Fast intro to R

Germain Poullot

April 2026

Contents

Where to find good courses on R :

- Kristina Schubert lecture notes: https://www.kristina-schubert.de/statistik-mit-r/
- R for Data Science: https://r4ds.hadley.nz/
- Harvard YouTube video (9h): https://www.youtube.com/watch?v=g_3IKHG-rfA

Where to start seaching for good data:

- In German https://www.destatis.de/DE/Home/_inhalt.html
- In French https://www.insee.fr/fr/accueil
- Worldwide https://en.wikipedia.org/wiki/List_of_national_and_international_statistical_services

1 Course 1: First steps, the vector-table mindset, descriptive statistics

1.1 Downloading R

Go to https://www.r-project.org/, and download the latest (stable) version of R. Install it.
Go to https://posit.co/downloads and download the latest (stable) version of RStudio. Install it.

1.2 Making my first table

1.2.1 Where am I working?

Open RStudio; you should see 4 sub-windows.

First, we need to decide where we will run our code. Currently, R is running wherever you initially asked it to run, but you probably forgot where it is, so let's ask:

```
getwd()
```

If you want to change it (which you probably do, because R needs to handle a lot of data at once, so you would prefer to make it run in its own folder):

```
setwd("C:/the_path_to_my_directory")
```

`wd` stands for "working directory". You can do things more easily with RStudio: in the top-right, click on the RStudio logo, and open a new directory with the name "Course1".

1.2.2 Can you do math?

Now, let's check that R knows math. In the console (bottom left), write

```
5+3
```

Then execute it.

Also check the result of $3 + 5$.

Always writing in the console will be very annoying, so let's open a new *script* (i.e. a text file that contains the commands to execute): either go to File > New File > R Script, or press Ctrl+Shift+N.

Write again $5 + 3$ on the first line of your script, and press Ctrl+Enter.

This executes your current line. If you want to execute everything from the beginning, press Ctrl+Shift+Enter.

1.2.3 Vectors, tables

R uses *vectors* and *tables* (a.k.a. *frames*).

To create a vector, write (`c` stands for "combine"):

```
c (2, 5, −2, 2, −1)
```

I want to save this vector in a variable with a meaningful name.

```
Temp = c (2, 5, −2, 2, −1)
# or
Temp <− c (2, 5, −2, 2, −1)
```

These are the temperatures for a week (at the beginning of spring). Hence, let's create the days of the week.

```
Week = c ("Monday", 'Tuedsday', 'Wednesday', 'Thursday', 'Friday',
"Saturday", 'Sunday')
```

Note that, in the top-right, you have the names of your variables and a summary of what they contain.

It is very annoying to have these two vectors separated; I want them in a common *table*.

```
my_table = data.frame(Week, Temp)
```

Problem: I have 5 temperatures for 7 days of the week, so I cannot make a table. Add 0 and -1 to the temperatures. Re-execute `my_table = ...`.

Let's view my table:

```
View(my_table)
```

Note that the names of the columns were automatically set to be the names of your variables. You can access the columns of a table using the $ sign, e.g. `my_table$Temp` is the column containing the temperatures.

Let's compute the number of different temperatures, their mean, and their variance.

```
unique(my_table$Temp)
length(unique(my_table$Temp))

mean(my_table$Temp)
var(my_table$Temp)
```

To access the data inside my table, I can call them directly (remember: in matrices, the first entry is the row number, the second is the column number). The numbering starts at 1.

```
my_table[1, 2]
my_table[2, 1]

my_table[3:4, 1:2]
my_table[c(1, 3), 2]

my_table[1, ]
my_table[5, ]
my_table[, 1]
my_table[, 2]

my_table$Week[3]
my_table$Week[1:4]

my_table[9]
```

1.2.4 Filters

R is very good at applying the same operation to a whole vector. I want to know how many days had a negative temperature.

```
my_table$Temp < 0
my_table$Temp <= 0
```

This gives you a vector of answers. Note that, unlike many languages, in R the words `TRUE` and `FALSE` are uppercase, which is (annoying but) important.

I want the part of my table that contains only the days with negative temperatures.

```
which(my_table$Temp < 0)
my_table[my_table$Temp < 0, ]
subset(my_table, Temp < 0)
```

I want the mean temperature of the negative days. Try:

```
mean(my_table[my_table$Temp < 0, ])
```

It doesn't work: `my_table[my_table$Temp < 0, ]$Temp` is still a table. You didn't specify which column you want the mean of. Let's correct that:

```
mean(my_table[my_table$Temp < 0, ]$Temp)
```

1.2.5 Coordinate-wise operations

I want to estimate how much I will pay for buying my fruits. Let's download some data on fruit prices (because I'm too lazy to invent some myself).

```
url = "https://gist.githubusercontent.com/MamathaReddygari/
0b52821de0d7d92928c00e648f67f00c/raw/
70e3c578bbf49755e99e2af528c20be11d47a3d5/fruits.csv"

retail = read.csv(url)
View(retail)
```

A `csv` file is a file with *comma-separated values*. You can open it with a text editor, but when it becomes very large, you can open it directly in R.

