Python for Econometrics and Operations Research

Sander Gribling Pieter Kleer Johan van Leeuwaarden Sven Polak

Table of contents

Chapter 1

Welcome

Welcome to the online "book" that serves as an introduction to the programming language Python, that you will see in various courses throughout the Econometrics and Operations Research (EOR) bachelor program.

The first part of this book is used for the "Python crash course" announced during the course Linear Algebra. The crash course consists of two (recommended) lectures that will form a useful basis for various programming assignments in the EOR bachelor program, including the assignment of the (mandatory) course Linear Optimization in the second quartile of the first year.

The two crash course lectures (taught only in English) are outlined below, including links to the lecture materials. To program in Python, we will use the Jupyter Notebook application in which we can execute Python code.

> **i** Overview
>
> **Lecture 1 (September 9, 12.45–14.30, Cube 242)** General introduction to Jupyter Notebook and *Chapter 3 - Python basics* (except Section 3.6).
> Right-click and use "Save link as…" for the materials below:
> Slides in PDF form.
> Exercises of Chapter 3 (see Section 2.2.2 on how to open this file in Jupyter Notebook after storing it on your computer).
> Solutions (also Jupyter Notebook file).
> **Lecture 2 (September 23, 14.45-16.30, Cube 241)** General introduction to packages, Numpy and Sympy, and *Chapter 4 - Linear Algebra*.
> Right-click and use "Save link as…" for the materials below:
> Slides in PDF form.
> Exercises of Chapter 5 (see Section 2.2.2 on how to open this file in Jupyter Notebook after storing it on your computer).
> Image.jpg (store this image in the same folder as the Jupyter Notebook)

> **i** Software requirements
>
> We will use the Jupyter Notebook application during our lectures, which is available on the university computers in the computer room where the lectures take place.

> If you want to use your private laptop, you can install Python and Jupyter Notebook by following the instructions in Chapter 2. Please do this **before** the first lecture.

Before we jump into coding with Python, we will start by discussing what programming is at the most basic level and motivating why we are learning how to code in Python in the first place.

1.1 What is a programming language?

Without getting into complicated details, a programming language is a way to communicate to a computer, via written text, tasks or operations that you want it to carry out. This is very different to how we often usually interact with a computer, which often involves pointing and clicking on different buttons and menus with your mouse.

In the EOR bachelor program, the goal is often to tell a computer to carry out complicated numerical computations or to visualize numerical data. To some extent, you have already done this in high school using a graphing calculator. In fact, everything that your graphing calculator can do, you can also do with Python, but the advantage of Python is that it can also handle much more complicated tasks.

To use Python in a correct fashion, it is important that you understand the grammar, i.e., "syntax", of the Python programming language. When humans speak to each other and someone makes a grammar mistake, it usually isn't a big deal. We usually know what they mean. But if you make a "syntax error", i.e, grammar mistake in a programming language, it won't understand what you mean. The computer will throw an error.

1.2 Why Python?

There are many different programming languages out there: C, C++, C#, Java, JavaScript, R, Julia, Stata, MATLAB, Fortran, Ruby, Perl, Rust, Go, Lua, Swift - the list goes on. So why should we learn Python over these other alternatives?

The best programming language depends on the task you want to accomplish. Are you building a website, writing computer software, creating a game, or analyzing mathematical data? While many languages could perform all of these tasks, some languages excel in some of them.

Python is by far the most popular programming language when it comes to "data science" tasks, that you will often encounter in the EOR bachelor program. It is also often used in web development, creating desktop applications and games, and for scientific computations. It is therefore a very versatile programming language that can complete a very wide range of tasks.

Python is also completely free and open source and can run on all common operating systems. This means you can share your code with anyone and they will be able to run it, no matter what computer they are on or where they are in the world.

There is also a very large active community that creates packages to do a wide-range of operations, keeping Python up to date with the latest developments. For example, excellent community help is available at Stackoverflow, so if you Google how to do something in Python most likely that question has already been answered on Stackoverflow. Funnily enough, a key skill to develop with programming is how to formulate your question into Google to land on the right Stackoverflow page.

More recently, "large-language" models like ChatGPT have become a very useful resource for Python. ChatGPT can write excellent Python code and also explains all the steps it takes, so we encourage you to use

it as a tool to help you when you are stuck.

You should keep in mind though, that throughout the bachelor program, you will not always be allowed to use tools like ChatGPT. It is also important that you understand basic programming concepts to catch errors that AI might introduce (do not forget that LLMs are merely predicting text and not “consciously” writing a script), or to help improve your AI-prompt writing skills.

These days employers are increasingly looking to hire people with programming skills. Knowing how to program in Python - one of the most commonly used languages by companies - is therefore a very valuable addition to your CV.

Chapter 2

Software

In this chapter we will learn how to install Python and run our very first command.

2.1 Installing Anaconda

The easiest way to install Python and Jupyter Notebook is by installing Anaconda, which is a software package that includes Python, Jupyter Notebook, and other software applications. You can install Anaconda by visiting https://www.anaconda.com/download.

You should see this page:

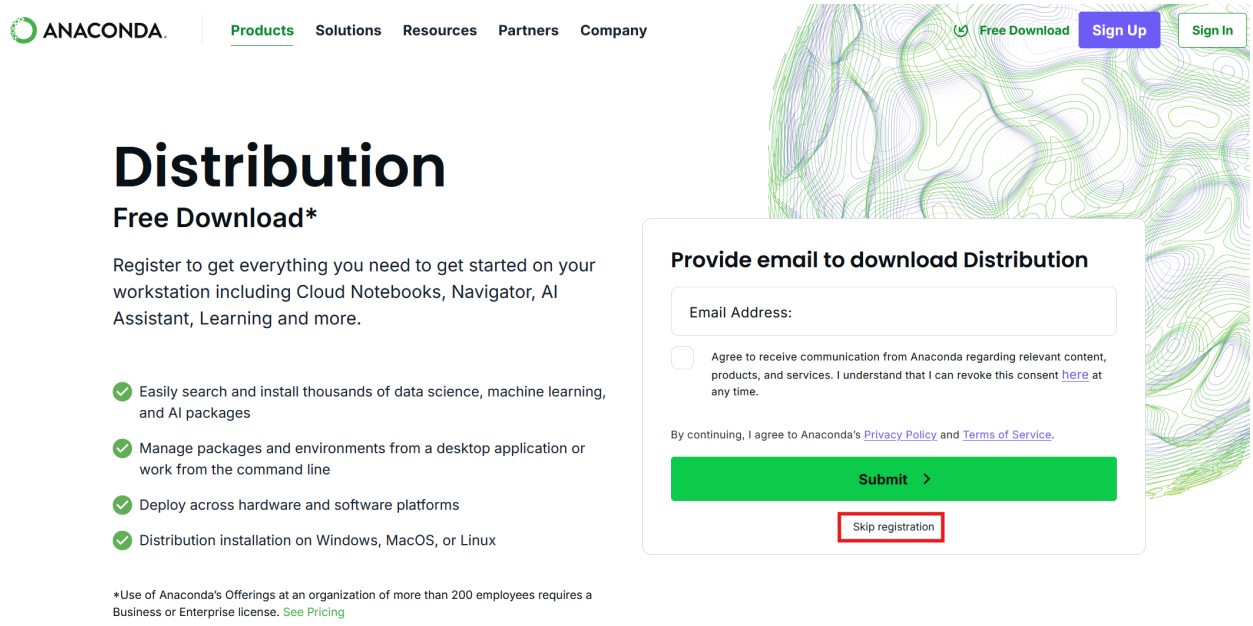


Figure 2.1: Anaconda Download Page

You should click the "Skip registration" button (although feel free to register if you like). You will then see the following page:

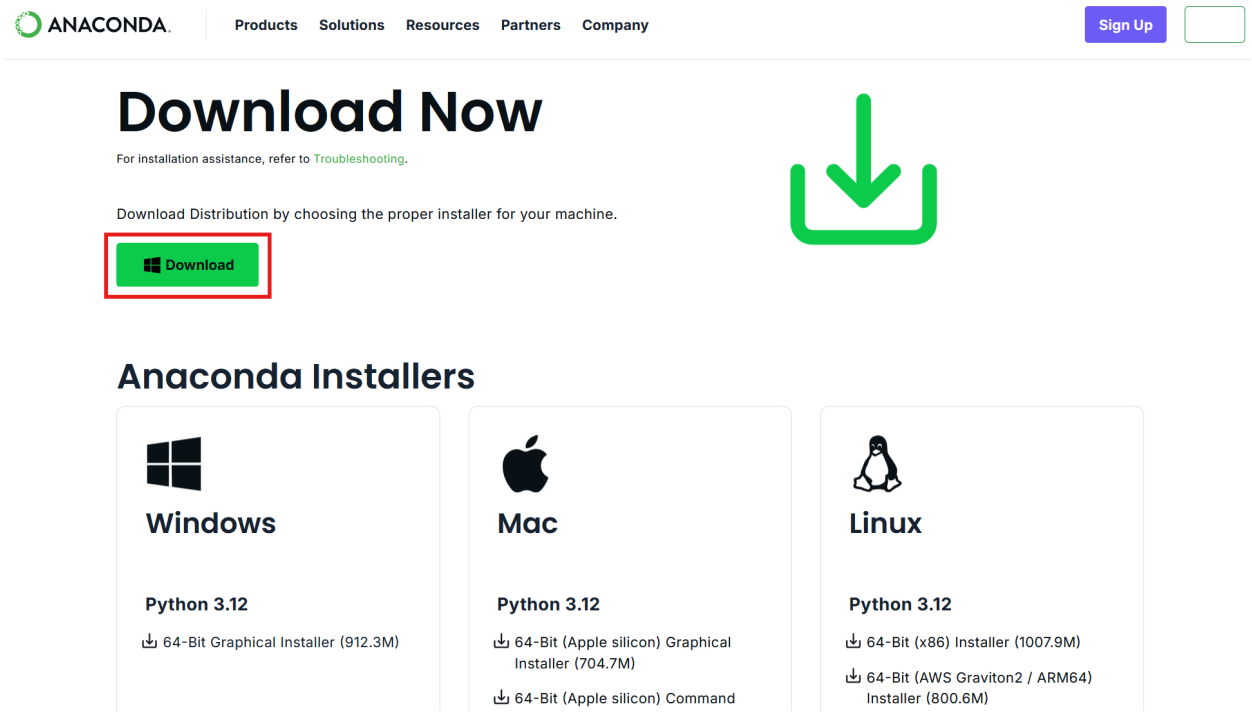


Figure 2.2: Anaconda Download Page

You should then click on the "Download" button. Mac users will see a Mac logo instead.

After downloading the file, click on it to install it. Follow the installation wizard and keep all the default options during installation.

After installation you will see a number of new applications on your computer. You can see all applications that were installed using *Anaconda Navigator*. You can open the navigator by searching for it in the Start menu (on Windows).

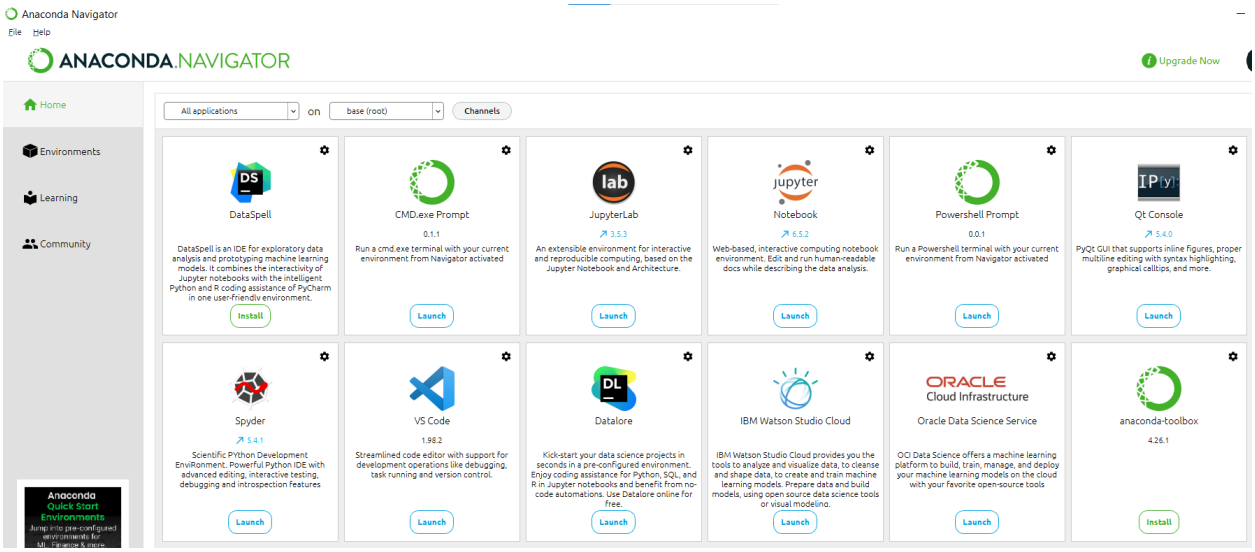


Figure 2.3: Anaconda Navigator

We highlight two applications:

- *Jupyter Notebook*. This is a web application that allows you to write a notebook (like a report) with text and Python code snippets with output. We will learn how to use this application later on.
- *Spyder/VS Code.* These are computer applications that allows you to write Python scripts and execute them to see the output. Such an application is called an Integrated Desktop Environment (IDE).

2.2 Jupyter Notebook

You can open the Jupyter Notebook application either by pressing 'Launch' in the Anaconda Navigator, or you can search for the application directly on your (university) computer via the Start menu.

The application will open as a tab in a web browser, and you should then see a list of folders.

Figure 2.4: Jupyter Notebook application

2.2.1 Creating new notebook

You can navigate to a folder and then create a new notebook by clicking on 'New' in the top-right and then selecting 'Python 3 (ipykernel)' under Notebook.

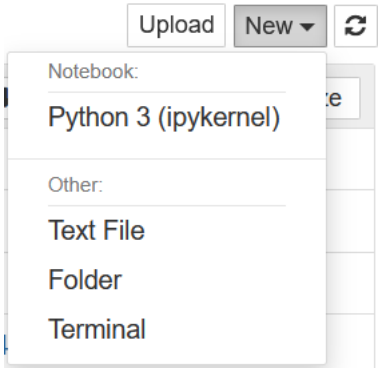


Figure 2.5: Creating notebook

The new notebook will open in another tab and is stored in the folder in which you created it, typically under the name 'Untitled.ipynb'. You can change the name of the file either in the folder in which you stored it, or via `File -> Rename` in the top-left corner. You should see the empty notebook as below.

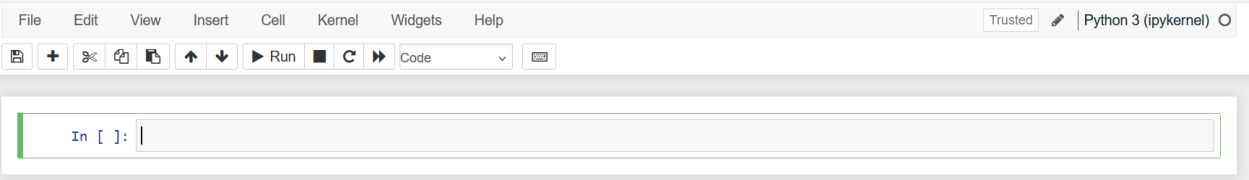


Figure 2.6: New notebook

In the bar you can type Python code. Let us execute our first code, which is a simple calculation $1 + 1$. To find $1 + 1$ in Python, we can use the command `1+1`, similar to how we would do it in Excel or in the Google search engine. Let's try this out. Type 1+1 in the code bar and click ▶ Run (or hit Shift + Enter on your keyboard). We will see the output 2 on the next line next to a red Out [1]:

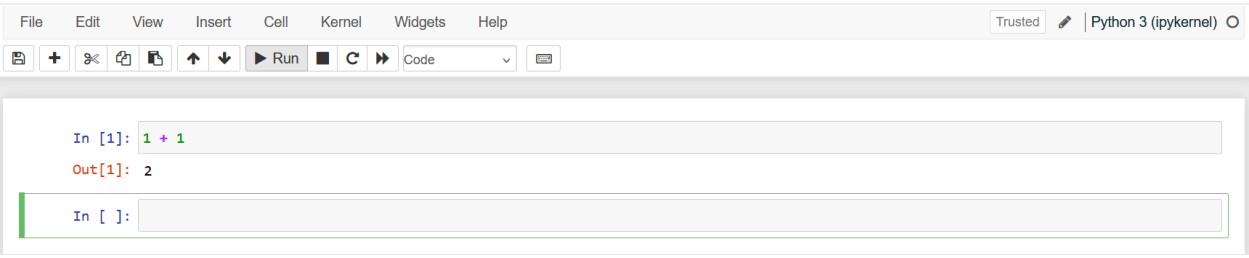


Figure 2.7: First Python code

The red Out [1] means this is the output from the code snippet In [1] executed in this notebook. If we continue typing code in the second bar, it will be called In [2] and its output Out [2]. However, the same happens if we would re-run the first code snippet with `1+1` : It will also be called In [2] and its output Out [2]. The index keeps track of how many code snippets have been executed in the notebook.

2.2.2 Opening existing notebook

The extension of a Jupyter Notebook file is `.ipynb` and sometimes denoted as Jupyter Source File. If you have stored such a file on your computer, you can open in it Jupyter Notebook as follows:

- Open Jupyter Notebook (see above) and navigate to the folder where you stored the file.
- Click on the .ipynb file and then it should open in a new tab of the browser.

> ⚠ WARNING about files on university computer
>
> If you store a file on a (TiU) university computer, for example in the Downloads folder, it will typically be deleted when you log out. To avoid this, either:
> - Copy the file onto a USB drive.
> - E-mail the file to yourself.
> - Store it on the M: drive (that is denoted by the drive that has your name) of the university computer, whose files are not deleted.
>
> Next time you want to use a file, put it again in the Downloads folder to work on it (and make sure to back it up properly again afterwards).

