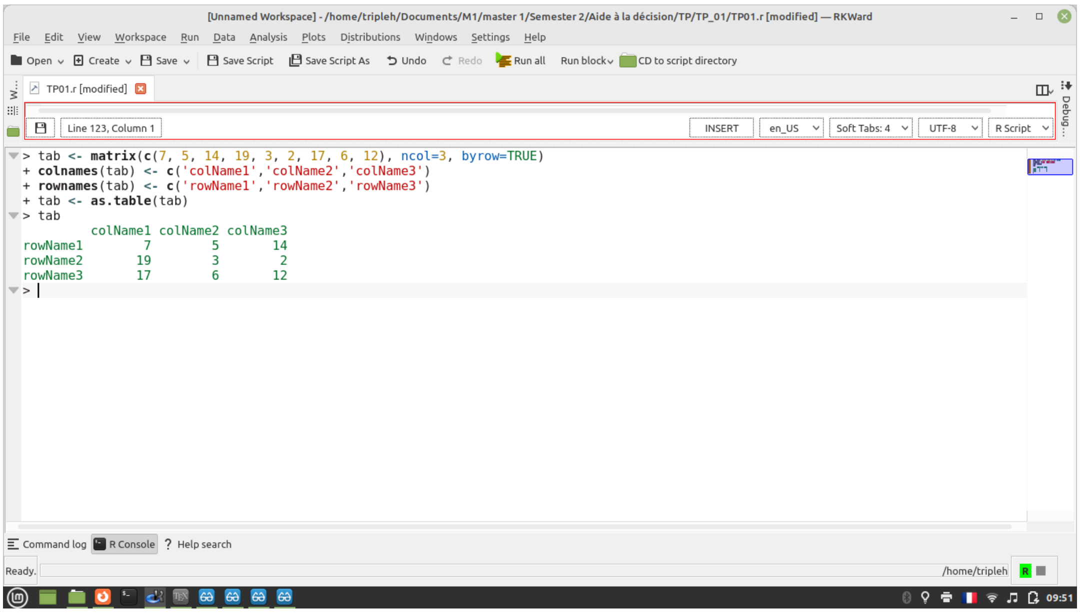

FIGURE 1.13: Tables 1

1.7.2 Create a Table from Existing Data

```
#make this example reproducible
set.seed(1)

#define data
df <- data.frame(team=rep(c('A', 'B', 'C', 'D'), each=4),
                 pos=rep(c('G', 'F'), times=8),
                 points=round(runif(16, 4, 20),0))

#view head of data
head(df)


#create table with 'position' as rows and 'team' as columns
```

```
tab1 <- table(df$pos, df$team)
tab1
```

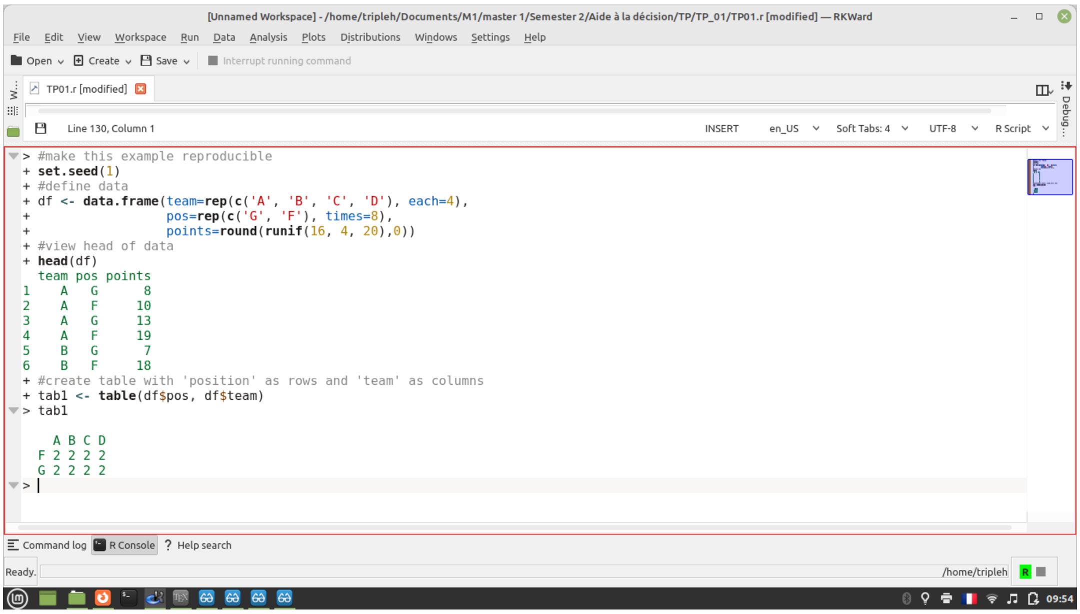

FIGURE 1.14: Tables 2

1.8 Data Frames

Data frames are used to store spreadsheet-like data. They can either be thought of as matrices where each column can store a different type of data, or non nested lists where each element is of the same length. We create data frames with the data.frame function:

```
(a_data_frame <- data.frame(
x = letters[1:5],
y = rnorm(5),
z = runif(5) > 0.5
))
```

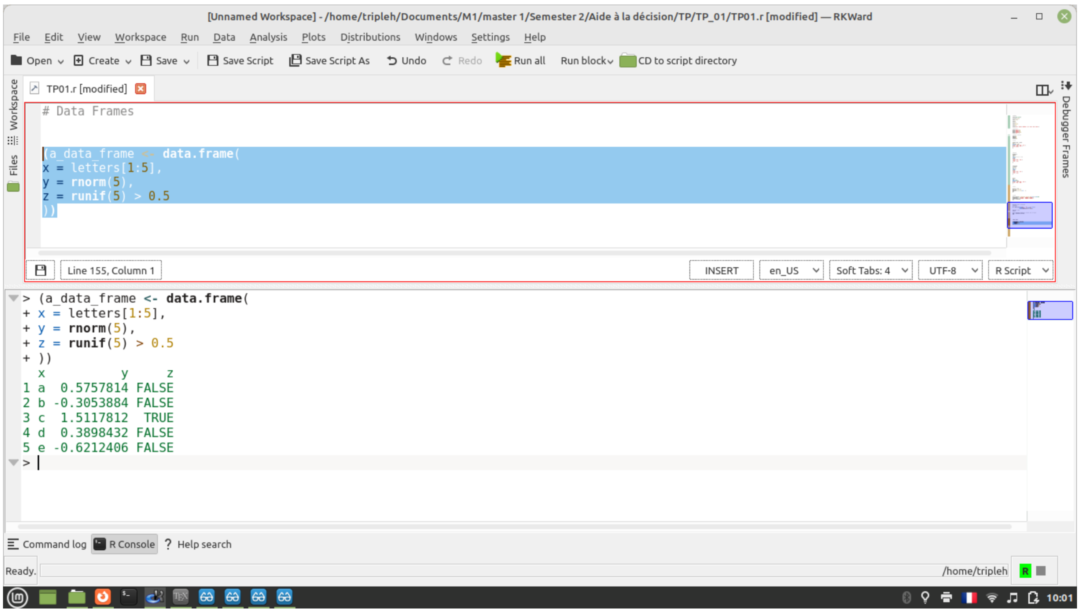

FIGURE 1.15: Data Frames 1

Notice that each column can have a different type than the other columns, but that all the elements within a column are the same type. Also notice that the class of the object is data.frame, with a dot rather than a space.