I am not buying every possible fruit, so let's list the ones I want and extract them from the data.

```
my_fruits = c ("Bananas", "Apples", "Pears", "Figs", "Blueberries")

my_groceries = subset(retail, Fruit %in% my_fruits)
View(my_groceries)
```

I don't like frozen fruit, so let's remove it. The symbol `!` in front of a boolean statement means "not".

```
my_groceries = subset(my_groceries, !Form == "Frozen")
```

Similarly, I only care about the name of the fruit, the form, and the price.

```
my_groceries = my_groceries[, c("Fruit", "Form", "RetailPrice")]
```

As I am a bit of a perfectionist, I don't like that the rows of my table currently have strange numbers (they are the numbers from the original table `retail`). To fix this, there is an easy way: just remove the row names and let R recreate them from scratch.

```
rownames(my_groceries) = NULL
```

I go to the supermarket: I buy 200g of bananas, 500g of apples, 100g of pears, 150g of figs, and 120g of blueberries. Let's add this to my table.

```
my_groceries$Quantities = c(0.2, 0.5, 0.12, 0.15, 0.1)
```

Now, let's compute the price I paid for each item. Remember that R is very good at performing coordinate-wise operations on vectors.

```
my_groceries$PricePaid = my_groceries$RetailPrice * my_groceries$Quantities
sum(my_groceries$PricePaid)
```

1.3 Final exercise of the first session

1.3.1 Installing a package

The next table I want to work with is a common example already available in R, but we need to download a package to access it.

```
# This takes time, but you need to do it only once:
install.packages("nycflights13")

# This you need to redo each time:
library(nycflights13)
```

This package contains (among other things) a variable `flights`. Write the following and read in the bottom-right panel to understand what this variable contains:

```
?flights
```

Q: How many rows and columns does this table contain?
Q: List the years appearing in the table. How many different departure times are there?

1.3.2 Means and standard deviation depending on another column

Q: How many flights were there in March? What is the mean departure delay?
Q: Create a table `DELAYS` with one column named `month`.
I want to add a column computing the number of flights for each month. Let's use the `for` syntax.

```
for (i in c(1:12)) {
  DELAYS$nbr_flights[i] = sum(flights$month == i)
}
```

Q: Don't forget to view your table. Check that the total number of flights is indeed the sum of the number of flights for each month.
Now, let's add the mean departure delay and its standard deviation. This data is in the column `flights$dep_delay`. We will see one way now and a better way next time.

```
for (i in c(1:12)) {
  DELAYS$mean_delay[i] = mean(flights$dep_delay[flights$month == i])
  DELAYS$sd_delay[i] = sd(flights$dep_delay[flights$month == i])
}
```

All the means and standard deviations are `NA`. Why?
Let's debug a few things. Execute `flights$month == 7` to see the result, then `flights$dep_delay[flights$month == 7]`. How many values are there in this vector? Do you see some `NA` values?
The word `NA` means *not available*. R requires you to decide how to handle missing data, so it will never decide on its own what to do. Here, we want to remove these `NA` values (which is very different from considering them equal to 0). Luckily, there is a straightforward way to do this in the `mean` and `sd` functions (the letters `rm` mean "remove"):

```
mean(flights$dep_delay[flights$month == 7], na.rm=TRUE)
```

Q: Now, correct your code to compute the mean and standard deviation of the departure delays.

2 Course 2: `tidyverse`, making drawings, intro to linear regression

2.1 Downloading `tidyverse`

The package **`tidyverse`** ("tidy" = sorted in a clean way; "-verse" stands for universe) is efficient for cleaning and visualizing data. First of all, let's install it and make it available in our session. Create a new directory **`SecondCourse`**, then write:

```
install.packages("tidyverse") # Takes time, may complain

library(tidyverse)
```

To discover what the package can do, we will work on the **`tibble`** called **`storms`**, in order to extract data on hurricanes. A **`tibble`** is just a variant of a **`table`**, but created by **`tidyverse`**.

Visualize the content of the **`tibble`** named **`storms`**.

`tidyverse` is not just one package; it actually contains many sub-packages with their own names: we will mainly use **`dplyr`** and **`ggplot2`** today.

2.2 Cleaning data

2.2.1 Selecting columns and rows, using piping

To call a function from a package, you can use double colons, e.g. **`dplyr::select`**. However, **`tidyverse`** loads many functions directly, which means you can just write **`select`** instead. Type:

```
?select
?dplyr::select
```

Then understand what the following does:

```
select(storms,
       -c(lat, long)
       )

select(storms,
       -c(lat, long, ends_with("diameter"))
)

filter(storms,
       status == "hurricane")
```

Typically, in R, you want to perform an operation on a table (or **`tibble`**), then perform another action on the result, and so on. Writing $f(g(h(\dots)))$ quickly becomes annoying, so R provides a convenient alternative called *piping*, denoted by **`|>`** (like a smaller version of $\mapsto$). This operator passes what precedes it as the first argument of the following function. Look at the following **`tibble`**.

```
storms |>
  select(-c(lat, long, ends_with("diameter"))) |>
  filter(status == "hurricane")
```

Save the resulting **`tibble`** in a variable named **`hurricanes`**, view it, compute the number of different hurricane names, and the mean pressure.

We now want to sort the data efficiently. To this end, we will use another function from **`tidyverse`** that arranges the **`tibble`** in ascending (or descending, if specified) order for a given column.

```
arrange(hurricanes, wind)
arrange(hurricanes, desc(wind))
```

```
arrange(my_tibble, desc(wind), name)

arrange(hurricanes, desc(wind), name) |>
  distinct(name, year)
```

The function `distinct` allows you to keep only the rows with unique values.

Now that we are almost satisfied with the data extracted from the original `tibble` named `storms`, we can save it as a `.csv` file.

```
clean_hurricanes = arrange(hurricanes, desc(wind), name) |>
  distinct(name, year, .keep_all = TRUE)

clean_hurricanes |>
  select(c(year, name, wind, pressure)) |>
  write.csv("hurricanes.csv", row.names = FALSE)
```

2.2.2 Grouping, slicing, summarizing

Open a new script, say `SecondScript.R`, and load the `tibble` named `hurricanes` that you have just saved.

