Python Programming for Mathematics

Julien Guillod

Sorbonne Université

2026

© 2026, Julien Guillod. License CC BY-NC-ND.

Table of contents

Preface

The purpose of this collection of exercises is to provide an introduction to the use of programming in various areas of mathematics. The programming language used is Python 3.6 or later. These exercises are used as a basis for the practical sessions LU2MA100 given at Sorbonne University in the framework of the undergraduate mathematics studies.

The exercises in this collection correspond, up to a few minor details, to those of the book *Python Programming for Mathematics*, published by CRC Press. The solutions are available in the print and ebook editions published by CRC Press.

This English version is a translation of the French edition *Programmation Python par la pratique* published by Dunod.

The notebooks used to work through the exercises are available on 

and can be launched directly online on 

.

1 Introduction

Python is a leading programming language in the scientific world. It is perfectly adapted to program mathematical problems. This book focuses on the practical use of the Python language in different areas of mathematics: sequences, linear algebra, integration, graph theory, finding zeros of functions, probability, statistics, differential equations, symbolic calculus, and number theory. Through 55 exercises of increasing difficulty, and corrected in detail, it gives a good overview of the possibilities of using programming in mathematics and to be able to solve complex mathematical problems.

It is not necessary to do the exercises in the order suggested, even if some exercises sometimes call upon notions seen in previous exercises. The more difficult exercises are indicated by exclamation marks:

- ! : longer or more difficult;
- !! : quite long and complex;
- !!! : challenge proposed without correction.

1.1 Acknowledgments

Thanks to Marie Postel and Nicolas Lantos for their careful proofreading of the first version of this manuscript and for many corrections and suggestions. Special thanks to Johann Faouzi and Louis Thiry.

Thanks also to the members of the Sorbonne University pedagogical teams who used these exercises for their feedback and contributions: Mathieu Barré, Constantin Bône, Jules Bonnard, Cédric Boutillier, Thibault Cimic, Jeanne Decayeux, Cécile Della Valle, Guillaume Duboc, Jean-Jil Duchamps, Jean-Merwan Godon, Elise Grosjean, Cindy Guichard, Sidi-Mahmoud Kaber, Nicolas Lantos, Mathieu Mari, David Michel, Leo Miolane, Anouk Nicolopoulos, Arnaud Padrol, Diane Peurichard, Marie Postel, Xavier Poulot-Cazajous, Alexandre Rege, Othmane Safsafi, Emmanuel Schertzer, Agustín Somacal, Didier Smets, Robin Strudel, Gauthier Tallec, Nicolas Thomas, Paul Vernhet, Jules Vidal, and Raphaël Zanella.

Finally, I would like to thank the students who worked on these exercises for their constructive feedback, which contributed to the improvement of this collection.

The attentive reader is thanked in advance for pointing out typos or other errors.

1.2 Why Python?

Python is a general-purpose interpreted programming language that has the particularity of being very readable and pragmatic. It has a very large base of external modules, especially scientific ones, which makes it particularly attractive for programming mathematical problems. The fact that Python is an interpreted language makes it slower than compiled languages, but it ensures a great speed of development which allows humans to work a little less while the computer has to work a little more. This particularity makes Python one of the main programming languages used by scientists.

1.3 Prerequisites

This book does not aim to explain the syntax and principles of the Python language, so the prerequisite is to know the basics. There are many resources to update yourself if needed, for example:

- the online course *Python Programming MOOC* by the University of Helsinki;
- the book *Python for Everybody* by Charles R. Severance;
- various online courses, like *Crash Course on Python*.

Moreover, the realization of the exercises requires access to a computer or an online service with Python 3.6 (or more recent) completed by the following modules: NumPy, SciPy, SymPy, Matplotlib, Numba, NetworkX, and Pandas. The use of a code editor allowing writing in Python is also highly recommended. It is suggested here to use Jupyter Lab, which allows both the writing of interactive notebooks and scripts and also the addition of one's own solutions below the statements, which is very practical. It is not necessary to use Jupyter Lab, other environments are also suitable, such as Spyder or Jupyter Notebook.

The following sections describe how to install and run the Python environment or use it online without installation.

1.4 Documentation

It is generally not useful (nor desirable) to know all the functions and subtleties of the Python language for occasional use. However, it is essential to know how to use the documentation efficiently. The official documentation is available at https://docs.python.org/. The language and version can be selected in the upper left corner. It is strongly recommended to look at how the documentation is written and to learn how to use it.

1.5 Installation

People who cannot or do not want to install Python can go directly to Section 1.6 for alternatives available online without installation.

There are basically three ways to install Python and the modules required to perform the exercises:

- *Miniforge* is a Python distribution manager. The modules required for the exercises must be installed manually. This is the preferred method on Windows or MacOS.
- *Linux repositories:* Most Linux distributions allow you to install Python and the core modules directly from the package repositories that come with them. This is the preferred method under Linux.
- *Virtual environment:* allows you to manage several versions of Python and its modules. This method provides finer and more advanced control over the installed modules than the previous methods.

1.5.1 Installation with Miniforge

The easiest way to install Python 3 and all the necessary dependencies on Windows and MacOS is to install Miniforge and then install the required modules manually. Detailed documentation is available here. In summary, the installation procedure is as follows:

1. Download Miniforge from https://conda-forge.org/download/.
2. Double-click the downloaded file to start the Miniforge installation, then follow the installation procedure.
3. Once the installation is complete, launch Miniforge Prompt from the Start menu or the application list.
4. In the terminal, type the command:

```
conda install numpy scipy sympy matplotlib numba networkx pandas
 ↪  jupyterlab jupyterlab-lsp python-lsp-server
```

1.5.2 Installing from repositories

Most Linux distributions allow to easily install Python and the most standard modules directly from the distribution repositories. The following procedure is for Ubuntu, but can be easily adapted to other distributions.

1. Install Python 3:

```
sudo apt install python3 python3-pip
```

2. Update Pip:

```
pip install --upgrade pip
```

3. Install the modules NumPy, SciPy, SymPy, Matplotlib, Numba, NetworkX, and Pandas:

```
sudo apt install python3-numpy python3-scipy python3-sympy
 ↪  python3-matplotlib python3-numba python3-networkx python3-pandas
```

4. Jupyter Lab is not available in the Ubuntu packages, so it must be installed with Pip:

```
pip install jupyterlab
```

5. Optionally (but recommended), install the LSP (Language Server Protocol) interface with the command:

```
pip install jupyterlab-lsp python-lsp-server[all]
```

1.5.3 Advanced installation

The following procedure describes how to install modules manually with the Pip package manager in a virtual environment.

1. If Python is not already installed by your operating system, install it from https://www.python.org/downloads/.
2. Create a virtual environment by entering the following command in a terminal:

```
python -m venv .venv
```

3. Activate the virtual environment on Windows by entering:

```
.venv\Scripts\activate.bat
```

or on MacOS and Linux by entering:

```
source .venv/bin/activate
```

4. Install the required modules by entering the following command in a terminal:

```
pip install numpy scipy sympy matplotlib numba networkx pandas
 ↪  jupyterlab jupyterlab-lsp python-lsp-server[all]
```

1.6 Launch of Jupyter Lab

1.6.1 On the command line

With Miniforge, launch Miniforge Prompt from the Start menu or application list. In other cases, simply open a terminal (if a virtual environment has been created, don't forget to activate it). To launch Jupyter Lab from the command line, type `jupyter lab` in the terminal. To quit, click on `Shutdown` in the `File` menu of the Jupyter Lab window. It is also possible to type `Ctrl+C` followed by `y` in the terminal where the command `jupyter lab` was executed.

Remark. If the default browser is installed as a Snap package (as it is by default on Ubuntu, for example), the browser reports that the file is inaccessible. To solve this problem, run the following command in a terminal:

```
echo "c.NotebookApp.use_redirect_file = False" >>
 ↪  ~/.jupyter/jupyter_notebook_config.py
```

1.6.2 Online without installation

For people who cannot or do not want to install Python and the necessary dependencies on their own computer, it is possible to use Jupyter Lab online with launch binder. No account is required, but modified documents are automatically deleted on exit, so it's essential to save them on your own computer before leaving. Otherwise, various services offer the possibility to use Jupyter Lab for free after creating an account:

- CoCalc
- Google Colaboratory

1.7 Use of Jupyter Lab

Once Jupyter Lab is launched, the window shown in Figure 1.1 should appear in a browser.

Jupyter Lab essentially allows us to process three types of documents: *notebooks*, *scripts*, and *terminals*. A notebook consists of cells that can contain either code or text in Markdown format. Code cells can be evaluated interactively on demand, which allows great flexibility. Text cells can contain comments, titles, or LaTeX formulas as represented in Figure 1.2.

A Python script is simply a text file containing Python instructions. It is executed in its entirety from A to Z and it is not possible to interact interactively with it during its execution (unless it has been explicitly programmed). To execute a Python script, it is necessary to open a terminal.

Basic commands

- Create a new file: click on the "+" button on the top left, then choose the type of file to create.

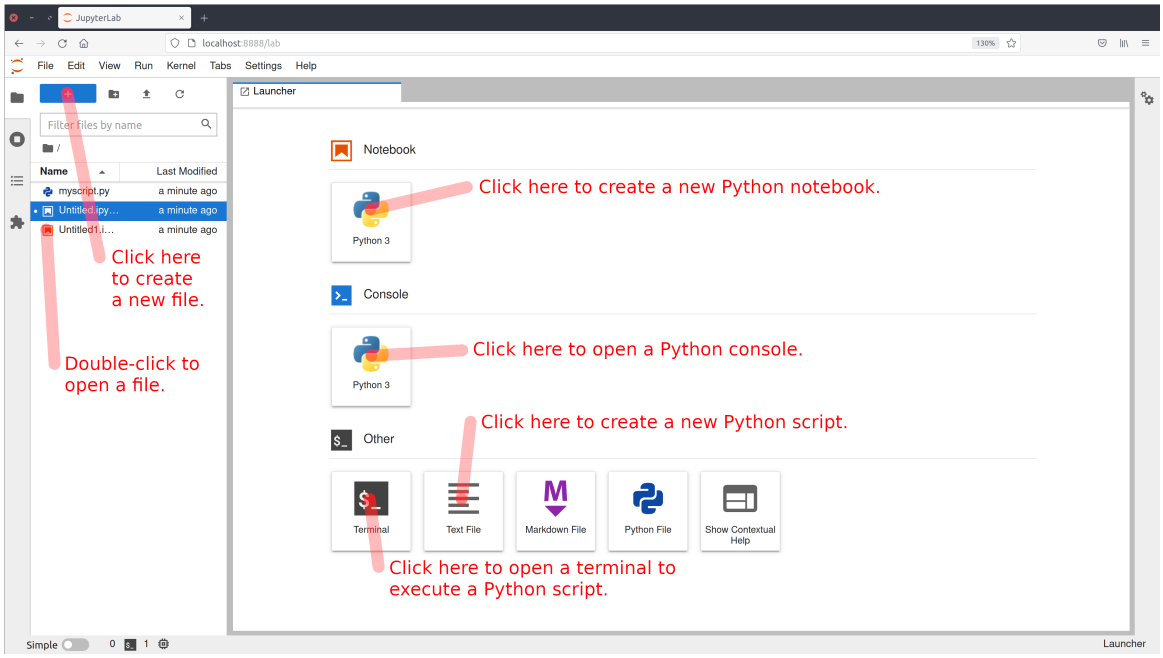


Figure 1.1: Jupyter Lab launch window.

- Rename a file: click with the second mouse button on the title of the notebook (either in the tab or in the file list).
- Change cell type: drop-down menu to choose between “Code” and “Markdown”.
- Execute a code cell: combination of keys `SHIFT+ENTER`.
- Format a text cell: combination of keys `SHIFT+ENTER`.
- Edit a text cell: double-click on the cell.
- Run a script: type `python scriptname.py` in a terminal to run the script `scriptname.py`.
- Rearrange cells: click and drop.
- Juxtaposing tabs: click and drop.

The detailed documentation of Jupyter Lab is available here.

1.8 Advanced use of Jupyter Lab

Recent versions of Jupyter Lab (3 and higher with ipykernel greater than 6) feature a particularly useful debugger and LSP (Language Server Protocol) interface. The debugger allows you to find errors in the code by stopping the program at particular points to understand what’s going on. Documentation and a tutorial on how to use the debugger are available here. The LSP interface provides access to documentation and function signatures, offers code diagnostics and autocompletion. Information on installing and using the LSP interface is available here.

Debugger When writing code, it’s natural to make mistakes, and one important aspect is to locate and identify them efficiently. To do this, it’s possible to put `print` commands in the right places, but it’s more appropriate to use a debugger for this. To activate Jupyter Lab’s debugger,

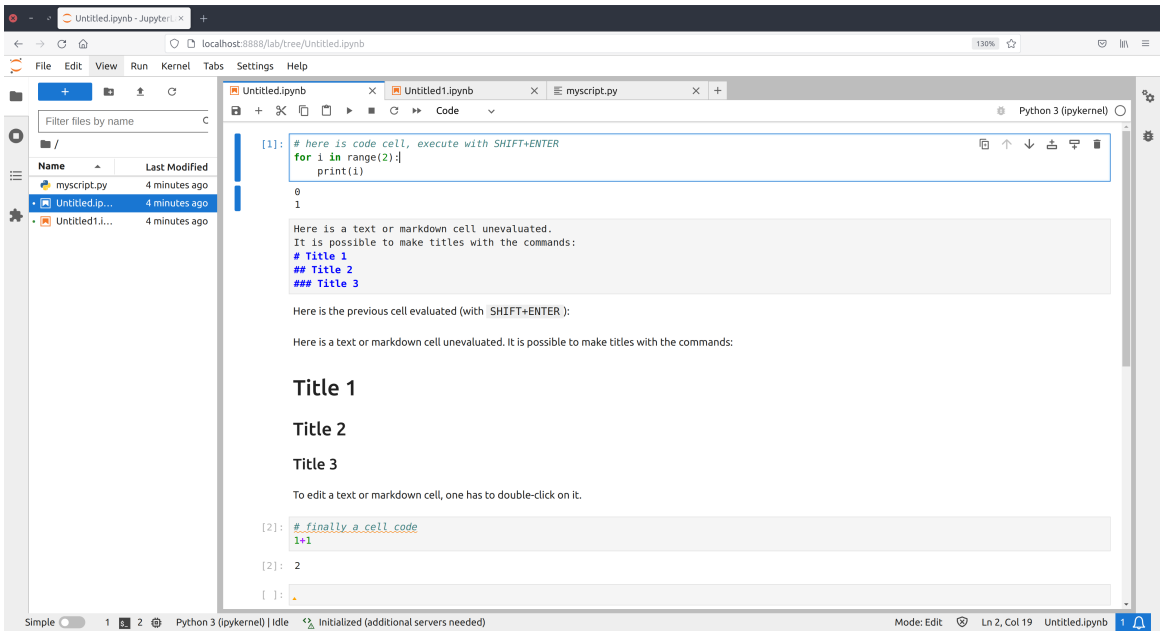


Figure 1.2: A Jupyter notebook is composed of several cells. Here a code cell is followed by an unevaluated text cell, then the same cell is evaluated, and finally a last code cell.

click on the beetle in the top right-hand corner so that it turns orange. When the debugger is activated, the list of global variables is available in the dedicated bar. The most useful aspect of the debugger is the definition of breakpoints, which allow you to execute the code up to a certain line and inspect the state of the program at that point. To do this, consider the following function, which adds two numbers:

```
def add(a, b):
    res = a + b
    return res
```

Clicking to the left of a code line number places a breakpoint indicated by a red dot. Here, we propose to click on the second line performing the addition. By executing the following function call code:

```
resultat = add(1, 2)
print(resultat)
```

```
3
```

the program will stop at the second line of the `add` function. You can view the values of variables `a` and `b` in the "Variables" tab and the relevant source code in the "Sources" tab. Breakpoints are grouped together in the "Breakpoints" tab. By navigating the "Callstack" tab, you can continue program execution up to the next breakpoint.