In this example, the rows have been automatically numbered from one to five. If any of the input vectors had names, then the row names would have been taken from the first such vector. For example, if y had names, then those would be given to the data frame:

```r
y <- rnorm(5)
names(y) <- month.name[1:5]
data.frame(
x = letters[1:5],
y = y,
z = runif(5) > 0.5
)
```

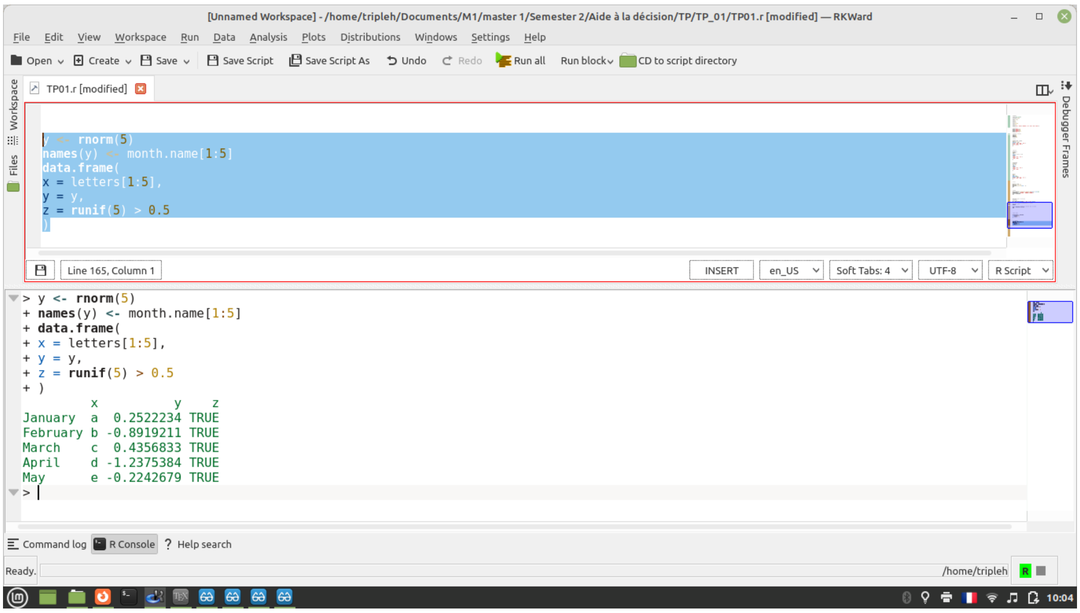

FIGURE 1.16: Data Frames 2

This behavior can be overridden by passing the argument row.names = NULL to the data.frame function:

```
data.frame(
x = letters[1:5],
y = y,
z = runif(5) > 0.5,
row.names = NULL
)
```

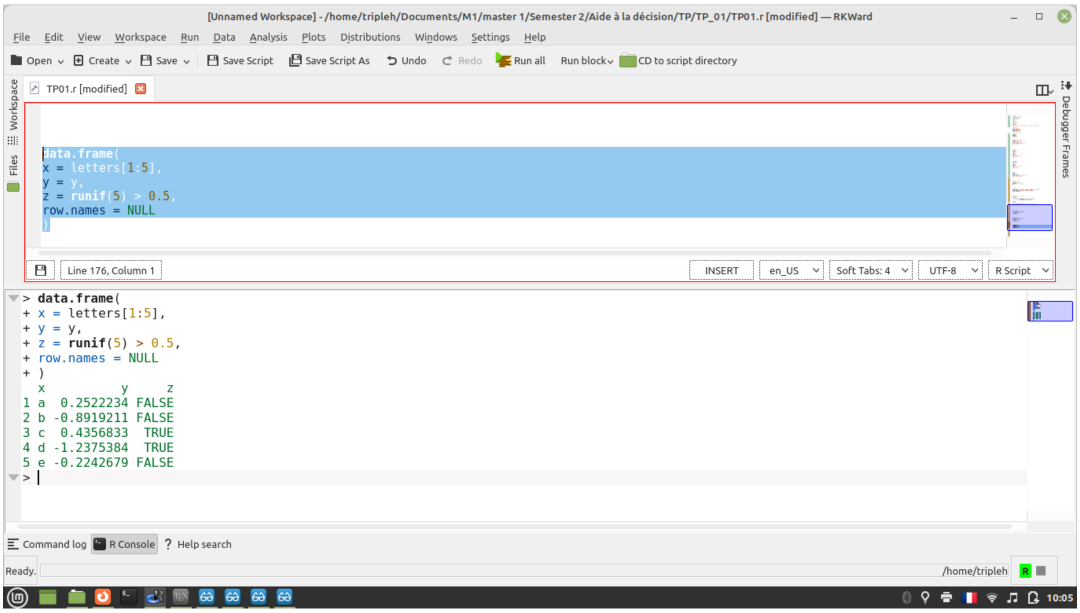

FIGURE 1.17: Data Frames 3

It is also possible to provide your own row names by passing a vector to row.names. This vector will be converted to character, if it isn't already that type:

```
data.frame(
x = letters[1:5],
y = y,
z = runif(5) > 0.5,
row.names = c("Jackie", "Tito", "Jermaine", "Marlon", "Michael")
)
```