If you want to open a .ipynb file that is located in the M: drive of a university computer, you need to open

Jupyter Notebook differently than explained above (because you cannot navigate to the M: drive if you open Jupyter Notebook via the Start menu). Instead, do the following:

1. Close all instances of Jupyter Notebook if any are running.
2. Look up the *Anaconda Prompt* application in the Start menu and Run it.
3. Type `jupyter notebook --notebook-dir=M:` and press Enter
4. Jupyter Notebook should now open in the web browser showing the files and folders of the M: drive
5. Open the desired .ipynb file to work on it.

2.3 Code snippets in this book

In this book, we won't always show screenshots like we did above. Instead we will show code snippets in boxes like this:

```
1 + 1
```

```
2
```

The part that is code will be in color and there will be a small clipboard icon on the right which you can use to copy the code to paste into your own Notebook to be able to experiment with it yourself. The output from the code will always be in a separate gray box below it (without a clipboard icon).

Chapter 3

Python basics

In this chapter we will learn how to use Python as a calculator and see some basic programming concepts.

3.1 Arithmetic operations

We start with the most basic arithmetic operations: Addition, subtraction, multiplication and division are given by the standard `+`, `-`, `*` and `/` operators that you would use in other programs like Excel. For example, addition:

```
2 + 3
```

```
5
```

Subtraction:

```
5 - 3
```

```
2
```

Multiplication:

```
2 * 3
```

```
6
```

Division:

```
3 / 2
```

```
1.5
```

It is also possible to do multiple operations at the same time using parentheses. For example, suppose we wanted to calculate:

$$\frac{2+4}{4 \cdot 2} = \frac{6}{8} = 0.75$$

We can calculate this in Python as follows:

```
(2 + 4) / (4 * 2)
```

```
0.75
```

With the `**` operator (two stars) we can raise a number to the power of another number. For example, $2^3 = 2 \times 2 \times 2 = 8$ can be computed as

```
2 ** 3
```

```
8
```

> ⚠ WARNING
>
> Do **not** use `^` for exponentiation. This actually does a very different thing in Python.

> Exercise 3.1
>
> Compute the following expressions using the operator $+, -, *, /$ and $**$:
>
> i) $3 + 5 \cdot 2$
>
> ii) $\frac{(10-4)^2}{3}$
>
> iii) $\frac{((2+3)\cdot 4-5)^2}{3+1}$

3.2 Variables

In Python we can assign single numbers to *variables* and then work with and manipulate those variables.

Assigning a single number to a variable is very straightforward. We put the name we want to give to the variable on the left, then use the `=` symbol as the *assignment operator*, and put the number to the right of the `=`. The `=` operator binds a number (on the right-hand size of `=`) to a name (on the left-hand side of `=`).

To see this at work, let's set $x = 2$ and $y = 3$ and calculate $x + y$:

```
x = 2
y = 3
x + y
```

```
5
```

When we assign $x = 2$, in our code, the number is not fixed forever. We can assign a new number to `x`. For example, we can assign the number 6 to `x` instead. The sum of x (which is 6) and y (which is 3), is now 9:

```
x = 6
x + y
```

```
9
```

Finally, you cannot set $x = 2$ with the command `2 = x`. That will result in an error. The name must be on the left of `=` and the number must be on the right of `=`.

Exercise 3.2

Define variables a, b, c with numbers $19, 3$ and 7, respectively. Compute the following expressions:

i) $a + b \cdot c$

ii) $\frac{(a-c)^2}{b}$

iii) $\frac{((b+c)\cdot a - c^2)^2}{a+b}$

If you want to print multiple expressions within the same code snippet, you can use the `print()` function of Python for each of the expressions.

```
x = 2
y = 3
print(x + y)
print(x - y)
```

```
5
-1
```

3.3 Lists

We can also store multiple variables in one object, a so-called *list*. A list with numbers is created by writing down a sequence of numbers, separated by commas, in between two brackets `[` and `]`.

```
z = [3, 9, 1, 7]
z
```

```
[3, 9, 1, 7]
```

We can also create lists with fractional numbers.

```
z = [3.1, 9, 1.9, 7]
z
```

```
[3.1, 9, 1.9, 7]
```

To access the numbers in the list, we can *index* the list at the position of interest. If we want to get the number at position i in the list, we use the syntax `z[i]`.

```
z[1]
```

```
9
```

Something strange is happening here… The left-most number in the list is 3.1, but `z[1]` returns 9. This happens because Python actually starts counting at index 0 (instead of 1).

Indexing convention in Python

The *left-most number* in a Python list is located at *position* 0. The number next to that at position 1, etc. That is, the i-th number in a list with n numbers can be found at position $i-1$ for $i = 1, \dots, n$

In other words, the "first" number in the list is located at position 0, and we can access it using `z[0]` instead.

Below we index the number of the list at positions $i \in \{0, 1, 2, 3\}$ separately.

```
z[0]
```

```
3.1
```

```
z[1]
```

```
9
```

```
z[2]
```

```
1.9
```

```
z[3]
```

```
7
```

Exercise 3.3

Consider the list $a = [11, 41, 12, 35, 6, 33, 7]$.

i) Compute the sum of the numbers at even positions in a (i.e., positions $0, 2, 4$, and 6).

ii) Compute the result of multiplying the first and last elements of a, then subtracting the middle element.

iii) Compute the square of the element at position 2, divided by the sum of the elements at the odd positions.

3.4 For-loop

Suppose we want to compute the sum of the numbers in the list a of Exercise 3.3. We could do this manually by indexing every number and adding them one by one.

```
a = [1, 4, 2, 5, 6, 3, 7]

a[0] + a[1] + a[2] + a[3] + a[4] + a[5] + a[6]
```

```
28
```

In fact, we can also define a new variable to store this number in. Let us call this variable `total_sum`.

```
a = [1, 4, 2, 5, 6, 3, 7]

total_sum = a[0] + a[1] + a[2] + a[3] + a[4] + a[5] + a[6]
total_sum
```

28

If the list is a is very long, for example containing thousands of elements, then it becomes very tedious to compute the total sum with the approach above. Such long lists are not uncommon in real-life data.

A much better way is to use a *for-loop*, which lets us go through each element in the list one at a time. Here's how we could compute the sum of the numbers in the list a using a for-loop:

```
a = [1, 4, 2, 5, 6, 3, 7]
total_sum = 0

for i in [0,1,2,3,4,5,6]:
    total_sum = total_sum + a[i]

print(total_sum)
```

28

Let's break down what is happening here.

- We first define the list $a = [1, 4, 2, 5, 6, 3, 7]$.
- We define the variable `total_sum` with initial value 0. This variable will be the *running total* of the numbers in a that we are adding up. After the full code has been executed, this variable will contain the sum of all numbers in a, and its value is printed at the end using `print(total_sum)`.
- The line `for i in [0,1,2,3,4,5,6]:` indicates that we want to carry out a piece of code multiple times with different values for the variable `i`.
  - The words `for` and `in` are *Python keywords*, meaning they have a very specific purpose in Python. They get a different color when you type them in a code block.
  - Note that the line should end with a colon `:`.
- The piece of code that has to be run multiple times for $i \in \{0, 1, 2, 3, 4, 5, 6\}$ is `total_sum = total_sum + a[i]`.
  - For Python to understand that `total_sum = total_sum + a[i]` has to be executed with different values for i, we indent this line (using Tab on the keyboard).
  - We *overwrite* the value of `total_sum` with its current value plus the number at position i in a, i.e., the number `a[i]`. In the table below this process is illustrated for all the values of i.

`i`	`a[i]`	`total_sum` after this iteration
0	1	$0 + 1 = 1$
1	4	$1 + 4 = 5$
2	2	$5 + 2 = 7$
3	5	$7 + 5 = 12$
4	6	$12 + 6 = 18$
5	3	$18 + 3 = 21$
6	7	$21 + 7 = 28$

`i`	`a[i]`	`total_sum` after this iteration

In the first *iteration* of the for-loop ($i = 0$), we have the initial number 0 for `total_sum` so adding `a[0]` results in a new value of `total_sum` being $0 + 1 = 1$.

In the second iteration, with now `total_sum` equal to 1, we add the number $a[1]$, which results in the new value of `total_sum` being 1 (current number of `total_sum`) plus `a[1]` (which is 4), resulting in a new running total of $1 + 4 = 5$.

If we would be interested in only computing, e.g., the sum of the first three numbers in a, we could replace the index list $[0, 1, 2, 3, 4, 5, 6]$ by $[0, 1, 2]$.

```
a = [1, 4, 2, 5, 6, 3, 7]
total_sum = 0

for i in [0,1,2]:
    total_sum = total_sum + a[i]

total_sum
```

```
7
```

Exercise 3.4

Create the list $a = [1, 4, 2, 5, 6, 3, 7]$.

i) Compute the sum of the numbers at the even indices using a for-loop.

ii) Compute the product of the numbers in a using a for-loop.

If you want to execute more lines of code in every iteration of the for-loop, you should indent all of them. In the code below we compute the running total `total_sum` and also use the `print()` command of Python to print the value of the running total after every addition. This results in all the values in the right column of the above table being printed.

```
a = [1, 4, 2, 5, 6, 99, 3]
total_sum = 0

for i in [0,1,2,3,4,5,6]:
    total_sum = total_sum + a[i]
    print(total_sum)
```

```
1
5
7
12
18
117
120
```

One final note to make is that this approach might still require a lot of typing if the list a contains many values. For example, if we are given a list of a thousand values, we would have to type a list with values 0 through 999 in the for-loop above.

There is a way to do this quicker, by using the `range()` function in Python. If we instead use the line `for i in range(7):`, then Python executes the indented code below for the values $i = 0, 1, 2, 3, 4, 5, 6$. Note that $i = 7$ is not included!

In general using `for i in range(n):` executes the indented lines below for the (in total n) values $i = 0, 1, 2, \ldots, n - 1$.

```
a = [1, 4, 2, 5, 6, 99, 3]
total_sum = 0

for i in range(7):
    total_sum = total_sum + a[i]
    print(total_sum)
```

```
1
5
7
12
18
117
120
```

3.5 Conditional statements

In many programming situations, we want the computer to make decisions based on certain conditions. For example, if a number is negative, we might want to handle it differently than if it were positive. In Python, we can do this using *conditional statements*, also known as *if/else statements*.

Let's look at a basic example. We first make a general remark about printing text in Python.

Printing text

If you want to print text in Python, you should put it in between quotation marks.

```
print("Hello world")
```

```
Hello world
```

If you want to print both text and variables, you can do that in the same `print()` command separating them with a comma.

```
x = 3
print("The value of x is", x)
```

```
The value of x is 3
```

> Note that in `"The value of x is"` part, the x-symbol is interpreted merely as a letter, not a variable.

Now let us look at an example.

```
x = 5

if x > 0:
    print("x is positive")
else:
    print("x is not positive")
```

```
x is positive
```

Here is what this code does:

- `x = 5` assigns the number 5 to the variable x.
- `if x > 0:` checks whether x is greater than zero, i.e, Python checks whether the condition `x > 0` is true or false. If the condition is true, it executes the indented code below, which is a print-statement in this case. Python then no longer checks the `else:` statement.
- If the condition is false (i.e., $x \leq 0$), then Python executes the indented code under `else:`, which is a different print-statement in this case.

We can also have multiple conditions using `elif`, which stands for "else if". Below we add the third statement that checks if x is precisely equal to zero. Also here, as soon as Python reaches a statement that is true, it does not check the remaining statements anymore.

```
x = 0

if x > 0:
    print("x is positive")
elif x == 0:
    print("x is zero")
else:
    print("x is negative")
```

```
x is zero
```

In the code above, we use the syntax `x == 0` to define the statement that checks whether x is precisely equal to 0. You should not use `x = 0` (otherwise Python would confuse this with assigning a value of 0 to the variable x, which is not what we want).

This checks the conditions one by one from top to bottom and executes the first indented code block where the condition is true. If you want more than three conditions, you should start with an `if` statement, then `elif` statements, and finish with an `else`.

Finally, if you want to execute multiple lines of code for one or more of the conditions, you should indent all those lines under the respective conditions.

```python
x = 1

if x > 0:
    print("x is positive")
    print(x)
elif x == 0:
    print("x is zero")
else:
    print("x is negative")
    print(x)
```

```
x is positive
1
```

Exercise 3.5

Create the list $a = [1, 4, -4, 0, 5, -3, -7]$ in Python.
Use a for-loop in combination with the code above to check for every number in a whether it is positive, zero, or negative. If a number is positive you should print the message `"The number is positive"`, if it is zero `"The number is zero"` and if it is negative `"The number is negative"`.
The output of your piece of code should be as follows.

```
The number is positive
The number is positive
The number is negative
The number is zero
The number is positive
The number is negative
The number is negative
```

Let us now look at an example from mathematics. Suppose we want to compute the roots x of a quadratic equation of the form:

$$ax^2 + bx + c = 0$$

Here a, b and c are known given numbers, and the goal is to find one or more x's that satisfy the above equation.

The solution(s) to this problam are given by the quadratic formula (Dutch: abc-formule):

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Here $\pm$ means that one solution is given by choosing a plus symbol in the place of $\pm$, and the other solution by choosing the minus symbol.

The expression under the square root, called the *discriminant* $D = b^2 - 4ac$, determines how many (real) roots exist:

- If $D > 0$, the equation has two real roots.
- If $D = 0$, the equation has exactly one real root.
- If $D < 0$, there are no (real) roots.

Exercise 3.6

Create variables $a = 3, b = 2$ and $c = -1$. Create a variable `D` for the discriminant (in terms of a, b and c).

i) Use conditional statements to determine how many roots the quadratic formula $ax^2 + bx + c$ has, based on the three possibilities for the discriminant. For each possibility, print an appropriate message in the indented code block. For the chosen a, b and c, the function has two roots (so this case should be printed in your code).

ii) Use conditional statements to print the roots x of the quadratic formula $ax^2 + bx + c$, based on the three possibilities for the discriminant (in the third case, do not print the roots, but a message saying there are no roots). *Hint: If you want to print two variables* `y` *and* `z` *you can use* `print(y,z)` *or use* `print(y)` *and* `print(z)` *on different indented lines.* For the chosen a, b and c, your output should show the roots -1 and 0.333 (possibly with more or less decimals).

You can play around with your code by choosing different numbers for a, b and c, and see if you get different output cases for both questions above.

Exercise 3.7

Create the lists $a = [3, 7, 1, 4], b = [2, 7, 4, 4]$ and $c = [11, 3, 0, 1]$. Write a for-loop that executes your code of Exercise 3.6(ii) for every combination (a_i, b_i, c_i) where a_i, b_i, c_i are the numbers at position i in the lists a, b, c, respectively, for $i = 0, 1, 2, 3$.

Your output should look like this.

```
The formula has no real roots
The formula has no real roots
-4.0 0.0
-0.5
```

Exercise 3.8 (bonus)

Suppose we are given a list $g = [9.1, 1.3, 5.4, 5.6, 5.74, 6.74, 8.25, 9.2, 7.1, 6.9]$ of student grades.

i) Write a Python code that rounds every grade to the nearest half integer, i.e., to the value in the set $\{0, 0.5, 1, 1.5, 2, 2.5, \dots, 8, 8.5, 9, 9.5, 10\}$ it is closest to, and print this value. *Hint: You can round a number to its closest integer by using the* `round()` *function. For example* `round(5.3)` *gives* 5*, and* `round(5.9)` *gives* 6*. Think of a way to use the* `round()` *function to round to half integers.*

On g as given, the output should be as follows.

```
9.0
1.5
5.5
5.5
5.5
6.5
8.0
9.0
7.0
```

```
7.0
```

ii) Adjust your code so that grades that lie in the interval $(5, 6)$ are rounded either up to 6 or down to 5 depending which of the two a number is closest to (in other words, rounding to 5.5 is no longer allowed). *Hint: You can use the* `and` *keyword to check multiple conditions in an if-statement.*

On g as given, the output should be as follows.

```
9.0
1.5
5
6
6
6.5
8.0
9.0
7.0
7.0
```

The latter procedure is in fact how your grades in the EOR bachelor program are rounded.

3.6 Math basics

In this chapter we will see some of the basic math functionality that Python has to offer. Many of these tasks can be carried out by your graphing calculator as well, but Python can also handle much more difficult problems that you will see in the course of you academic career.

We start with the basics of defining a function, such as a quadratic formula.

3.6.1 Python function

If we want to compute a certain mathematical expression for many different variables, it is often convenient to use a Python function for this.

For example, consider the quadratic function $f(x) = x^2 + 2x^2 - 1$. Say we want to know the values of $f(-3), f(-2.5), f(1)$ and $f(4)$. What we would like to do is to 'automate' the computation of a function value, so that we do not have to write out the whole function everytime.

For this we can use a Python function for this as follows.

```
def f(x):
    return x**2 + 2*x - 1
```

What does the code above do? First of all the syntax to tell Python we want to define a function called `f` that takes as input a number `x` is `def f(x):`.

We next have to tell Python what the function is supposed to compute. On the second line, with one tab indented, we have the `return` statement. Here we write down the expression that the function should return (or compute), which in our case is the function value $f(x) = x^2 + 2x^2 - 1$.

We can now compute the function value $f(x)$ for any value of x. What happens is that Python *calls* the function f with input the chosen value of x, and then returns the function value $f(x)$, i.e., the expression in the `return` statement.

```
f(-2)
```

```
-1
```

```
f(1)
```

```
2
```

Note that you can also name the function differently, for example we could also have done `def quadratic_function(x):`. You should then use the name `quadratic_function` too in the command well you call the Python function to compute the function value $f(x)$.

```
def quadratic_function(x):
    return x**2 + 2*x - 1

quadratic_function(1)
```

```
2
```

Just as your graphing calculator we can plot a Python function, search for its roots, integrate a certain area under the curve and much more! More advanced tasks that Python can handle will introduced in later courses in the EOR bachelor program.

If you want to get a better understanding of the codes in the coming sections, you could already have a look at Chapter 9 of this course document of another course taught at the Tilburg School of Economics and Management. We do not explain the code here, but give it as a teaser what more is possible with Python!