Now, we can do what we did at the end of the last session, but faster and more cleanly. First, we group the content of the table by year.

```
hurricanes |>
  group_by(year)
```

The content is not only visually grouped by year; it is also interpreted by R as a collection of "big rows" named 1980, 2019, 1988, 2005, etc., each containing sub-rows. Hence, we can perform operations on this structure. For instance, let's remove all rows except the first one: if we do this on a normal `tibble`, we keep only one row, but here we keep one row per group.

```
hurricanes |>
  group_by(year) |>
  slice_head()

hurricanes |>
  group_by(year) |>
  slice_min(order_by = pressure) |>
  filter(year >= 1980 & year >= 2000)
```

We can also create summaries of the groups, e.g. count the number of elements per group or compute the mean of some value:

```
hurricanes |>
  group_by(year) |>
  summarise(n())

hurricanes |>
  group_by(year) |>
  summarise(number_of_hurricanes = n())
```

Q: Construct a `tibble` with 3 columns indicating the mean wind speed and the number of hurricanes for each year.

2.2.3 Fixing poorly formatted data

Q: Download the `tibble` named `students` and understand the problem.
As this problem is quite common, there is a direct solution in the `tidyverse`:

```
students = pivot_wider(students,
             id_cols = name,
             names_from = attribute,
             values_from = value
             )
```

Q: Try to create a `tibble` containing the mean GPA (Grade Point Average) in each major.
Q: Notice that you have a problem, identify it, and correct it (you can use `as.numeric(...)` for instance).
Q: Type `?pivot_longer` and strengthen your understanding.
As you can imagine, there are many other problems that can occur when working with data produced by others. You now know the fundamentals: check whether `tidyverse` can handle your problem. In the future, you can use the `tidyverse` documentation, search the Internet, or ask an AI which commands to use; but you should not reinvent the wheel!

2.3 Final exercise on cleaning data: cleaning strings

Q: Download `favorite_subjects.csv` into a `tibble` named `best`.
Q: Group your data to count how many times each subject has been voted for, and sort them in decreasing order.
Q: Notice that there are several problems. Visualize the beginning of your `tibble` to understand what needs to be done.
To remove spaces at the beginning and end (respectively inside) of a string, you can use `str_trim()` (respectively `str_squish()`). To manipulate case (upper/lower), you can use `str_to_title()`, `str_to_upper()`, or `str_to_lower()`.
Q: Start cleaning your data, and look at your poll again.
You still have a German versus English issue. This is more delicate, so let's handle it manually.
Q: Create the following two vectors, and use a clever `for` loop to replace German words with English ones in `best` (use the next two commands and understand what they do). Then compile your poll and save the result in a variable named `poll`.

```
German = c('Statistik', 'Mathematik', 'Informatik')
English = c('Statistics', 'Mathematics', 'Computer Science')
```

```
any(best$Subject %in% German)
all(best$Subject %in% English)
```

Q: Think about (do not implement it) a way to combine the results for algebra, statistics, and mathematics, and those for data science and computer science.

2.4 Making nice drawings

2.4.1 The basics of `ggplot`

To create a drawing, R proceeds step by step. The first step is to create a blank page with `ggplot(...)`. We are going to visualize the recent `poll` that you just made, so let's pass this variable to our plot: `ggplot(poll)`. Nothing happens, because R (or more precisely `tidyverse`) does not yet know what you want to plot exactly, but the data are loaded.
Second, we need to specify what goes on the x-axis and what goes on the y-axis. This is done using *aesthetics*, which is a somewhat convoluted concept: each function that we are going to use is supposed to do something on its own, and you can pass it some aesthetics to explain how it is

supposed to do its job. Hence, we ask `ggplot` to load the data from `poll`, and we specify that the x-axis should come from the vector `poll$Subject` while the y-axis should come from `poll$votes`. As `poll` is already loaded, you only need to write:

```
ggplot(poll, aes(x = Subject, y = votes))
```

Err... that's interesting, but where is my plot? Well, you did not tell R what you wanted to plot: you have set up the scenery, but not the play. The "play" here is called a *geometry*. As we are doing a poll, the geometry that we want to use is probably a histogram, that is to say `geom_col()`. You can add it (literally) to your plot, as follows:

```
ggplot(poll, aes(x = Subject, y = votes)) +
  geom_col()

ggplot(poll, aes(y = Subject, x = votes)) +
  geom_col()

ggplot(poll, aes(x = Subject, y = votes)) +
  geom_col() + coord_flip()
```

Note that, in the last call, we have added `coord_flip()`, which flips the x-axis and the y-axis. This is a basic example of how plotting works: you create a plot, you add the geometry, then you add a bunch of modifiers on top. Everything is done with addition (which is commutative, so it does not depend on the order[1]).

You have probably noticed that the column plot has reordered your data alphabetically. This is annoying, so we bypass that. We also add some labels to our plot.

```
ggplot(poll, aes(x = reorder(Subject, votes, decreasing = TRUE), y = votes)) +
  geom_col() +
  labs(x = "Subject", y = "nb-votes", title = "Favorite-subjects")
```

The following are discussions of some common aesthetics and modifiers that you may want to apply to your plot. You should try them at least once: even though it may be a bit boring, it will make you more comfortable with plotting. There are many more options, and many variations of each. I leave it to you to imagine the possibilities[2].

Changing the limits of your scale

```
ggplot(poll, aes(x = Subject, y = votes)) +
  geom_col() +
  scale_y_continuous(limits = c(0, 40))
```

Filling the columns with colors

```
ggplot(poll, aes(x = reorder(Subject, votes, decreasing = TRUE), y = votes)) +
  geom_col(aes(fill = Subject))
  labs(x = "Subject", y = "nb-votes", title = "Favorite-subjects")
```

Adapting for colorblind people (Viridis is the name of a color standard in R , and `d` stands for "discrete")

```
ggplot(poll, aes(x = reorder(Subject, votes, decreasing = TRUE), y = votes)) +
  geom_col(aes(fill = Subject)) +
  scale_fill_viridis_d("Theme") +
  labs(x = "Subject", y = "nb-votes", title = "Favorite-subjects")
```

[1] But the human being reading your code might care deeply about the order...