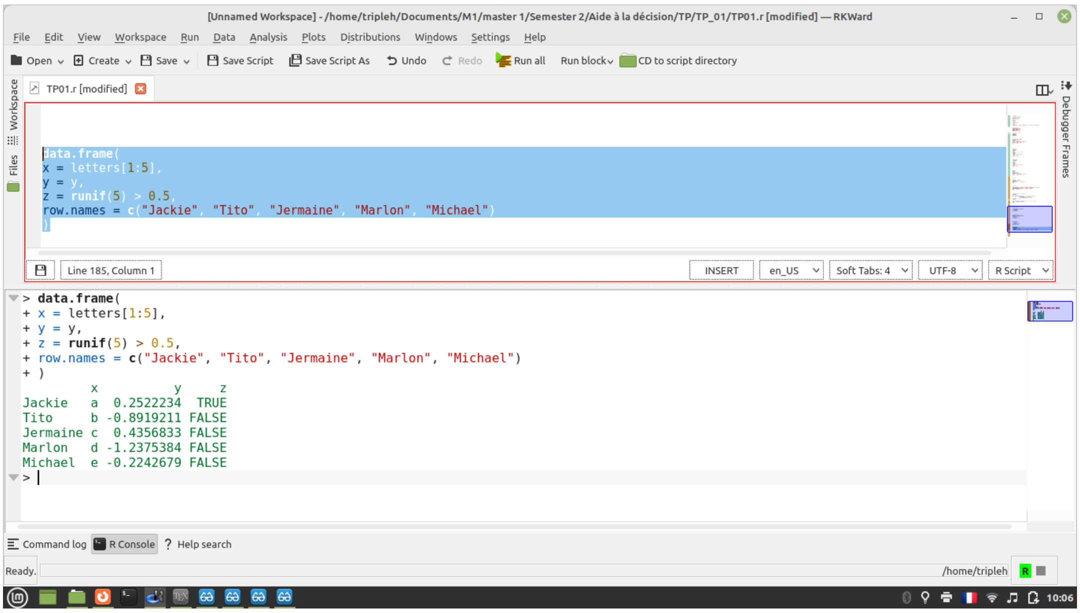

FIGURE 1.18: Data Frames 4

1.8.1 Lists vs Data Frames

- Lists can contain different sizes and types of variables in each element.
- Lists are recursive variables, since they can contain other lists.
- You can index lists using [], [[]], or $.
- NULL is a special value that can be used to create "empty" list elements.
- Data frames store spreadsheet-like data.
- Data frames have some properties of matrices (they are rectangular), and some of lists (different columns can contain different sorts of variables).
- Data frames can be indexed like matrices or like lists.
- merge lets you do database-style joins on data frames.

1.9 Sampling and simulation

The sample command is a useful way of drawing random samples in R. (Technically, they are pseudo-random since there is an underlying deterministic algorithm, but they "look like" random samples for almost all practical purposes.) For example,

```r
n <- 10; k <- 5
sample(n,k)
```

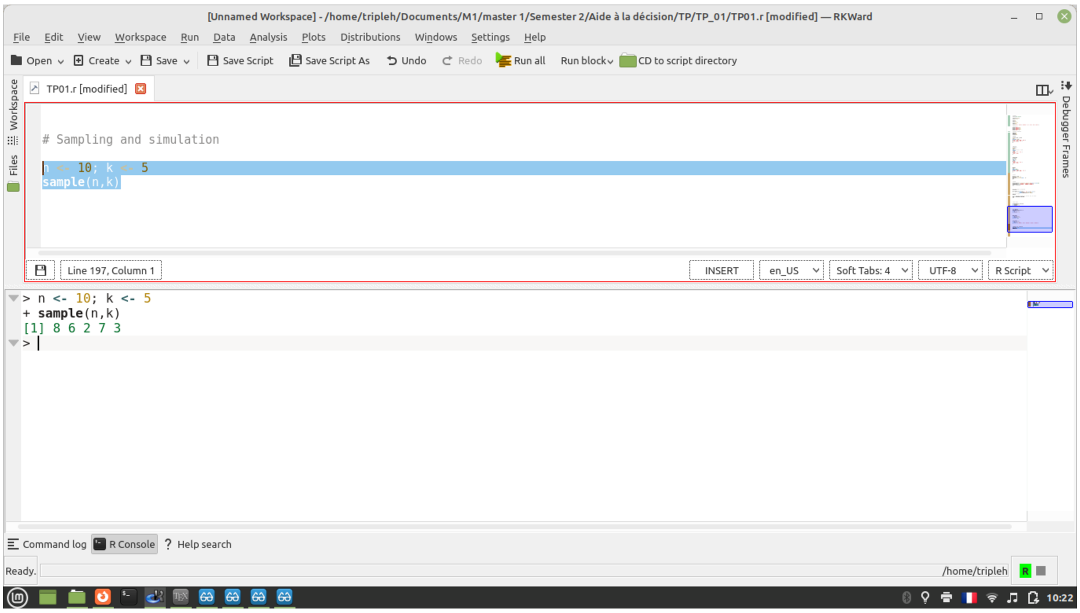

FIGURE 1.19: Sampling 1

generates an ordered random sample of 5 of the numbers from 1 to 10, without replacement, and with equal probabilities given to each number. To sample with replacement instead, just add in replace = TRUE:

```r
n <- 10; k <- 5
sample(n,k,replace=TRUE)
```

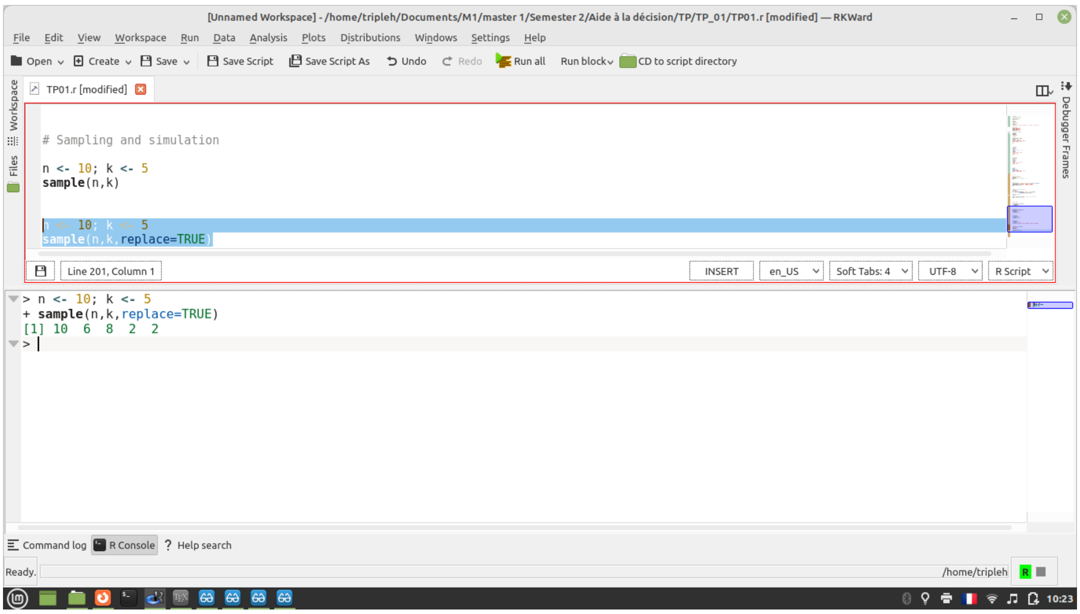

FIGURE 1.20: Sampling 2

To generate a random permutation of 1, 2, . . . , n we can use sample(n,n), which because of R's default settings can be abbreviated to sample(n). We can also use sample to draw from a non-numeric vector. For example, letters is built into R as the vector consisting of the 26 lowercase letters of the English alphabet, and sample(letters,7) will generate a random 7-letter "word" by sampling from the alphabet, without replacement.[1]

The sample command also allows us to specify general probabilities for sampling each number. For example,

```
sample(4, 3, replace=TRUE, prob=c(0.1,0.2,0.3,0.4))
```