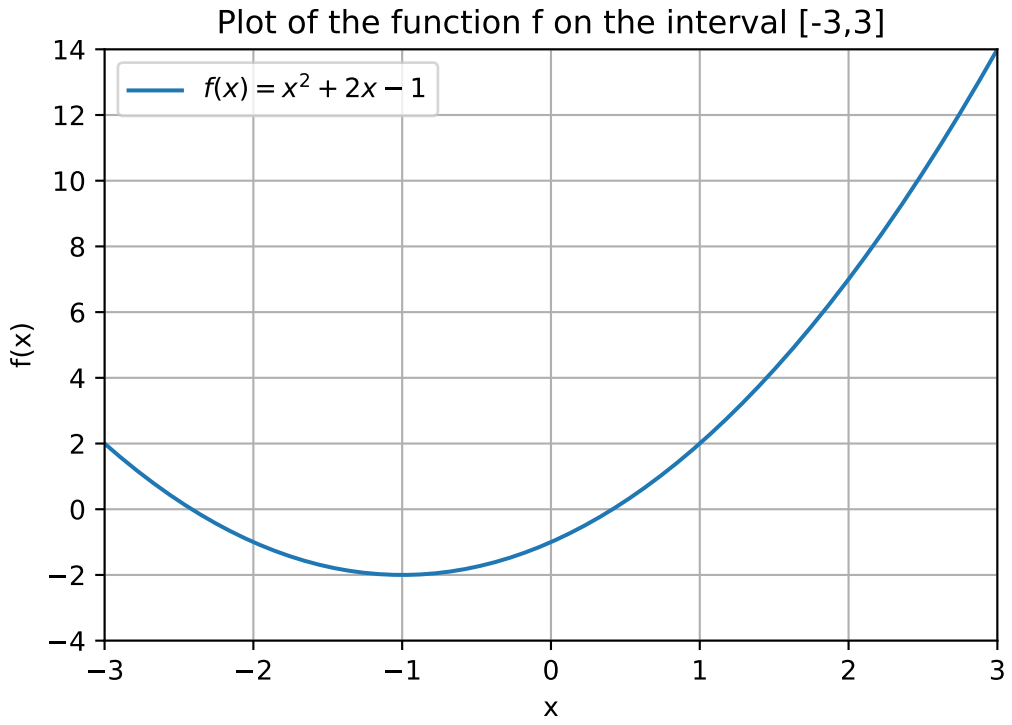
3.6.2 Plotting

Consider again the function $f(x) = x^2 + 2x - 1$. A visualization of this function is given below. If you want to plot a function in Python you have to make use of functionality from NumPy and Matplotlib which are so-called Python packages.

Packages are functions written by other people to make our live easy, i.e., so that we do not have to write every code file from scratch in Python.

```
import numpy as np
import matplotlib.pyplot as plt

# Define the x range
x = np.linspace(-3, 3, 600)

# Define the function f
def f(x):
    return x**2 + 2*x -1

# Create the plot
plt.figure(figsize=(6, 4))
plt.plot(x, f(x), label='$f(x) = x^2 + 2x - 1$')
```

```
# Add labels and title
plt.title('Plot of the function f on the interval [-3,3]')
plt.xlabel('x')
plt.ylabel('f(x)')

# Add a grid
plt.grid(True)

# Set range
plt.xlim(-3,3)
plt.ylim(-4,14)

# Add a legend
plt.legend()

# Show the plot
plt.show()
```



3.6.3 Root finding

Similarly, the SciPy package can be used to carry out various mathematical tasks and algorithms, making it very important for data analysis purposes.

The code below uses a pre-written Python function called `fsolve()` from SciPy to compute the roots of a function f. In other words, `fsolve()` is a mathematical algorithm for finding the root of a function, such as

Newton's method, that someone implemented in Python and made available publicly for the whole world to use. If you are interested in the *source code* of this function, you can look it up in the *documentation* of Python (more specifically, SciPy in this case).

```
import scipy.optimize as optimize

def f(x):
    return x**2 + 2*x - 1

guess = 3
f_zero = optimize.fsolve(f,guess)

print("A root of the function f is given by", f_zero)
```

```
A root of the function f is given by [0.41421356]
```

The function `fsolve()` takes two inputs: a function of which we want to find a root, and an initial guess (3 in our case) of where the root is.

Entering an initial guess for where the root is located, is in some sense the equivalent of giving a bracket in which the root should lie on your graphing calculator. In fact, there are other root finding functions available in Python that work in this way, i.e., that require you to give an initial bracket, just as you do on your graphing calculator.

Different initial guesses might lead to different roots found by Python. In fact, as you can see the function f has two roots of which the above code finds the right one. We could find the left root by filling in a different initial guess, e.g., -3 instead of 3.

```
guess = -3
f_zero = optimize.fsolve(f,guess)

print("A root of the function f is given by", f_zero)
```

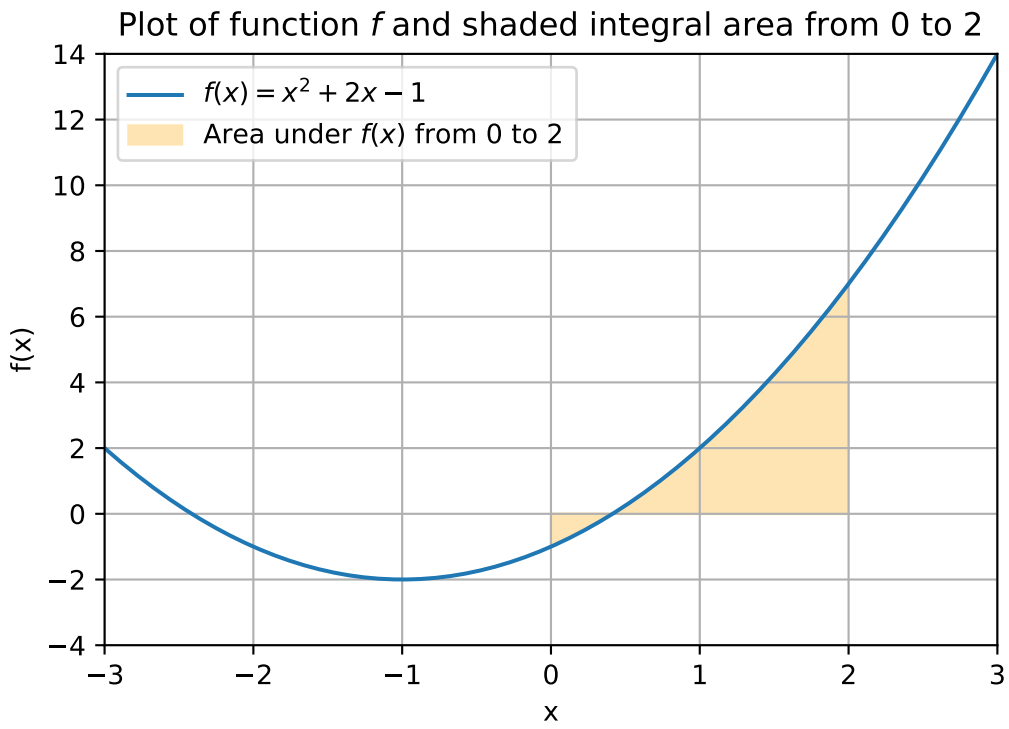
```
A root of the function f is given by [-2.41421356]
```

3.6.4 Integration

Finally, it is also possible to use built-in functionality from SciPy to integrate a function. Below we integrate the function f from 0 to 2. This integral area is illustrated in the figure below.

```
import numpy as np
import matplotlib.pyplot as plt

# Define the x range for the full plot
x = np.linspace(-3, 3, 600)

# Define the function f
def f(x):
    return x**2 + 2*x - 1
```

```
# Create the plot
plt.figure(figsize=(6, 4))
plt.plot(x, f(x), label='$f(x) = x^2 + 2x - 1$')

# Define the interval for shading (0 to 2)
x_fill = np.linspace(0, 2, 300)
plt.fill_between(x_fill, f(x_fill), alpha=0.3, color='orange',
    label='Area under $f(x)$ from 0 to 2')

# Add labels and title
plt.title('Plot of function $f$ and shaded integral area from 0 to 2')
plt.xlabel('x')
plt.ylabel('f(x)')

# Add a grid
plt.grid(True)

# Set axis limits
plt.xlim(-3, 3)
plt.ylim(-4, 14)

# Add a legend
plt.legend()

# Show the plot
plt.show()
```




```python
from scipy.integrate import quad

# Define the function to integrate
def f(x):
    return x**2 + 2*x - 1

# Perform the integration from 0 to 2
result, error = quad(f, 0, 2)

# Print the result
print("Integral of f(x) from 0 to 2 is:", result)
print("Numerical error in integral computation is at most", error)
```

```
Integral of f(x) from 0 to 2 is: 4.666666666666666
Numerical error in integral computation is at most 5.666271351443603e-14
```

3.6.5 Why Python and not my calculator?

So far we have illustrated task with Python that you graphing calculator can also carry out. The advantage of Python is that it can handle much more complicated computing tasks and handle much more difficult mathematical functions, that your graphing calculator is not able to handle.

Many of these tasks you will come across in various courses of the EOR bachelor program, already starting with the course Linear Optimization in the second quartile of year 1.

Furthermore, throught the EOR bachelor program you will also see some other programming languages such as R and Matlab. Many of the general programming ideas, such as for-loops and conditional statements, exist

in those languages as well, but sometimes the syntax (i.e., the grammar of the programming language) is different than that of Python.

Chapter 4

Linear algebra

In this chapter we will learn how to use Python for linear algebra.

4.1 Packages

In the first session of this crash course you have learned the basics of Python: how to use Python as a calculator, use lists, for-loops, and if-else-statements.

Python can be used for much more advanced operations as well. As you can imagine, code quickly grows more complicated. Fortunately, we can reuse code written by other through so-called **packages** (or libraries). In the optional Section 3.6 you might have seen several packages already.

Using packages has several advantages. If you use a standard package, then it makes your code more readable. It also reduces the risk of errors: Python packages are typically developed by expert programmers and thoroughly tested before they are released to the public. For the mainstream packages that we will be using, you can thus be reasonably confident that they deliver what they promise.

Another major advantage of packages is that they are often heavily optimized in terms of speed and memory efficiency.[1]

Today we will use several well-known packages:

- NumPy,
- Sympy,
- and Matplotlib.

In a nutshell: packages are functions written by other people to make our life easy, i.e., so that we do not have to write every code file from scratch in Python.

The NumPy, SymPy and Matplotlib packages should be installed in a standard Anaconda installation. If you have another Python installation, typically using the PIP package manager should allow you to install packages. In this case, commands like `pip install numpy` *and* `pip install sympy` *should do the job.*

[1] Under the hood, numpy for instance relies on BLAS and LAPACK for most of its linear algebraic subroutines. BLAS and LAPACK are written in a programming language called Fortran, see for example the LAPACK documentation. LAPACK is used by many other programming languages, including Matlab.

4.2 Why `numpy` and `sympy`?

The package `numpy` is designed for fast numerical calculations, e.g., with matrices. We can compute matrix products, solve linear systems, and compute inverses with `numpy` very quicky. The package `sympy`, on the other hand, is a symbolic mathematics library. It can do exact algebraic manipulations (also with variables x and y), but it is less quick than numerical computations with `numpy`. `sympy` also contains a method to do exact row-reduction, to bring a matrix into reduced echelon form. This may also be useful if you did row-reduction by hand and want to verify your result with the computer.

4.3 Basic matrix and vector operations with Numpy

Suppose we have the following vectors $\mathbf{x}$ and $\mathbf{y}$, and matrix A:

$$\mathbf{x} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \mathbf{y} = \begin{bmatrix} 4 \\ 5 \\ 7 \end{bmatrix}, A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

We can define vectors in Python as so-called NumPy arrays. Think of these as lists (that we saw earlier) on which we can perform numerical computations. To create such arrays, we have to import the NumPy package `numpy`. We do this under the *alias* `np` (so that we can use the short-hand notation `np` for `numpy` everywhere).

```
import numpy as np

# Define the vectors and matrix
x = np.array([1, 2, 3])
y = np.array([4, 5, 7])
A = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])
```

In Python, we can add vectors, compute the matrix-vector product and multiply matrices. Note: To compute a matrix-vector product you use the command `A@x`. What happens if you use the command `A*x`? Now, try to compute $A\mathbf{x}$, $A\mathbf{y}$, and $A(\mathbf{x}+\mathbf{y})$ with Python. What do you notice?

```
# your code here
```

4.4 Application: input-output models

(From Python Linear Algebra Notebook by Herbert Hamers)

A chemical plant receives oil from three different regions: the Middle East, South America, and the NorthSea. The quality of the oil is different for each region. These oils will be used to produce gasoline, diesel fuel, and bike chain oil. The oil from the Middle east will be only used for the production of gasoline. Of the oil of South America, 20 % will be used for the production of gasoline, and 80 % for the production of diesel fuel. Finally, 25 % of the North Sea oil will be used to produce gasoline, 25 % to produce diesel fuel, and 50 % for the production of bike chain oil.

Suppose today a shipment arrives of 5000 barrels of oil from the Middle East, 9000 barrels of oil from South America, and 1000 barrels from the North Sea. The plant wants to know how much gasoline, diesel fuel, and bike chain oil it can produce from this shipment.

Obviously, the 5000 barrels of oil from the Middle East will completely be used for the production of gasoline, i.e. $1 \cdot 5000 = 5000$ barrels of gasoline. Of the 9000 barrels of South America oil, 20 % will be used for the production of gasoline. Hence, this leads to $0.2 \cdot 9000 = 1800$ barrels of gasoline. Finally, of the 1000 barrels of North sea oil 25 % will be used for the production of gasoline. Hence, we obtain $0.25 \cdot 1000 = 250$ barrels of gasoline. Hence, the total number of barrels of gasoline that will be produced is

$$1 \cdot 5000 + 0.2 \cdot 9000 + 0.25 \cdot 1000 = 5000 + 1800 + 250 = 7050.$$

By a similar computation, one can compute the total number of barrels of diesel fuel:

$$0 \cdot 5000 + 0.8 \cdot 9000 + 0.25 \cdot 1000 = 7450.$$

Finally, one can show that the total number of barrels of bike chain oil is 500, using a similar computation.

The production process can be summarized using the matrix-vector product. The production process is represented by the following matrix

$$\begin{bmatrix} 1 & 0.2 & 0.25 \\ 0 & 0.8 & 0.25 \\ 0 & 0 & 0.5 \end{bmatrix},$$

where the first column represents the division in the three end products of the Middle East oil, the second column represents the division in the three end products of the South America oil, and the third column represents the division in the three end products of the North Sea oil.

```
A=np.array([[1,0.2,0.25],[0,0.8,0.25],[0,0,0.5]])
print(A)
```

```
[[1.   0.2  0.25]
 [0.   0.8  0.25]
 [0.   0.   0.5 ]]
```

The shipment of the oil can be summarized by the following vector:

$$q = \begin{bmatrix} 5000 \\ 9000 \\ 1000 \end{bmatrix}$$

Using the matrix-vector product, it can easily be calculated how much gasoline, diesel fuel, and bike chain oil can be produced from this oil-delivery:

$$Aq = \begin{bmatrix} 1 & 0.2 & 0.25 \\ 0 & 0.8 & 0.25 \\ 0 & 0 & 0.5 \end{bmatrix} \begin{bmatrix} 5000 \\ 9000 \\ 1000 \end{bmatrix} = \begin{bmatrix} 1 \cdot 5000 + 0.2 \cdot 9000 + 0.25 \cdot 1000 \\ 0 \cdot 5000 + 0.8 \cdot 9000 + 0.25 \cdot 1000 \\ 0 \cdot 5000 + 0 \cdot 9000 + 0.5 \cdot 1000 \end{bmatrix} = \begin{bmatrix} 7050 \\ 7450 \\ 500 \end{bmatrix}$$

Hence, using the matrix-vector product we come to the same conclusion: the oil-delivery will result in 7050 barrels of gasoline, 7450 barrels of diesel fuel, and 500 barrels of bike chain oil.

The next day a shipment of 10000 barrels of oil from the Middle East, 1000 barrels of South America, and 200 barrels of the North Sea arrives. Determine the output of gasoline, diesel fuel, and key chain oil of this oil-delivery.

```
# your code here
```

Let us continue with a variation on the above problem.

Now, a chemical plant receives oil from five different regions: the Middle East, South America, the North Sea,Africa and Asia. Again, the quality of the oil is different for each region. These oils will be used to produce gasoline, diesel fuel, bike chain oil, fuel 95 and fuel 98. The oil from the Middle east will be only used for the production of gasoline. Of the oil of South America, 20% will be used for the production of gasoline, and 80% for the production of diesel fuel. 25% of the North Sea oil will be used to produce gasoline, 25% to produce diesel fuel, and 50% for the production of bike chain oil. Of the oil of Africa, 10% will be used for the production of gasoline, 10% for the production of bike chain oil, 30% for the production of fuel 95 and 50% for the production of fuel 98. Finally, 30% of the Asia oil will be used to produce gasoline, 5 % to produce diesel fuel, 50 % for the production of fuel 95, and 15% for the production of fuel 98.

Suppose today a shipment arrives of 5000 barrels of oil from the Middle East, 9000 barrels of oil from South America, 1000 barrels from the North Sea, 4000 barrels of oil from Africa, and 4000 barrels of oil from Asia. The plant wants to know how much gasoline, diesel fuel, bike chain oil, fuel 95, and fuel 98 it can produce from this shipment.

With the new information A will be a 5x5 matrix and q will be a 5x1 vector. Determine the output of gasoline, diesel fuel, bike chain oil, fuel 95, and fuel 98 of this oil-delivery.

```
# your code here
```

4.5 Matrix operations: inverse, determinant, solving linear systems

Assuming we imported NumPy under the alias `np`, the module `np.linalg` from NumPy contains several standard linear algebra methods that we will encounter today.