[2] Note that R accepts both British and American spelling for words like `color`/`colour`.

Removing the legend

```
ggplot(poll, aes(x = reorder(Subject, votes, decreasing = TRUE), y = votes)) +
  geom_col(aes(fill = Subject), show.legend = FALSE) +
  scale_fill_viridis_d("Theme") +
  labs(x = "Subject", y = "nb votes", title = "Favorite subjects") +
  theme_classic()
```

By now, you should be satisfied with your first plot, so we can save it. Remember that a plot is an object that exists in your computer's memory, so you can store it in a variable. The unit `px` stands for "pixel": the default unit is inches (≈ 2.54 cm), which is not convenient in most cases.

```
p = ggplot(poll, aes(x = reorder(Subject, votes, decreasing = TRUE), y = votes)) +
  geom_col(aes(fill = Subject), show.legend = FALSE) +
  scale_fill_viridis_d("Theme") +
  labs(x = "Subject", y = "nb votes", title = "Favorite subjects") +
  theme_classic()

ggsave("FavoriteSubject.png", plot = p, height = 900, width = 2400, units = "px")
```

2.5 Final exercise on plotting: drawing the Saffir–Simpson scale

We want to know whether there is some dependence between wind speed and pressure in hurricanes. (If you have unloaded the variable `hurricanes`, remember that you saved it in a `.csv` file, so you can reload it at any time.)

Q: Make a plot of the `hurricanes` variable, with pressure on the x-axis and wind speed on the y-axis, using points (not columns). The theme should remain classic, and the title of your plot should be "Hurricanes".

Q: Set the shape of the points to `shape = 1`, and color them according to the year the hurricane occurred.

Q: It is hard to distinguish between the points; can you jitter them (use `geom_jitter`)? Now that you can see better, what kind of hurricanes would you like to observe to complete your data?

Q: (Go back to the non-jittered point plot.) We want to categorize hurricanes. By definition, a hurricane is a storm with a wind speed greater than 64 knots (1 knot ≈ 1.852 km/h, so roughly 118 km/h), but according to the Saffir–Simpson scale, we can subdivide hurricanes further into categories depending on their wind speed:

- between 64 knots and 82 knots: category I;
- between 83 knots and 95 knots: category II;
- between 96 knots and 112 knots: category III;
- between 113 knots and 136 knots: category IV;
- beyond 137 knots: category V;

Using the following vector and the function `findInterval(x, vec)`, which finds between which values of the vector `vec` the values in `x` lie, add a column `category` to your `tibble` named `hurricanes`.

```
SaffirSimpson = c(64, 83, 96, 113, 137)
```

Q: We can indicate in the aesthetics of our plot that our data should be treated as several subtables. Using the following beginning, draw a point plot (with shapes 1), called "Hurricanes", in the classic theme, without a legend.

```
p = ggplot(hurricanes, aes(x = pressure, y = wind, color = factor(category))) + ...
```

Q: I do not like the colors, so let's change them using the following vector. To do so, you need to scale the colors manually, in the same way as you scaled the limits of the y-axis.

```
my_colors = c('green', 'orange', 'red', 'purple', 'black')
```

Q: Do not forget to store your plot in a variable named `p`. I want to add horizontal lines to the plot to separate the categories. Remember that we are adding elements to our plot, so nothing prevents us from adding another geometry. The geometry for a horizontal line is `geom_hline(...)`, and you can pass it an argument `yintercept = ...` to indicate the y-value of the line. Draw 6 horizontal lines, colored according to `my_colors`, at the heights given in the Saffir–Simpson scale (add 0.5 for cleaner separation). Use `linetype = 3` to obtain dotted lines.

2.6 Towards linear regression

As we have factored our data, not only can we treat their colors separately, but we can also apply other methods separately. For instance, let's add the line obtained by linear regression for each category of hurricanes (`lm` stands for "linear model"; smooth geometries are tools for computing smooth trends, and by default they use LOESS[3]).

Try to understand what `se` could mean.

```
p + geom_smooth(method = "lm", se = FALSE, show.legend = FALSE)
p + geom_smooth(method = "lm", se = TRUE, show.legend = FALSE)
```

Finally, for the sake of exploration (and because we may not have time to cover it in detail), let's briefly look at linear regression:

```
model <- lm(wind ~ pressure, data = hurricanes)
summary(model)
```

To get the coefficients, you can write:

```
model$coefficients[1]
model$coefficients[2]
```

If you want the *coefficient of determination* (usually denoted R^2), you can access it via:

```
r2 <- summary(model)$r.squared
r2
```

Finally, let me add the global regression and its coefficient of determination, instead of computing a regression for each category:

```
q = ggplot(hurricanes, aes(x = pressure, y = wind)) +
  geom_point(shape = 1, show.legend=FALSE) +
  labs(title = "Hurricanes") +
  theme_classic()
for (i in 1:6) {
  q = q + geom_hline(linetype = 3, yintercept = SaffirSimpson[i]+0.5, color=my_colors[i])
}
q + geom_smooth(method = "lm", se = TRUE, show.legend = FALSE) +
  annotate("text",
           x = Inf, y = Inf,
           label = paste0("R^2=", round(r2, 3)),
           hjust = 1.1, vjust = 1.5)
```

[3] Local polynomial regression

3 Course 3: Creating and sampling probability distributions

3.1 Visualizing usual distributions

Usual distributions are already implemented in R. Each implemented distribution comes in 4 ways (usually): *density*, *distribution*, *quantiles*, and *sampling*. For instance, for the uniform distribution, the built-in functions are `dunif`, `punif`, `qunif`, and `runif`. For disambiguation, say that $X \sim \text{Unif}([0,1])$, let f be the density function (i.e. the derivative of $t \mapsto \mathbb{P}(X \leq t)$), then we have:

- `dunif(t)` $= f(t)$
- `punif(t)` $= \mathbb{P}(X \leq t)$
- `qunif(z)` $= \min\{t \in \mathbb{R} \ ; \ \mathbb{P}(X \leq t) \geq z\}$ for $z \in [0,1]$
- `runif(n)` is a vector of length n where each entry is a sample from the uniform distribution

You can get more information using:

```
?dunif
?punif
?qunif
?runif
```

Let's make a table to visualize all this stuff.

```
x = (0:100) / 100
density = dunif(x)
distrib = punif(x)
quantile = qunif(x)

Unif = data.frame(x, density, distrib, quantile)
View(Unif) # It is weird to put quantile in this table...
```

Q: Explain why it is weird to put everything in the same table (the next example will be more striking).

We draw the density and the distribution using `ggplot`. **Nota Bene**: If you do not manage to install `tidyverse` (and `ggplot`), you can use `hist(...)` instead of `geom_histogram(...)`, and `curve(...)` instead of `stat_function(...)`. Good luck!

```
library(tidyverse)

ggplot(data.frame(x = c(-0.33,1.33)), aes(x = x)) +
  stat_function(fun = dunif, linewidth = 0.5, color='blue') +
  stat_function(fun = punif, linewidth = 0.1, color='red') +
  labs(title = "Density and distribution of Uniform(0,1)", y = "Density, Distribution") +
  ylim(0, 1.2) +
  theme_classic()
```

We do the same with the normal distribution, because we want to compare them. Comment on the change of `x`, and look at the `quantile` column in the `Norm` table.

```
?dnorm
?pnorm
?qnorm
?rnorm

x = (-500:500) / 100
density = dnorm(x)
distrib = pnorm(x)
quantile = qnorm(x)
```

```
Norm = data.frame(x, density, distrib, quantile)
View(Norm) # here, it is clear that quantile is a bad idea!

ggplot(data.frame(x = c(−5,5)), aes(x = x)) +
  stat_function(fun = dnorm, linewidth = 0.5, color='blue') +
  stat_function(fun = pnorm, linewidth = 0.1, color='red') +
  labs(title = "Density and distribution of Normal(0,1)", y = "Density, Distribution") +
  ylim(0, 1.1) +
  theme_classic()
```

We finish this introduction by comparing the quantile functions for the uniform and the normal distributions.

```
ggplot(data.frame(x = c(−0.1,1.1)), aes(x = x)) +
  stat_function(fun = qunif, color = 'green') +
  stat_function(fun = qnorm, color = 'orange') +
  ylim(−2.5, 2.5) +
  theme_classic()
```

3.2 Covariance matrix to analyze a shape via Monte Carlo method

Suppose you want to determine the shape of some plane object (formally fix $K \subseteq \mathbb{R}^2$), but you can only try asking whether a point is in the shape or not (formally, you have access to an oracle $f : (x, y) \mapsto \big((x, y) \in K\big)$). You know that the whole shape is contained in the square $[0, 100]^2$.

You can of course pick points regularly inside $[0, 100]^2$ and ask if they are in the shape (formally fix a large n and ask $f(x_i, y_j)$ for $x_i = \frac{100i}{n}$ and $y_j = \frac{100j}{n}$ for all $0 \leq i, j \leq n$), then draw the shape using a computer. However, this can be very costly, and it is usually impossible to do in real life (due to both measurement errors and experimental setups). One way around this is to use a Monte Carlo method: sample points at random in your shape. Sampling uniformly inside the shape directly is usually tricky[4], yet one can sample uniformly inside a larger shape (e.g. a disk or a square) and filter to keep only the points lying inside your shape.

The same principle applies if the shape is the result of a certain function: one can sample in the domain of definition, then apply the function to every sample point, obtaining a sample of the image shape. This sample will not necessarily be uniform enough, but there are ways to make it more uniform. We do not explore this aspect here, but focus on sampling uniformly in a square and then testing whether the points belong to the shape.

3.2.1 Sampling in my shape

First, let's sample uniformly inside a square.

```
XY = cbind(x = runif(100, ), y = runif(100))
View(XY)

nrow(XY)

cov(XY)
```

Q: What were you expecting the covariance matrix to be?

Then, we need to import a shape. I have made an auxiliary script in R that you need to download and save next to your current script, named `what_is_my_shape.R`. You can open it manually, but that would be cheating.

You can import the functions therein and test them. My shape is inside the box $[-1.5, +3] \times [-2, +2]$.

[4] For instance, I know no efficient algorithm to sample points uniformly at random in a polytope.

```
source("what_is_my_shape.R")

in_my_shape(4, 2)
in_my_shape(0, 0)
```

Now, we sample points uniformly at random inside $[-1.5, +3] \times [-2, +2]$, and visualize them.

```
XY = data.frame(x = runif(1000, min = -1.5, max = 3), y = runif(1000, min = -2, max = 2))
nrow(XY)

ggplot(XY, aes(x = x, y = y)) +
  geom_point()
```

We want to apply the function `in_my_shape` to every row of the table `XY`, but "sadly", I did not *vectorize* my function, that is to say I did not adapt it to use a vector (nor a data frame) as an argument, and return a vector (or a data frame). We can easily make do with that (`mapply` stands for "apply map to"):

```
mapply(in_my_shape, XY$x, XY$y)
```

This should be a vector of `TRUE` and `FALSE`, indicating which of the points you have randomly chosen are inside my shape, and which are not. We make a table out of that, and visualize it.

```
MonteCarlo = XY[mapply(in_my_shape, XY$x, XY$y), ]
View(MonteCarlo)

nrow(MonteCarlo)

ggplot(MonteCarlo, aes(x = x, y = y)) +
  geom_point()
```

Q: Without writing any new line, what would be your guess of the area of my shape? How can you make your guess more precise? (math question: does the method you are proposing converge? under which mode of convergence? how fast?)