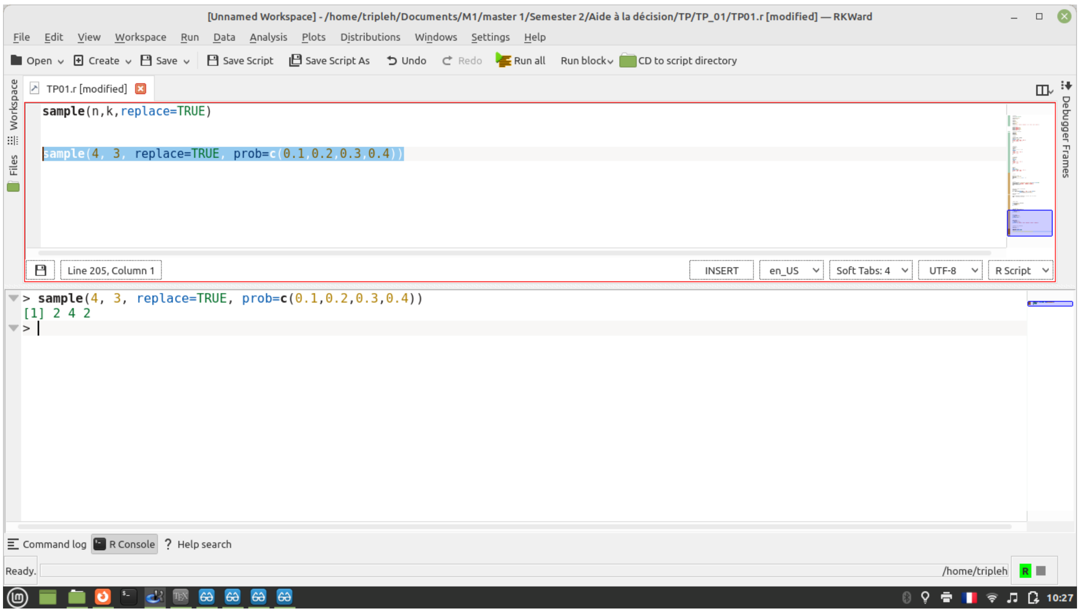

FIGURE 1.21: Sampling 3

samples three numbers between 1 and 4, with replacement, and with probabilities given by (0.1, 0.2, 0.3, 0.4). If the sampling is without replacement, then at each stage the probability of any not-yet-chosen number is proportional to its original probability. Generating many random samples allows us to perform a simulation for a probability problem. The replicate command, which is explained below, is a convenient way to do this.[1]

Command	What it does
`sample(7)`	random permutation of $1, 2, \ldots, 7$
`sample(52,5)`	picks 5 times from $1, 2, \ldots, 52$ (don't replace)
`sample(letters,5)`	picks 5 random letters of the alphabet (don't replace)
`sample(3,5,replace=TRUE,prob=p)`	picks 5 times from $1, 2, 3$ with probabilities p (replace)
`replicate(10^4,`*experiment*`)`	simulates 10^4 runs of *experiment*

FIGURE 1.22: Sampling 4

1.10 set seed

When you generate "random numbers" in R, you are actually generating pseudorandom numbers. These numbers are generated with an algorithm that requires a seed to initialize. Being pseudorandom instead of pure random means that, if you know the seed and the generator, you can predict (and reproduce) the output. In this tutorial you will learn the meaning of setting a seed, what does set.seed do in R, how does set.seed work, how to set or unset a seed, and hence, how to make reproducible outputs.[**noauthor_set_2020**]

1.10.1 What is to set seed in R?

Setting a seed in R means to initialize a pseudorandom number generator. Most of the simulation methods in Statistics require the possibility to generate pseudorandom numbers that mimic the properties of independent generations of a uniform distribution in the interval (0,1)(0, 1)(0,1).[**noauthor_set_2020**]

In order to obtain these sequences of pseudorandom numbers, we need a recursive algorithm called Random Number Generator (RNG):

$$xi = f(xi - 1, xi - 2, xi - 3, ..., xi - k)$$

where k is the order of the generator and $(x0, x1, x2, ..., xk - 1)(x_0, x_1, x_3, \ldots, x_{k-1})(x0, x1, x2, ..., xk - 1)$ is the seed (or initial state of the generator).[**noauthor_set_2020**]

There are several generators, that can be selected with the RNGkind function or with the argument kind of the R set.seed function, that uses by default the Mersenne-Twister generator.[**noauthor_set_2020**]

1.10.2 Why set seed in R?

When using functions that sample pseudorandom numbers, each time you execute them you will obtain a different result. Consider, for instance, that you want to sample 5 numbers from a Normal distribution. For that purpose you could type:

```
rnorm(5)
```

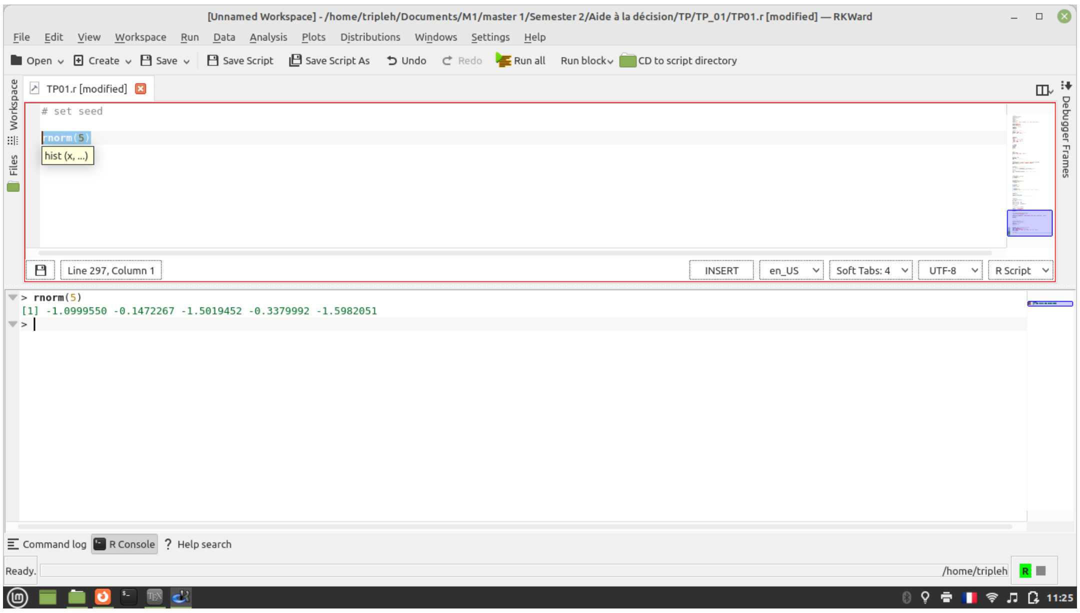

FIGURE 1.23: Set Seed 1

Nevertheless, if you execute the previous code you will obtain a different output. This implies that the code is not reproducible, because you don't know the seed that R used to generate that sequence.[**noauthor_set_2020**]

It is possible that you don't want your code to be reproducible, but there are several cases where reproducibility is desired. Set a seed in R is used for:

1. Reproducing the same output of simulation studies.
2. Help to debug the code when dealing with pseudorandom numbers.

1.10.3 How to set seed in R?

The purpose of the R set.seed function is to allow you to set a seed and a generator (with the kind argument) in R. It is worth to mention that:

1. The state of the random number generator is stored in .Random.seed (in the global environment). It is a vector of integers which length depends on the generator.
2. If the seed is not specified, R uses the clock of the system to establish one.