- `det(A)` : Determinant of matrix A
- `inv(A)` : Inverse of matrix A
- `solve(A,b)` : Solution to linear system $Ax = b$

To use such a method, you should use the syntax `np.linalg.method_name` with `method_name` replaced by one of the three options above.

```
import numpy as np

A = np.array([[0,1],[1,0]]) #Define A
print(A)

print("The determinant of A is", np.linalg.det(A))
```

```
[[0 1]
 [1 0]]
The determinant of A is -1.0
```

Exercise 5.1:

i) Compute the determinant of the following matrices

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \qquad B = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 1 & 1 & 1 \end{pmatrix}, \qquad C = \begin{pmatrix} 1 & 3 & 2 \\ 2 & 3 & 7 \\ 1 & 3 & 1 \end{pmatrix}$$

ii) Which of the matrices A, B, C are invertible? For each of these matrices, compute the inverse.

```
# your code here
```

Exercise 5.2:

In the Linear Algebra course we have seen that if we multiply a single row by a constant k, then the determinant gets multiplied by k as well.

i) Verify this (using Python) for the matrix obtained from C by multiplying the first row by 10:

$$D = \begin{pmatrix} 10 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} C$$

ii) What is $\det(10C)$?

```
# your code here
```

We can use Python to solve linear systems of equations using `numpy`. Consider for example the following system of equations:

$$\begin{cases} x_1 - 3x_2 = 5 \\ -x_1 + x_2 + 5x_3 = 2 \\ x_2 + x_3 = 0 \end{cases}$$

We can solve the system very efficiently numerically with the command `np.linalg.solve(A,b)`:

```
import numpy as np

# Coefficient matrix A
A = np.array([[1, -3, 0],
              [-1, 1, 5],
              [0, 1, 1]])

# Right-hand side vector b
b = np.array([5, 2, 0])

# Solve the system
x = np.linalg.solve(A, b)
print("Solution:", x)
```

```
Solution: [ 2. -1.  1.]
```

As you can see, the solution is $x_1 = 2.0, x_2 = -1.0, x_3 = 1.0$. Note that the output is *numerical* i.e., it is an approximation computed with finite precision arithmetic inside the computer. `numpy` is faster for large systems than `sympy`, but it may introduce tiny rounding errors (e.g., 2.00000000001 instead of 2).

Exercise 5.3: Use `numpy` to solve the following system:

$$\begin{cases} x_1 - 3x_2 + 4x_3 &= -4 \\ 3x_1 - 7x_2 + 7x_3 &= -8 \\ -4x_1 + 6x_2 + 2x_3 &= 4 \end{cases}$$

```
# your code here
```

4.6 Modifying matrix entries

Consider the following large matrix A, and a zero matrix B of the same size as A:

```
A = np.array([
    [5, 12, 7, 3, 14, 6, 9, 2, 11, 8],
    [1, 13, 4, 10, 7, 5, 12, 6, 8, 3],
    [9, 2, 11, 5, 13, 7, 4, 10, 6, 12],
    [8, 1, 14, 6, 9, 3, 11, 2, 5, 13],
    [7, 10, 3, 12, 6, 9, 2, 8, 4, 11],
    [2, 6, 9, 5, 11, 7, 3, 12, 10, 1],
    [4, 8, 2, 10, 5, 13, 6, 9, 1, 7],
    [12, 3, 6, 11, 2, 8, 5, 14, 7, 10],
    [10, 5, 1, 7, 12, 4, 8, 3, 6, 9],
    [3, 9, 5, 8, 1, 10, 7, 11, 2, 12]
])
B=np.zeros((10,10))
```

We can modify entries of B separately. E.g., we can modify the top-left entry of B to be 100:

```
B[0,0]=100
print(B)
```

```
[[100.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]]
```

Exercise 5.4: Use for loops over the row indices i and column indices j to modify the entries of B as follows, for each entry of A:

- If $A_{ij} < 6$, set $B_{ij} = 100$.
- If $A_{ij} \geq 6$, set $B_{ij} = A_{ij}$.

Then, compute the sum of all entries in B with `np.sum(B)`. What is the result?

```
# your code here
```

4.7 Linear transformations of images

In this section we will illustrate the concept of applying linear transformations to real-life images. We use the `cv2` library, the `numpy` library, and the `matplotlib` packages for this.

The above packages should be installed in a standard Anaconda installation. If you have another Python installation, typically using the PIP package manager should allow you to install packages. In this case, commands like "pip install opencv-python" "pip install numpy" "pip install matplotlib" in a (Windows) powershell should do the job.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
```

A linear transformation for a vector in $\mathbb{R}^2$ can be represented by

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

For images, (x, y) are the pixel coordinates of the original image and (x', y') are the pixel coordinates of the transformed image. First we load in an image (this can be any image with the name 'image.jpg', but it must be in the same folder as the Python notebook):

```
img2 = cv2.imread('image.jpg')

plt.imshow(img2[:,:,::-1])
plt.axis('off')
```

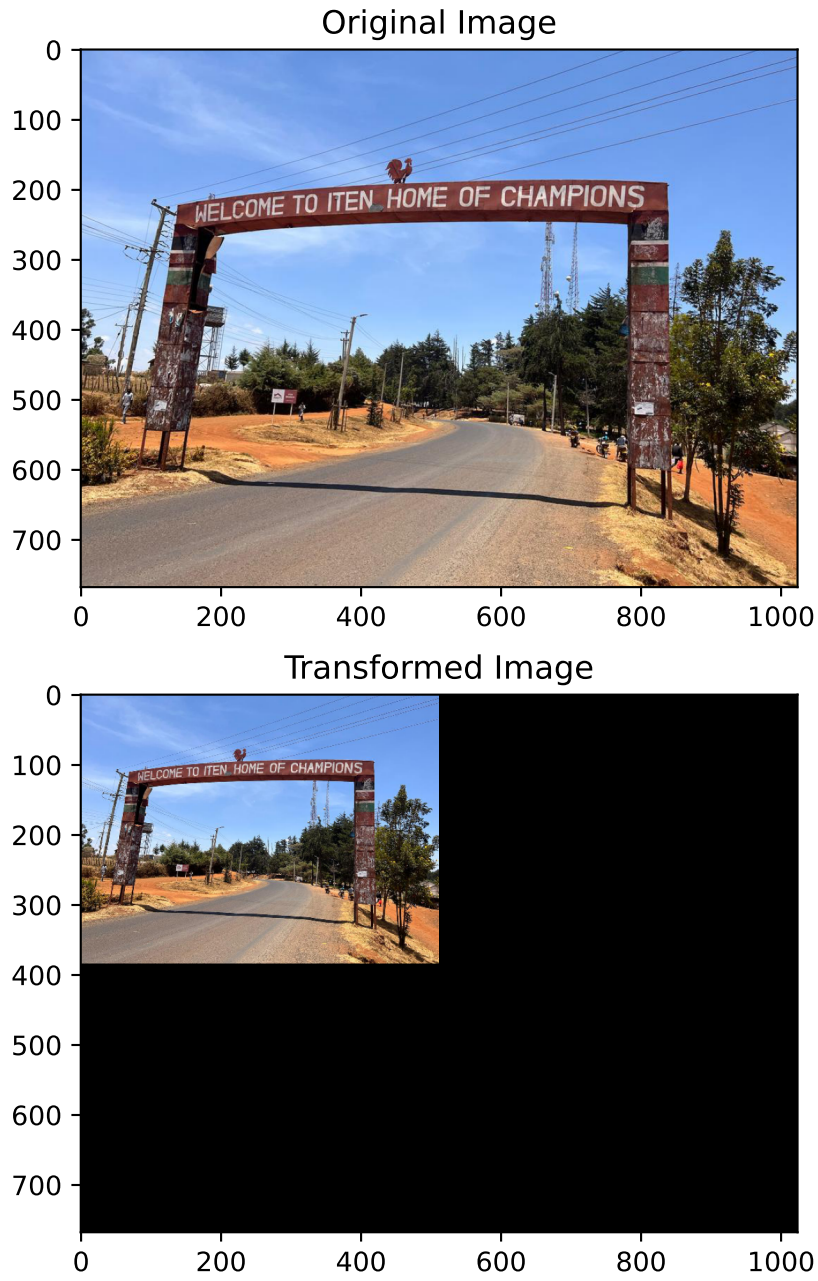
The picture can be seen as a set of points in $\mathbb{R}^2$, corresponding to the pixels, where each point is assigned a color. We are now going to move the points according to a linear transformation $\mathbb{R}^2 \to \mathbb{R}^2$ which maps $\mathbf{x}$ to $A\mathbf{x}$.

We first predefine a Python function called `perform_Transformation()` to apply a transformation to the image. You don't have to understand this code, but only remember that its purpose is to apply transformations to images. The function takes as input an image and a transformation matrix.

```
# Function to apply transformation and visualize the result (source: kaggle.com)
def perform_Transformation(image, A):
    rows,cols,ch = image.shape

    M = np.array([[A[0,0], A[0,1], 0],
                  [A[1,0], A[1,1], 0]])

    dst = cv2.warpAffine(image,M,(cols,rows))


    plt.figure(figsize = (24,8))
    plt.subplot(211); plt.imshow(image[:,:,::-1]); plt.title('Original Image')

    plt.subplot(212); plt.imshow(dst[:,:,::-1]); plt.title("Transformed Image")
```

Next, we create a 2x2 (transformation) matrix A and use the image that we stored in `img2` variable and apply the transformation function to these inputs. The function prints both the original and transformed image.

```
# The matrix A scales the picture down 50% in both the x direction and y direction
A = np.array([[0.5,0],
                [0,0.5]])

perform_Transformation(img2, A)
```



Exercise 5.5:

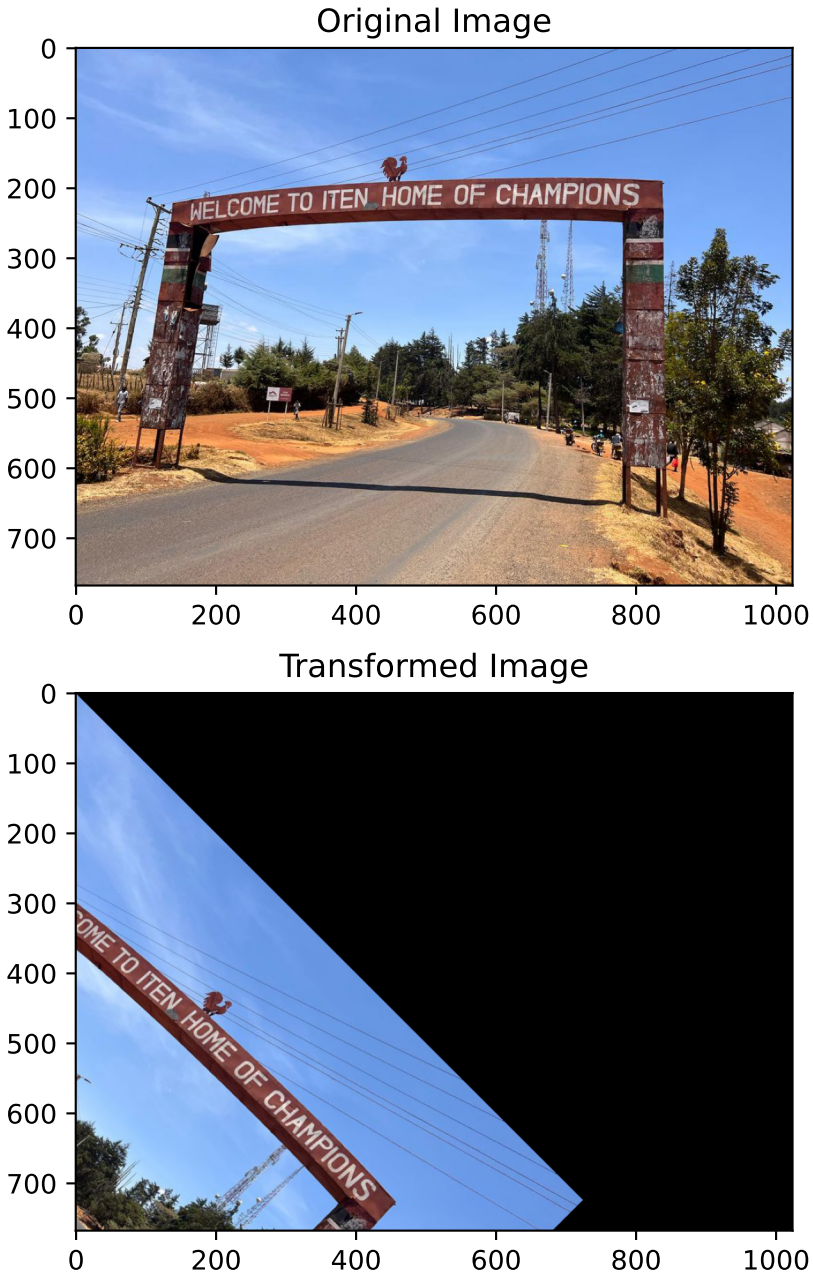
i. Can you scale the picture 50% down in the x-direction only?

ii. Investigate what the matrix $A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ does to the picture. Can you explain why?

```
# your code here
```

The following matrix rotates the picture approximately 45 degrees (note that $\sin(\pi/4) = \cos(\pi/4) \approx 0.707$). Sadly, the image is only partly visible then.

```
# The matrix A rotates the picture 45 degrees
A = np.array([[0.707,-0.707],
              [0.707,0.707]])

perform_Transformation(img2, A)
```



Exercise 5.6:

i. Let $A = \begin{bmatrix} 0.5 & 0 \\ 0 & 1 \end{bmatrix}$ and $B = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$. Compute the matrix products AB and BA.

ii. Investigate what the matrix $BA = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0.5 & 0 \\ 0 & 1 \end{bmatrix}$ does to the picture. Can you explain why?

iii. What does the matrix $AB = \begin{bmatrix} 0.5 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ to the picture and why?

iv. Can you scale the picture 50% down in the x-direction only, and then rotate the picture (approximately) 45 degrees? What matrix corresponds to this?

```
# your code here
```

4.8 Symbolic computations: sympy

The `sympy` library is for symbolic computations. We will now demonstrate how this library works. `sympy` also contains a function to bring a matrix into reduced echelon form. This is very useful if you want to verify manual calculations.

```
import sympy as sp
```

We now show how to solve the following system of equations:

$$\begin{cases} x_1 - 3x_2 + 4x_3 = -4 \\ 3x_1 - 7x_2 + 7x_3 = -8 \\ -4x_1 + 6x_2 + 2x_3 = 4 \end{cases}$$

The augmented matrix of the system of linear equations is

$$\left[\begin{array}{ccc|c} 1 & -3 & 4 & -4 \\ 3 & -7 & 7 & -8 \\ -4 & 6 & 2 & 4 \end{array}\right]$$

In Python you can do this with `sympy` as follows.

```
Ab=sp.Matrix([[1,-3,4,-4],[3,-7,7,-8],[-4,6,2,4]])
```

Now we can find the reduced form of the system using `rref()` from the `sympy` library. The method returns two elements. The first is the row-reduced echelon form of the matrix, and the second is a list of the pivot columns. By typing `Ab_rref[0]`, only the row-reduced echelon form is printed.

```
Ab_rref = Ab.rref()
Ab_rref[0]
```

$$\begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

The system is consistent and we can immediately see the general solution: $x_1 = 2$, $x_2 = 2$ and $x_3 = 0$. The solution is exact, e.g., Python did not round the number 2 to 2.00000 as it does when using the `numpy`

module. `sympy` allows us to do exact computation with matrices, as opposed to approximate computations with `numpy`. Computations with `sympy` are generally slower than with `numpy`, so for very large systems, `numpy` is preferred.

Exercise 5.7: Use Python to solve the following system of equations:

$$\begin{cases} x_1 - 3x_2 = 5 \\ -x_1 + x_2 + 5x_3 = 2 \\ x_2 + x_3 = 0 \end{cases}$$

```
# your code here
```

4.9 Application: supply and demand model

(From Linear Algebra notebook by Herbert Hamers)

The demand q_d and supply q_s of an item depend on the price p and income Y. Suppose that the relation between demand, price and income can be described by the equation

$$q_d = 8 - 0.2p + 0.1Y,$$

and the relation between supply and price by the equation

$$q_s = 6 + 0.3p.$$

The market is in equilibrium if $q_s = q_d$. Replacing both q_s and q_d by the new variable q, the market equilibrium can be found by solving the following system of two linear equations for the unknowns p, q and Y:

$$\begin{cases} q = 8 - 0.2p + 0.1Y \\ q = 6 + 0.3p. \end{cases}$$

First, we will rewrite this system of linear equations. We place all terms with a variable to the left of the equal sign and moreover we put terms with the same variable in the different equations right below each other:

$$\begin{cases} q + 0.2p - 0.1Y = 8 \\ q - 0.3p = 6. \end{cases}$$

```
Ab = sp.Matrix([[1, 0.2, -0.1, 8], [1, -0.3, 0, 6]])
Ab
```

$$\begin{bmatrix} 1 & 0.2 & -0.1 & 8 \\ 1 & -0.3 & 0 & 6 \end{bmatrix}$$

Exercise 5.8:

How many solutions does this system of linear equations have? You can use the function `rref()` again to find the row reduced echelon form of the above matrix. Determine the solution if $Y = 50$.

```
# your code here
```

4.10 Symbolic computation of the determinant in Python with `sympy`

The `sympy` package can also be used to calculate determinants of symbolic matrices.

Exercise 5.9:

We now investigate how to compute determinants of *symbolic* matrices using sympy. Let

$$A = \begin{pmatrix} 1 & 0 & x \\ 1 & -x & 0 \\ x & 0 & -x \end{pmatrix}.$$

Our goal will be to compute the determinant of A in terms of x. First we define the variable x as a "symbolic" variable in Sympy.

```
import sympy as sp
x = sp.symbols('x')
```

Next, carry out the following steps.

- Define the matrix A using the variable x.
- Compute the determinant of A using the command `A.det()`. For which values of x does the determinant equal zero?
- To be able to see more easily for which values of x the determinant equals 0, you can use Sympy's `sp.factor()` function to factor the determinant. The input of this function is the determinant of A. Apply this function and argue for which values of x the matrix A is invertible.

```
# your code here
```

4.11 (Optional) Row reduction with intermediate steps

This optional section describes a more advanced code snippet that implements a row-reduction algorithm from scratch, and allows you to output the intermediate operations. You should be able to read most of the code and recognize the algorithm, but some details have not been explained in this notebook. More specifically, you will see again for-loops and if-statements, but also "new" functions such as .copy(), .as_mutable(), .append(). If you are interested, we invite you to look up their documentation. You can also use this algorithm to revisit some examples that you have seen in the linear algebra course.

As a disclaimer: this code has been generated using ChatGPT.