Hover When hovering over any part of the code with the mouse, if a part of the code becomes underlined it is then possible to get information about the function with the CTRL key. For example, by hovering the mouse over the following code:

```
from numpy import linalg
```

and by pressing the CTRL key with the mouse on numpy or linalg a window with explanations about these modules is displayed. This is also the case for manually defined functions if they contain a docstring:

```
def square(x):
    """Definition of the function x -> x^2"""
    return x*x
```

Moving the mouse over the word square:

```
r = square(4)
```

underlines it, and with the key CTRL the definition appears.

Diagnostics Critical errors or warnings are indicated by an underline in red or orange, for example, in the case of an undefined variable:

```
def f(x):
    if x:
        undefined_variable
    return x
```

Suggestions By typing linalg. in a cell, suggestions of functions available in this module are displayed. In other cases, the suggestions are activated with the TAB key. This is the case, for example, with a manually defined dictionary:

```
dic = {'key1':3, 'key2':5}
```

By typing dic[in a cell followed by the TAB key, the suggestions 'key1' and 'key2' come up.

Signatures By typing linalg.solve(then comes the help and signature of this function, *i.e.*, the way the arguments are to be used in this function. By placing the mouse on the word solve with the CTRL key, there comes also a description of the function.

References By clicking on a symbol, its other uses are highlighted.

Definition By clicking with the right mouse button on a symbol and then on "Jump to definition", it is possible to go to the definition of the function in question. It is possible to test on the following code for example:

```
f(None)
```

Renaming It is possible to rename a variable intelligently (*i.e.*, without renaming local variables, for example) by right-clicking on the variable in question and selecting "Rename symbol".

Diagnostics panel It is possible to sort and navigate through the diagnostics using the "Diagnostics panel". To open it, simply select "Show diagnostics panel" from the context menu of a cell (right mouse button).

Personalization The “Settings” menu of Jupyter Lab allows you to customize the working environment, especially to choose the theme, the font size, the default indentation, but also many other more advanced options.

2 Data structures

To represent data structures, Python offers four basic types: lists (type `list`), tuples (type `tuple`), sets (type `set`), and dictionaries (type `dict`). The purpose of this chapter is to show the fundamental differences between these data structures and to explain what they are best suited for. Detailed documentation on data structures is available here.

Concepts covered

- data structures (list, tuple, set, dictionary)
- mutable and immutable types
- hashable type
- list, set and dictionary comprehensions
- numerical series

Exercise 2.1 - Lists

A list is a structure allowing to store heterogeneous elements:

```
list0 = [0, 5.4, "string", True]
```

Lists are mutable, *i.e.*, it is possible to modify an element, add one or delete one, without having to redefine the whole list.

```
list0[3] = False # replace True by False
list0.append("new") # add the string "new" to the list
list0.insert(2, 34) # insert 34 in place of 2
list0.remove(0) # remove 0
list0
```

```
[5.4, 34, 'string', False, 'new']
```

In particular, care must be taken when copying a list. If we execute the following code:

```
list1 = list0
list1[2] = "change"
list0
```

```
[5.4, 34, 'change', False, 'new']
```

then `list0` is also modified and is equal to `list1`. To create a real copy, you have to use the following code:

```
list2 = list0.copy()
list2[2] = "rechange"
list0
```

```
[5.4, 34, 'change', False, 'new']
```

which does not modify `list0`. Note that it is possible to modify the elements of a list inside a function:

```
def f(l):
    l[0] = 0
f(list0)
list0
```

```
[0, 34, 'change', False, 'new']
```

Finally, it is possible to create lists with the help of list comprehension:

```
list1 = [2*i+1 for i in range(10)]
list1
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

a. Search the documentation for the syntax to concatenate two lists.

> **Hint**
>
> See the documentation here.

b. Look in the documentation for the syntax to extract a slice from a list, *i.e.*, if a is, for example, a list of length 10, return the elements from 6 to 9.

> **Hint**
>
> See the documentation here.

c. Search the documentation for the syntax to return the length of a list.

d. Write a function `fibonacci(N)` that returns the list of N first terms of the Fibonacci sequence defined by $u_{n+2} = u_{n+1} + u_n$ with $u_0 = 0$ and $u_1 = 1$.

e. Write a function `pascal(N)` that returns the N-th line of Pascal's triangle:

$$
\begin{array}{ccccccccccc}
 & & & & & 1 & & & & & \\
 & & & & 1 & & 1 & & & & \\
 & & & 1 & & 2 & & 1 & & & \\
 & & 1 & & 3 & & 3 & & 1 & & \\
 & 1 & & 4 & & 6 & & 4 & & 1 & \\
1 & & 5 & & 10 & & 10 & & 5 & & 1
\end{array}
$$

f. Let $(u_n)_{n\in\mathbb{N}}$ and $(v_n)_{n\in\mathbb{N}}$ be the sequences defined by $u_0 = 1$, $v_0 = 1$, and

$$u_{n+1} = u_n + v_n\,, \qquad\qquad v_{n+1} = 2u_n - v_n\,,$$

for $n \geq 0$. Calculate u_{100} and v_{100}.

g. Write a function `vk(n0,K)`, which for two integers n_0 and $K \geq 1$ computes the sequence of values v_k defined by $v_0 = n_0$ and

$$v_{k+1} = \begin{cases} 3v_k + 1 & \text{if } v_k \text{ is odd,} \\ \frac{v_k}{2} & \text{if } v_k \text{ is even,} \end{cases}$$

for $0 \leq k < K$. For $K = 1\,000$ and various values of $n_0 \in \{10, 100, 1\,000, 10\,000\}$, display the last five calculated values, *i.e.*, $(v_{K-4}, v_{K-3}, v_{K-2}, v_{K-1}, v_K)$.

Exercise 2.2 - Tuples

Tuples allow, just like lists, to store heterogeneous elements:

```
tuple0 = (0, 5.4, "string", True)
```

But unlike lists, tuples are not mutable. It is not possible to modify an element, add one or delete one, without redefining the whole tuple. The advantage of a tuple over a list is that it is hashable, *i.e.*, it can be used as a key in a dictionary.

Finally, it is possible to assign variables inside a tuple, for example:

```
(a,b) = (1,9)
```

This is especially useful for exchanging two variables without having to use an additional variable:

```
(a,b) = (b,a)
```

a. Check that a tuple is immutable.

b. Define a function `mdlast(lst,val)` having as argument a list of integer tuples `lst` and an integer `val` and return the list of tuples with the last element of each tuple replaced by `val`. For example, if `lst = [(10, 20), (30, 40, 50, 60), (70, 80, 90)]`, then `mdlast(lst,100)` should return `[(10, 100), (30, 40, 50, 100), (70, 80, 100)]`.

c. How to convert a tuple into a list and vice versa?

Exercise 2.3 - Sets

Sets are used to store heterogeneous elements in the mathematical sense of set theory:

```
set0 = {0, 5.4, "string", True}
```

It is possible to test if an element belongs to a set:

```
if "string" in set0:
    print("inside")
```

```
inside
```

Sets are mutable, so it is possible to add or remove an element from a set:

```
set0.add(18) # add 18 to the set
set0.add(0) # add 0 to the set (this does nothing as 0 already belongs to
 ↪  the set)
set0.remove("string") # remove "string to the set
```

On the other hand, sets can only contain hashable elements, *i.e.*, immutable. In particular a set cannot contain another set:

```
set1 = { {1,2}, {3}, {4} }
```

```
TypeError: cannot use 'set' as a set element (unhashable type: 'set')
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[23], line 1
----> 1 set1 = { {1,2}, {3}, {4} }
TypeError: cannot use 'set' as a set element (unhashable type: 'set')
```

Note that in Python there are also immutable sets, called `frozenset`:

```
frozenset0 = frozenset([0, 5.4, "string", True])
```

A string can be transformed into a set:

```
set1 = set('abracadabra')
```

As with lists, it is possible to make set comprehensions:

```
set2 = {x for x in 'abracadabra' if x not in 'abc'}
```

In this example, strings are automatically transformed into sets. Note that the empty set is defined by `set()`.

a. Define a function `divisible(n)` that returns the set of integers divisible by `n` less than or equal to 100.

b. Search the literature to find the intersection, union, and difference of two sets. Determine the numbers less than or equal to 100 that are not divisible by 2 but divisible by 3 and 5.

> **Hint**
>
> See the documentation of `set` here.

Exercise 2.4 - Dictionaries

Dictionaries are a structure allowing to store heterogeneous elements indexed by keys (also heterogeneous):

```
dict0 = {"apples": 0, "pears": 4, 12: 2}
```

The elements of a dictionary are accessible through the keys:

```
dict0["apples"]
dict0[12]
```

```
2
```

A dictionary can be seen as an associative array associating to each key a value. The list of keys and the list of values are accessible, respectively, with `dict0.keys()` and `dict0.values()`. Dictionaries are mutable, so it is possible to modify a key-value association and to add or remove one:

```
dict0["apples"] = 3 # modify the value associated to apples
dict0["oranges"] = "many" # add oranges as key with value "many"
del dict0["pears"] # remove the key pears, hence the value
dict0.pop("apples") # remove the key apples, hence the value
```

```
3
```

Although a dictionary is mutable, the keys that compose it must be hashable objects, *i.e.*, immutable. Thus, a list or a set cannot be used as keys in a dictionary:

```
dict0[list0] = "test"
dict0[set0] = "retest"
```

```
TypeError: cannot use 'list' as a dict key (unhashable type: 'list')
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[33], line 1
----> 1 dict0[list0] = "test"
      2 dict0[set0] = "retest"
TypeError: cannot use 'list' as a dict key (unhashable type: 'list')
```

On the other hand, it is possible to have a tuple or a frozenset as a key:

```
dict0[tuple0] = "test"
dict0[frozenset0] = "rest"
```

hence the interest of frozensets. As for lists and sets, it is possible to make dictionary comprehensions:

```
dict1 = {x: x**2 for x in range(5)}
```

Finally, an interesting thing about dictionaries is the unpacking illustrated by the following example:

```
def add(a=0, b=0):
    return a + b
d = {'a': 2, 'b': 3}
add(**d)
```

```
5
```

a. How to define an empty dictionary?

b. How to concatenate several dictionaries together?

c. We consider a list of words:

```
words = ['Apricot', 'Cranberry', 'Pineapple', 'Banana', 'Blackcurrant',
 ↪  'Cherry', 'Lemon', 'Clementine', 'Quince', 'Date', 'Strawberry',
 ↪  'Raspberry', 'Pomegranate', 'Gooseberry', 'Persimmon', 'Kiwi',
 ↪  'Litchi', 'Mandarin', 'Mango', 'Melon', 'Mirabelle', 'Nectarine',
 ↪  'Orange', 'Grapefruit', 'Papaya', 'Peach', 'Pear', 'Apple', 'Plum',
 ↪  'Grape']
```

Write a function `position(words, x, n)` that returns the list of words with the character `x` as their `n`-th letter (starting from zero, as in Python).

d. Assuming that the list of words is very long, then each time the `position` function is evaluated, the whole set of words is searched, which takes quite a long time. To improve this, build a dictionary `mots_dict` having as keys the tuples `(x,n)` and as values the list of words having the character `x` as `n`-th letter, *i.e.*, such that `mots_dict[x,n]` returns the same thing as `position(words, x , n)` except for the order. Thus, the `words` list is traversed only once during dictionary construction and then dictionary evaluation is extremely fast for any query.

3 Homogeneous structures

Python's default data structures can handle heterogeneous data (for example, integers and strings). This feature makes Python data structures extremely flexible at the expense of performance. Indeed, since heterogeneous data must be supported, it is not possible to allocate a fixed memory range for a data structure, which slows down its use. Particularly in mathematics, homogeneous datasets of fixed size (list of integers, real or complex vectors, matrices...) appear very regularly. The NumPy module defines the `ndarray` type that is optimized for such homogeneous data structures of fixed sizes. The NumPy documentation is available here.

To load the NumPy module, it is usual to proceed as follows:

```
import numpy as np
```

Concepts covered

- homogeneous data table
- slicing
- vector operations
- indexing and selection

Exercise 3.1 - Introduction to NumPy

Creation. The size and type of the elements of a NumPy array must be known in advance. The first way to create a NumPy array is to construct an array filled with zeros by specifying the size and type:

```
array0 = np.zeros(3, dtype=int) # vector of 3 integers
array1 = np.zeros((2,4), dtype=float) # array of floats of size 2x4
array2 = np.zeros((2,2), dtype=complex) # square matrix of complex numbers
 ↪  of size 2x2
array3 = np.zeros((5,6,4)) # three-dimensional array of floats
```

The second way is to pass the data directly:

```
array4 = np.array([1,4,5]) # vector of integers
array5 = np.array([[1.1,2.2,3.3,4.4],[1,2,3,4]]) # matrix of floats of size
 ↪  2x4
array6 = np.array([[1+1j,0.4],[3,1.5]]) # matrix of complex numbers of size
 ↪  2x2
```

NumPy will then determine itself the type and the size of the array. Note that it is possible to force the type:

```
array0 = np.array([1,4,5], dtype=complex) # vector of complex numbers
```

The type of the elements of the NumPy array `array1` can be determined by `array1.dtype`. The size of this array is given by `array1.shape`. The following commands are used to access the array elements:

```
array4[1] # return 4
array5[1,3] # return 4.0
```

```
np.float64(4.0)
```

Note that the indices start at 0 and not at 1. NumPy arrays are mutable in the sense that the data can be modified while keeping the same type and size:

```
array0[1] = 4
array1[1,3] = 3.3
array3[3,4,2] = 3
```

Slicing. Slicing allows you to access certain parts of a table:

```
array4[2:3] # return the elements of indices between 2 and 3
array1[0,:] # return the first row of array1
array1[:,-1] # return the last column of array1
array3[3,3:5,1:4] # return the corresponding sub-matrix
```