Run again the previous example where we sampled five random numbers from a Normal distribution, but now specify a seed before:

```
# Specify any integer
set.seed(1)

rnorm(5) # -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078
```

If you execute the previous code, you will obtain the same output. However, note that if you run rnorm(5) twice, it gives different results:

```
set.seed(1)

rnorm(5) # -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078
rnorm(5) # -0.8204684  0.4874291  0.7383247  0.5757814 -0.3053884
```

It should be noted that the previous block of code returns the same pseudorandom numbers than the following:

```
set.seed(1)
rnorm(10)

-0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078
-0.8204684  0.4874291  0.7383247  0.5757814 -0.3053884
```

This is due to when calling a random number generation function, the output depends on the values of .Random.seed, that changes after executing these functions. If you store the value of .Random.seed you can get the current seed state.

```
set.seed(1)
x <- .Random.seed
rnorm(5)

y <- .Random.seed
rnorm(5)

# .Random.seed is not equal in both cases
identical(x, y) # FALSE
```

In consequence, in case you want to output the same numbers twice, you have to set the same seed twice:

```
set.seed(1)
rnorm(5)   # -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078

set.seed(1)
rnorm(5)   # -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078
```

As we pointed out before, setting a seed in R is useful when working with simulation studies. Suppose that you want to calculate the median of some values from a uniform distribution:

As we used the set.seed function, if you execute the previous code you will obtain the following result:

```
# Set seed
set.seed(1234)

n_rep <- 10     # Number of repetitions
n <- 2          # Number of points

Median <- numeric(n_rep)

for (i in 1:n_rep) {
```

```
    Median[i] <- median(runif(n))
}

Median



0.3680014 0.6163271 0.7506130 0.1210231 0.5901674
0.6192831 0.6030835 0.5648057 0.2765220 0.2094744
```

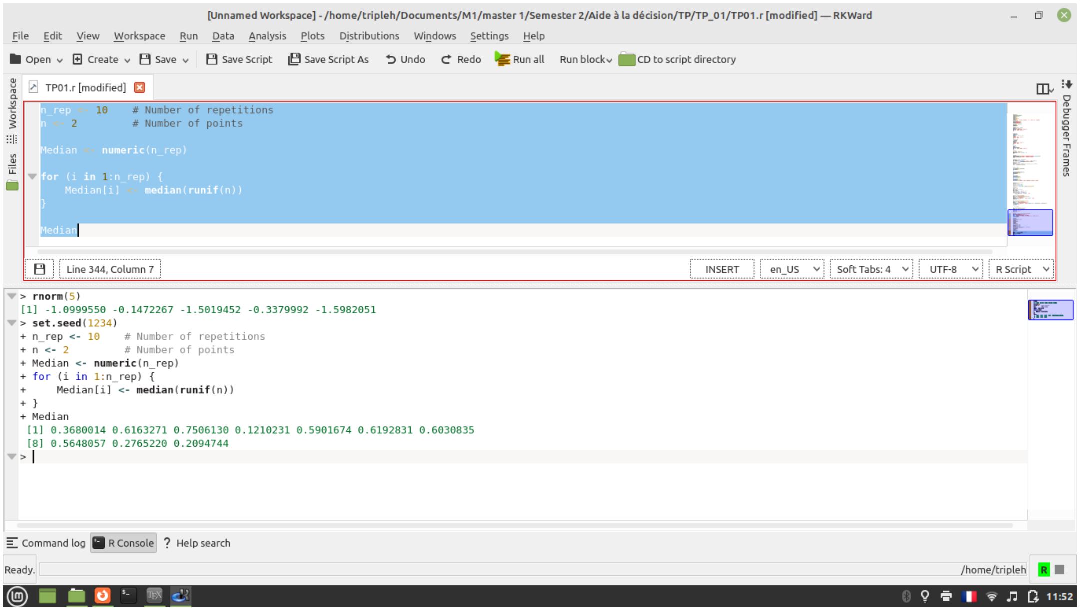

FIGURE 1.24: Set Seed 2

Nonetheless, if for some reason an error appears at some iteration you won't be able to reproduce the error. In order to solve this issue you have two options: saving the value of .Random.seed or changing the seed at each iteration:

```
set.seed(5)

for (i in 1:n_rep) {
  seed <- .Random.seed
  # If an error arises you can debug with: .Random.seed <- seed

  # Code

}
```

```
for (i in 1:n_rep) {
    set.seed(i)
    # If an error arises you can debug with set.seed(i)

    # Code

}
```

1.11 Plotting

the oldest system, having been around as long as R itself. base graphs are easy to get started with, but they require a lot of fiddling and magic incantations to polish, and are very hard to extend to new graph types. To remedy some of the limitations of base, the grid graphics system was developed to allow more flexible plotting. grid lets you draw things at a very low level, specifying where to draw each point, line, or rectangle. While this is wonderful, none of us have time to write a couple of hundred lines of code each time we want to draw a scatter plot. The second plotting system, lattice, is built on top of the grid system, providing high level functions for all the common plot types. It has two standout features that aren't available in base graphics. First, the results of each plot are saved into a variable, rather than just being drawn on the screen. This means that you can draw something, edit it, and draw it again; groups of related plots are easier to draw, and plots can be saved between sessions. The second great feature is that plots can contain multiple panels in a lattice, so you can split up your data into categories and compare the differences between groups.[2]

1.11.1 Line Plots

A line chart is a graph that connects a series of points by drawing line segments between them. These points are ordered in one of their coordinate (usually the x-coordinate) value. Line charts are usually used in identifying the trends in data.

The **plot()** function in R is used to create the line graph.

```
# Create some variables
x <- 1:10
y1 <- x*x
y2  <- 2*y1

# Create a basic stair steps plot
plot(x, y1, type = "S")


# Show both points and line
plot(x, y1, type = "b", pch = 19,
     col = "red", xlab = "x", ylab = "y")
```