```
from sympy import Matrix

def rref_with_steps(mat):
    """
```

```
    Perform row-reduction to RREF while recording intermediate steps and row operations.
    Returns:
        rref_matrix (Matrix),
        pivot_columns (tuple),
        steps (list of (Matrix, str))  # Each step is (matrix_snapshot, operation_descriptio
    """
    A = mat.as_mutable().copy()
    rows, cols = A.shape
    pivots = []
    steps = [(A.copy(), "Initial matrix")]

    row = 0
    for col in range(cols):
        if row >= rows:
            break

        # Find pivot row
        pivot_row = None
        for r in range(row, rows):
            if A[r, col] != 0:
                pivot_row = r
                break

        if pivot_row is None:
            continue

        # Swap rows if needed
        if pivot_row != row:
            A.row_swap(pivot_row, row)
            steps.append((A.copy(), f"Swap R{pivot_row+1}  R{row+1}"))

        # Scale pivot row
        pivot_val = A[row, col]
        if pivot_val != 1:
            A.row_op(row, lambda x, _: x / pivot_val)
            steps.append((A.copy(), f"R{row+1} → (1/{pivot_val})·R{row+1}"))

        # Eliminate other rows
        for r in range(rows):
            if r != row and A[r, col] != 0:
                factor = A[r, col]
                A.row_op(r, lambda x, j: x - factor * A[row, j])
                steps.append((A.copy(), f"R{r+1} → R{r+1} - ({factor})·R{row+1}"))

        pivots.append(col)
        row += 1
```

```
    return A, tuple(pivots), steps
```

```
A = Matrix([[1, 2, 1],
            [2, 4, 3],
            [3, 6, 5]])

rref_matrix, pivots, steps = rref_with_steps(A)

print("RREF:")
print(rref_matrix)
print("Pivot columns:", pivots)

print("\nSteps:")
for i, (s, op) in enumerate(steps):
    print(f"Step {i}: {op}")
    display(s)
```

```
RREF:
Matrix([[1, 2, 0], [0, 0, 1], [0, 0, 0]])
Pivot columns: (0, 2)

Steps:
Step 0: Initial matrix
```

$$\begin{bmatrix}1 & 2 & 1\\2 & 4 & 3\\3 & 6 & 5\end{bmatrix}$$

```
Step 1: R2 → R2 - (2)·R1
```

$$\begin{bmatrix}1 & 2 & 1\\0 & 0 & 1\\3 & 6 & 5\end{bmatrix}$$

```
Step 2: R3 → R3 - (3)·R1
```

$$\begin{bmatrix}1 & 2 & 1\\0 & 0 & 1\\0 & 0 & 2\end{bmatrix}$$

```
Step 3: R1 → R1 - (1)·R2
```

$$\begin{bmatrix}1 & 2 & 0\\0 & 0 & 1\\0 & 0 & 2\end{bmatrix}$$

```
Step 4: R3 → R3 - (2)·R2
```

$$\begin{bmatrix}1 & 2 & 0\\0 & 0 & 1\\0 & 0 & 0\end{bmatrix}$$

Chapter 5

Linear optimization

In this chapter we will show how to implement linear optimization problems using Python. The goal is to model optimization problems in Python and then solve them using Gurobi, which is a software package dedicated to solving optimization problems.

> **Prerequisites**
>
> This chapter assumes familiarity with Sections 3.1-3.5 of this online book.

In the next section we explain how to install Gurobi on your personal laptop, and how to connect it to Anaconda. On the university computers Gurobi is already installed and configured correctly, so if you are using such a computer, you can skip the next section.

5.1 Gurobi installation

It is assumed that you already have Python installed on your system. Otherwise, you need to install Python first. The Anaconda distribution is a good choice. If you get stuck anywhere in the installation process, then you can find more information on Gurobi's Quick Start Guide.

5.1.1 Register at Gurobi

Visit the Gurobi website and click on the register button. Open the registration form and make sure that you select the *Academic* account type, and select Student for the academic position.

5.1.2 Download Gurobi

Download Gurobi from the website. Note that you need to login with your Gurobi account before you can download Gurobi. Select the distribution that corresponds to your system (e.g., Windows or macOS), and select the regular "Gurobi Optimizer", not any of the AMPL variations. Unless mentioned otherwise, download the most recent version.

5.1.3 Install Gurobi

Run the installer and follow the installation steps. At some point, the installer may ask whether to add Gurobi to your execution path. This is probably useful to accept.

5.1.4 Gurobi license

You cannot use Gurobi without a license, so you need to apply for a license. As a student you can request a free academic license; take the **Named-User Academic** one. After you have obtained the license, you need to activate it for your Gurobi installation. If you open the license details on the Gurobi website, you can see what you need to do: open a command prompt and run the `grbgetkey` command with the code that corresponds to your license. This command will create your license file. Make sure that you remember where the license file is saved. The default location is probably the best choice.

Note that the `grbgetkey` command will check that you are on an academic domain, so you need to perform this step on the **university network** (possibly via a VPN connection). Once installed correctly, you can also run Gurobi without an active VPN connection.

5.1.5 Using Gurobi in Python

Gurobi should now be correctly installed, but we also want to be able to use it from Python. Therefore, we need to install Gurobi's Python package. Open your Anaconda prompt (if you have the Anaconda installation) and run the following commands.

- `conda config --add channels http://conda.anaconda.org/gurobi`
- `conda install gurobi`

If you don't have the Anaconda installation, then you can do something similar with the `pip` command.

5.1.6 Test your installation

Now you can test whether everything is setup correctly. Open an interactive Python session, for instance using Jupyter Notebook. Try the following commands:

```
from gurobipy import Model
model = Model()
```

```
Set parameter Username
Set parameter LicenseID to value 2715549
Academic license - for non-commercial use only - expires 2026-09-29
```

If both these commands succeed, then you are done.

If the first command fails, then the Gurobi python module has not been installed correctly. If the second command fails, then the license has not been setup correctly (make sure the license file is at the right location).

If at any point, you need more information about Gurobi, then you can always go to the official Gurobi documention online.

5.2 Basics

All the relevant functionality needed to use Gurobi via Python is contained in the `gurobipy` package. For our purposes, we only need two *modules* from this package:

- The `Model` module that we will use to build optimization problems
- The `GRB` module that contains various "constants" that we use to include words such as *maximize* and *minimize*, as well as *binary* and *integer*.

In Python you can include the modules by adding the following line at the top of a script.

```
from gurobipy import Model, GRB
```

5.2.1 Example: Duplo problem

To illustrate how to implement optimization problems in Python, we consider the Duplo problem from the lectures:

$$\begin{array}{lll} \text{maximize} & 15x_1 + 20x_2 & \text{(profit)} \\ \text{subject to} & x_1 + 2x_2 \leq 6 & \text{(big bricks)} \\ & 2x_1 + 2x_2 \leq 8 & \text{(small bricks)} \\ & x_1, x_2 \geq 0 & \end{array}$$

with decision variables

- x_1: Number of chairs
- x_2: Number of tables

Let us implement the Duplo problem in Gurobi.

Model object

To start, we will create a variable (or object) `model` that will contain all the information of the problem, such as the decision variables, objective function and constraints.

In programming terms, we create an *object* from the `Model()` class. We can give the instance a name by adding the `name` *keyword* argument, with value `'Duplo problem'` in our case. We add this name with quotations so that Python knows this is plain text.

```
# Initialize Gurobi model.
model = Model(name='Duplo problem')
```

You can think of the Model class object `model` as a large box that we are going to fill with decision variables, an objective function, and constraints. We will also reserve a small space in this box to store information about the optimal solution to the linear optimization problem, once we have optimized the problem.

The concept of having "objects" is in fact what the programming language Python is centered around. This is known as *object-oriented programming*, which you will learn about more later in your first full programming course. You can have a look, for example, here already if you are interested in this concept.

Decision variables

To add decision variables, an objective function, and constraints, to our Model object we use so-called *methods* which are Python functions.

To create a decision variable, we can use the `addVar()` method. As input argument, we include a name for the decision variables.

```
# Declare the two decision variables.
x1 = model.addVar(name='chairs')
x2 = model.addVar(name='tables')
```

> If your Model object is not called `model`, but for example `model_Duplo`, then you should use `model_Duplo.addVar(name='chairs')` instead. The same applies for all later methods that are introduced. Never call a model `Model` (with capital M) because this spelling is reserved for the `Model()` class in Gurobi.

When creating a variable, you can specify more properties. For example, we might have an upper and lower bound for a decision variable (such as a nonnegativity constraint). You can use the keyword arguments `lb` and `ub`, respectively, for this.

To see all these properties, you can check out the `addVar` documentation by typing `help(Model.addVar)` in Python. Looking up the documentation in this way can be done for every method.

```
help(Model.addVar)
```

```
Help on cython_function_or_method in module gurobipy._model:

addVar(self, lb=0.0, ub=1e+100, obj=0.0, vtype='C', name='', column=None)
    ROUTINE:
      addVar(lb, ub, obj, vtype, name, column)

    PURPOSE:
      Add a variable to the model.

    ARGUMENTS:
      lb (float): Lower bound (default is zero)
      ub (float): Upper bound (default is infinite)
      obj (float): Objective coefficient (default is zero)
      vtype (string): Variable type (default is GRB.CONTINUOUS)
      name (string): Variable name (default is no name)
      column (Column): Initial coefficients for column (default is None)

    RETURN VALUE:
      The created Var object.

    EXAMPLE:
      v = model.addVar(ub=2.0, name="NewVar")
```

If we would want to create a variable, with name *test variable*, having lower bound 3 and upper bound 7, we can do this with `x = model.addVar(lb=3,ub=7,name='test variable')` using in particular the `lb` and `ub` keyword arguments.

As you can see above, all the keyword arguments have default values, meaning that if we do not specify them, Gurobi uses the specified default value for them. For example, by default a variable is continuous (`vtype` keyword argument) and non-negative (`lb` keyword argument), so we do not have to specify these for the Duplo exmpale, because there we have $x_1, x_2 \geq 0$.

Also note that `ub` is set to 10^{100} which roughly speaking indicates that the decision variable has no upper bound value by default.

Objective function

Now that we have our decision variables, we can use them to define the objective function and the constraints.

We use the `setObjective()` method to do so by refering to the variables created above. The `sense` keyword argument must be used to specify whether we want to maximize or minimize the objective function. For this, we use `GRB.MAXIMIZE` or `GRB.MINIMIZE`, respectively.

```
# Specify the objective function.
model.setObjective(15*x1 + 20*x2, sense=GRB.MAXIMIZE)
```

Constraints

The next step is to declare the constraints using the `addConstr()` method. Also here, we can specify the constraints refering to the variables `x1` and `x2`. We give the constraints a name as well.

```
# Add the resource constraints on bricks.
model.addConstr(x1 + 2*x2 <= 6, name='big-bricks')
model.addConstr(2*x1 + 2*x2 <= 8, name='small-bricks')
```

```
<gurobi.Constr *Awaiting Model Update*>
```

Now the model is completely specified, we are ready to compute the optimal solution. We can do this using the `optimize()` method applied to our model `model`.

```
# Optimize the model
model.optimize()
```

```
Gurobi Optimizer version 12.0.1 build v12.0.1rc0 (win64 - Windows 10.0 (19045.2))

CPU model: 12th Gen Intel(R) Core(TM) i7-1265U, instruction set [SSE2|AVX|AVX2]
Thread count: 10 physical cores, 12 logical processors, using up to 12 threads

Optimize a model with 2 rows, 2 columns and 4 nonzeros
Model fingerprint: 0xadc88607
Coefficient statistics:
  Matrix range     [1e+00, 2e+00]
  Objective range  [2e+01, 2e+01]
  Bounds range     [0e+00, 0e+00]
  RHS range        [6e+00, 8e+00]
Presolve time: 0.00s
Presolved: 2 rows, 2 columns, 4 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    3.5000000e+31   3.500000e+30   3.500000e+01      0s
       2    7.0000000e+01   0.000000e+00   0.000000e+00      0s

Solved in 2 iterations and 0.00 seconds (0.00 work units)
Optimal objective  7.000000000e+01
```

We can see some output from Gurobi, and the last line tells us that Gurobi found an optimal solution with objective value 70.

To summarize all the step above, we have given the complete code below (but not executed this time).

```
from gurobipy import Model, GRB

# Initialize Gurobi model.
model = Model(name='Duplo problem')

# Declare the two decision variables.
x1 = model.addVar(name='chairs')
x2 = model.addVar(name='tables')

# Specify the objective function.
model.setObjective(15*x1 + 20*x2, sense=GRB.MAXIMIZE)

# Add the resource constraints on bricks.
model.addConstr(x1 + 2*x2 <= 6, name='big-bricks')
model.addConstr(2*x1 + 2*x2 <= 8, name='small-bricks')

# Optimize the model
model.optimize()
```

Recalling our linear optimization problem object as being a "box" filled with decision variables, an objective function and constraints, the box now also contains a small space where the optimal solution is stored.

Let us have a look at how to access information about the optimal solution. The objective function value of the optimal solution, that was also displayed in the output when optimization the model with `model.optimize()` is stored in the `ObjVal` *attribute*. An attribute is a piece of information about an object in Python.

```
# Print text 'Objective value' and the model.ObjVal attribute value
print('Objective value:', model.ObjVal)
```

```
Objective value: 70.0
```

The optimal values of a decision variables can be obtained by accessing the `X` *attribute* of a variable.

```
print('x1 = ', x1.X)
print('x2 = ', x2.X)
```

```
x1 =  2.0
x2 =  2.0
```

If the model has many variables, printing variables in this way is a cumbersome approach. It is then easier to iterate over all variables of the model using the `getVars()` method. This method creates a list of all the decision variables of the model.

```
print(model.getVars())
```

```
[<gurobi.Var chairs (value 2.0)>, <gurobi.Var tables (value 2.0)>]
```

As you can see above, we already see the optimal values of the decision variables, but also a lot of unnecessary information.

Let us print these values in a nicer format by iterating over the list `model.getVars()` using a for-loop with index variable `var`; you can choose another name than `var` for the index variable if you want. We print for every variable `var` its name, stored in the attribute `VarName`, and its value, stored in the attribute `X`. In between, we print the $=$ symbol in plain text (hence, the quotations).

Recall that if you want to print multiple data types (such as a variables and text) in a `print()` statement, you should separate them with commas.

```
for var in model.getVars():
    print(var.VarName, "=", var.X)
```

```
chairs = 2.0
tables = 2.0
```

Exercise 1

Médecins sans Frontières (MSF) (Dutch: Artsen zonder Grenzen) wants to build medical kits to use in region with an epidemic. The following constraints should be taken into account.

- MSF has raised € 6400 to build two types of medical medical kits: vaccination and surgical kits. Assembling a surgical kit costs € 320; a vaccination kit € 210.
- MSF has in total 80 labour hours available in which employees can build these kits. Building a surgical kit requires 2.5 labour hours; a vaccination kit requires 4.5 labour hours.
- At most 15 vaccination kits can be built.

A linear optimization problem that maximizes the total number of produced kits is given below. The decision variables are the number of surgical kits (x_1) and the number of vaccination kits (x_2) to assemble.

$$
\begin{array}{llrcrcrl}
\text{max} & z = & x_1 & + & x_2 & & & \textit{total number of kits} \\
\text{s.t.} & & 320x_1 & + & 210x_2 & \leq & 6400 & \textit{budget constraint} \\
& & 2.5x_1 & + & 4.5x_2 & \leq & 80 & \textit{labor hours constraint} \\
& & & & x_2 & \leq & 15 & \textit{at most } 15 \textit{ vac. kits} \\
& & x_1, & & x_2 & \geq & 0 & \textit{nonneg. constraints}
\end{array}
$$

Implement this model in Python with a Model object called `model_msf`, with decision variable whose keyword arguments for `name` are `'Surgical kits'` and `'Vaccination kits'`, respectively. Optimize your model.
If you include the following lines at the end of your code, you should get the output as indicated.

```
# Replace model_msf by chosen name of Model object if different.
# For example, if your object is called model, then use model.getVars() instead
for var in model_msf.getVars():
    print(var.VarName, "=", var.X)
```

```
Surgical kits = 13.114754098360653
Vaccination kits = 10.491803278688531
```

As you can see in the output of Exercise 1, the optimal values of the decision variables are not integer, although this is a desired property because we cannot build, e.g., 13.11 kits. We will later see how you can enforce the decision variables to be integer-valued as well. You can ignore this shortcoming for now.

Exercise 2

A brewery produces beer in Haarlem and Eindhoven (plants), and ships to Amsterdam, Amersfoort, Gouda, Den Bosch and Breda (customers).
We can ship beer units from the plants to the customers, taking into account transportation costs, demand of the customers and supply of the plants. In the table below we have indicated the following input data:

- Unit transport costs (in euros) from the **plants** (Haarlem, Eindhoven) to the **customers** (Amsterdam, Breda, Gouda, Amersfoort, Den Bosch);
- Demand of each customer: what needs to *at least* be delivered;
- Maximum supply capacity of each plant: what can *at most* be supplied.

	A'dam	Breda	Gouda	A'foort	Den Bosch	Supply
Haarlem	131	405	188	396	485	47
Eindhoven	554	351	479	366	155	63
Demand	28	16	22	31	12	

We define decision variables x_{pc} that model the number of units shipped from plant p to customer c, with the meaning of the indices p and c as follows:

- Plants: $p = 1$ for Haarlem, $p = 2$ for Eindhoven.
- Customers: $c = 1$ for Amsterdam, $c = 2$ for Breda, $c = 3$ for Gouda, $c = 4$ for Amersfoort, $c = 5$ for Den Bosch.

A linear optimization problem that minimizes the total transportation costs subject to the demand and supply constraints is given below.

$$
\begin{array}{lll}
\text{min} & 131x_{11} + 405x_{12} + 188x_{13} + 396x_{14} + 485x_{15} + & \\
& 554x_{21} + 351x_{22} + 479x_{23} + 366x_{24} + 155x_{25} & \\
\text{s.t.} & x_{11} + x_{12} + x_{13} + x_{14} + x_{15} \leq 47 & \textit{supply Haarlem} \\
& x_{21} + x_{22} + x_{23} + x_{24} + x_{25} \leq 63 & \textit{supply Eindhoven} \\
& x_{11} + x_{21} \geq 28 & \textit{demand A'dam} \\
& x_{12} + x_{22} \geq 16 & \textit{demand Breda} \\
& x_{13} + x_{23} \geq 22 & \textit{demand Gouda} \\
& x_{14} + x_{24} \geq 31 & \textit{demand A'foort} \\
& x_{15} + x_{25} \geq 12 & \textit{demand Den Bosch} \\
& x_{11}, x_{12}, x_{13}, x_{14}, x_{15}, x_{21}, x_{22}, x_{23}, x_{24}, x_{25} \geq 0 & \textit{ship nonneg. amounts}
\end{array}
$$

Implement this model in Python with a Model object called `model_beer`. In the `name` keyword argument of a decision variable, indicate the plant-customer combination it represents (for example, you could define `x11 = model_beer.addVar(name='Haarlem-Amsterdam')` for the decision variable x_{11} modeling the Haarlem-Amsterdam combination).
If you include the following lines at the end of your code, you should get the output as indicated.