```
array([[0., 0., 0.],
       [0., 3., 0.]])
```

Iteration. It is possible to iterate an array on its first dimension, for example, to return the sum of the rows:

```
for i in array5:
    print(np.sum(i))
```

```
11.0
10.0
```

a. Study the documentation for the `arange` function and use this function to generate the vectors $(5, 6, 7, 8, 9)$ and $(3, 5, 7, 9)$.

> **Hint**
>
> The documentation of the `arange` function is available here.

b. Study the documentation for the function `linspace` and use it to generate 10 evenly spaced numbers in the interval $[2, 5]$.

c. Read the documentation for the `reshape` function and perform the following transformations in succession:

$$(1,2,3,4,5,6) \to \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \to \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \to \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$$

Exercise 3.2 - Operations on arrays

The basic arithmetic operations on NumPy arrays are performed element by element:

```
mat1 = np.array([[1,2.5,3],[5,6.1,8],[3,2,5]])
mat2 = np.array([[1,0.5,0],[0,0.9,8],[2,0,0]])
mat1 + mat2 # return the sum element by element
mat1 * mat2 # return the product element by element (not the matrix
 ↪  product)
10*mat1**2 # return 10 times the square of the elements of mat1
```

```
array([[ 10. ,  62.5,  90. ],
       [250. , 372.1, 640. ],
       [ 90. ,  40. , 250. ]])
```

Most of the mathematical functions defined by NumPy (see https://numpy.org/doc/stable/reference/routines.math.html) are also performed element by element:

```
np.cos(mat1) # return the cosine element by element of mat1
np.exp(mat1) # return the exponential element by element of mat1
```

```
array([[2.71828183e+00, 1.21824940e+01, 2.00855369e+01],
       [1.48413159e+02, 4.45857770e+02, 2.98095799e+03],
       [2.00855369e+01, 7.38905610e+00, 1.48413159e+02]])
```

The matrix product can be performed in one of three ways:

```
np.dot(mat1,mat2)
mat1.dot(mat2)
mat1 @ mat2
```

```
array([[ 7.  ,  2.75, 20.  ],
       [21.  ,  7.99, 48.8 ],
       [13.  ,  3.3 , 16.  ]])
```

a. Given a vector $(v_0, v_1, \dots, v_{n-1})$, the discrete derivative of this vector is defined by the vector $(d_0, d_1, \dots, d_{n-2})$ given by $d_i = v_{i+1} - v_i$ for $i = 0, 1, \dots, n-2$.

Write a function `diff_list` that computes the discrete derivative of a list and a function `diff_np` that does the same operation but on NumPy vectors using slicing.

b. Let `a_list` and `a_np` be, respectively, a list and an array of 1 000 elements drawn at random in the interval $[0, 1]$:

```
a_list = [np.random.random() for _ in range(1000)]
a_np = np.random.random(1000)
```

Compare the execution time of `diff_list(a_list)` and `diff_np(a_np)`.

Hint

In Jupyter Lab, it is very easy to determine the time taken by a cell to evaluate itself, just start the cell with `%%time`, for example:

```
%%time
result = diff_list(a_list)
```

```
CPU times: user 48  s, sys: 3  s, total: 51  s
Wall time: 55.3  s
```

To evaluate the cell multiple times and average the execution time to get a more accurate result, replace `%%time` with `%%timeit`. The documentation is available here.

Exercise 3.3 - Vandermonde matrix

For $p, n \in \mathbb{N}^*$ and $x = (x_1, \dots, x_p)$ a vector of size p, the corresponding Vandermonde matrix is defined by:

$$V(x,n) = \begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} & x_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & x_{p-1} & x_{p-1}^2 & \cdots & x_{p-1}^{n-1} & x_{p-1}^n \\ 1 & x_p & x_p^2 & \cdots & x_p^{n-1} & x_p^n \end{pmatrix}.$$

a. Write a function that constructs the matrix $V(x,n)$ element by element using a double loop.

b. After establishing a relationship to write the k-th column of $V(x,n)$ solely as a function of x and k, write a second function that constructs the matrix $V(x,n)$ column by column using this relationship.

c. After establishing a relationship between the k-th column of $V(x,n)$, its $(k-1)$-th column, and the vector x, write a third function that constructs the matrix $V(x,n)$ column by column using this relationship.

d. Compare the execution times of these three functions for $n = 150$, $p = 100$, and x generated randomly.

Exercise 3.4 - Array indexing !

Slicing allows you to select blocks in an array, but it is also possible to select disparate elements using an array as indexing:

```
a = np.arange(12)**2 # array of perfect squares
i = np.array([1,3,8,5]) # array of indices
a[i] # array of the elements with indices i
```

```
array([ 1,  9, 64, 25])
```

Note that it is also possible to index by an array of higher dimension. The result is then an array of the same shape as the index:

```
j = np.array([[3,4],[9,7]]) # two-dimensional array of indices
a[j] # select the elements with indices j
```

```
array([[ 9, 16],
       [81, 49]])
```

For a multi-dimensional array:

```
b = np.array([[0,1,2,3],[4,5,6,7],[8,9,10,11]])
i = np.array([0,1,2,2]) # array of first indices
j = np.array([1,0,3,1]) # array of second indices
b[i,j] # select the elements of indices ij
```

```
array([ 1,  4, 11,  9])
```

Finally, it is possible to index an array by an array of Booleans:

```
c = np.array([[0,1,2,3],[4,5,6,7],[8,9,10,11]])
cond = (c >= 5) # array of Booleans defined by True if >= 5 and False
 ↪  otherwise
c[cond] = 5 # assign the value 5 to all entries greater than 5
```

For the following, we consider the numbers:

```
[0.9602, -0.99, 0.2837, 0.9602, 0.7539, -0.1455, -0.99, -0.9111, 0.9602,
 ↪  -0.1455, -0.99, 0.5403, -0.99, 0.9602, 0.2837, -0.99, 0.2837, 0.9602]
```

```
[0.9602,
 -0.99,
 0.2837,
 0.9602,
 0.7539,
 -0.1455,
 -0.99,
 -0.9111,
 0.9602,
 -0.1455,
```

```
-0.99,
0.5403,
-0.99,
0.9602,
0.2837,
-0.99,
0.2837,
0.9602]
```

as the results of a measurement made every 0.1 second at times between 2 and 3.7 seconds.

a. Since the measurements are supposed to be positive, change the data to 0 when the values are negative.

b. Calculate the times for which the previous measurements are maximum.

c. For each maximum measure, return the previous measure, the maximum measure, and the next measure. If the previous or the next measure does not exist, replace them with `np.nan`.

4 Plotting

Plotting results from numerical or symbolic calculations is essential to analyze or interpret them. The Matplotlib module allows making very varied visualizations. The documentation of Matplotlib is available here.

To use it, it is usual to import it like this:

```
import matplotlib.pyplot as plt
```

and often NumPy is also very useful:

```
import numpy as np
```

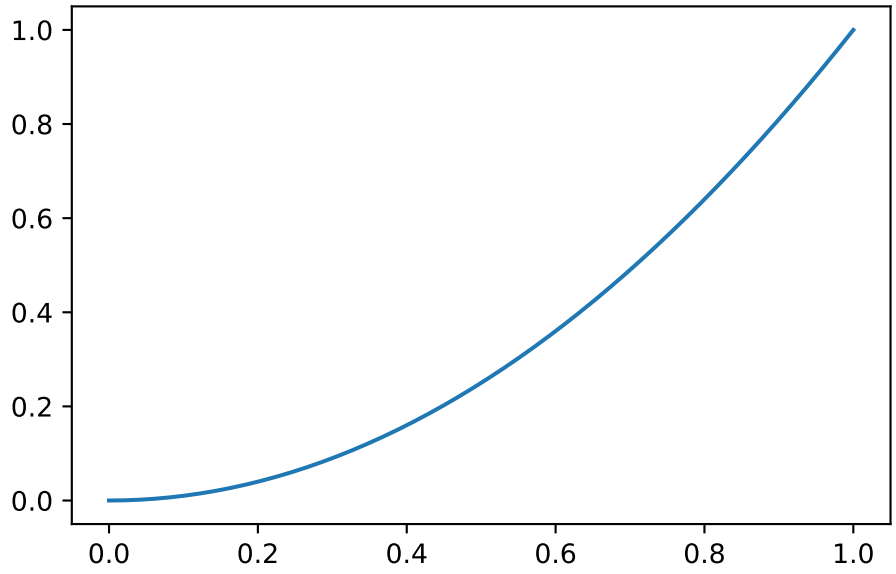
Concepts covered

- one-dimensional plots
- two- and three-dimensional plots
- cobweb diagram
- bifurcation diagram
- optimization by parallelization

Exercise 4.1 - Plots

For example, the function `plot` can be used to represent the function x^2:

```
x = np.linspace(0,1,50)
y = x**2
plt.plot(x,y)
plt.show()
```



In order to define a nice figure that can be exported as in Figure 4.1, the syntax is as follows:

```
plt.figure(figsize=(8,5)) # size of the figure (in inches)
plt.title(r'Plot of $x^2$') # title of the figure (LaTeX code can be
 ↪  included)
plt.xlabel(r'$x$') # label of the horizontal axis
plt.ylabel(r'$y$') # label of the vertical axis
plt.plot(x, y, marker='o', label=r"$x^2$") # legend
plt.legend() # display the legend
plt.savefig("test.pdf") # export the figure to PDF
plt.savefig("test.png", dpi=100) # export the figure to PNG
plt.show()
```

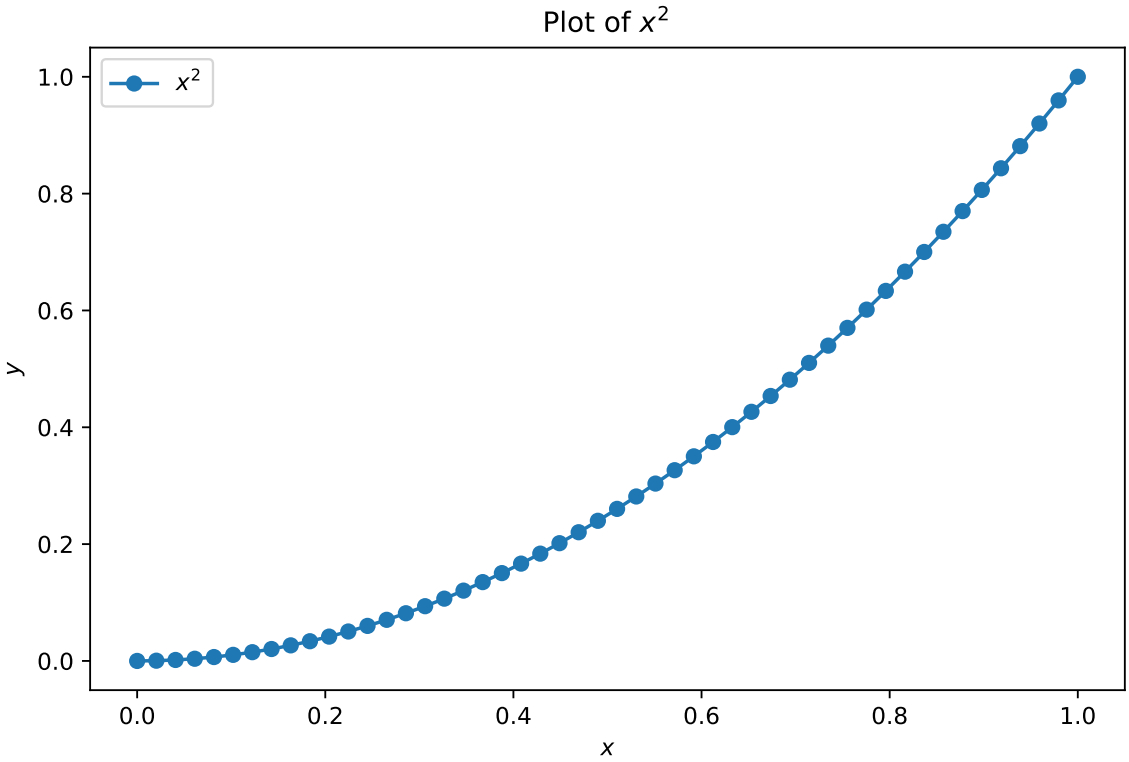


Figure 4.1: Plot of x^2.

a. Plot the functions $\sin(kx)$ and $\cos(kx)$ for $k = 1, 2, 3$ on $x \in [0, 2\pi]$ in the same figure. Make the graduations on the horizontal axis all $\frac{\pi}{2}$, as in Figure 4.2.

> **Hint**
>
> Use the `xticks` function described here.

b. Look at the help for the `imshow` function and use it to plot a matrix of random numbers in $[0, 1]$ of size 10×10 as in Figure 4.3.

c. Plot the density and the contour lines of the function $f(x, y) = \frac{-y}{5} + e^{-x^2-y^2}$ for $x \in [-3, 3]$ and $y \in [-3, 3]$ as in Figure 4.4.

Exercise 4.2 - Deterministic chaos

The goal is to study an extremely elementary model of evolution of a population. In spite of its simplistic character, this model presents many very interesting properties, in particular chaotic ones.

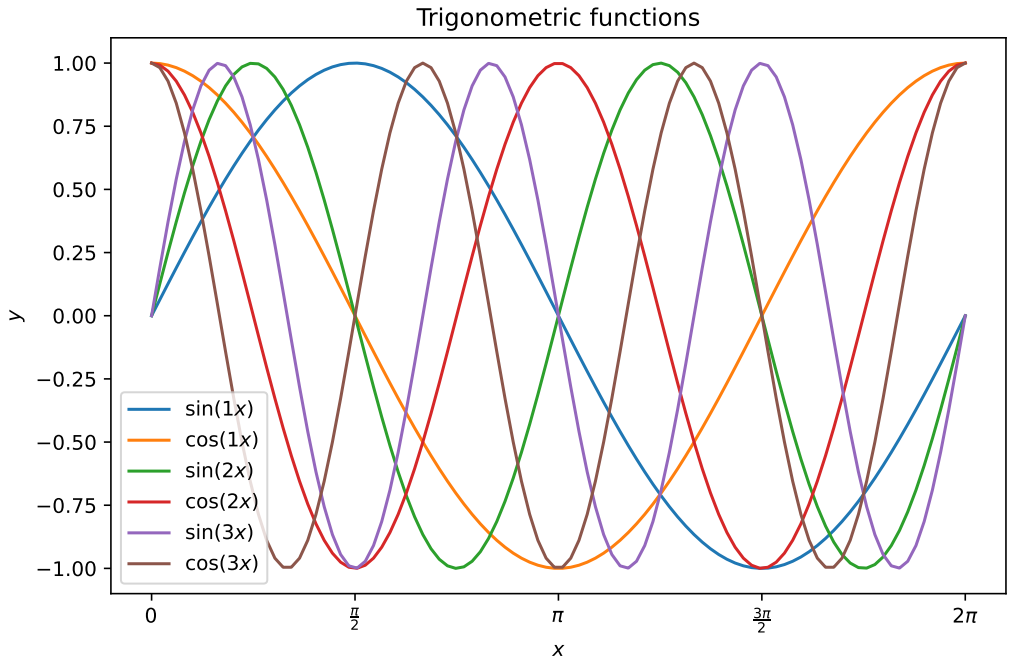


Figure 4.2: Plot of $\sin(kx)$ and $\cos(kx)$ for $k = 1, 2, 3$.

The proportion of a population at a discrete time $i \in \mathbb{N}$ is noted $x_i \in [0, 1]$. The evolution of the population is given by $x_{i+1} = f_\mu(x_i)$, where $f_\mu : [0, 1] \to [0, 1]$ is the function defined by:

$$f_\mu(x) = \mu x(1 - x)\,.$$

The parameter $\mu \in [0, 4]$ describes the population growth. The application f_μ is called the logistic map of parameter μ.

a. Define the function f_μ in Python as `f(x, mu)` $= f_\mu(x)$ and plot it for several values of μ.

b. Why can't the value of μ be greater than 4 in the model?

c. Define a function that takes as argument a value of μ, an initial data $x_0 \in [0, 1]$ and a number $n \in \mathbb{N}$ and returns the list $S_\mu^n = (x_0, x_1, x_2, x_n)$. Modify this function so that it can take an optional parameter $m \in \mathbb{N}$ and return the list $S_\mu^{m,n} = (x_m, x_{m+1}, \dots, x_n)$, *i.e*, remove the first $m - 1$ elements from S_μ^n.

d. Test this function by making a graphical representation of the list S_μ^n for different values of the parameters x_0 and μ. Observe the different behaviors of the sequence.

Cobweb diagram. One way to study more specifically what happens when μ varies is to make a cobweb diagram that consists in connecting by straight lines the points:

$$\big\{(x_0, 0), (x_0, x_1), (x_1, x_1), (x_1, x_2), (x_2, x_2), \dots, (x_n, x_n), (x_n, x_{n+1})\big\}.$$

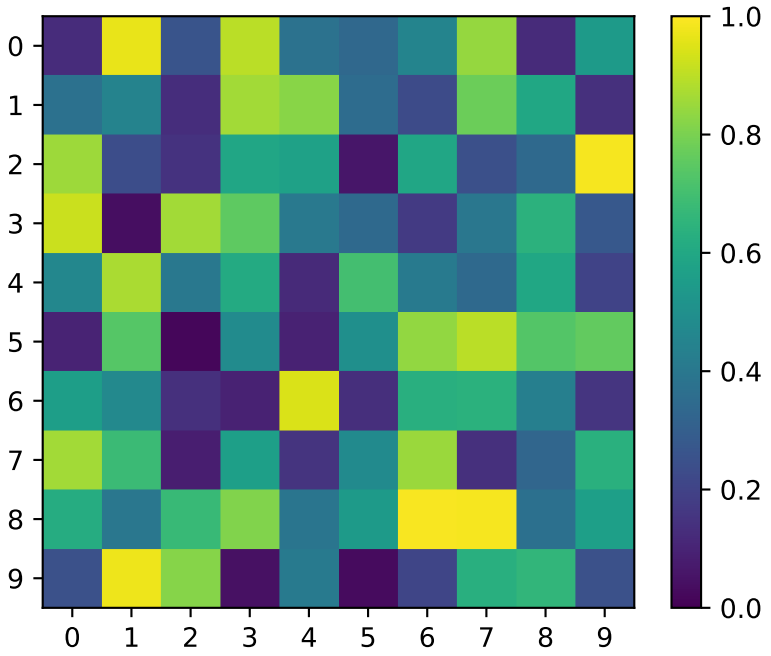


Figure 4.3: Plot of a random matrix. The colors correspond to the values of the entries of the matrix.

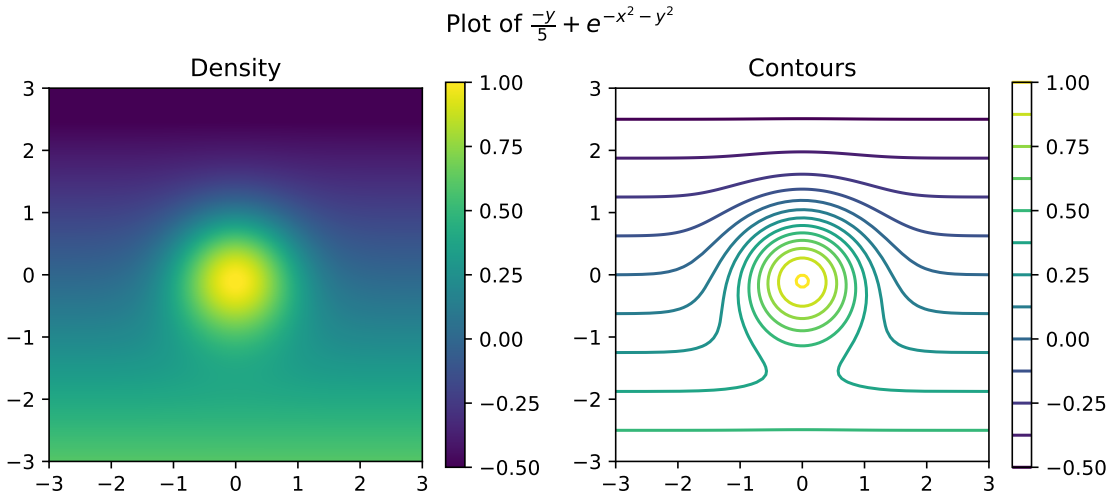


Figure 4.4: Density and contour lines of the function f.

e. Define a function that returns the list of points needed to build the cobweb diagram.

> **Hint**
>
> In order to use Matplotlib afterwards, it is sometimes simpler to represent a list of points $\{(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)\}$ as the vector of x-coordinates and the vector of y-coordinates, *i.e.*, $(x_0, x_1, \dots, x_n)$ and $(y_0, y_1, \dots, y_n)$.

f. Define a function that draws the graph of the function f_μ, the graph of the identity function, as well as the segments connecting the points of the previous list.

g. By experimenting, study qualitatively the effects of the parameters and x_0. Describe the behavior observed when μ increases.

Bifurcation diagram. The previous experiments suggest that the behavior of the sequence x_i in large time (*i.e.*, when i is large) is independent of the choice of the initial condition x_0 but depends a lot on the value of the parameter μ. The aim of this section is to represent graphically for each value of μ the set of points $S_\mu^{m,n} = (x_m, x_{m+1}, \dots, x_n)$, *i.e.*, by putting μ on the horizontal axis and all the values of $S_\mu^{m,n}$ on the vertical axis. A good choice of parameters to clean up the diagram and keep only the long-time behavior of the system is $n = 200$ and $m = 100$, for example.

h. Define a function that for a given list L of values of μ, an initial data $x_0 \in [0, 1]$ and integers $m, n \in \mathbb{N}$ returns the list of points $\{(\mu, x) : x \in S_\mu^{m,n} \text{ for } \mu \in L\}$.

i. Define a function that plot this list of points. It is suggested to take for L a list of 1 000 values in the interval $[0, 4]$.

> **Hint**
>
> Use the `scatter` function of Matplotlib with `s=1` and `edgecolor='none'`.

j. Interpret the obtained diagram, in particular what it says about the long-time behavior of the system. Determine approximately for which values of μ the system :

- has zero as its only fixed point;
- has a unique nonzero fixed point;
- oscillates between two distinct values (cycle of length two);
- oscillates between four distinct values (cycle of length four);
- oscillates between three distinct values (cycle of length three).

Representation of the attractor. For values of μ close to 4, the values of the population x_i seem to be more or less random. However, the system is purely deterministic in the sense that for a given initial value x_0, the population x_i is defined without randomness. This *a priori* random behavior is called deterministic chaos. In order to notice that the points x_i are not randomly determined, the goal is to represent graphically the points (x_n, x_{n+1}) to see that x_{n+1} is not random at all with respect to x_{n+1}.

k. For each given value of μ, define a function that returns the list of points:

$$\{(x_m, x_{m+1}), (x_{m+1}, x_{m+2}), \dots, (x_n, x_{n+1})\},$$

l. Plot these points for different values of μ. For example, $n = 5000$ and $m = 100$ is a good choice of parameters.

m. How would the previous plot look like if each x_i were drawn randomly in the interval $[0, 1]$ independently of x_{i-1}?

Exercise 4.3 - Mandelbrot set

The Mandelbrot set is defined as the set of points $c \in \mathbb{C}$ for which the sequence of complex numbers defined recursively by $z_0 = 0$ and

$$z_{n+1} = z_n^2 + c,$$

is bounded. It is possible to show that $c \in \mathbb{C}$ is in the Mandelbrot set if and only if $|z_n| \leq 2$ for any integer n.

a. Write a function `mandelbrot(c)` that checks if the point $c \in \mathbb{C}$ is in the Mandelbrot set approximately by testing the first hundred iterations.

b. Test the previous function with $c = 0$ and $c = 1 + i$. What is expected theoretically?

> **Hint**
>
> For example, the complex number $2 + 3i$ is defined in Python as `2+3j`.

c. Write a function `mandelbrot_set(N)` that generates an array of size N representing the set $c \in \{x + iy : x \in [-2, 2] \text{ and } y \in [-2, 2]\}$ and that returns an array of Booleans of size N determining if the associated point is in the Mandelbrot set or not.

d. Using the previous function with $N = 100$, plot with `imshow` an approximation of the set of points belonging to the Mandelbrot set.

e. Adapting the previous functions, plot the logarithm of the number of iterations required before $|z_n| \leq 2$ is no longer satisfied instead of a Boolean, as represented in Figure 4.5.

f. !! The previous method has the disadvantage of computing each value of c sequentially, which makes the evaluation rather slow. Propose a new implementation allowing to compute in parallel all the values using NumPy indexing.

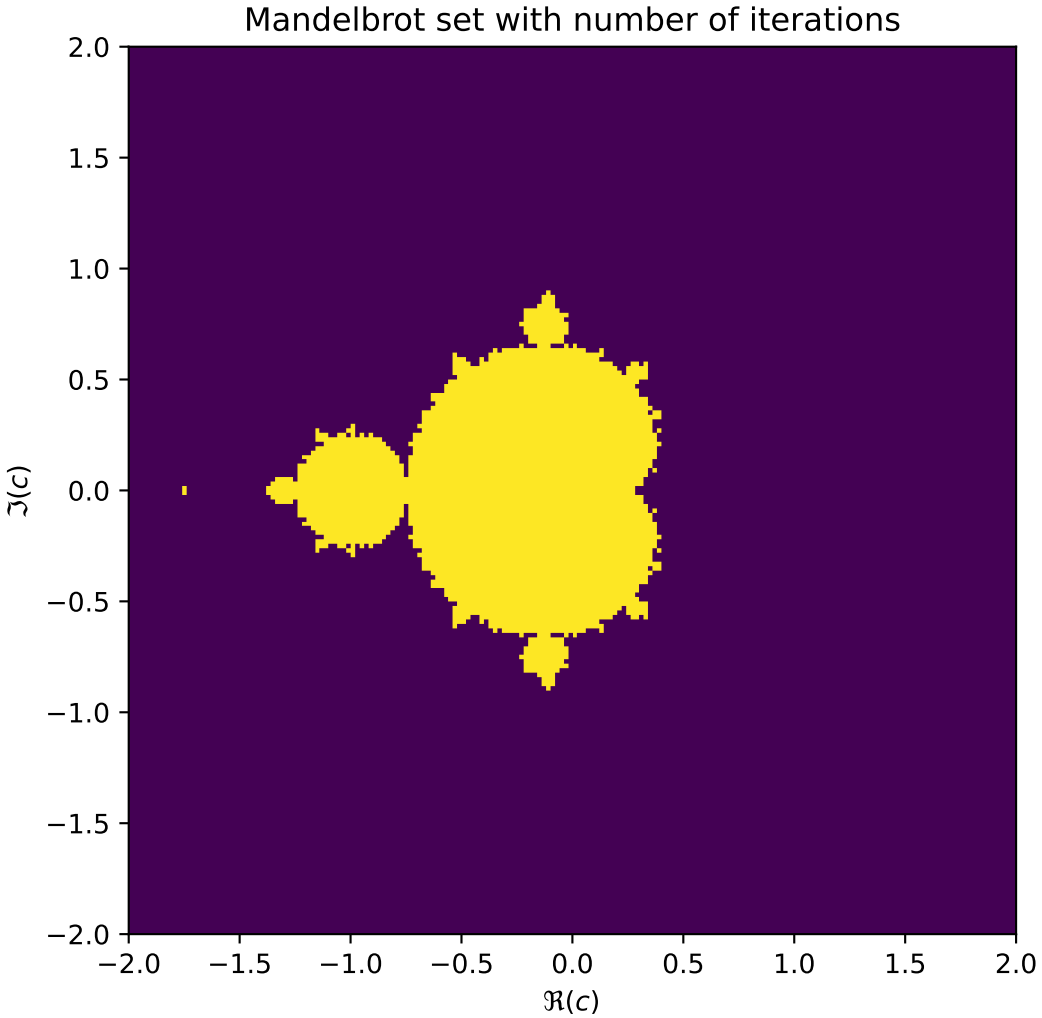


Figure 4.5: Graphical representation of the Mandelbrot set, where the color represents the logarithm of the number of iterations needed to get out of the disk of diameter two.

Exercise 4.4 - Advanced graphics !

The purpose of this exercise is to discover a range of possibilities offered by Matplotlib.

a. Draw the stream lines of the vector field of the Van der Pol oscillator:

$$\begin{pmatrix} y \\ -x + \mu(1 - x^2)y \end{pmatrix}$$

for different values of $\mu \in \mathbb{R}$.

b. Plot the parametric curve:

$$\begin{pmatrix} (1 + t^2)\sin(2\pi t) \\ (1 + t^2)\cos(2\pi t) \\ t \end{pmatrix}$$

for $t \in [-2, 2]$.

c. Plot the function of two variables:

$$f(x, y) = \sin\left(\sqrt{x^2 + y^2}\right)$$

in three dimensions for $x \in [-5.5]$ and $y \in [-5.5]$.

d. !! Represent the given Möbius strip as a parametric surface:

$$\begin{pmatrix} (3 + v\cos(u/2))\cos u \\ (3 + v\cos(u/2))\sin u \\ v\sin(u/2) \end{pmatrix}$$

for $u \in [0, 2\pi]$ and $v \in [-1, 1]$.

e. !! Look at the examples available here and choose two to understand and modify.

5 Integration

The goal is to obtain an approximation of a definite integral of the type:

$$J = \int_a^b f(x)\,\mathrm{d}x$$

for some function $f : [a,b] \to \mathbb{R}$ too complicated to *a priori* determine the value of J by hand. Deterministic and probabilistic approximation methods will be introduced to obtain an approximation $\tilde{J}$ of J.

Concepts covered

- classical methods (rectangles, trapezoids, and Simpson)
- Monte Carlo method
- convergence speed
- statistics

Exercise 5.1 - Rectangle rule

The rectangle rule is based on the definition of the integral in the Riemann sense. The first step is to split the interval $[a,b]$ into N intervals $[x_n, x_{n+1}]$ of the same size $\delta = \frac{b-a}{N}$, *i.e.*, $x_n = a + n\delta$ for $n \in \{0, 1, \dots, N-1\}$. The second step consists in assuming that the function f is constant on each interval $[x_n, x_{n+1}]$, thus to make the approximation:

$$J_n = \int_{x_n}^{x_{n+1}} f(x)\,\mathrm{d}x \approx \delta f(\tilde{x}_n)\,,$$

for $\tilde{x}_n$ a certain value to choose in the interval $[x_n, x_{n+1}]$. The choice of $\tilde{x}_n$ can, for example, be done by $\tilde{x}_n = x_n + \alpha\delta$ with $\alpha \in [0,1]$. Finally, the approximation of J is given by the sum of the approximations of J_n,

$$\tilde{J} = \sum_{n=0}^{N-1} \delta f(\tilde{x}_n)\,.$$

Assuming that $f \in C^1([a,b])$, it is then possible to show that the rectangle rule converges and that its speed of convergence is of order one. A numerical method is of order k if the error between the numerical approximation and the exact result is of order N^{-k}.

a. Choose a continuous function $f : [a,b] \to \mathbb{R}$ and define the corresponding Python function `f(x)`. To test the code, it is wise to choose a function f whose integral can be easily computed by hand.

> **Hint**
>
> The list of basic mathematical functions available in Python in the `math` module is available here. Note that NumPy also defines mathematical functions, see the documentation here.

b. Write a function `rectangles(f,a,b,N)` that returns the approximation of the integral J by the rectangle rule, for example, by choosing $\tilde{x}_n = x_n$, *i.e.*, the left edge of the interval $[x_n, x_{n+1}]$.

> **Hint**
>
> It is not necessary to store all the values of the approximations of J_n, but it is possible to increment a variable for each approximation of J_n.

c. Modify the previous function so that it takes an optional parameter `alpha` determining the choice of parameter $\alpha \in [0,1]$.

d. Write a function `plot_rectangles(f,a,b,N,alpha=0.5)` that graphically represents the approximation by the rectangle rule.

e. Determine empirically the speed of convergence of the rectangle rule as a function of N.

f. Determine analytically the convergence of the rectangle rule. What are the necessary assumptions on f?

> **Hint**
>
> Use the mean value theorem.

Exercise 5.2 - Trapezoidal rule

The trapezoidal rule is based on a linear approximation on each interval $[x_n, x_{n+1}]$, more specifically:

$$J_n = \int_{x_n}^{x_{n+1}} f(x)\,\mathrm{d}x \approx \delta \frac{f(x_n) + f(x_{n+1})}{2}.$$

a. Write a Python function `trapezes(f,a,b,N)` that returns the approximation of the integral J by the trapezoidal rule. Test the function `trapezes(f,a,b,N)` for different functions f.

b. Is your implementation of the function `trapezes(f,a,b,N)` optimal in terms of the number of evaluations of f performed compared to the number of evaluations needed? An optimal implementation of the function `trapezes(f,a,b,N)` should perform $N+1$ evaluations of f.

c. Determine empirically the speed of convergence of the trapezoidal rule as a function of N.

d. ! Determine analytically the convergence of the trapezoidal rule. What are the necessary assumptions on f?

Exercise 5.3 - Monte Carlo method

The Monte Carlo method (named after casinos, not a person) is a probabilistic approach to approximate the value of an integral. The basic idea is that the integral J can be seen as the expectation of a uniform random variable X on the interval $[a,b]$:

$$J = \int_a^b f(x)\,\mathrm{d}x = (b-a)\mathbb{E}(f(X))\,.$$

By the law of large numbers, this expectation can be approximated by the empirical mean:

$$\tilde{J} = \frac{b-a}{N}\sum_{i=0}^{N-1} f(x_i)\,,$$

where x_i are drawn randomly in the interval $[a,b]$ with a uniform probability distribution.

a. Write a function `montecarlo(f,a,b,N)` that determines an approximation $\tilde{J}$ of J by the Monte Carlo method.

> **Hint**
>
> To generate a vector of random numbers, the `random` sub-module of NumPy can be useful, see the documentation here.

b. Modify the previous function, so that it returns in addition to the mean $\tilde{J}$ also the empirical variance:

$$\tilde{V} = \frac{(b-a)^2}{N}\sum_{i=0}^{N-1}\left(f(x_i) - \frac{\tilde{J}}{b-a}\right)^2\,.$$

c. Study empirically the convergence of the Monte Carlo method as a function of N by making for each value of N a statistic on k executions. More precisely, this consists in making k evaluations of $\tilde{J}$ by the Monte Carlo method and to calculate the mean and the variance of the k results obtained.

d. Determine analytically the convergence of the Monte Carlo method. What are the necessary assumptions on f?

Hint

Use the central limit theorem.

Exercise 5.4 - Simpson's rule !

Simpson's rule consists in approximating the function f on each interval $[x_n, x_{n+1}]$ by a polynomial of degree 2. The most natural choice is the polynomial p_n of degree 2 passing through the points $(x_n, f(x_n))$, $(\frac{x_n+x_{n+1}}{2}, f(\frac{x_n+x_{n+1}}{2}))$, and $(x_{n+1}, f(x_{n+1}))$.

a. Determine the explicit form of the polynomial p_n.

Hint

The polynomial $L(x) = \frac{(x-c)(x-b)}{(a-c)(a-b)}$ takes the value 1 at $x = a$ and the value 0 at $x = b$ and $x = c$. Make a linear combination of three such polynomials.

b. Compute the approximation given by $J_n \approx \int_{x_n}^{x_{n+1}} p_n(x)\,\mathrm{d}x$.

Hint

It is possible to calculate this integral by hand or to do it with the SymPy module, see the documentation here.