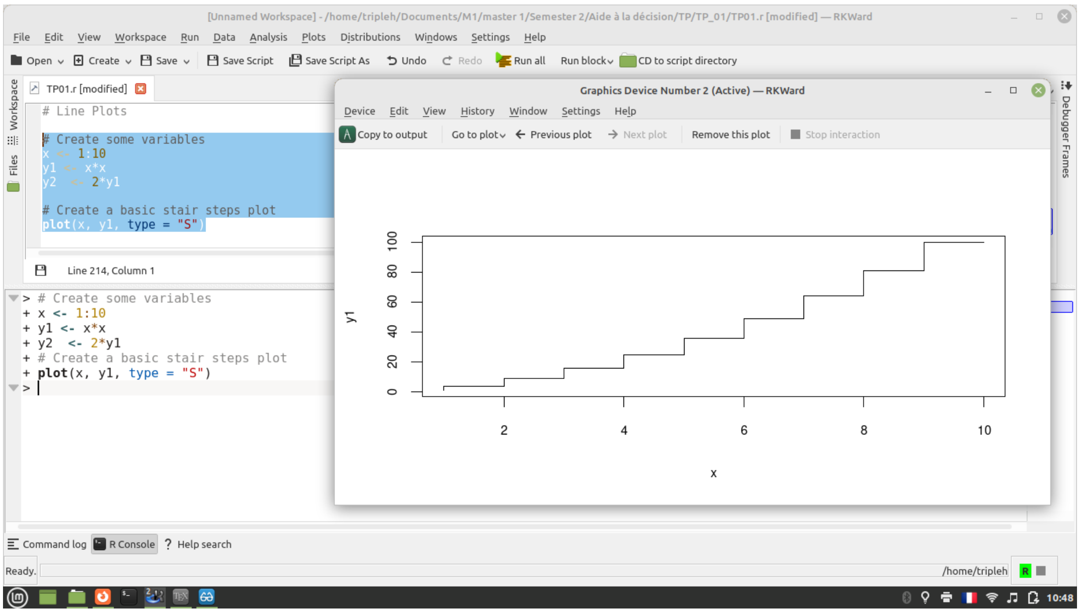

FIGURE 1.25: Line Plot 1

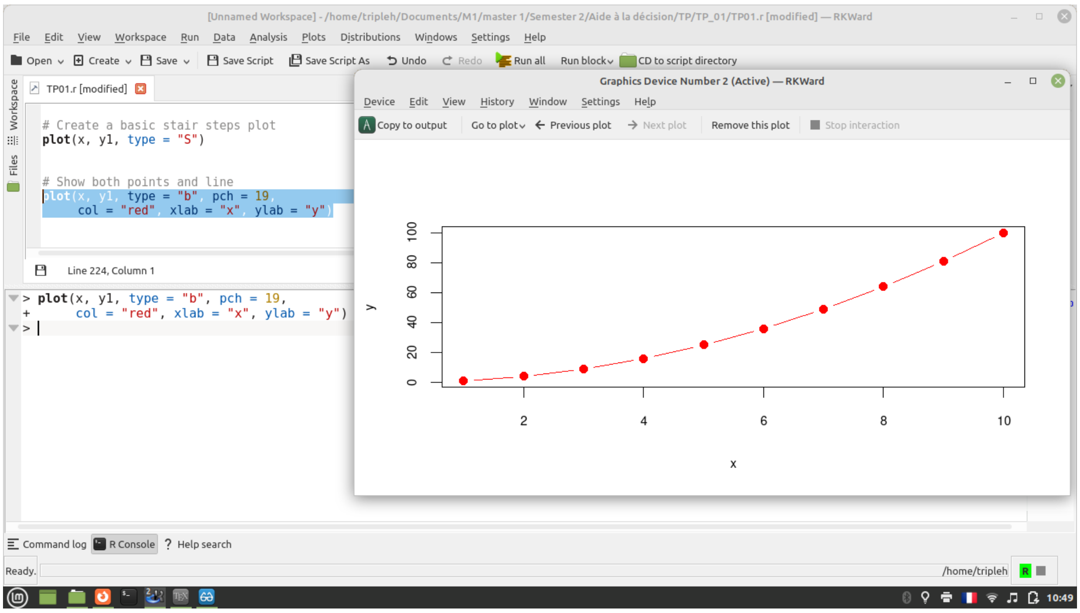

FIGURE 1.26: Line Plot 2

1.11.2 Histograms

A histogram represents the frequencies of values of a variable bucketed into ranges. Histogram is similar to bar chat but the difference is it groups the values into continuous ranges. Each bar in histogram represents the height of the number of values present in that range.

R creates histogram using **hist()** function. This function takes a vector as an input and uses some more parameters to plot histograms.

```
# Create data for the graph.
v <-  c(9,13,21,8,36,22,12,41,31,33,19)
```

```
# Give the chart file a name.
png(file = "histogram.png")

# Create the histogram.
hist(v,xlab = "Weight",col = "yellow",border = "blue")

# Save the file.
dev.off()
```

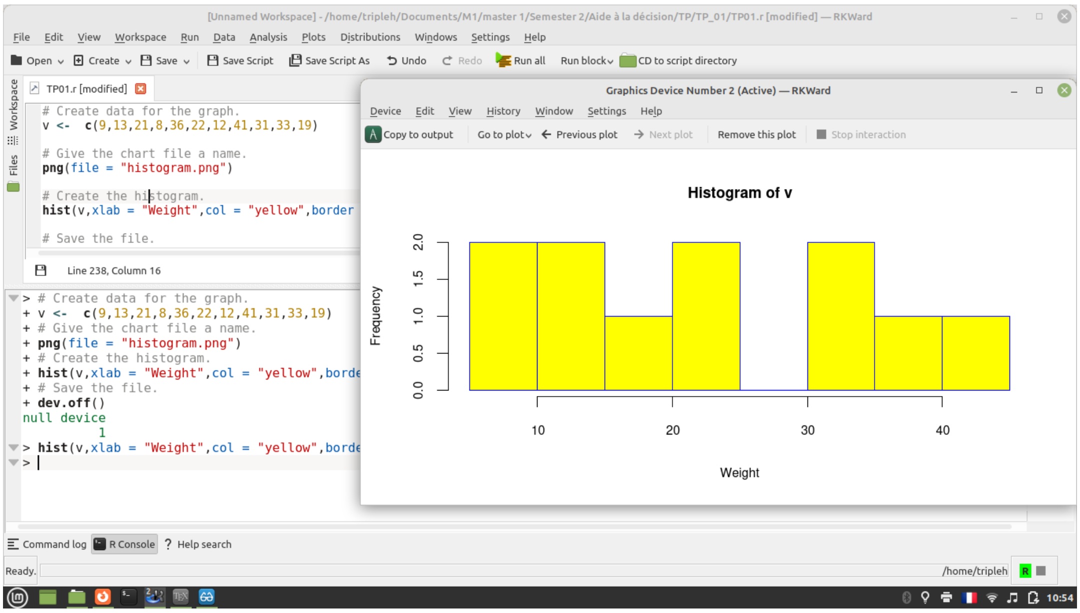

FIGURE 1.27: Histogram 1

To specify the range of values allowed in X axis and Y axis, we can use the xlim and ylim parameters.