```
for var in model_beer.getVars():
    print(var.VarName, "=", var.X)
```

```
Haarlem-Amsterdam = 28.0
Haarlem-Breda = 0.0
Haarlem-Gouda = 19.0
Haarlem-Amersfoort = 0.0
Haarlem-Den Bosch = 0.0
Eindhoven-Amsterdam = 0.0
Eindhoven-Breda = 16.0
Eindhoven-Gouda = 3.0
Eindhoven-Amersfoort = 31.0
Eindhoven-Den Bosch = 12.0
```

As you might have experienced in Exercise 2, once the number of decision variables and/or constraints grows, it becomes more tedious to write out the full problem in Python. Luckily, there are more efficient ways to implement the above problem, that only require a couple of lines of code, even if there are many more plants and customers! This is beyond the scope of this course.

5.2.2 Infeasible models

Not every linear optimization problem has an optimal solution. For example, the problem

$$\begin{array}{lrcrcr} \min & z = & 2x_1 & + & x_2 & \\ \text{s.t.} & & x_1 & + & x_2 & \leq & -1 \\ & & x_1, & & x_2 & \geq & 0 \end{array}$$

has no feasible solution, because the sum of two nonnegative numbers ($x_1, x_2 \geq 0$) can never sum up to something smaller or equal than -1 ($x_1 + x_2 \leq -1$). The implementation of this problem is given below. It can be seen from the output (on the last line) that the model is indeed infeasible.

```
# Initialize Gurobi model.
model = Model(name='Infeasible model')

# Declare the two decision variables.
x1 = model.addVar() # We leave out the name keyword argument
x2 = model.addVar()

# Specify the objective function.
model.setObjective(x1 + x2, sense=GRB.MINIMIZE)

# Add the resource constraints on bricks.
model.addConstr(x1 + x2 <= -1)

# Optimize the model
model.optimize()
```

```
Gurobi Optimizer version 12.0.1 build v12.0.1rc0 (win64 - Windows 10.0 (19045.2))

CPU model: 12th Gen Intel(R) Core(TM) i7-1265U, instruction set [SSE2|AVX|AVX2]
Thread count: 10 physical cores, 12 logical processors, using up to 12 threads

Optimize a model with 1 rows, 2 columns and 2 nonzeros
Model fingerprint: 0xf5807d19
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [1e+00, 1e+00]
  Bounds range     [0e+00, 0e+00]
  RHS range        [1e+00, 1e+00]
Presolve removed 0 rows and 2 columns
Presolve time: 0.00s

Solved in 0 iterations and 0.00 seconds (0.00 work units)
Infeasible model
```

You can look up the type of solution to a problem in the `Status` attribute of a model.

```
model.Status
```

```
3
```

The above output number is not informative right away. The number represents the *Optimization Status Code*, whose meaning you can look up in Gurobi's online documentation.

> Exercise 3
>
> When implementing and optimizing the following problem
>
> $$\begin{array}{lrrcrcl} \max & z = & 2x_1 & + & x_2 & & \\ \text{s.t.} & & x_1 & - & x_2 & \leq & 5 \\ & & x_1, & & x_2 & \geq & 0 \end{array} \quad ,$$
>
> Python returns an Optmization Status Code of 4. Look up in the online documentation, more specifically the *Reference Manual*, what this means.

5.3 Integer variables

If you are not interested in finding fractional solutions, but want to enforce the decision variables to take integer values, you can do this with the `vtype` keyword argument.

Consider the following linear optimization problem (without integrality constraints).

$$\begin{array}{lrrcrcl} \max & z = & 2x_1 & + & x_2 & & \\ \text{s.t.} & & x_1 & + & x_2 & \leq & 2.5 \\ & & x_1, & & x_2 & \geq & 0 \end{array} \quad ,$$

```
# Initialize Gurobi model.
model_frac = Model(name='Fractional problem')

# Suppress Gurobi Optimizer output
model_frac.setParam('OutputFlag',0)

# Declare the two decision variables.
x1 = model_frac.addVar(name="x1")
x2 = model_frac.addVar(name="x2")

# Specify the objective function.
model_frac.setObjective(2*x1 + x2, sense=GRB.MAXIMIZE)

# Add the resource constraints on bricks.
model_frac.addConstr(x1 + x2 <= 2.5)

# Optimize the model
model_frac.optimize()

# Print optimal values of decision variables
for var in model_frac.getVars():
    print(var.VarName, "=", var.X)
```

```
x1 = 2.5
x2 = 0.0
```

If we specify `vtype=GRB.INTEGER` in the `addVar()` method when creating the variables, Gurobi will find an optimal solution with all decision variables being integers.

```
# Initialize Gurobi model.
model_int = Model(name='Integer problem')

# Suppress Gurobi Optimizer output
model_int.setParam('OutputFlag',0)

# Declare the two decision variables.
x1 = model_int.addVar(name="x1",vtype=GRB.INTEGER)
x2 = model_int.addVar(name="x2",vtype=GRB.INTEGER)

# Specify the objective function.
model_int.setObjective(2*x1 + x2, sense=GRB.MAXIMIZE)

# Add the resource constraints on bricks.
model_int.addConstr(x1 + x2 <= 2.5)

# Optimize the model
model_int.optimize()

# Print optimal values of decision variables
for var in model_int.getVars():
    print(var.VarName, "=", var.X)
```

```
x1 = 2.0
x2 = -0.0
```

Anoter common case is where the decision variables are supposed to be binary, meaning they can only take values in $\{0, 1\}$. Setting variables to be binary can be done with `vtype=GRB.BINARY`.

```
# Initialize Gurobi model.
model_bin = Model(name='Binary problem')

# Suppress Gurobi Optimizer output
model_bin.setParam('OutputFlag',0)

# Declare the two decision variables.
x1 = model_bin.addVar(name="x1",vtype=GRB.BINARY)
x2 = model_bin.addVar(name="x2",vtype=GRB.BINARY)

# Specify the objective function.
model_bin.setObjective(2*x1 + x2, sense=GRB.MAXIMIZE)
```

```
# Add the resource constraints on bricks.
model_bin.addConstr(x1 + x2 <= 2.5)

# Optimize the model
model_bin.optimize()

# Print optimal values of decision variables
for var in model_bin.getVars():
    print(var.VarName, "=", var.X)
```

```
x1 = 1.0
x2 = 1.0
```

Exercise 4

Take your solution from Exercise 1, and modify it so that the decision variables only take integer values. Optimize the model again.
Your output should now be as follows:

```
# Print optimal values of decision variables
for var in model_msf.getVars():
    print(var.VarName, "=", var.X)
```

```
Surgical kits = 9.0
Vaccination kits = 15.0
```

5.4 Beyond the basics

You can also do more complicated things with `gurobipy` such as defining many decision variables with one command, or use a for-loop to create many constraints in one go.

To illustrate these concepts, we will write a compact code to solve a general standard form problem with m constraints and n decision variables: $\min \mathbf{c}'\mathbf{x}$ subject to $\mathbf{Ax} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}$ where $\mathbf{c} \in \mathbb{R}^n$, $\mathbf{b} \in \mathbb{R}^m$, $\mathbf{A} \in \mathbb{R}^{m \times n}$, and $x = (x_1, \ldots, x_n)$.

More explicitly, the problem is as follows:

$$\begin{array}{lrcccl} \min & c_1 x_1 & + & \cdots & + & c_n x_n & \\ \text{s.t.} & a_1 x_1 & + & \cdots & + & a_{1n} x_n & = b_1 \\ & & & \vdots & & & \\ & a_{m1} x_1 & + & \cdots & + & a_{mn} x_n & = b_m \\ & x_1, & & \cdots & & x_n & \geq 0 \end{array}$$

Suppose we are given the following input data in Numpy arrays (see the Linear Algebra chapter).

```
import numpy as np
```

```
# Objective function coefficients
c = np.array([3, 1, 7, 2, -1]) # n = 5

# Matrix A (m = 3, n = 5)
A = np.array([
    [2, 1, 0, 3, -1],
    [1, 0, 4, 0, 2],
    [0, 3, -2, 1, 1]
])

# Right-hand side vector b
b = np.array([10, 12, 5]) # m = 3
```

Decision variables

To create the decision variables $x_1, \ldots, x_5$, we can use `addVar()` five times, but this is rather cumbersome, especially if we would have many more decision variables.

Instead we can use the related method `addVars()` that allows us to create many variables simultaneously.

```
# We define m and n here for convenience later
m = 3 # Alternatively, m = len(b)
n = 5 # Alternatively, n = len(c)

# Create Model object
model = Model("Standard form problem")

# Add decision variables
x = model.addVars(n)
```

With the syntax `x = model.addVars(n)` we tell Python to create $n(=5)$ decision variables. You can use the i-th decision variable to define constraints and the objective function by indexing `x` at position $i-1$, i.e, by using `x[i-1]` . Recall that Python starts counting from zero when indexing data objects such as lists.

If you want you can also add a list of names for the decision variables using the `name` keyword argument. Make sure that the number of elements in the list matches the value of n.

```
# Add decision variables
x = model.addVars(n,name=["Var1","Var2","Var3","Var4","Var5"])
```

Objective function

We continue with creating the objective function by indexing the variables in `x` and the array `c` containing the objective function coefficients.

```
# Create objective function
model.setObjective(x[0]*c[0] + x[1]*c[1] + x[2]*c[2] + x[3]*c[3] + x[4]*c[4])
```

We can do this even more compactly by using the `quicksum()` function, which has to be imported from `gurobipy`. In the code below we do two things (recall that $n = 5$):

1. Create the list `y = [x[0]*c[0], x[1]*c[1], x[2]*c[2], x[3]*c[3], x[4]*c[4]]` using the concept of *list comprehension*.
2. Use the `quicksum()` function to add up the elements in the list `y` and set the resulting quantity as our objective function.

```
from gurobipy import quicksum

# Create objective function
y = [x[j]*c[j] for j in range(n)] # List with terms of the objective function
model.setObjective(quicksum(y)) # Sum up the terms in the list y
```

The part `[x[j]*c[j] for j in range(n)]` is the compact *list comprehension* syntax for carrying out the piece of code below where we repeatedly append the terms `x[j]*c[j]` as elements to an (initially empty) list `y` using the `append()` method for a list.

```
y = [] # Start with empty list y
for j in range(n): # Iterated over j = 0,1,2,...,n-1
    y.append(x[j]*c[j]) # Append the element x[j]*c[j] to the list y
```

Note that these computations are actually still quite abstract up until this point, because the decision variables have no actual values yet! You should think of these more as symbolic computations.

Constraints

We continue with adding the equality constraints. We use a for-loop to loop over the equalities $a_{i1}x_1 + \cdots + a_{in}x_n = b_i$ for $i = 1, \dots, m$ (note that Python starts counting at 0, though). Note that the array `A` is a list of lists, and that `A[i]` corresponds to the coefficients $a_{i1}, \dots, a_{in}$.

```
print(A[2]) # Last row i = 2 of the matrix A
```

```
[ 0  3 -2  1  1]
```

Furthermore, `A[i][j]` is then the j-th element on the i-th row.

```
print(A[2][3]) # Last row i = 2 and element j = 3 of the matrix A
```

```
1
```

For a fixed i, the list `[A[i][j]*x[j] for j in range(n)]` then contains the coefficients of the i-th row multiplied by the variables in x, i.e, $[a_{i1}x_1, ..., a_{in}x_n]$. Applying `quicksum()` to this list then computes the expression $a_{i1}x_1 + \cdots + a_{in}x_n$. This should equal b_i, so we model the constraint by `quicksum([A[i][j]*x[j] for j in range(n)]) == b[i]`. This is done below.

```
# Create constraints
for i in range(m):
    y = [A[i][j]*x[j] for j in range(n)]
    model.addConstr(quicksum(y) == b[i])
```

All together, our code looks as below. The beauty of this is that even if the input data arrays A, b and c get larger, the code below does not.

```
import numpy as np
from gurobipy import Model, GRB, quicksum

# Objective function coefficients
c = np.array([3, 1, 7, 2, -1]) # n = 5

# Matrix A (m = 3, n = 5)
A = np.array([
    [2, 1, 0, 3, -1],
    [1, 0, 4, 0, 2],
    [0, 3, -2, 1, 1]
])

# Right-hand side vector b
b = np.array([10, 12, 5]) # m = 3

# We define m and n here for convenience later
m = 3
n = 5

# Create Model object
model = Model("Standard form problem")

# Suppres Gurobi output message
model.setParam('OutputFlag',0)

# Add decision variables
x = model.addVars(n,name=["Var1","Var2","Var3","Var4","Var5"])

# Create objective function
model.setObjective(quicksum(x[j]*c[j] for j in range(n)))

# Create constraints
for i in range(m):
    model.addConstr(quicksum(A[i][j]*x[j] for j in range(n)) == b[i])

# Optimize the model
model.optimize()
```

```
# Print optimal values of decision variables
for var in model.getVars():
    print(var.VarName, "=", var.X)
```

```
Var1 = 4.75
Var2 = 0.0
Var3 = 0.0
Var4 = 1.3749999999999998
Var5 = 3.625
```

The critical reader, however, might note that, if we want to include the variable names with the list `["Var1","Var2","Var3","Var4","Var5"]`, then this list would need to get longer when n increases.

You can replace this list by the list comprehension `[f"Var{i+1}" for i in range(n)]` to quickly generate the list `["Var1","Var2", ..., "Varn"]`. This comprehension loops over the values $i = 0, 1 \ldots, n-1$.

The part `f"Var{i+1}"` is called a formatted *string (of text)*. It allows us to include the *value* of the expression `i+1` in plain text (we do $+1$ because we want the variable names to start at 1). The value you want to include in plain text should be put in curly brackets `{}`, and you have to add an `f` in front of the string of text in quotations so that Python knows it should format the value of $i+1$ into text.

Now we get a code which truly is independent of the values of m and n. That is, for any input data A, b and c, the code below creates a model to solve the standard form problem.

```
import numpy as np
from gurobipy import Model, GRB, quicksum

# Objective function coefficients
c = np.array([3, 1, 7, 2, -1]) # n = 5

# Matrix A (m = 3, n = 5)
A = np.array([
    [2, 1, 0, 3, -1],
    [1, 0, 4, 0, 2],
    [0, 3, -2, 1, 1]
])

# Right-hand side vector b
b = np.array([10, 12, 5]) # m = 3

# We define m and n here for convenience later
m = 3
n = 5

# Create Model object
model = Model("Standard form problem")
```

```
# Suppres Gurobi output message
model.setParam('OutputFlag',0)

# Add decision variables
x = model.addVars(n,name=[f"Var{i+1}" for i in range(n)])

# Create objective function
model.setObjective(quicksum(x[j]*c[j] for j in range(n)))

# Create constraints
for i in range(m):
    model.addConstr(quicksum(A[i][j]*x[j] for j in range(n)) == b[i])

# Optimize the model
model.optimize()

# Print optimal values of decision variables
for var in model.getVars():
    print(var.VarName, "=", var.X)
```

```
Var1 = 4.75
Var2 = 0.0
Var3 = 0.0
Var4 = 1.3749999999999998
Var5 = 3.625
```

Chapter 6

Probability and statistics

In this chapter we will show various visualizations and Python basics of concepts we have seen in the course Probability and Statistics (35B402).

More precisely, this chapter consists of two parts: links to interactive applications illustrating concepts and examples seen in class, and Python basics for working with probability distributions and randomly generated data.

> **Note**
>
> This document will be updated throughout the course Probabilisty and Statistics during the academic year 2025/2026.

6.1 Probability distributions

The `stats` module of SciPy has many built-in probability distributions. Each distribution can be seen as an object on which various methods can be performed (such as accessing its probability density function or summary statistics like the mean and median).