3.2.2 Principal Component Analysis

My shape is weird, but it clearly has some fundamental axis. In the same way, clouds of points coming from real-life data usually have a fundamental axis. A very simple way to detect such a fundamental axis is called *Principal Component Analysis*, which means computing the eigenvector of the covariance matrix associated with the largest eigenvalue. Remember that the covariance matrix is a symmetric matrix, so all its eigenvalues are real, and the covariance matrix is diagonalizable.

We can directly compute the spectral decomposition of a matrix in R. The result is usually sorted in decreasing order of eigenvalues.

```
Sigma = cov(MonteCarlo)
eigen(Sigma)
```

Using `tidyverse`, we add arrows in the directions given by the eigenvectors, i.e. the columns of `eigen(Sigma)$vectors`. We scale each arrow according to its eigenvalue: hence their relative sizes indicate the importance of each direction compared to the other direction. If all the arrows have the same size, or if they are very small, then one should be doubtful about the existence of a fundamental axis: the point cloud is either very symmetric, or completely chaotic, or has many different axes/clusters, or the representation is not adapted (take a semi-log scale, or consider another transformation), or else.

```
ei = eigen(Sigma)$values
P = eigen(Sigma)$vectors

ggplot(MonteCarlo, aes(x = x, y = y)) +
  geom_point() +
  geom_segment(aes(x = -ei[1]*P[1, 1], y = -ei[1]*P[2, 1], xend = ei[1]*P[1, 1], yend = ei[1]*P[2, 1]),
               arrow = arrow(ends = 'both', length = unit(0.3, "cm")),
               color = "red", linewidth = 1.2) +
  geom_segment(aes(x = -ei[2]*P[1, 2], y = -ei[2]*P[2, 2], xend = ei[2]*P[1, 2], yend = ei[2]*P[2, 2]),
               arrow = arrow(ends = 'both', length = unit(0.3, "cm")),
               color = "blue", linewidth = 1.2) +
  theme_classic()
```

Nota Bene: The covariance matrix is a symmetric real matrix; hence, not only is it diagonalizable, but its eigenvectors form an orthogonal basis. Thus, in the plane, the second eigenvector is necessarily perpendicular to the first. Its direction does not provide much information; only its magnitude does. See the next section for a more involved method.

We can now reveal what my shape originally was:

```
reveal_shape()
```

You can also play with a second shape using `in_my_shape2` and `reveal_shape2`.

Nota Bene: This game also works perfectly in higher dimensions, of course! For instance, in dimension 3, the first eigenvector of the covariance matrix will indicate the fundamental direction, then the second one will indicate which direction in the plane orthogonal to the first is the most important, and so on. The relative sizes of the eigenvalues indicate the relative importance of these directions.

3.2.3 Independent Component Analysis

Obviously, there exists a method that is far more powerful (and more costly) to describe the fundamental axes of a point cloud, namely *Independent Component Analysis*. We will not present the mathematical details of this method, but only give the  implementation. Please refer to Wikipedia or any good book to learn more about the subject.

To run the following, you will need a package, so execute once `install.packages("ica")`. The value `nc` is the number of components, which is basically the dimension. There is a lot of information in the result of an independent component analysis, but we only care about `M`, the matrix whose columns are the directions of the components, and `vafs`, which is the *variance-accounted-for* by each component, i.e. the relative importance of each direction. Note that the result is often more accurate.

```
#install.packages("ica")
library(ica)
?ica

fit = ica(MonteCarlo, nc = 2, method = "fast")
View(fit)

P = fit$M
vafs = fit$vafs

ggplot(MonteCarlo, aes(x = x, y = y)) +
  geom_point() +
  geom_segment(aes(x = -vafs[1]*P[1, 1], y = -vafs[1]*P[2, 1], xend = vafs[1]*P[1, 1], yend = vafs[1]*P[2, 1]),
               arrow = arrow(ends = 'both', length = unit(0.3, "cm")),
```

```
            color = "red", linewidth = 1.2) +
  geom_segment(aes(x = -vafs[2]*P[1, 2], y = -vafs[2]*P[2, 2], xend = vafs[2]*P[1, 2], yend = vafs[2]*P[2, 2]),
            arrow = arrow(ends = 'both', length = unit(0.3, "cm")),
            color = "blue", linewidth = 1.2) +
  theme_classic()
```

3.3 Verifying Wigner semi-circle law

Recall that an *eigenvalue* of a matrix A is a number λ such that there exists a non-zero vector x satisfying $Ax = \lambda x$. There are at most n eigenvalues, which may be complex numbers even if A has only real coefficients, but if A is a real symmetric matrix, then A has exactly n real eigenvalues (counted with multiplicities).
The aim is to verify experimentally the following statement from Eugene Wigner:

> Let W_n be a random symmetric $n \times n$ matrix, e.g. a symmetric matrix whose entries are normally distributed, and let λ_n be an eigenvalue of W_n chosen at random (uniformly) among all eigenvalues of W_n. Then, asymptotically, up to normalization, λ_n follows a *Wigner semi-circle distribution* whose density function is the curve of a semi-circle, i.e. $t \mapsto \frac{2}{\pi}\sqrt{1-t^2}$ for $t \in [-1, +1]$, and 0 elsewhere (without normalization, one can use the general semi-circle distribution with density $t \mapsto \frac{2}{R^2\pi}\sqrt{R^2-t^2}$ for $t \in [-R, +R]$, and 0 elsewhere, for some radius R).

In particular, looking at all the eigenvalues of W_n at once for large n should give n samples from the Wigner semi-circle distribution. Hence, drawing the histogram of the eigenvalues of W_n should yield a semi-circle. We want to verify this fact and try to check that we indeed have convergence.