The width of each of the bar can be decided by using breaks.

```
# Create data for the graph.
v <- c(9,13,21,8,36,22,12,41,31,33,19)

# Give the chart file a name.
png(file = "histogram_lim_breaks.png")

# Create the histogram.
hist(v,xlab = "Weight",col = "green",border = "red", xlim = c(0,40)
   , ylim = c(0,5),
   breaks = 5)

# Save the file.
dev.off()
```

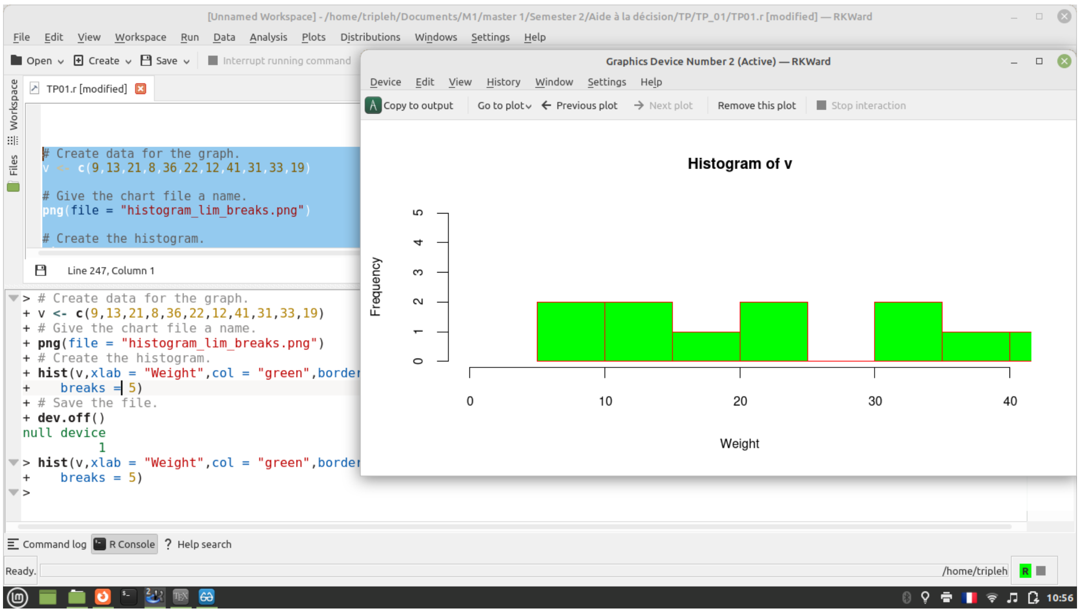

FIGURE 1.28: Histogram 2

1.11.3 Bar Charts

Create barplots with the barplot(height) function, where height is a vector or matrix. If height is a vector, the values determine the heights of the bars in the plot. If height is a matrix and the option beside=FALSE then each bar of the plot corresponds to a column of height, with the values in the column giving the heights of stacked "sub-bars". If height is a matrix and beside=TRUE, then the values in each column are juxtaposed rather than stacked. Include option names.arg=(character vector) to label the bars. The option horiz=TRUE to createa a horizontal barplot.

```
# Create the data for the chart
H <- c(7,12,28,3,41)

# Give the chart file a name
png(file = "barchart.png")

# Plot the bar chart
barplot(H)

# Save the file
dev.off()
```

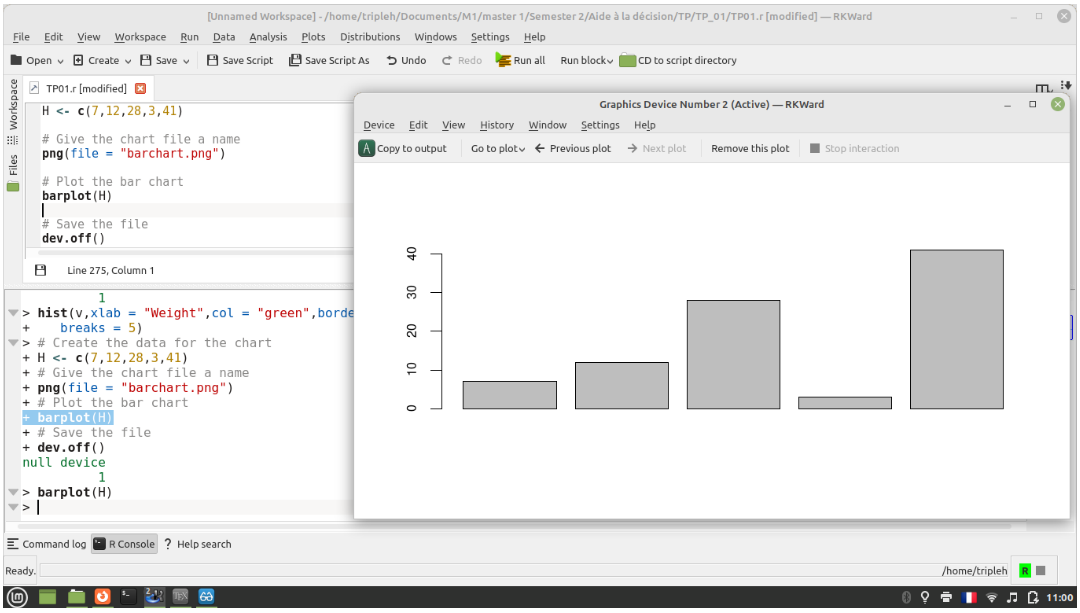

FIGURE 1.29: Bar Chart 1

The features of the bar chart can be expanded by adding more parameters. The main parameter is used to add title. The col parameter is used to add colors to the bars. The args.name is a vector having same number of values as the input vector to describe the meaning of each bar.

```r
max.temp <- c(22, 27, 26, 24, 23, 26, 28)
# barchart with added parameters
barplot(max.temp,
main = "Maximum Temperatures in a Week",
xlab = "Degree Celsius",
ylab = "Day",
names.arg = c("Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"),
col = "darkred",
horiz = TRUE)
```