```
import numpy as np
import scipy.stats as stats
```

In this section we will focus on continuous probability distributions. SciPy also has many built-in discete probability distributions.

A list of all continuous distributions that are present in the `stats` module can be found here; they are so-called `stats.rv_continuous` objects. We can instantiate a distributional object by using `scipy.stats.dist_name` where `dist_name` is the name of a built-in (continuous) probability distribution in the mentioned list.

Many distributions have input parameters `scale` and `loc` that model the scale and location of the distribution, respectively. Depending on the distribution that is considered, these parameters have different meanings.

As an example, the normal distribution has probability density function

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

which is parameterized by μ and σ.

In Python μ is the `loc` parameter, and σ the `scale` parameter. To figure out the function of the `scale` and `loc` parameter, you can check the documentation (which can be found here for the Normal distribution).

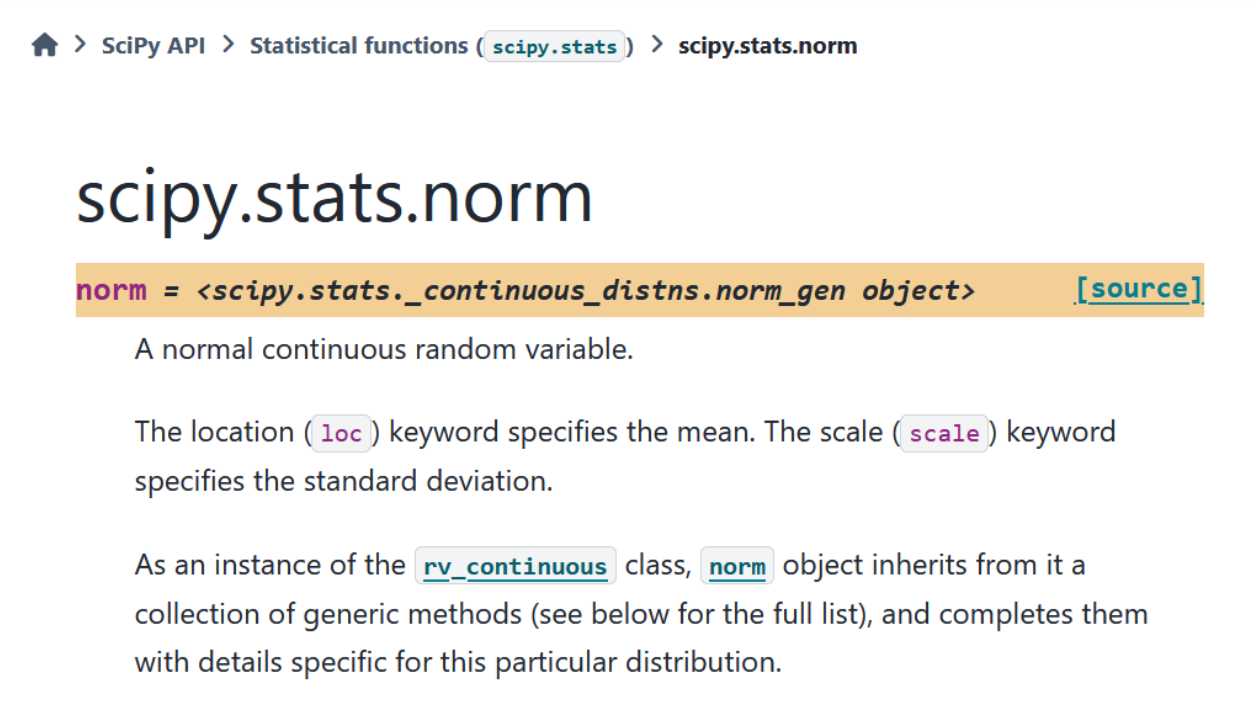


Figure 6.1: Documentation of the normal distribution

All distributions have default values for these parameters, which are typically `loc=0` and `scale = 1`.

```
# Create normal distribution object with mu=0, sigma=1
dist_norm = stats.norm(loc=0, scale=1)
```

Once a distribution object has been instantiated, we can use methods (i.e., functions) to obtain various properties of the distribution, such as its probability density function (pdf), cumulative density function (cdf) and summary statistics such as the mean, variance and median (or, more general, quantiles).

We give a list of some common methods for a distribution object named `dist_name`. We start with common functions associated with a probability distribution.

- `dist_name.pdf(x)`: Value $f(x)$ where f is the pdf of the distribution.
- `dist_name.cdf(x)`: Value $F(x)$ where F is the cdf of the distribution.

```
x = 1
print(dist_norm.pdf(x))
```

```
0.24197072451914337
```

All the above functions are *vectorized*, meaning here that they can also take one-dimensional Numpy arrays as input, in which case they return the requested value for every element in the array.

```
x = np.array([1, 3, 5.5])
print(dist_norm.pdf(x)) # Gives probability density function evaluated in x = 1, 3, 5.5
```

```
[2.41970725e-01 4.43184841e-03 1.07697600e-07]
```

We can also access various summary statistics:

- `dist_name.mean()` : Returns mean of the distribution
- `dist_name.var()` : Returns variance of the distribution
- `dist_name.median()`: Returns median of the distribution

```
dist_norm = stats.norm(loc=0,scale=2)

mean =  dist_norm.mean()
variance = dist_norm.var()
median = dist_norm.median()

print("Mean of the distribution is", mean)
print("Variance of the distribution is", variance)
print("Median of the distribution is", median)
```

```
Mean of the distribution is 0.0
Variance of the distribution is 4.0
Median of the distribution is 0.0
```

As a final application of density functions, we show how they can be used to compute convolutions.

Insight 3.19: Convolution of three distributions

Following the setting of Insight 3.19 seen in class, let $Z = X + Y + U$ with $X \sim \text{LogNormal}(\mu, \sigma^2), Y \sim \text{Exp}(\lambda)$ and $U \sim U(a, b)$, that is, Z is the sum of a log-normal, exponential and uniform distribution.

The goal is to create for a grid of z-values, the corresponding probability density $f_Z(z)$-values. Computing convolutions can be done easily with the `convolve()` function from SciPy's `signal` package. In the code below this is done in a two-step approach, which we will first elaborate on.

First, the pdf of $W = X + Y$,

$$f_W(w) = \int_{-\infty}^{\infty} f_X(x) f_Y(w - x) \text{dx},$$

of the convolution of X and Y is computed for various values of w, and then afterwards the pdf $f_Z = F_{(X+Y)+U}$ as the convolution of $W = X + Y$ and U is computed in a similar way.

Because we cannot compute the integral above analytically, we approximate it by a discrete sum over a finite range of x-values. In the code below, we take the discretized grid

$$D = \{0, 001, 002, \dots, 19.999, 20\}$$

of x-values with now the interpretation that dx $= 0.001$, so that the discrete approximation becomes

$$f_{X+Y}(w) \approx \sum_{x \in D} f_X(x) f_Y(w-x)\text{dx}.$$

To quickly compute the grid values, we can use the `arange()` function from NumPy. For three inputs a, b and step this function returns a NumPy array with the values $[a, a + \text{step}, a + 2 \cdot \text{step}, \ldots, b - \text{step}]$. Note that the value of b itself is excluded. Let us illustrate this with an example.

```
a = 2
b = 5
step = 0.2

x = np.arange(a,b,step)

print(x)
```

```
[2.  2.2 2.4 2.6 2.8 3.  3.2 3.4 3.6 3.8 4.  4.2 4.4 4.6 4.8]
```

To compute $\sum_{x \in D} f_X(x) f_Y(w-x)\text{dx}$, we evaluate the probability density function of the distributions of X and Y in the grid points in `x` in the code below and store these in `f_X` and `f_Y`. The function `convolve()` then computes the quantity

$$\sum_{x \in D} f_X(x) f_Y(w-x)$$

for many values of w (explained below). To obtain the approximation for $f_{X+Y}(w)$ we have to multiply this expression by dx, which can be considered a constant in the discrete approximation.

Because we evaluate f_X and f_Y on the domain D, all the values that the sum $W = X + Y$ can attain are in the set

$$\{0, 001, 002, \ldots, 19.999, 20, 20.001, \ldots, 39.999, 40.000\}.$$

The elements in `f_W` correspond to $f_W(w)$ for the w's in this set. We compute these w-values as well for completeness.

The above steps are then repeated for the second convolution. The final z-values, for which $f_Z(z)$ has been computed and stored in `f_Z`, are computed in the array `z`.

Although this is not a complete description of what is going on in the code below, it does illustrate that convolutions can be numerically computed with relatively small (programming) effort as opposed to a typically hard analysis of the corresponding integral analytically.

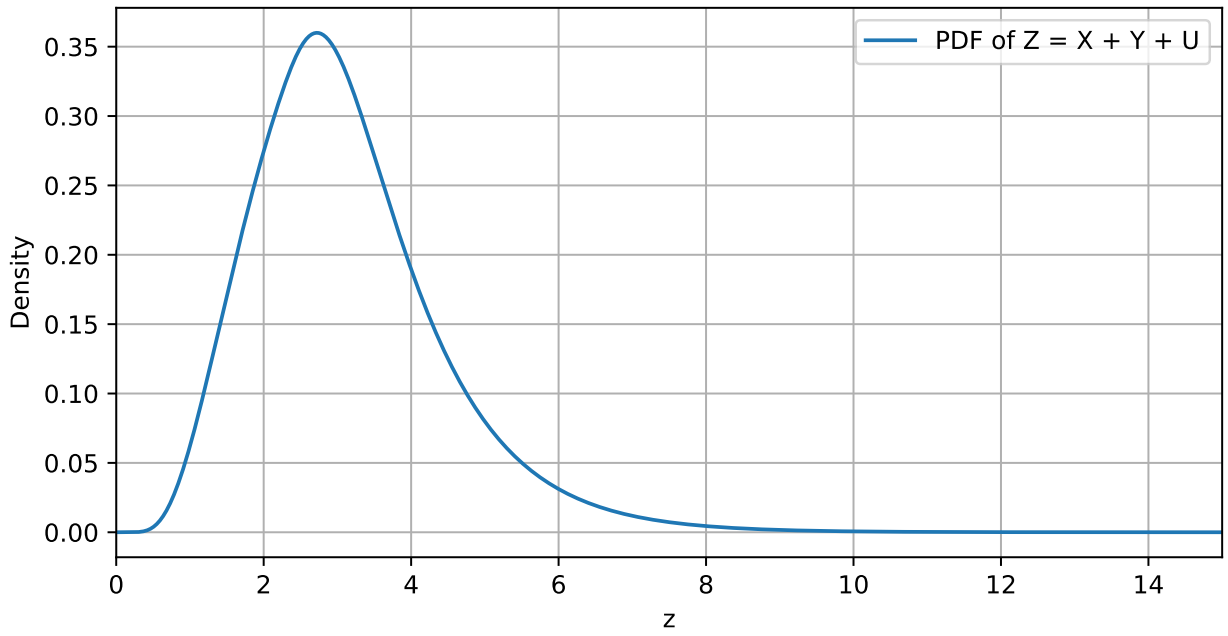
```
import numpy as np
from scipy.stats import lognorm, expon, uniform
from scipy.signal import convolve
import matplotlib.pyplot as plt

# Parameters (as in Insight 3.19)
mu = 0.0
sigma = 0.5
```

```
lam = 1.0
a, b = 0.0, 2.0

# Grid D (create with built-in function np.arange())
dx = 0.001
x_max = 20
x = np.arange(0, x_max + dx, dx)

# PDFs
f_X = lognorm.pdf(x, s=sigma, scale=np.exp(mu))
f_Y = expon.pdf(x, scale=1/lam)
f_U = uniform.pdf(x, loc=a, scale=b - a)

# First convolution: W = X + Y.
f_W = convolve(f_X, f_Y, mode="full") * dx

# Corresponding grid for W
w = np.arange(0, len(f_W)) * dx

# Second convolution: Z = (X + Y) + U = W + U:
f_Z = convolve(f_W, f_U, mode="full") * dx

# Corresponding grid for Z
z = np.arange(0, len(f_Z)) * dx
```

In the code above the final array z contains various values so that `f_Z[i]` is the pdf value of Z of the i-th element `z[i]` in `z`. We can also visualize the resulting pdf by plotting `z` against `f_Z`.

If you want to know more about plotting functions with Python, you can have a look, e.g., here .

```
# Plot
plt.figure(figsize=(8, 4))
plt.plot(z, f_Z, label="PDF of Z = X + Y + U")
plt.xlim(0, 15)
plt.xlabel("z")
plt.ylabel("Density")
plt.legend()
plt.grid()
plt.show()
```



6.2 Simulation-based inference

6.2.1 Data generation

Here come general remarks about generating data (samples) from distributions.

6.2.2 Examples from Chapter 7

Here come various examples from Chapter 7.

6.3 Interactive visualizations from class

In this section we present various interactive visualizations that were demonstrated in class. Note that every example also contains a link to a stand-alone website with the application, in case this notebook does not format them well on your device.

These applications have been constructed using Python's Vega-Altair package. You can access the code of all applications as well by clicking on the "Show code generating the plot below" button.

You can copy this code into a Jupyter notebook to run it yourself (although the sliders might appear at a different point in the figure). Furthmore, you might need to run the following line in the Anaconda prompt (if you are using Python via the Anaconda installation) so that the line `alt.data_transformers.enable("vegafusion")` in some codes does not raise an error.

```
conda install -c conda-forge vegafusion vl-convert-python
```

Example 2.33

In this section we provide an interactive application for the visualization of the bivarate Normal distribution. This tool gives the option to vary the covariance matrix entries in four different ways (each with their own characteristic). You can also control which probability contour is plotted.

The means are fixed at $\mu_X = 5$ and $\mu_Y = 50$. The reason these cannot be varied is because this only give rise to a linear translation of the elements in the application, which is mathematically not interesting.

If the app is not well-formatted on your device, you can also find it here.

```
import pandas as pd
import numpy as np
import altair as alt
from scipy.stats import chi2

# For Altair to handle large dataframes
alt.data_transformers.enable("vegafusion")


# Fixed mean values
mu_x = 5
mu_y = 50


# ------------------------------
# Interactive sliders
# ------------------------------
# Values of p for probability contours
p_values = np.arange(0.05, 1.0, 0.05)

# Slider for probability p
p_slider = alt.param(
        name="p",
        value=0.95,
        bind=alt.binding_range(min=p_values.min(),
                               max=p_values.max(),
                               step=0.05,
                               name="Probability contour for p = ")
)

# Different regimes for correlation/covariance
sigma_values = [(4,25,8),
                (4,25,-8),
                (4,25,0),
                (3,6,4)]

# Dropdown menu for sigma regimes
```

```
sigma_selector = alt.param(
    name="sigma",
    value=sigma_values[0],  # Set the default value
    bind=alt.binding_select(options=sigma_values,
                            name="Select covariance matrix  ,  ,    = ")
)


# ----------------------------
# Initial data frames
# ----------------------------
df_points = pd.DataFrame({
    "x": pd.Series(dtype="float64"),
    "y": pd.Series(dtype="float64"),
    "name": pd.Series(dtype="string"),
    "sigma": pd.Series(dtype="object"),
})

df_line = pd.DataFrame({
    "x": pd.Series(dtype="float64"),
    "y": pd.Series(dtype="float64"),
    "name": pd.Series(dtype="string"),
    "sigma": pd.Series(dtype="object"),
})

df_ellipses = pd.DataFrame({
    "x": pd.Series(dtype="float64"),
    "y": pd.Series(dtype="float64"),
    "p": pd.Series(dtype="float64"),
    "theta": pd.Series(dtype="float64"),
    "name": pd.Series(dtype="string"),
    "sigma": pd.Series(dtype="object"),
})

# ----------------------------
# Create data for frames
# ----------------------------
for sigmas in sigma_values:
    sigma_x = sigmas[0]
    sigma_y = sigmas[1]
    sigma_xy = sigmas[2]

    # Means and covariance
    mu = np.array([mu_x, mu_y])
    Sigma = np.array([[sigma_x, sigma_xy],
                      [sigma_xy, sigma_y]])
```

```
    # Random data
    np.random.seed(42)
    data = np.random.multivariate_normal(mu, Sigma, size=500)
    data_x, data_y = data[:,0], data[:,1]

    # Correlation line
    corr = np.corrcoef(data_x, data_y)[0,1]
    slope = corr * (np.std(data_y)/np.std(data_x))
    intercept = np.mean(data_y) - slope * np.mean(data_x)
    x_line = np.linspace(np.min(data_x), np.max(data_x), 100)
    y_line = intercept + slope * x_line

    df_points_new = pd.DataFrame({"x": data_x,
                                  "y": data_y,
                                  "name": "Generated data (points)",
                                  "sigma" : [sigmas] * len(data_x)})
    df_points = pd.concat([df_points, df_points_new],ignore_index=True)

    df_line_new = pd.DataFrame({"x": x_line,
                                  "y": y_line,
                                  "name": "Correlation (line)",
                                  "sigma" : [sigmas] * len(x_line)})
    df_line = pd.concat([df_line, df_line_new],ignore_index=True)

    # Precompute ellipses for all p
    def probability_ellipse(mu, Sigma, p, n_points=200):
        c = chi2.ppf(p, df=2)
        vals, vecs = np.linalg.eigh(Sigma)
        axes = np.sqrt(vals * c)
        theta = np.linspace(0, 2*np.pi, n_points)
        ellipse = vecs @ np.vstack([axes[0]*np.cos(theta), axes[1]*np.sin(theta)])
        ellipse[0] += mu[0]
        ellipse[1] += mu[1]
        return pd.DataFrame({"x": ellipse[0],
                             "y": ellipse[1],
                             "theta": theta,
                             "p": p,
                             "name": "Probability contour (ellipse)",
                             "sigma" : [sigmas] * len(ellipse[0])}).sort_values("theta")

    df_ellipses_new = pd.concat([probability_ellipse(mu, Sigma, p) for p in p_values])
    df_ellipses = pd.concat([df_ellipses, df_ellipses_new],ignore_index=True)

# ----------------------------
# Create the visualization
# ----------------------------
# Define the domain limits
```

```
x_min, x_max = -2, 12
y_min, y_max = 30, 70

# Info for legend
color_legend=alt.Scale(domain=["Generated data (points)",
                               "Correlation (line)",
                               "Probability contour (ellipse)"],
                                   range=["grey", "red", "blue"])

# Filter points within domain
df_points_filtered = df_points[
    (df_points["x"] >= x_min) & (df_points["x"] <= x_max) &
    (df_points["y"] >= y_min) & (df_points["y"] <= y_max)
]

# Now use df_points_filtered for the data points
points = alt.Chart(df_points_filtered).transform_filter(
    (alt.datum.sigma[0] == sigma_selector[0]) & # Match with selected sigma regime
    (alt.datum.sigma[1] == sigma_selector[1]) &
    (alt.datum.sigma[2] == sigma_selector[2])
).mark_circle(size=40, opacity=0.5).encode(
    x="x:Q", y="y:Q",
    color=alt.Color("name:N",
                    scale=color_legend,
                    legend=alt.Legend(title="Element type"))
)

# Line can also be filtered
df_line_filtered = df_line[
    (df_line["x"] >= x_min) & (df_line["x"] <= x_max) &
    (df_line["y"] >= y_min) & (df_line["y"] <= y_max)
]

line = alt.Chart(df_line_filtered).transform_filter(
    (alt.datum.sigma[0] == sigma_selector[0]) &
    (alt.datum.sigma[1] == sigma_selector[1]) &
    (alt.datum.sigma[2] == sigma_selector[2])
).mark_line(strokeWidth=2).encode(
    x="x:Q",
    y="y:Q",
    color=alt.Color("name:N",
                    scale=color_legend,
                    legend=alt.Legend(title="Element type"))
)

# Ellipse can be filtered similarly
```

```
df_ellipses_filtered = df_ellipses[
    (df_ellipses["x"] >= x_min) & (df_ellipses["x"] <= x_max) &
    (df_ellipses["y"] >= y_min) & (df_ellipses["y"] <= y_max)
]

ellipse = alt.Chart(df_ellipses_filtered).transform_filter(
   (alt.datum.p - p_slider) ** 2 < 1e-6,
    (alt.datum.sigma[0] == sigma_selector[0]) &
    (alt.datum.sigma[1] == sigma_selector[1]) &
    (alt.datum.sigma[2] == sigma_selector[2])
).mark_line(strokeDash=[5,5]).encode(
    x="x:Q", y="y:Q", order="theta",
    color=alt.Color("name:N",
                    scale=color_legend,
                    legend=alt.Legend(title="Element type"))
)

# Combine charts
chart = (points + line + ellipse).add_params(sigma_selector,p_slider).encode(
    x=alt.X("x:Q", scale=alt.Scale(domain=[x_min, x_max])),
    y=alt.Y("y:Q", scale=alt.Scale(domain=[y_min, y_max]))
).properties(width=500, height=350,
    title={"text": {
        "expr":
        "'Bivariate Normal distribution with probability contour for p = ' + format(p, \".2f
            "anchor": "middle"
        }
)

chart
```

```
alt.LayerChart(...)
```

Example 2.50

Here we give an interactive visualizaiton of Example 2.50 illustrating the tail behaviour of the Gamma(α, β)-Mixed Poisson distribution as α varies.

If the app is not well-formatted on your device, you can also find it here.