3.3.1 Matrices and histograms

We have seen matrices (a bit). Of course, we can create matrices made from samples and visualize them!

```
n = 50
A = matrix(rnorm(n^2), n, n)

View(A)
plot(A)
image(A)
image(A, col = heat.colors(100))
image(A, col = topo.colors(100))
image(A, col = colorRampPalette(c("white", "skyblue", "navy"))(100))
```

Getting the spectral decomposition of a matrix is easy; reading it is slightly harder, because it contains both the eigenvalues and the eigenvectors, possibly as complex numbers.

```
Ei = eigen(A)
View(Ei)
```

Using `Ei$values` yields the vector of eigenvalues. Note that running `eigen(A)` several times yields the same result, but running the line `A = matrix(rnorm(n^2), n, n)` changes all the subsequent results.

First, we need to create a random **symmetric** matrix. There are plenty of ways to do it; the simplest one is to compute $A + A^T$ where A is a random matrix (A^T is the transpose of A).

Q (math): If A is a random matrix whose entries are independent and normally distributed variables with a common mean and variance, justify that $A+A^T$ is a symmetric matrix whose entries are normally distributed with a common mean and variance.

Q (math): Show that this is not the case when replacing $A+A^T$ by $A \cdot A^T$, even though $A \cdot A^T$ is symmetric.

Now, we sample such a random symmetric matrix and visualize its eigenvalues. The argument `binwidth` indicates that the width of each column shall be 3: R will take care of starting and ending the bins at the correct spot.

```
n = 500
A = matrix(rnorm(n^2), n, n)
W = A + t(A)

Ei = eigen(W)$values

ggplot(data.frame(ei = Ei), aes(x = ei)) +
  geom_histogram(binwidth = 3, fill = "lightblue", color='black')

# alternatively without ggplot:
# hist(Ei, probability = TRUE)
# ?hist
```

3.3.2 Finding the growth of the radius

Q: Increase and decrease `n = 500` in the previous code, and observe that the limits of the x-axis change according to `n`.

This would be very annoying if we want to compare the distributions of the eigenvalues of W_n for different n (e.g. to understand the convergence phenomenon), and it will moreover start to generate very large values, which will both mess up our graphical visualization and impact our computational speed. Our first task is to find the growth rate of this x-range, that is to say, to determine $\max \mathrm{Spec}(W_n)$ as a function of n. To this end, we sample one matrix of size n for different n: we need to be clever here and not take all n from 1 to (say) 100 because it would be too costly and may not go high enough. Here, I chose to work with sizes $n = \lfloor 1.5^k \rfloor$ for $k \in \{4, 19\}$ because I want a geometric growth, but not too fast.

```
test_range = data.frame(n = as.integer(1.5^(4:19)))
for (i in 1:16) {
  n = test_range$n[i]
  A = matrix(rnorm(n^2), n, n)
  W = A + t(A)
  test_range$max_ei[i] = max(eigen(W)$values)
}

View(test_range)
```

The column `max_ei` is increasing with respect to n, as we feared, but how fast? To understand this, we can plot it, especially in a log-log scale.

Nota Bene: Note that we sampled one matrix per size n, so our experimental data might be very noisy. If we do not manage to see anything in the upcoming visualization, then we should sample several matrices for each size n and compute the mean of our observation for each n.

```
ggplot(test_range, aes(x = n, y = max_ei)) +
  geom_line() +
  theme_classic()

ggplot(test_range, aes(x = n, y = max_ei)) +
  geom_line() +
  scale_x_log10() +
  scale_y_log10() +
  theme_classic()
```

This seems to be a line of slope $\frac{1}{2}$. This means that, up to some constant c, we should have $\log(\max \mathrm{Spec}(W_n)) = \frac{1}{2}\log n + c$, or equivalently that $\max \mathrm{Spec}(W_n)$ is proportional to $\sqrt{n}$. Hence, dividing `W` by $\sqrt{n}$ should keep the eigenvalues in a bounded range. By convention, we take the normalization $\frac{1}{2\sqrt{2n}}W_n$, so that the eigenvalues lie (almost surely) between -1 and $+1$. Remember that this is a mathematical experiment, not a proof of anything: if you want a proof, use pen and paper instead!

Let us check our normalization.

```
test_range = data.frame(n = as.integer(1.5^(4:19)))
for (i in 1:16) {
  n = test_range$n[i]
  A = matrix(rnorm(n^2), n, n)
  W = (A + t(A)) / (2 * sqrt(2 * n))
  test_range$max_ei[i] = max(eigen(W)$values)
}

ggplot(test_range, aes(x = n, y = max_ei)) +
  geom_line() +
  theme_classic()

ggplot(test_range, aes(x = n, y = max_ei)) +
  geom_line() +
  scale_x_log10() +
  scale_y_log10() +
  theme_classic()
```

3.3.3 Implementing the Wigner semi-circle law

We have our histogram, properly scaled: we need to construct the distribution that we want to compare it with. To this end, we need to define a *function*. If you have seen functions in other languages (Python, Java, C, etc.), then you should not be surprised. Otherwise, this is not the place to detail the fundamental construction, so we will say very little about it.

A function takes some arguments and returns a value. In between, you can perform as many operations as you want; the content of these operations will not be accessible outside the function. A function (as well as a number, a graphic, a table, etc.) can be stored in a named variable. To declare the arguments `x`, `y`, and `z` of a function, we just write `function(x, y, z)`. If we want to indicate a default value for an argument, say we want the value of `z` to be 3 if nothing else is chosen by the user, we just write `z = 3` inside the function declaration. By default, R returns the last computed value inside a function, so you can write `return(y)` at the end or just `y`; it will have exactly the same effect.