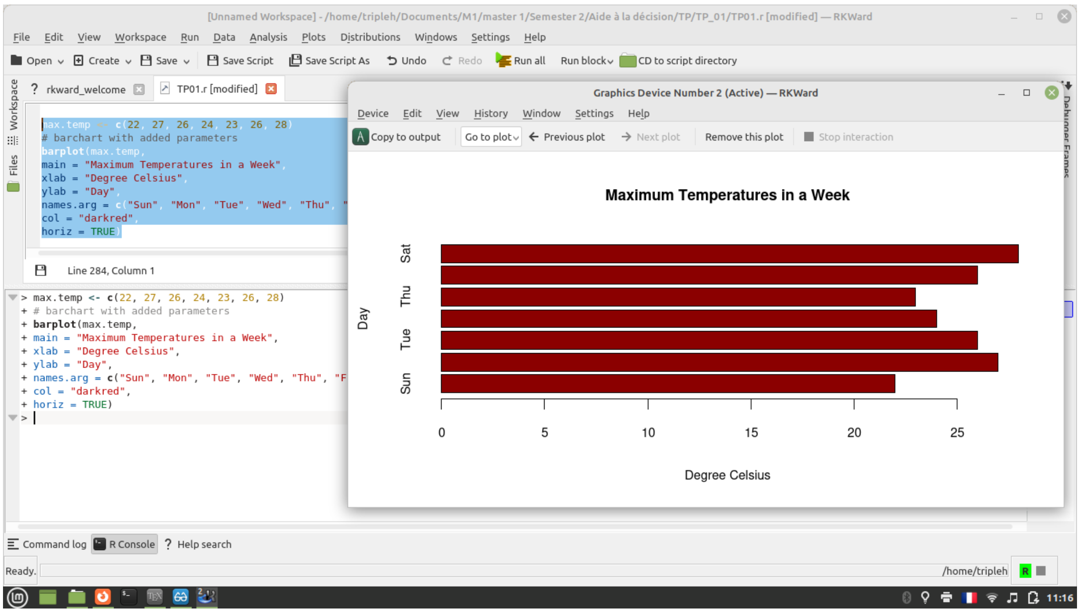

FIGURE 1.30: Bar Chart 2

Appendix A

Appendix A

A.1 R code

```
# Classes

class(c(TRUE, FALSE))

class(sqrt(1:10))

class(3 + 1i)

class(1)

class(1L)

class(0.5:4.5)

class(1:5)

class(c("she", "sells", "seashells", "on", "the", "sea", "shore"))


# Vectors in R

vector("numeric", 5)
vector("complex", 5)
vector("logical", 5)
vector("character", 5)
vector("list", 5)


# shortcut to Vectors

numeric(5)
complex(5)
logical(5)
character(5)


# Arrays


(three_d_array <- array(
1:24,
dim = c(4, 3, 2),
```

```
dimnames = list(
c("one", "two", "three", "four"),
c("ein", "zwei", "drei"),
c("un", "deux")
)
))






# Matrices

(a_matrix <-
 matrix(
1:12,
nrow = 4,
 #ncol = 3 works the same
dimnames =
 list(
c("one",
 "two", "three", "four"),
c("ein",
 "zwei", "drei")
)
))




(two_d_array
 <- array(
1:12,
dim = c(4,
 3),
dimnames =
 list(
c("one",
 "two", "three", "four"),
c("ein",
 "zwei", "drei")
)
))





matrix(
1:12,
nrow = 4,
byrow = TRUE,
dimnames = list(
c("one", "two", "three", "four"),
c("ein", "zwei", "drei")
)
```

```
)



# Lists


(a_list <- list(
c(1, 1, 2, 5, 14, 42),
month.abb,
matrix(c(3, -8, 1, -3), nrow = 2),
asin
))




# Tables

tab <- matrix(c(7, 5, 14, 19, 3, 2, 17, 6, 12), ncol=3, byrow=TRUE)
colnames(tab) <- c('colName1','colName2','colName3')
rownames(tab) <- c('rowName1','rowName2','rowName3')
tab <- as.table(tab)
tab






#make this example reproducible
set.seed(1)

#define data
df <- data.frame(team=rep(c('A', 'B', 'C', 'D'), each=4),
                 pos=rep(c('G', 'F'), times=8),
                 points=round(runif(16, 4, 20),0))

#view head of data
head(df)


#create table with 'position' as rows and 'team' as columns
tab1 <- table(df$pos, df$team)
tab1








# Data Frames


(a_data_frame <- data.frame(
x = letters[1:5],
```

```
y = rnorm(5),
z = runif(5) > 0.5
))





y <- rnorm(5)
names(y) <- month.name[1:5]
data.frame(
x = letters[1:5],
y = y,
z = runif(5) > 0.5
)




data.frame(
x = letters[1:5],
y = y,
z = runif(5) > 0.5,
row.names = NULL
)



data.frame(
x = letters[1:5],
y = y,
z = runif(5) > 0.5,
row.names = c("Jackie", "Tito", "Jermaine", "Marlon", "Michael")
)




# Sampling and simulation

n <- 10; k <- 5
sample(n,k)


n <- 10; k <- 5
sample(n,k,replace=TRUE)


sample(4, 3, replace=TRUE, prob=c(0.1,0.2,0.3,0.4))




# Plotting

# Line Plots

```

```
# Create some variables
x <- 1:10
y1 <- x*x
y2  <- 2*y1

# Create a basic stair steps plot
plot(x, y1, type = "S")


# Show both points and line
plot(x, y1, type = "b", pch = 19,
     col = "red", xlab = "x", ylab = "y")


 # Histograms



# Create data for the graph.
v <-  c(9,13,21,8,36,22,12,41,31,33,19)

# Give the chart file a name.
png(file = "histogram.png")

# Create the histogram.
hist(v,xlab = "Weight",col = "yellow",border = "blue")

# Save the file.
dev.off()




# Create data for the graph.
v <- c(9,13,21,8,36,22,12,41,31,33,19)

# Give the chart file a name.
png(file = "histogram_lim_breaks.png")

# Create the histogram.
hist(v,xlab = "Weight",col = "green",border = "red", xlim = c(0,40)
   , ylim = c(0,5),
   breaks = 5)

# Save the file.
dev.off()




 # Bar Charts



# Create the data for the chart
H <- c(7,12,28,3,41)

```

```r
# Give the chart file a name
png(file = "barchart.png")

# Plot the bar chart
barplot(H)

# Save the file
dev.off()






max.temp <- c(22, 27, 26, 24, 23, 26, 28)
# barchart with added parameters
barplot(max.temp,
main = "Maximum Temperatures in a Week",
xlab = "Degree Celsius",
ylab = "Day",
names.arg = c("Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"),
col = "darkred",
horiz = TRUE)


# set seed

rnorm(5)


# Specify any integer
set.seed(1)

rnorm(5)


set.seed(1)

rnorm(5)
rnorm(5)

set.seed(1)
rnorm(10)


set.seed(1)
x <- .Random.seed
rnorm(5)

y <- .Random.seed
rnorm(5)

# .Random.seed is not equal in both cases
identical(x, y) # FALSE


set.seed(1)
```

```
rnorm(5)

set.seed(1)
rnorm(5)

# Set seed
set.seed(1234)

n_rep <- 10     # Number of repetitions
n <- 2          # Number of points

Median <- numeric(n_rep)

for (i in 1:n_rep) {
    Median[i] <- median(runif(n))
}

Median
```

Bibliography

[1] Joseph K. Blitzstein and Jessica Hwang. *Introduction to probability*. Second edition. Boca Raton: CRC Press, 2019. ISBN: 9780429766732 9780429428357.

[2] Richard Cotton. *Learning R*. First Edition. OCLC: ocn830813799. Beijing ; Sebastopol, CA: O'Reilly, 2013. ISBN: 9781449357108.

[3] Garrett Grolemund. *Hands-on programming with R*. First edition. OCLC: ocn887746093. Sebastopol, CA: O'Reilly, 2014. ISBN: 9781449359010.

[4] Norman S. Matloff. *Probability and statistics for data science: math + R + data*. Boca Raton: CRC Press, Taylor & Francis Group, 2019. ISBN: 9780367260934 9781138393295.

[5] William Mendenhall, Robert J. Beaver, and Barbara M. Beaver. *Introduction to probability and statistics*. 14th ed./Student edition. Boston, MA, USA: Brooks/-Cole, Cengage Learning, 2013. ISBN: 9781133103752.

[6] Seema Singh. *Probability Distributions: Discrete and Continuous*. en. Apr. 2019. URL: `https://medium.com/@seema.singh/probability-distributions-discrete-and-continuous-7a94ede66dc0` (visited on 02/13/2022).