```
import numpy as np
import pandas as pd
import altair as alt
from scipy.stats import nbinom

# ----------------------------
# Parameters
# ----------------------------
```

```
k2 = np.arange(0, 31)
beta_nb = 1.0
p_nb = beta_nb / (beta_nb + 1)

# Range of alpha values for slider
alpha_values = np.arange(1.0, 15.1, 0.5)

# ------------------------------
# Build DataFrame
# ------------------------------
rows = []
for alpha in alpha_values:
    pmf = nbinom(n=alpha, p=p_nb).pmf(k2)
    tail = 1 - nbinom(n=alpha, p=p_nb).cdf(k2 - 1)

    for k, p, t in zip(k2, pmf, tail):
        rows.append({
            "k": k,
            "value": p,
            "type": "PMF",
            "alpha": alpha
        })
        if k > 0:  # avoid log(0)
            rows.append({
                "k": k,
                "value": t,
                "type": "Tail",
                "alpha": alpha
            })

df = pd.DataFrame(rows)

# ------------------------------
# Slider parameter
# ------------------------------

alpha_slider = alt.param(
    name="alpha",
    value=5.0,
    bind=alt.binding_range(
        min=alpha_values.min(),
        max=alpha_values.max(),
        step=1,
        name=r"Dispersion control value   =  "
    )
)
```

```
slider_chart = (
    alt.Chart(pd.DataFrame({"x": [0]}))
    .add_params(alpha_slider)
    .mark_point(opacity=0)
    .properties(
        height=10,
        title=" "
    )
)


# ----------------------------
# PMF plot
# ----------------------------
pmf_chart = (
    alt.Chart(df[df["type"] == "PMF"])
    .transform_filter(alt.datum.alpha == alpha_slider)
    .mark_line(interpolate="step-after", strokeWidth=2, color="blue")
    .encode(
        x=alt.X("k:Q", title=r"k",scale=alt.Scale(domain=[0, 30]),
                axis=alt.Axis(
                titleFontSize=15,
                titleFontWeight='lighter')),
        y=alt.Y("value:Q", title=r"P(X = x)",scale=alt.Scale(domain=[0, 0.5]),
                axis=alt.Axis(
                titleFontSize=15,
                titleFontWeight='lighter')
               )
    )
    .properties(
        title={"text": {
                        "expr": "'Gamma( =' + alpha + ', =1)-Mixed Poisson (Negative Binomial
                    },
                    "anchor": "middle"
                },
        width=225,
        height=200
    )
)

# ----------------------------
# Tail probability plot (log-log)
# ----------------------------
tail_chart = (
    alt.Chart(df[df["type"] == "Tail"])
    .transform_filter(alt.datum.alpha == alpha_slider)
```

```
    .mark_line(strokeWidth=2, color="blue")
    .encode(
        x=alt.X("k:Q", title=r"k", scale=alt.Scale(type="log"),axis=alt.Axis(
                titleFontSize=15,
                titleFontWeight='lighter',
            )),
        y=alt.Y(
            "value:Q",
            scale=alt.Scale(type="log", domain=[1e-9, 1]),
            title=r"P(X   x)",
            axis=alt.Axis(
                titleFontSize=15,
                titleFontWeight='lighter',
            )
        )
    )
    .properties(
        title={"text": {
                        "expr": "'Corresponding tail probability (Negative Binomial)'"
                    },
                    "anchor": "middle"
                },
        width=225,
        height=200
    )
)

# ----------------------------
# Final layout
# ----------------------------
final_chart = alt.vconcat(
    slider_chart,
    pmf_chart | tail_chart
)

final_chart
```

```
alt.VConcatChart(...)
```

Example 4.18

Here we show the convergence of a standardized Gamma$(k, 1)$ distribution to a Normal distribution as k increases.

If the app is not well-formatted on your device, you can also find it here.

```
import numpy as np
import pandas as pd
import altair as alt
from scipy import stats
alt.data_transformers.enable("vegafusion")

# -----------------------------
# Slider values
# -----------------------------
k_values = np.arange(2, 100, 1)  # integer k from 1 to 15
lam = 1.0  # lambda parameter

# x-axis for standardized values
x = np.linspace(-3, 3, 100)

# -----------------------------
# Build long-form DataFrames for all k values
# -----------------------------
rows_pdf = []
rows_cdf = []

for k_slider in k_values:
    # Gamma distribution
    gamma_dist = stats.gamma(a=k_slider, scale=1/lam)
    mu = gamma_dist.mean()
    sigma = gamma_dist.std()

    gamma_pdf = gamma_dist.pdf(mu + sigma * x) * sigma
    gamma_cdf = gamma_dist.cdf(mu + sigma * x)

    # Standard Normal reference
    normal_pdf = stats.norm.pdf(x)
    normal_cdf = stats.norm.cdf(x)

    # PDF rows
    for xi, gpdf, npdf in zip(x, gamma_pdf, normal_pdf):
        rows_pdf.append({"x": xi, "value": gpdf, "distribution": "Standardized Gamma", "k_sl
        rows_pdf.append({"x": xi, "value": npdf, "distribution": "Normal(0,1)", "k_slider":

    # CDF rows
    for xi, gcdf, ncdf in zip(x, gamma_cdf, normal_cdf):
        rows_cdf.append({"x": xi, "value": gcdf, "distribution": "Standardized Gamma", "k_sl
        rows_cdf.append({"x": xi, "value": ncdf, "distribution": "Normal(0,1)", "k_slider":

df_pdf = pd.DataFrame(rows_pdf)
df_cdf = pd.DataFrame(rows_cdf)
```

```
# ----------------------------
# Slider parameter
# ----------------------------
k_slider_param = alt.param(
    name="k",
    value=5,
    bind=alt.binding_range(min=k_values.min(),
                           max=k_values.max(),
                           step=1,
                           name=r"Value k = ")
)

slider_chart = (
    alt.Chart(pd.DataFrame({"x": [0]}))
    .add_params(k_slider_param)
    .mark_point(opacity=0)
    .properties(height=10)
)

# ----------------------------
# PDF chart
# ----------------------------
pdf_chart = (
    alt.Chart(df_pdf)
    .transform_filter(alt.datum.k_slider == k_slider_param)
    .mark_line()
    .encode(
        x=alt.X("x", scale=alt.Scale(domain=[-3, 3]), title="x",axis=alt.Axis(
                    titleFontSize=15, # Size
                    titleFontWeight='lighter')),
        y=alt.Y("value", scale=alt.Scale(domain=[-0.05, 0.6]), title="f(x)", axis=alt.Axis(
                    titleFontSize=15, # Size
                    titleFontWeight='lighter')),
        color=alt.Color("distribution:N", legend=alt.Legend(title="Distribution"))
    )
    .properties(width=225, height=200, title="Probability Density Function")
)

# ----------------------------
# CDF chart
# ----------------------------
cdf_chart = (
    alt.Chart(df_cdf)
    .transform_filter(alt.datum.k_slider == k_slider_param)
    .mark_line()
    .encode(
```

```
        x=alt.X("x", scale=alt.Scale(domain=[-3, 3]), title="x",axis=alt.Axis(
                    titleFontSize=15, # Size
                    titleFontWeight='lighter')),
        y=alt.Y("value", scale=alt.Scale(domain=[-0.05, 1]), title="F(x)",axis=alt.Axis(
                    titleFontSize=15, # Size
                    titleFontWeight='lighter')),
        color=alt.Color("distribution:N", legend=alt.Legend(title="Distribution"))
    )
    .properties(width=225, height=200, title="Cumulative Density Function")
)

# ----------------------------
# Combine slider + charts
# ----------------------------
final_chart = alt.vconcat(
    slider_chart,
    pdf_chart | cdf_chart
).properties(title={
                "text": {
                    "expr": "'Standardized Gamma(k=' + k + ',1) and Normal(0,1) distribution
                },
                "anchor": "middle"
            })

final_chart
```

```
alt.VConcatChart(...)
```

Example 4.28

Here we show convergence of the standardized sum of Poisson(1) random variables to a Normal distribution. Note that this concerns "convergence" of a discrete distribution to a continuous distribution.

If the app is not well-formatted on your device, you can also find it here.

```
import numpy as np
import pandas as pd
import altair as alt
from scipy import stats
# alt.data_transformers.enable("vegafusion")

# ----------------------------
# Slider values
# ----------------------------
k_values = np.arange(1, 50, 1)

# ----------------------------
```

```
# Build DataFrames separately
# ----------------------------
rows_pois = []
rows_norm = []

for k in k_values:
    # --- Poisson sum ---
    pois_dist = stats.poisson(mu=k)
    multiple = 4  # This value should be the same as the x_range later one;
                  # otherwise there are also x-values outside the plotted range
                  # which makes the legend jump around. This is a rule of thumb, though.
    s_vals = np.arange(0, k + multiple * np.sqrt(k)) # Most mass falls in this support

    z_vals = (s_vals - k) / np.sqrt(k)
    pmf_vals = pois_dist.pmf(s_vals)

    bin_width = 1 / np.sqrt(k)

    for z, p in zip(z_vals, pmf_vals):
        rows_pois.append({
            "z": z,
            "value": p / bin_width,
            "k_slider": k,
            "legend": "Standardized Poisson Sum"
        })

    # --- Normal reference ---
    x_range = multiple
    x = np.linspace(-x_range, x_range, 100)
    normal_pdf = stats.norm.pdf(x)

    for xi, yi in zip(x, normal_pdf):
        rows_norm.append({
            "z": xi,
            "value": yi,
            "k_slider": k,
            "legend": "Normal(0,1)"
        })

df_pois = pd.DataFrame(rows_pois)
df_norm = pd.DataFrame(rows_norm)

# ----------------------------
# Slider parameter
# ----------------------------
k_slider_param = alt.param(
```

```
    name="k",
    value=1,
    bind=alt.binding_range(
        min=k_values.min(),
        max=k_values.max(),
        step=1,
        name="Value k = "
    )
)

slider_chart = (
    alt.Chart(pd.DataFrame({"x": [0]}))
    .add_params(k_slider_param)
    .mark_point(opacity=0)
    .properties(height=0)
)


# ----------------------------
# Common legend properties
# ----------------------------
color_encoding = alt.Color(
    "legend:N",
    scale=alt.Scale(
        domain=["Standardized Poisson Sum", "Normal(0,1)"],
        range=["pink", "blue"]
    ),
    legend=alt.Legend(title="Distribution", orient="right")
)

# ----------------------------
# Poisson bars
# ----------------------------
# Define the width calculation
pmf_chart = (
    alt.Chart(df_pois)
    .transform_filter(alt.datum.k_slider == k_slider_param)  # Filter based on slider value
    .mark_bar(opacity=0.7,width= 25 * k_slider_param ** (-0.5)) # Set heuristically by inspe
    .encode(
        x=alt.X("z:Q", title="Standardized value", scale=alt.Scale(domain=[-x_range, x_range
                axis=alt.Axis(
                    titleFontSize=15, # Size
                    titleFontWeight='lighter')),
        y=alt.Y("value:Q", title=r"Density",
                axis=alt.Axis(
                    titleFontSize=15, # Size
```

```
                    titleFontWeight='lighter')), # Amount of bold face
        color=color_encoding
    )
)


# ----------------------------
# Normal line
# ----------------------------
normal_curve = (
    alt.Chart(df_norm)
    .transform_filter(alt.datum.k_slider == k_slider_param)
    .mark_line(size=3)
    .encode(
        x=alt.X("z:Q", scale=alt.Scale(domain=[-x_range,x_range])),
        y="value:Q",
        color=color_encoding
    )
)


# ----------------------------
# Final chart
# ----------------------------
final_chart = alt.vconcat(
    slider_chart,
    (pmf_chart + normal_curve).properties(
        title={
            "text": {
                "expr": "'Standardized sum of k = ' + k + ' Poisson(1) variables'"
            },
            "anchor": "middle"
        }, width=350, height=300
    )
)

final_chart
```

```
alt.VConcatChart(...)
```

Example 4.29

Here we show the convergence of the sum of bimodal PDFs to a Normal distribution with the same mean and variance.

If the app is not well-formatted on your device, you can also find it here.

```
import numpy as np
import pandas as pd
import altair as alt
from scipy.signal import convolve
alt.data_transformers.enable("vegafusion")

def bimodal_pdf(x, mu1, sigma1, mu2, sigma2, w1, w2):
    """Compute a bimodal PDF as a weighted sum of two normal distributions."""
    pdf1 = (1 / (np.sqrt(2 * np.pi) * sigma1)) * np.exp(-0.5 * ((x - mu1) / sigma1) ** 2)
    pdf2 = (1 / (np.sqrt(2 * np.pi) * sigma2)) * np.exp(-0.5 * ((x - mu2) / sigma2) ** 2)
    return w1 * pdf1 + w2 * pdf2

def compute_convolution(pdf1, pdf2, dx):
    """Compute the convolution of two PDFs and normalize the result."""
    conv = convolve(pdf1, pdf2, mode='full') * dx
    return conv / np.sum(conv * dx)

# Parameters for the bimodal distribution
mu1, sigma1 = 2.0, 0.5
mu2, sigma2 = 6.0, 1.0
w1, w2 = 0.5, 0.5

# Compute mean and variance of the bimodal distribution
mu_bimodal = w1 * mu1 + w2 * mu2
var_bimodal = w1 * (sigma1**2 + mu1**2) + w2 * (sigma2**2 + mu2**2) - mu_bimodal**2
sigma_bimodal = np.sqrt(var_bimodal)

# Initial x range and PDF (wide enough)
x = np.linspace(-5, 25, 100)
dx = x[1] - x[0]
y = bimodal_pdf(x, mu1, sigma1, mu2, sigma2, w1, w2)

# DataFrames for Altair plotting
df_list = []
df_normal_list = []

# n values
n_values = np.arange(1, 21, 1)

for n in n_values:
    pdf_n = y.copy()
    x_n = x.copy()

    # Perform (n-1) convolutions
    for _ in range(n - 1):
        x_n_new = np.linspace(x_n[0] + x[0], x_n[-1] + x[-1], len(pdf_n) + len(y) - 1)
```

```
        pdf_n = compute_convolution(pdf_n, y, dx)
        x_n = x_n_new

    # Compute cumulative sum for approximate CDF
    cdf_n = np.cumsum(pdf_n) * dx
    # Determine 0.01 and 0.99 quantiles
    x_min = np.interp(0.0001, cdf_n, x_n)
    x_max = np.interp(0.9999, cdf_n, x_n)

    # Mask to keep only points in [x_min, x_max]
    mask = (x_n >= x_min) & (x_n <= x_max)
    x_n_trim = x_n[mask]
    pdf_n_trim = pdf_n[mask]

    # Add bimodal convolution data
    for xi, pi in zip(x_n_trim, pdf_n_trim):
        df_list.append({"x": xi, "PDF of bimodal sum": pi, "n": n, "legend": "Sum of Bimodal

    # Normal approximation data (same x-range)
    mu_normal = n * mu_bimodal
    sigma_normal = np.sqrt(n) * sigma_bimodal
    pdf_normal = (1 / (np.sqrt(2 * np.pi) * sigma_normal)) * np.exp(-0.5 * ((x_n_trim - mu_n
    for xi, pi in zip(x_n_trim, pdf_normal):
        df_normal_list.append({"x": xi, "PDF of bimodal sum": pi, "n": n, "legend": "Normal

# Convert to DataFrames
df = pd.DataFrame(df_list)
df_normal = pd.DataFrame(df_normal_list)

# Slider
n_selector = alt.param(
    name="n",
    value=1,
    bind=alt.binding_range(min=n_values.min(),
                           max=n_values.max(),
                           step=1,
                           name="Value n = ")
)

# ------------------------------
# Common legend properties
# ------------------------------
color_encoding = alt.Color(
    "legend:N",
    scale=alt.Scale(
        domain=["Sum of Bimodal PDFs", "Normal with same mean/variance"],
```

```
        range=["pink", "blue"]
    ),
    legend=alt.Legend(title="Distribution", orient="right")
)

# Altair plot
chart_bimodal = alt.Chart(df).add_params(n_selector).transform_filter(
    alt.datum.n == n_selector
).mark_area(opacity=0.3).encode(
    x='x:Q',
    y='PDF of bimodal sum:Q',
    color=color_encoding
)

chart_normal = alt.Chart(df_normal).transform_filter(
    alt.datum.n == n_selector
).mark_line(strokeDash=[5,2]).encode(
    x='x:Q',
    y='PDF of bimodal sum:Q',
    color=color_encoding
)

(chart_bimodal + chart_normal).properties(
    title={"text":{"expr":"'Sum of n = ' + n + ' bimodal PDFs'"},"anchor":"middle"}, width=3
)
```

```
alt.LayerChart(...)
```