That being said, we want to construct the density function and the distribution of the Wigner semi-circle distribution. As usual, we name them `dwigner` and `pwigner`. The mathematical formula for `dwigner` has been explained at the beginning of this section, and the formula for `pwigner` is obtained via a straightforward integration.

```
dwigner <- function(x, radius = 1){
  y = numeric(length(x))
  inside = abs(x) <= radius
  y[inside] = (2 / pi*radius^2) * sqrt(radius^2 - x[inside]^2)
  return(y)
}

pwigner <- function(x, radius = 1){
  y = numeric(length(x))
```

```
  inside = abs(x) <= radius
  y[x <= radius] = 0
  y[x >= radius] = 1
  z = x[inside] / radius
  y[inside] = 1/2 + (z*sqrt(1 - z^2) + asin(z))/pi
  y # return(...) is useless in R because it automatically returns the last computed value, here 'y'
}
```

Q: To check that everything works, we draw the density and the distribution. What properties are you expecting to see in these graphics?

```
ggplot(data.frame(x = c(-1.2,1.2)), aes(x = x)) +
  stat_function(fun = dwigner, linewidth = 0.5, color='blue') +
  stat_function(fun = pwigner, linewidth = 0.1, color='red') +
  labs(title = "Density and distribution of Wigner(0,1)", y = "Density, Distribution") +
  ylim(0, 1.1) +
  theme_classic()
```

3.3.4 Checking that we have the correct limit distribution

We draw the density of the Wigner semi-circle distribution together with our histogram of eigenvalues (using an aesthetic that specifies that the histogram should display densities rather than counts). Observe how well they coincide!

```
n = 500
A = matrix(rnorm(n^2), n, n)
W = (A + t(A)) / (2 * sqrt(2*n))
Ei = eigen(W)$values

ggplot(data.frame(ei = Ei), aes(x = ei)) +
  geom_histogram(aes(y = after_stat(density)), fill = "lightblue", color='black') +
  stat_function(fun = dwigner, linewidth = 0.5, color='blue') +
  labs(title = "Density of Wigner(0,1) versus histogram of sample", y = "Density, Distribution") +
  xlim(-1.1, 1.1) + ylim(0, 1) +
  theme_classic()
```

3.3.5 Estimating the speed of convergence (in distribution)

First and foremost, we can check whether the expectation and variance (of the eigenvalues of normalized symmetric random matrices) converge when $n \to +\infty$. The expectation of the Wigner semi-circle distribution is 0 (you can clearly see the symmetry in the graph), and the variance is $\frac{1}{4}$ (for radius 1), which can be computed directly via an integral. Let's check this on one example.

```
mean(Ei) # 0 ?
var(Ei) # 1/4 ?
```

Going further, we construct a table `test_convergence`, with the same sample of sizes as before (see the numerous comments in Section 3.3.2, notably we sample only one matrix per size, which is not ideal). Then we plot it and observe the speed of convergence.

```
test_convergence = data.frame(n = as.integer(1.5^(4:19)))
for (i in 1:16) {
  n = test_convergence$n[i]
  A = matrix(rnorm(n^2), n, n)
  W = ( A + t(A) ) / (2 * sqrt(2*n))
```

```
  Ei = eigen(W)$values
  test_convergence$mean[i] = mean(Ei)
  test_convergence$var[i] = var(Ei)
}
View(test_convergence)

ggplot(test_convergence, aes(x = n)) +
  geom_line(aes(y = mean, color = 'mean')) +
  geom_line(aes(y = var, color = 'variance')) +
  labs(title = "Convergence of mean and variance of eigenvalues", y = 'means and variances') +
  theme_classic()
```

Further, we can compute for each n the *Kolmogorov distance*, that is $\sup_t |F_n(t) - F(t)|$. The software R provides many ways of comparing samples and distributions, including the Kolmogorov distance. The standard approach is to compute the Kolmogorov–Smirnov test, named `ks.test`.

```
?ks.test
KS = ks.test(Ei, pwigner)
View(KS)
```

Some clarification of the results of `ks.test(sample, F)`, where the sample is interpreted empirically and `F` as a distribution function:

- `D` $= \sup_t |G(t) - F(t)|$, where $G(t)$ is the empirical cumulative distribution function of the sample
- `p-value` $= 1 -$ (answer to "If `Ei` were truly distributed according to `dwigner`, how surprising is the observed data?")
- `two-sided` means that R tests whether the empirical distribution deviates from `pwigner` in either direction
- `ks.test` uses an asymptotic approximation; it is not computing an exact `D`, but a very accurate estimate

We only need `D` here, so let's extract it and add a new column to our table `test_convergence`.

```
D = KS$statistic
D

test_convergence = data.frame(n = as.integer(1.5^(4:19)))
for (i in 1:16) {
  n = test_convergence$n[i]
  A = matrix(rnorm(n^2), n, n)
  W = ( A + t(A) ) / (2 * sqrt(2*n))
  Ei = eigen(W)$values
  test_convergence$D[i] = ks.test(Ei, pwigner)$statistic
}
View(test_convergence)

ggplot(test_convergence, aes(x = n, y = D)) +
  geom_line() +
  labs(title = "Convergence of the Kolmogorov distance") +
  theme_classic()

ggplot(test_convergence, aes(x = n, y = D)) +
  geom_line() +
  scale_x_log10() + scale_y_log10() +
  labs(title = "Convergence of the Kolmogorov distance in log−log") +
  theme_classic()
```

We can be satisfied: this decreases quite rapidly to 0! More precisely, it seems that D_n is roughly proportional to 1.75^{-n} (on my computer).

3.4 Testing a convergence in probability? a convergence almost sure?

That is an excellent project: you should definitely do it... on your own!

4 Course 4: χ^2 test, Student test, confidence intervals, etc.