c. Simplify by hand the sum $\tilde{J}$ of the approximations of J_n.

d. Write a function `simpson(f,a,b,N)` to approximate J with Simpson's rule.

e. Compare the accuracy, *i.e.*, the convergence speed of the rectangle, trapezoid, and Simpson methods as a function of N for a smooth function and the function $f(x) = \sqrt{1-x^2}$ on $[0,1]$ (whose integral is $\frac{\pi}{4}$).

f. If not already done, propose a parallel implementation of Simpson's rule using NumPy indexing.

Exercise 5.5 - Integration with SciPy !!

The above and other integration methods are defined in the `integrate` module of SciPy. This module allows in particular to handle more complicated cases: singular, generalized, or multidimensional integrals.

a. Define a function `E(n,x)` computing numerically the following integral:

$$E_n(x) = \int_1^\infty \frac{e^{-xt}}{t^n}\mathrm{d}t\,.$$

Hint

Read the documentation of the SciPy `integrate` sub-module available here.

b. Determine an approximation of the double integral:

$$I = \int_0^{\pi} \left(\int_0^{y} x \sin(xy) \, \mathrm{d}x \right) \mathrm{d}y \, .$$

6 Algebra

First, a method for solving a linear system by a direct algorithm is studied, then an iterative method will be used to determine the eigenvector associated to the largest eigenvalue of a matrix. Finally, the groups generated by a set of permutations will be studied.

Concepts covered

- direct solver (LU decomposition)
- *in place* algorithm
- iterative solver (iterated power)
- groups of permutations
- orbit and stabilizer

Exercise 6.1 - LU decomposition

The LU decomposition consists in decomposing a matrix A of size $n \times n$ into the form $A = LU$, where L is a lower triangular matrix with 1 on the diagonal and U an upper triangular matrix. Once the decomposition $A = LU$ of a matrix is known, it is then very easy to solve the linear problem $Ax = b$ by solving first $Ly = b$ then $Ux = y$. The advantage of the LU decomposition over the Gauss algorithm, for example, is that once the LU decomposition is known, it is possible to solve the linear system quickly with different right-hand sides.

Since $l_{ik} = 0$ if $k > i$, we have:

$$a_{ij} = \sum_{k=1}^{n} l_{ik} u_{kj} = l_{ii} u_{ij} + \sum_{k=1}^{i-1} l_{ik} u_{kj} \,,$$

and therefore as $l_{ii} = 1$:

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \,.$$

On the contrary, since $u_{kj} = 0$ if $k > j$, then:

$$a_{ij} = \sum_{k=1}^{n} l_{ik} u_{kj} = l_{ij} u_{jj} + \sum_{k=1}^{j-1} l_{ik} u_{kj} \,,$$

and therefore if $u_{jj} \neq 0$:

$$l_{ij} = \frac{1}{u_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right) .$$

Thus, if the first $(i-1)$ rows of U and the first $(i-1)$ columns of L are known, it is possible to determine the i-th row of U by:

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \, , \quad j \geq i \, ,$$

then, the i-th column of L by:

$$l_{ji} = \frac{1}{u_{ii}} \left(a_{ji} - \sum_{k=1}^{i-1} l_{jk} u_{ki} \right) , \quad j > i \ .$$

This algorithm for the LU decomposition of a matrix A requires that u_{ii} is never zero. This is indeed the case when the matrix A and all its principal submatrices are invertible.

a. Write a function `LU(A)` that returns the result of the LU decomposition of a matrix.

b. Given the LU decomposition of a matrix A, write a function `solve(L,U,b)` that solves the linear system $Ax = b$.

c. Write a function `LU_inplace(A)` that does not create new arrays for L and U but modifies A so that its lower triangular part (without the diagonal) matches L and its upper triangular part (with the diagonal) matches U. Also make a version `solve_inplace` that takes as input the output of `LU_inplace` and returns the solution x, without using any new arrays.

d. Using the LU decomposition of the matrix A, write a function `det(A)` that returns its determinant.

Exercise 6.2 - Power iteration method

The goal of this exercise is to determine the eigenvector of a matrix associated to the largest eigenvalue (in modulus), assuming that this one is unique. Given a real matrix A of size $n \times n$ and a vector $x_0 \in \mathbb{R}^n$, the sequence of vectors $(x_k)_{k \in \mathbb{N}}$ is defined by:

$$x_{k+1} = \frac{A x_k}{\| A x_k \|} \, .$$

Assuming, for example, that the matrix A is diagonalizable, it is then possible to show that the sequence $(x_k)_{k \in \mathbb{N}}$ converges up to a sign to the eigenvector associated to the largest eigenvalue of A in absolute value. The convergence takes place almost surely for all choices of x_0.

a. Define a function `power(A, x0, k)` that returns x_k. With this function, determine the largest eigenvector of the matrix:

$$A = \begin{pmatrix} 0.5 & 0.5 \\ 0.2 & 0.8 \end{pmatrix}.$$

b. Determine the eigenvalue associated with the previous eigenvector.

> **Hint**
>
> If v is a normalized eigenvector of A, then the associated eigenvalue is given by $\lambda = Av \cdot v$.

c. Write a function to automatically compute the largest eigenvalue (in modulus) and the associated eigenvector of a square matrix with a given precision. We will choose the initial vector x_0 randomly.

d. Assuming the matrix A is diagonalizable with a single eigenvalue of largest modulus, show that the sequence x_k converges up to one sign to the eigenvector associated with this largest eigenvalue for almost all choices of x_0.

> **Hint**
>
> Decompose x_0 in the eigenvector basis of A. For simplicity, we can assume that the eigenvalue of largest modulus is positive.

e. Look at the NumPy documentation to find the functions to compute the eigenvectors and eigenvalues of a matrix.

f. Compare the speed of the previous code and the NumPy functions for matrices of size $n \times n$ with $n = 10, 100, 1\,000$.

> **Hint**
>
> Taking for example matrices whose coefficients are randomly generated in the interval $(0, 1)$, the Perron-Frobenius theorem ensures the existence of a single eigenvalue of maximum modulus that is positive.

Exercise 6.3 - Exponential of matrices

The goal of this exercise is to develop an algorithm to calculate the exponential of a real square matrix. If A is a real square matrix, its exponential is defined by the series:

$$\exp(A) = \sum_{k=0}^{\infty} \frac{A^k}{k!},$$

by analogy with the exponential on real numbers. Here, A^k represents the matrix product of A with itself k times.

a. Define the NumPy arrays corresponding to the matrices A_1 and A_2 defined by:

$$A_1 = \begin{pmatrix} 1 & 0.8 & 0.6 \\ 0.8 & 0.2 & 0.8 \\ 0 & 1.2 & 0.9 \end{pmatrix}, \qquad A_2 = \begin{pmatrix} 2 & 3 & 2 \\ 1 & 2 & 3 \\ 4 & 3 & 5.2 \end{pmatrix}.$$

b. Define a function `matrix_power(A,n=20)` returning an approximation of $\exp(A)$ obtained by keeping only the first $n+1$ terms of the series, *i.e.*, the terms from $k=0$ to $k=n$.

c. Test on the matrices A_1 and A_2 and compare with the results of the function `expm` of the module `linalg` of SciPy.

d. Without using the `norm` function of NumPy or SciPy, define a function computing the Frobenius norm $\|A\|_F$ of a matrix A of size $m \times m$ defined by:

$$\|A\|_F^2 = \operatorname{tr}(AA^t) = \sum_{i=1}^{m}\sum_{j=1}^{m} |a_{ij}|^2 .$$

Compute the Frobenius norms of the matrices A_1 and A_2.

e. For matrices A_1 and A_2, plot the error in the Frobenius norm between the result of `matrix_power` and the result of `expm` as a function of the number of terms n kept. Put a logarithmic scale on the y-axis.

From a theoretical point of view, it is possible to show that the error is bounded by:

$$\left\| \exp(A) - \sum_{k=0}^{n} \frac{A^k}{k!} \right\|_F \leq \frac{e^{\|A\|_F}}{(n+1)!} \|A\|_F^{n+1} .$$

f. Plot this bound as a function of n for different values of the $\|A\|_F$ ranging from 2 to 20, with also a logarithmic scale on the y-axis. Roughly deduce the number of terms to keep so that the bound is lower than the machine precision of 10^{-15} if $\|A\|_F = 20$. Compare the theoretical bound with what was observed in the previous question.

A basic idea to improve the convergence of the series when the norm of the matrix is large is to perform a rescaling using the relation:

$$\exp A = \left(\exp(A/p)\right)^p ,$$

for $p \geq 1$ a well-chosen integer such that $\|A/p\|_F$ is small, for example, less than one.

g. Using the previous property, write a function `matrix_power_opt(A,n=20)` based on this property.

h. Redo the same graph as at point *e* but with this new function and comment.

Exercise 6.4 - Groups of permutations

The goal is to study the groups of permutations by developing algorithms to characterize them. A group of permutations $G \subset S_n$ is defined as being generated by a number of permutations: $G = \langle g_1, g_2, \dots, g_k \rangle$, with $g_i \in S_n$ a permutation of the set $\{1, 2, \dots, n\}$. A permutation:

$$g = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 1 & 2 & 3 \end{pmatrix},$$

can be represented in Python by the tuple `g = (0, 4, 1, 2, 3)`. The zero is added in order to conform with Python's zero-based indexing and thus simplify the implementations a bit. This simply means that vertex `0` goes on vertex `0`. Note that this exercise lends itself particularly well to an object-oriented implementation, and in this case the questions can be adapted accordingly.

a. Write a function `product(g1,g2)` that computes the product of two permutations.

b. Write a function `inverse(g)` that computes the inverse of a permutation.

c. Write a function that returns the decomposition of a permutation into cycles represented by a list of tuples.

d. Write a function that returns the permutation corresponding to a list of cycles, *i.e.*, that does the inverse of the previous function.

e. ! In Python, a group $G = \langle g_1, g_2, \dots, g_k \rangle$ generated by a family of permutations, can be represented by a list of permutations `G = [g1,g2,...,gk]`. Write a function `orbit(G,x)` that returns the orbit of a point $x \in \{1, 2, \dots, n\}$:

$$O_x = Gx = \{gx, g \in G\}.$$

Hint

Do not compute the set of elements of the group, it makes a list much too long. Construct the list (X^0, X^1, X^N) of disjoint sets defined recursively by $X^0 = \{x\}$ and X^n as the set of new elements of the form $g_i y$ with $1 \leq i \leq k$ and $y \in X^{n-1}$:

$$X^n = \left(\bigcup_{i=1}^{k} g_i X^{n-1} \right) \setminus \left(\bigcup_{i=1}^{n-1} X^i \right).$$

f. !! Define a function `stabilizer(G,x)` that returns the stabilizer of a point $x \in \{1, 2, \dots, n\}$:

$$G_x = \{g \in G : gx = x\},$$

in the form of a set of generators.

Hint

Use Schreier's lemma.

7 Graph theory

A graph is a pair $G = (X, E)$ consisting of a finite non-empty set X, and a set E of pairs of elements of X. The elements of X are the vertices of the graph G, those of E are the edges of the graph G. A graph is oriented if the edges have a direction, *i.e.*, if the pairs of elements of E are ordered lists such that $(i, j) \in E$ is not equivalent to $(j, i) \in E$. Here only undirected graphs, *i.e.*, whose pairs of elements of X are unordered sets ($(i, j) \in E$), are considered.

For example, the complete graph with n vertices K_n is defined as the graph of vertices $X = \{1, 2, \dots, n\}$ and of edges being the two-element of the power set of X. In particular, $K_4 = (X, E)$, where $X = \{1, 2, 3, 4\}$ and $E = \big\{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}\big\}$.

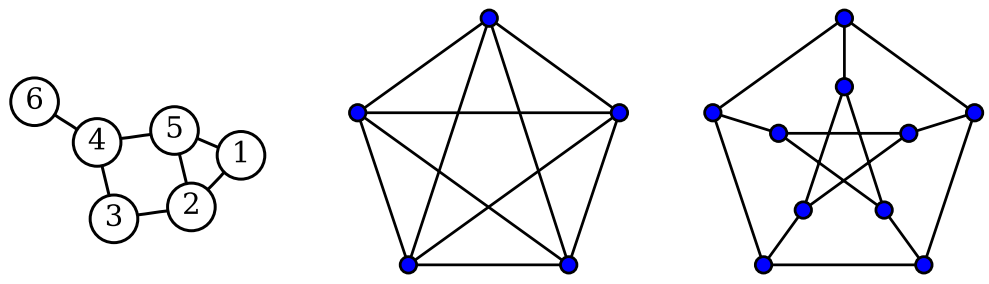
Concepts covered

- undirected graphs
- graphs as dictionaries
- use of frozensets
- adjacency matrix
- search for paths and triangles
- recursive function

Exercise 7.1 - Graphs as dictionaries

One way to represent a graph G is with a dictionary whose keys are the vertices and the value associated to each key $x \in X$ is a set containing the neighbors of x.

a. Construct the following graphs in dictionary form:



(a) Graph with 6 vertices (b) Complete graph (c) Petersen graph

b. Write a function `complete(n)` that constructs the complete graph K_n as a dictionary.

c. A graph given as a dictionary contains the information several times. Write a function `correct(graph)` to add the missing elements of an improperly defined graph so that for any vertex `x`, if `y` belongs to `graph[x]`, then `y` is also a key and `x` belongs to `graph[y]`. Test this function, in particular, on the improperly defined graph `{1:{3,4,2},3:{2}}`.

d. Write a function that returns the set (type `set`) of all edges of a graph represented by a dictionary.

> **Hint**
>
> Sets are mutable and therefore not hashable.

e. ! Write a function to determine whether two vertices are connected by a path or not and return the path if yes.

> **Hint**
>
> Use a recursive function.

f. ! Write a function that returns all paths between two vertices (without cycles).

Exercise 7.2 - Triangles in a graph

A triangle in a graph is a set of three vertices connected by three edges. Finding and analyzing triangles in a graph is important for understanding its structure.

a. Determine mathematically the number of subsets of cardinal three that a set of vertices X has. Is it reasonable to iterate over these elements?

b. Write a function that returns the set of all triangles in a graph.

To each graph $G = (X, E)$ corresponds a unique symmetric matrix A of size $n \times n$ with $n = |X|$ defined by:

$$A_{ij} = \begin{cases} 1, & \text{if } \{i, j\} \in E, \\ 0, & \text{if } \{i, j\} \notin E. \end{cases}$$

This matrix is called the adjacency matrix of graph G.

c. Define a function that returns the adjacency matrix of a graph.

> **Hint**
>
> Be careful that the vertices are not necessarily indexed by integers between 0 and n in the dictionary.

d. Define a function having as argument an adjacency matrix and returning the corresponding graph as a dictionary.

e. Using the adjacency matrix A and the matrix $B = A^2$, write a function returning the set of triangles of a graph.

f. Using the adjacency matrix A, write a function computing the number of triangles.

> **Hint**
>
> Interpret the entries of the matrix A^n.

Exercise 7.3 - Module NetworkX !!

Many graph theory algorithms are implemented in the NetworkX module, see the documentation here.

a. Follow the NetworkX tutorial available here.

b. Analyze one of the downloadable graphs at: https://github.com/gephi/gephi/wiki/Datasets or https://snap.stanford.edu/data/.

8 Symbolic calculation

As a general purpose language, Python does not include by default some mathematical concepts. An example already seen concerns vectors and numerical matrices which are implemented in the NumPy module. The goal here is to introduce the SymPy module which allows to do symbolic calculation.

For example, the number $\sqrt{8}$ is represented by default in Python as a float. The advantage of SymPy is that $\sqrt{8}$ is kept as a root and even automatically simplified:

```
import sympy as sp
sp.sqrt(8)
```

$2\sqrt{2}$

Note that the second instruction is not necessary, but allows to present the results in a more elegant way in Jupyter Lab. SymPy documentation is available here.

Concepts covered

- symbols and symbolic expressions
- simplification
- infinitesimal analysis (limit, derivation, integration, series)
- computer-assisted proof
- pathological function
- Green's function
- spherical coordinates

Exercise 8.1 - Introduction to SymPy

Before you can use symbolic variables, you have to declare them as symbols:

```
x = sp.Symbol("x") # define the symbol x
y = sp.Symbol("y", real=True) # define a real varaible y
e = sp.Symbol(r"\varepsilon", real=True, positive=True) # define a positive
 ↪  variable
```

Then, it is possible to perform operations between symbols:

```
x + 2*y + e/4 + x**2 + 3*x + 2*y
```

$$\frac{\varepsilon}{4} + x^2 + 4x + 4y$$

Most of the mathematical functions are implemented symbolically in SymPy and it is also possible to simplify them:

```
expr = sp.cos(x)**2 + sp.sin(x)**2 + (y**3 + y**2 - y - 1)/(y**2 + 2*y + 1)
 ↪  + sp.exp(-e)
sp.simplify(expr)
```

$$y + e^{-\varepsilon}$$

Finally, it is possible to make substitutions:

```
expr.subs(x,y) # substitute x by y
expr.subs({y:x, e:y}) # substitute y by x and e by y
```

$$\sin^2(x) + \cos^2(x) + e^{-x} + \frac{x^3 + x^2 - x - 1}{x^2 + 2x + 1}$$

then, for example, simplify the expression and plot its graph as a function of x as in Figure 8.1:

```
f = sp.simplify(expr.subs({y:x, e:y}))
sp.plot(f,(x,-2,6), title=f"Plot of ${sp.latex(f)}$")
```

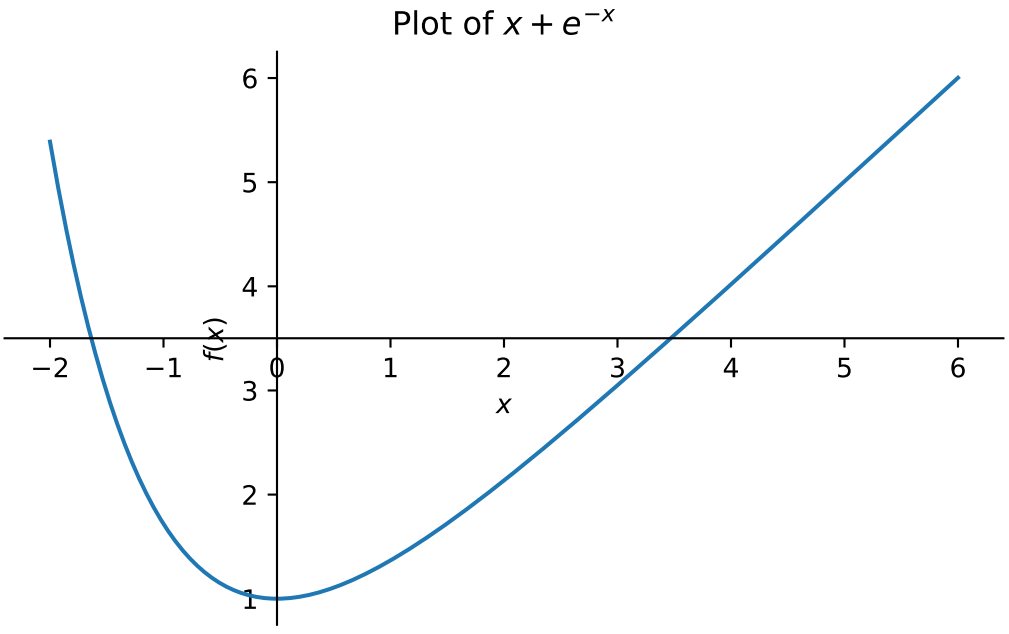


Figure 8.1: Plot of a SymPy function.

a. Read the documentation for the `solve` function and use it to calculate the roots of a general polynomial of degree two, then of degree three.

> **Hint**
>
> The documentation on solving algebraic equations is available here.

b. Read the documentation for the functions `evalf` and `N` to evaluate numerically the expression $\frac{\pi^2}{4}$.

> **Hint**
>
> The documentation on the numerical evaluation is available here.

c. Read the documentation for the `Rational` function and numerically evaluate the rational number $\frac{43609}{999}$ to 50 decimal places.

d. Determine the real and imaginary part of the expression:

$$\left(\frac{1+i\sqrt{3}}{1+i}\right)^{20}.$$

> **Hint**
>
> See the documentation here.

e. Read the documentation for the function `diff` and calculate the derivative of $xe^{x^{x^x}}$ with respect to x.

> **Hint**
>
> The documentation on derivatives is available here.

f. Read the documentation of the function `integrate` and calculate the following integrals:

$$I_1 = \int x^5 \sin(x)\,\mathrm{d}x\,, \qquad I_2 = \int_0^\infty \sin(x^2)\,\mathrm{d}x\,.$$

g. Calculate with SymPy the following limits:

$$L_1 = \lim_{x\to 0} \frac{\sin(x)}{x}\,, \qquad L_2 = \lim_{x\to 0} \sin\left(\frac{1}{x}\right), \qquad L_3 = \lim_{x\to\infty} \frac{5x^2+3x+2y}{y(x-4)(x-y)}\,.$$

h. Compute the series expansion of $\tan(x)$ at $x = 0$ to order 10 and the asymptotic expansion of $\left(1+\frac{1}{n}\right)^n$ for $n \to \infty$ to order 5.

i. Determine the eigenvalues of the matrix:

$$\begin{pmatrix} 1 & a & 0 \\ a & 2 & a \\ 0 & a & 3 \end{pmatrix}.$$

> **Hint**
>
> The documentation on symbolic matrices is available here.

Exercise 8.2 - Applications

The goal is to use SymPy to solve symbolically different mathematical problems by calculating the least possible things by hand.

a. Determine the number of zeros in the integer 123!.

b. Determine the ratio between the height and radius of a cylinder so as to minimize its area at a fixed volume.

c. For $x, y \in \mathbb{R}$ such that $xy < 1$, show that

$$\arctan(x) + \arctan(y) = \arctan\left(\frac{x+y}{1-xy}\right).$$

> **Hint**
>
> Derive the equation with respect to x and justify.

d. Prove the following formula due to Gauss:

$$\frac{\pi}{4} = 12\arctan\left(\frac{1}{38}\right) + 20\arctan\left(\frac{1}{57}\right) + 7\arctan\left(\frac{1}{239}\right) + 24\arctan\left(\frac{1}{268}\right).$$

It is imperative to use SymPy, the original demonstration of Gauss being 25 pages long, see pages 477 to 502 of the second volume of his complete works available here.

> **Hint**
>
> Apply the tangent function on each side of the equation and simplify. The documentation on the different simplification functions is available here.

e. Determine the volume of the region:

$$\left\{(x, y, z) \in \mathbb{R}^3 \,:\, x^2 + y^2 < z < 2x^2 + 4xy + 6y^2, |y| < 5, |x| < 4\right\}.$$

f. Determine the expression of the real Fourier coefficients of the 2π-periodic function f defined by $f(x) = |\sin(x)|$.

Exercise 8.3 - Conjecture due to Euler

Euler conjectured in 1769 that at least k powers of strictly positive integers are necessary for the sum to be itself a k power. In other words, if $n \geq 2$, $k \geq 1$, $a_1, a_2, \ldots, a_n \geq 1$ and $b \geq 1$ are integers such that:

$$\sum_{i=1}^{n} (a_i)^k = b^k$$

then necessarily $n \geq k$. This conjecture was disproved in 1966 by Lander & Parkin (doi:10.1090/S0002-9904-1966-11654-3) in what appears to be the shortest mathematical paper ever written with a counterexample for $k = 5$:

$$27^5 + 84^5 + 110^5 + 133^5 = 144^5 .$$

The goal is to show that this counterexample is the simplest possible, in the sense that it is the only counterexample with $k \leq 5$ and $b \leq 144$.

a. Show that Euler's conjecture is true for $k = 1$ and $k = 2$.

b. Check with Python the above counterexample.

c. Write a function `powers(bmax,k)` that returns the set (type `set`) of all integers from 1 to `bmax` raised to the power `k`.

d. Check that there is no counterexample with $k = 3$ and $b \leq 144$.

e. Write a function `combinations(lst,n)` that for a list of integers `lst` and an integer `n` returns the list of all combinations of `n` integers in `lst` in ascending order. For example, `combinations([1,2,3,4],2)` should return `[(1,1), (1,2), (1,3), (1,4), (2,2), (2,3), (2,4), (3,3), (3,4), (4,4)]`.

> **Hint**
>
> Use a recursive function on `n`.

f. Write a function `test(bmax,n,k)` that for three given integers `bmax`, `n`, and `k`, iterates over all combinations of `n` integers returned by `combinations` and checks whether the sum of these `n` integers raised to the power `k` is an integer present in the list `powers(bmax,k)`. Use this function to check that there is no counterexample to Euler's conjecture for $k = 4$ and $b \leq 144$. Depending on the power of your computer, it is possible to choose also $k = 5$ and thus to check that the counterexample of the introduction is indeed the simplest one.

For $k = 5$, the previous method of iterating over all combinations is rather slow. A faster method is to observe that the set of sums of type:

$$(a_1)^5 + (a_2)^5 + (a_3)^5 + (a_4)^5 ,$$

can be written as $S_1 + S_2$ where S_1 and S_2 are sums of two integers to the power 5.

g. Write a function `sum2(bmax,k)` which returns a dictionary having for keys the sums $(a_1)^k + (a_2)^k$ with the associated value (a_1, a_2) for $0 \leq a_1 \leq a_2 \leq$ `bmax`. We will take care to remove the trivial element zero from the dictionary. When building the dictionary, make sure that it is uniquely defined in the sense that there is no other possible value for an existing key. Test `sum2(300,5)` and `sum2(300,3)`.

h. Use the dictionary constructed earlier to determine the set of counterexamples for $k = 5$ and $b \leq 300$ by iterating over all elements of `powers(bmax,5)` and `sum2(bmax,5)`.

> **Hint**
>
> An optimal implementation takes at most a few seconds to run.

Exercise 8.4 - Pathological function

The goal is to construct a function that visually looks regular, but in fact is not. Let the function $f : \mathbb{R} \to \mathbb{R}$ be defined by:

$$f(x) = \sum_{k=1}^{\infty} \frac{\sin(k^2 x)}{k^5} .$$

Since the series converges absolutely, the function f is well defined.

a. With the help of SymPy calculate the function $g : \mathbb{R}$ defined by keeping the first hundred terms of the series:

$$g(x) = \sum_{k=1}^{100} \frac{\sin(k^2 x)}{k^5} ,$$

and plot the function g.

b. Estimate by hand the error between the functions f and g in absolute value.

c. Calculate the first derivative and the second derivative of g and plot these two derivatives. What can you conclude?

d. Explain mathematically what is going on.

Exercise 8.5 - Green's function of the Laplacian !

The goal of this exercise is to compute fully automatically the Green's function of the Laplacian in $\mathbb{R}^3$, *i.e.*, the solution satisfying:

$$\Delta G(x) = \delta(x)\,,$$

in $\mathbb{R}^3$, where $\delta(x)$ is the Dirac distribution.

For this, we introduce the spherical coordinates $x' = (r, \theta, \varphi)$ with $r > 0$, $0 \leq \theta \leq \pi$ and $0 \leq \varphi < 2\pi$ characterized by:

$$\begin{aligned} x_1 &= r\cos\varphi\sin\theta \\ x_2 &= r\sin\varphi\sin\theta \\ x_3 &= r\cos\theta\,. \end{aligned}$$

a. Define a function `to_spherical(expr)` to convert an expression given in Cartesian coordinates to spherical coordinates.

b. Define a function `to_cartesian(expr)` allowing to convert into cartesian coordinates an expression given in spherical coordinates. For simplicity, we can only deal with the case of an expression `expr` invoking the variables r and θ but not φ.

c. Calculate the scale factors of the spherical coordinates:

$$h_i = \left\| \frac{\partial x}{\partial x'_i} \right\|.$$

d. Define a function `gradient(f)` allowing to calculate the gradient of a function $f : \mathbb{R}^3 \to \mathbb{R}$ in spherical coordinates:

$$\nabla f = \left(\frac{1}{h_i} \frac{\partial f}{\partial x'_i} \right)_{i=1}^{3}.$$

e. Do the same to define the Laplacian in spherical coordinates:

$$\Delta f = \sum_{i=1}^{3} \frac{1}{J} \frac{\partial}{\partial x'_i} \left(\frac{J}{h_i^2} \frac{\partial f}{\partial x'_i} \right) \quad \text{where} \quad J = \prod_{i=1}^{3} h_i\,.$$

f. Find the radial solutions (*i.e.*, depending only on the variable r) of the equation $\Delta G = 0$ in $\mathbb{R}^3 \setminus \{0\}$.

Hint

Look at the documentation of the function `dsolve` to solve a differential equation.

g. Determine the equations that the integration constants must satisfy for the above solution to satisfy in Cartesian coordinates:

$$\lim_{|x|\to\infty} G(x) = 0 \quad \text{and} \quad \Delta G(x) = \delta(x)\,.$$

Hint

We must transform the two conditions into spherical coordinates. The first condition is expressed in spherical coordinates by:

$$\lim_{r\to\infty} G(r) = 0\,,$$

and the second is equivalent to:

$$\lim_{r\to 0} \int_0^{\pi} \int_0^{2\pi} \big(\nabla G(r)\cdot e_r\big) J(r,\theta,\varphi)\,\mathrm{d}\varphi\mathrm{d}\theta = 1\,.$$

h. Solve the equations on the integration constants and substitute in the radial solution of $\Delta G = 0$ to obtain the expression of the Green's function of the laplacian in spherical coordinates. Finally, determine the Green's function G of the Laplacian in $\mathbb{R}^3$ in Cartesian coordinates.

i. !! Let $g : \mathbb{R}^3 \to \mathbb{R}$ be a smooth function with compact support invariant by rotations along the vertical axis. Determine the asymptotic behavior at large distances of the solution of the equation:

$$\Delta f(x) = g(x)$$

up to order two, *i.e.*, the terms decreasing as $|x|^{-1}$ and as $|x|^{-2}$.

Hint

The solution is given by Green's formula:

$$f(x) = \int_{\mathbb{R}^3} G(x-x_0)g(x_0)\,\mathrm{d}^3x_0 = \int_{B_R} G(x-x_0)g(x_0)\,\mathrm{d}^3x_0\,,$$

where R is chosen so that the support of g is contained in B_R. To compute the asymptotic expansion of this integral, the first step is to convert $G(x-x_0)$ into spherical coordinates for x and x_0. Since g is invariant by rotations along the vertical axis, then in spherical coordinates g is independent of φ and is given by $g(r,\theta)$. The second step is to transform the integral in spherical coordinates with a triple integral on r_0, θ_0, and φ_0. The third step is to compute the asymptotic development of the integrand when $r \to \infty$, up to order

two. Finally, since r_0, θ_0, and φ_0 are bounded, the integration then commutes with the asymptotic expansion and the final result is given by the individual integration of the two terms of the asymptotic expansion.

9 Root finding

The aim of this series of exercises is to determine the roots of a function in an approximate way, in particular by Newton's method. This allows in particular to find approximate solutions of nonlinear equations. This method is fundamental both from a numerical and an analytical point of view.

Concepts covered

- Newton's method in one and several dimensions
- Jacobian matrix
- Newton's method attractor
- fractal set
- optimization by parallelization
- non-linear differential equation
- finite differences

Exercise 9.1 - Newton's method in one dimension

In one dimension, Newton's method consists in finding an approximate solution of a single equation. This equation can be put in the general form $F(x) = 0$, where $F : \mathbb{R} \to \mathbb{R}$ is a fairly regular function so, the goal is to find a zero of the function F. The equation $F(x) = 0$ is equivalent (if $F'(x) \neq 0$) to the equation $G(x) = x$, where G is the function defined by:

$$G(x) = x - \frac{F(x)}{F'(x)}\,.$$

Newton's method consists in finding a fixed point of G, *i.e.*, solving $G(x) = x$ by successive iterations:

$$x_{i+1} = G(x_i) = x_i - \frac{F(x_i)}{F'(x_i)}\,,$$

from an initial value $x_0 \in \mathbb{R}$. When the sequence $(x_i)_{i\in\mathbb{N}}$ converges, then the limit x is a solution of $G(x) = x$ therefore of $F(x) = 0$.

a. Write a function `newton1d(F, DF, x0, eps=1e-10, N=1000)` that given a function F, its derivative F', and an initial value x_0 computes Newton's iterations until $|F(x_i)| < \varepsilon$ and returns x_i. If N iterations were not enough to reach this convergence criterion, then return an error.

b. Using the function defined above, find an approximate solution of the equation $e^{-x} = x$.

c. Without using the function `sqrt`, `log`, or fractional powers, define a function `root(x,n)` that computes $\sqrt[n]{x}$.

Sometimes, the derivative of the function F cannot be calculated analytically, so it is necessary to approximate it numerically:

$$F'(x_i) \approx \frac{F(x_i) - F(x_{i-1})}{x_i - x_{i-1}},$$

which leads to the secant method:

$$x_{i+1} = x_i - F(x_i)\frac{x_i - x_{i-1}}{F(x_i) - F(x_{i-1})} = \frac{x_{i-1}F(x_i) - x_iF(x_{i-1})}{F(x_i) - F(x_{i-1})}$$

where x_0 and x_1 must be chosen.

d. Write a method `secant1d(F, x0, x1, eps=1e-10, N=1000)` implementing the secant method and test it on the previous example.

Exercise 9.2 - Newton's method in several dimensions

Newton's method in one dimension is easily generalized to several dimensions to solve equations of the form $F(x) = 0$, where $F : \mathbb{R}^n \to \mathbb{R}^n$ is a fairly regular function. Conceptually, the method is identical: the equation $F(x) = 0$ is equivalent to $G(x) = x$ with the function G defined by:

$$G(x) = x - \big(F'(x)\big)^{-1} F(x),$$

where $F'(x)$ denotes the Jacobian matrix of size $n \times n$ of F in x. Thus, Newton's iterations are written:

$$x_{i+1} = x_i - \big(F'(x_i)\big)^{-1} F(x_i).$$

a. Write a function `newton(F, DF, x0, eps=1e-12, N=10000)` implementing Newton's method in more than one dimension.

> **Hint**
>
> To have an optimal performance, one should not invert the Jacobian matrix but solve a linear system with $F(x_i)$ as second member.

b. Use the previous function to solve the following system:

$$\cos(x) = \sin(y), \qquad e^{-x} = \cos(y).$$

Exercise 9.3 - Newton's method attractor

The goal of this exercise is to solve the equation $z^3 = 1$ in the complex plane using Newton's method and to analyze to which of the three roots of unity the method will converge depending on the choice of the initial point z_0.

a. If necessary, adapt the function `newton1d` so that it also applies to complex numbers and test it to solve $z^3 = 1$ from different values of z_0.

b. Determine for each $z_0 \in \{x_0 + iy_0 : x_0 \in [-3, 3] \text{ and } y_0 \in [-3, 3]\}$ to which root of unity Newton's method will converge. Represent graphically this set as in Figure 9.1.

> **Hint**
>
> NumPy's `meshgrid` function can be useful to construct the matrix corresponding to the set of z_0.

c. ! The previous method has the disadvantage of proceeding sequentially to the calculation for each value of z_0, which makes this evaluation rather slow. Propose a new implementation allowing to compute in parallel all the values of z_0 using NumPy indexing.

> **Hint**
>
> To speed-up the method even more, Newton's iterations of $F(z) = z^3 - 1$ can be calculated by hand:
>
> $$z_{n+1} = \frac{1}{3z_n^2} + \frac{2z_n}{3}\,.$$

Exercise 9.4 - Nonlinear differential equation !!

The goal is to solve the following differential equation with boundary conditions:

$$u''(x) + u^3(x) = \sin(x)\,, \qquad\qquad u(0) = u(2\pi) = 0\,,$$

on the interval $[0, 2\pi]$. This equation is a simplified model for a nonlinear Schrödinger equation.

The method used is finite differences that consists in looking for the values of u at the points $x_n = \frac{2\pi n}{N}$ for $n = 0, 1, \ldots, N$. The unknowns are then the numbers $u_n = u(x_n)$ and form a vector of dimension $N+1$. The finite difference method consists in approximating the second derivative by:

$$u''(x) \approx \frac{u(x+h) - 2u(x) + u(x-h)}{h^2}\,,$$

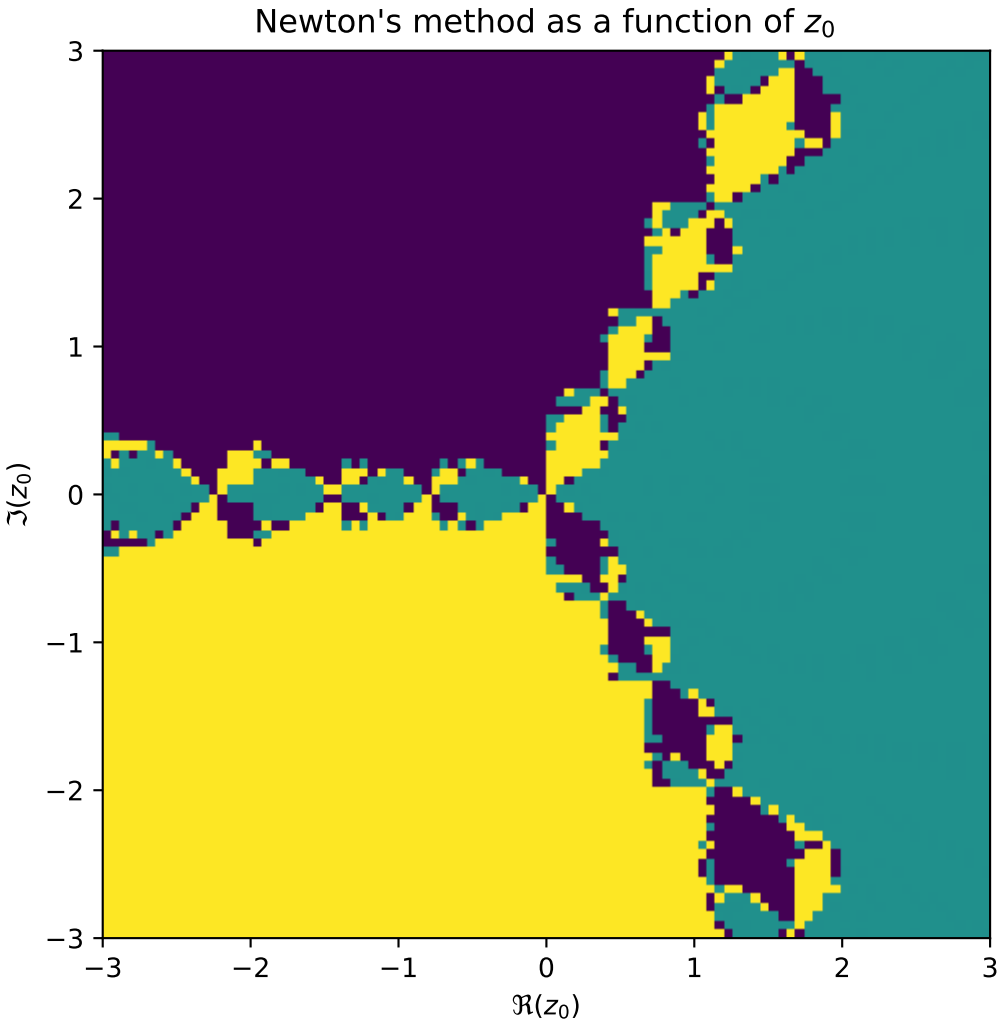


Figure 9.1: Graphical representation of the convergence of Newton's method as a function of the initial value z_0.

when h is small. Taking $h = \frac{2\pi}{N}$, then:

$$u''(x_n) \approx \frac{u_{n+1} - 2u_n + u_{n-1}}{h^2},$$

and so the initial equation is approximated by:

$$\frac{u_{n+1} - 2u_n + u_{n-1}}{h^2} + u_n^3 = \sin(x_n), \qquad u_0 = u_N = 0,$$

for $n = 1, 2, \dots, N-1$. This equation can be seen as an equation of the type $F(u) = 0$ for $u = (u_n)_{n=0}^{N+1}$ and thus be solved by Newton's method.

a. Show the following approximation:

$$u''(x) = \frac{u(x+h) - 2u(x) + u(x-h)}{h^2} + O(h^2) \quad \text{as} \quad h \to 0.$$

> **Hint**
>
> Use Taylor's theorem.

b. Define a vector `x` representing $N+1$ evenly spaced points in $[0, 2\pi]$ and `h` the distance between the points, with, for example, $N = 200$.

c. Define a function `F(u)` representing the function $F : \mathbb{R}^{N+1} \to \mathbb{R}^{N+1}$ allowing to put the approximated equation in the form $F(u) = 0$.

> **Hint**
>
> To have a fast implementation, it is imperative to use the NumPy slicing instead of a loop to build F.

d. Define a function `DF(u)` representing the Jacobian of the previous function.

> **Hint**
>
> The Jacobian is the derivative of $F(u) = F(u_0, u_1, \dots, u_N)$ with respect to $u = (u_0, u_1, \dots, u_N)$, *i.e.*:
>
> $$F'(u) = \begin{pmatrix} \partial_0 F(u) & \partial_1 F(u) & \partial_2 F(u) & \cdots & \partial_{N-1} F(u) & \partial_N F(u) \end{pmatrix},$$
>
> and can be calculated explicitly by hand.

e. Use the `newton` function defined earlier to calculate an approximate solution of the equation. By changing the initial values, is it possible to find other solutions?

Hint

Try with the initial data $u_0(x) = (1 + k)\sin(kx)$ for $k = 1, 2, 3, 4$ as the starting point of Newton's method.

10 Probability and statistics

In a first step, the statistics of the proportion of numbers beginning with a certain digit will be studied. Then in a second step, important probabilistic models will be introduced and simulated, such as random walks, illustrations of the central limit theorem, or percolation.

Concepts covered

- statistics and probability
- random harmonic series
- random walk
- central limit theorem
- random vectors
- percolation
- phase transition
- histograms
- optimization by compilation

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

Exercise 10.1 - Harmonic series of random sign

The goal of this exercise is to simulate the convergence of a harmonic series whose sign is drawn randomly. More precisely, if $(X_i)_{i\in\mathbb{N}}$ is a sequence of independent random variables worth -1 or 1 with probability $\frac{1}{2}$, then we define the partial sum:

$$W_0 = 0\,, \qquad\qquad W_n = \sum_{i=1}^{n} \frac{X_i}{i}\,,$$

and the question is to determine if the sequence $(W_n)_{n\in\mathbb{N}}$ converges and if so toward what.

a. Write a function `sign()` that simulates the random variable X_i.

b. Write a function `simulate(n)` that returns a realization of $(W_0, W_1, \dots, W_n)$.

c. Plot the function $n \mapsto W_n$ for different realizations, for example, for $0 \leq n \leq 1\,000$ and make a conjecture about the convergence of the sequence $(W_n)_{n\in\mathbb{N}}$.

d. ! Determine the histogram of $W_{1\,000}$ for 10^4 or 10^5 realizations to get an idea of the law of the limiting random variable.

Exercise 10.2 - Gambler's ruin

The goal is to simulate the evolution of the amount of money of a gambler playing heads or tails. At each toss, the player wins one euro if it is heads and loses one if it is tails. The probability of getting tails is noted p, that of getting heads q. In particular, $p = q = \frac{1}{2}$ if the coin is balanced.

Mathematically, the sum S_i owned by the player at time i is given by a random walk:

$$S_i = \begin{cases} 0\,, & \text{if } S_{i-1} = 0\,, \\ S_{i-1} + X_i\,, & \text{if } S_{i-1} \geq 1\,, \end{cases}$$

where $(X_i)_i$ are independent random variables of law $\mathbb{P}(X_i = 1) = p$ and $\mathbb{P}(X_i = -1) = q$.

a. Write a function `simulate(p,k,N)` that generates a realization of length N of the process from $S_0 = k$, *i.e.*, returns $(S_0, S_1, S_2, \dots, S_N)$. Represent graphically several realizations.

b. Simulate a player who, starting with a sum k, plays until he loses everything or has the amount $n \geq k$.

c. If T is the time at which the game stops, *i.e.*, when $S_T = 0$ or $S_T = n$, recover by simulation the theoretical results on the average time:

$$\mathbb{E}(T) = \begin{cases} k(n-k)\,, & \text{if } p = q\,, \\ \dfrac{n}{p-q}\dfrac{1-\rho^k}{1-\rho^n} - \dfrac{k}{p-q}\,, & \text{if } p \neq q\,, \end{cases}$$

and the place of exit:

$$\mathbb{P}(S_T = 0) = \begin{cases} \dfrac{n-k}{n}\,, & \text{if } p = q\,, \\ \dfrac{\rho^k - \rho^n}{1-\rho^n}\,, & \text{if } p \neq q\,, \end{cases}$$

where $\rho = q/p$. For this, we can plot these quantities as a function of p or just consider the case $p = q = \frac{1}{2}$.

Exercise 10.3 - Pólya urn

An urn initially contains (at $t = 0$) r_0 red balls and b_0 white balls. At each time, we pick a ball uniformly at random from the urn. This ball is then returned to the urn and a ball of the same color is added. Such a system is called a *Pólya urn*. The purpose of this exercise is to study the behavior of the fraction of red balls in the urn, *i.e.*, the number of red balls out of the total number. We will call r_n and b_n, respectively, the number of red and white balls in the urn at time n.

a. Write a function `density` taking as argument a tuple representing the number of red and white balls in an urn, and which returns the density of red balls.

We want to recursively construct the distribution of the number of red balls at time n, *i.e.*, the list of probabilities that the number of red balls is equal to a given integer k (which will be the index of the list). This is done by writing two functions: `next_dist_red`, which takes as argument the distribution at time n and returns the one at time $n + 1$, which is thus the function that does all the work, and `dist_red`, which is the wrapper function, taking as argument r_0, b_0, and time n and returning the distribution at time n by a recursive call. We will use the following useful facts (make a small drawing):

- The distribution passed as an argument to `next_dist_red` is a list `r`, and `r[k]` represents the probability of having `k` red balls in the urn at time n. The indices for `r` vary from 0 to the total number `s` of balls at time n.
- At time $n + 1$, to have k red balls, we need:
  - either having had k red balls at the previous time and not having drawn a red ball;
  - or having had $k - 1$ red balls at the previous time and having drawn a red ball.
- If $n = 0$, the result of `dist_red` is completely deterministic and the coefficients of the list are only 0 and 1, depending on r_0 and b_0.

b. Write the functions `next_dist_red` and `dist_red` using the directions provided. Look at the result of `dist_red(0,1,n)` and `dist_red(1,1,n)` for different values of `n` $(1, 2, 5, 10, 20, \dots)$ and comment.

Rather than theoretically computing for each n the sequence of theoretical probabilities, we will do statistics on a large number of Pólya urn realizations, after a large number of steps. For this, we need a function to evolve a Pólya urn.

c. Define a function `polya_step(r,b)` which, given the composition of an urn passed as two parameters `r` and `b`, returns the (random) evolution after one step of the composition of the urn as a tuple. Also define a function `polya(r0,b0,N)` taking as arguments r_0, b_0 and N as parameters and returning the (random) composition of a Pólya urn after N steps, also as a tuple.

d. Write a function `data_rdens_polya(r0,b0,N,nbexp)` that returns a list of length `nbexp` containing the densities of `nbexp` realizations of Pólya urns at time `N` initialized with `r0` red balls and `b0` white balls.

e. Store the result of `data_rdens_polya(2,3,1000,10_000)` in a variable and draw a histogram to see the distribution of densities. Be careful, we want the heights of the bars to be normalized

so that their surface represents the proportion of points, and not so that they give the number of points per bin.

Hint

A good rule of thumb is to choose the number of bins for a histogram of the order of the square root of the number of points. See the documentation of the `hist` function of Matplotlib.

Exercise 10.4 - Central limit theorem

The central limit theorem (also known as the central limit theorem, the central limit theorem or the central limit theorem) establishes the convergence of the sum of a sequence of random variables toward the normal distribution. Intuitively, this result states that a sum of identical and independent random variables tends (under certain conditions) toward a Gaussian random variable. Here's how it works:

Theorem: Let (X_n) be a sequence of independent real random variables of the same distribution with expectation μ and standard deviation $\sigma \neq 0$. Let $(\bar{X}_n)$ be the sequence defined by:

$$\bar{X}_n = \frac{1}{n} \sum_{k=1}^{n} X_k \,.$$

For n large enough, the distribution of $\bar{X}_n$ can be approximated by the normal distribution $\mathcal{N}(\mu, \frac{\sigma^2}{n})$.

The aim of this exercise is to check whether this theorem is valid for different laws of probability:

- *Poisson distribution*: discrete distribution on $\mathbb{N}$, of parameter λ, defined by:

$$\mathbb{P}(X = k) = \exp(-\lambda)\frac{\lambda^k}{k!}\,, \quad \forall k \in \mathbb{N}\,.$$

- *Normal distribution*: continuous distribution on $\mathbb{R}$, of parameters m and σ, defined by density:

$$\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-m)^2}{2\sigma^2}\right), \quad \forall x \in \mathbb{R}\,.$$

- *Cauchy distribution*: continuous distribution on $\mathbb{R}$, of parameters a and γ, defined by density:

$$\frac{1}{\pi\gamma\left(1+\left(\frac{x-a}{\gamma}\right)^2\right)}, \quad \forall x \in \mathbb{R}.$$

a. Define a function `normal_density(x, mu var)` taking as an argument :

- `x` (array): an array of floating-point numbers;
- `mu` (floating number): average μ;
- `var` (strictly positive floating number): variance σ^2;

and which returns the density of the normal distribution evaluated for each number x in `x`:

$$\mathcal{N}(\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right).$$

b. Look at the documentation for the `numpy.random.poisson` function and generate 10 random values according to a Poisson distribution with parameter $\lambda = 2$.

c. Write a function `samples_poisson(lam, N, M)` which takes as arguments:

- `lam` (strictly positive real): parameter λ for the Poisson distribution;
- `N` (strictly positive integer): number of experiments;
- `M` (strictly positive integer): number of random variables generated for each experiment;

and which generates `N` experiments with `M` random variables generated per experiment, and returns:

- the average value (floating number) over the `N * M` random variables generated;
- the standard deviation (floating number) on the `N * M` random variables generated;
- a numpy vector of size `N` where each element is the mean of the random variables in an experiment.

For the values `lam=2` and `N=10_000`, run `samples_poisson(lam, N, M)` for $M \in \{10, 100, 1\,000\}$ and save the results in variables.

d. For each value of `M`, display the distribution of the numpy vector containing the means of each experiment, as well as the distribution of the expected normal distribution if the central limit theorem is verified. You can use Matplotlib `hist` function to display the histogram of a table, with the `bins=50` parameter to set the number of columns and `density=True` to display a probability distribution. Use the empirical means and standard deviations returned by the `samples_poisson` function for normal distribution parameters. Choose relevant values for the x-axis boundary values. Make a hypothesis on the validity of the central limit theorem for the Poisson distribution.

e. Repeat questions *c)* and *d)* for the normal distribution. Using the function `numpy.random.normal`, generate averages for `loc=2`, `scale=1`, `N=10_000`, and $M \in \{10, 100, 1\,000\}$. Plot the histograms and make a hypothesis about the validity of the central limit theorem for the normal distribution.

f. Repeat questions *c)* and *d)* for the Cauchy distribution with $a = 0$ and $\gamma = 1$. To do this, use the function `numpy.random.standard_cauchy`. Use `bins=np.arange(-10, 10.1, 0.1)` and make an assumption about the validity of the central limit theorem for the Cauchy distribution.

Exercise 10.5 - Random generation of unit vectors

The aim of this exercise is to find an efficient method for randomly generating unit vectors in $\mathbb{R}^n$ according to a uniform distribution. We will start with the case $n = 2$, where a real vector can be represented by a complex number.

a. Consider the following strategy for randomly generating a unit vector in $\mathbb{R}^2$:

1. Generate x randomly according to the uniform distribution on $[-1, 1]$;
1. Generate y randomly according to the uniform distribution on $[-1, 1]$;
1. Return the unit complex $z = \frac{x}{\sqrt{x^2+y^2}} + \frac{y}{\sqrt{x^2+y^2}}i$.

Write a function `generate_complex` which takes as argument a positive integer N and returns a NumPy array of size N, where each element is a complex generated by the above strategy.

> **Hint**
>
> It is much more efficient to generate N random variables directly using the `size` argument of the function `numpy.random.uniform` than to generate them one by one with a `for` loop.

b. For $n = 2$, one way to check whether the distribution of vectors is uniform is to look at the distribution of angles/arguments of the complex numbers (*i.e.*, $\arg z$): this should be uniform. Use the `generate_complex` function with $N = 10^6$ and display the distribution of angles/arguments. Does the strategy described in the previous question generate unit vectors uniformly?

> **Hint**
>
> You can use the function `numpy.angle`.

c. The following modification to the previous strategy is proposed:

1. Generate x randomly according to the uniform distribution on $[-1, 1]$;
1. Generate y randomly according to the uniform distribution on $[-1, 1]$;
1. If $x^2 + y^2 \leq 1$, return the unit complex $z = \frac{x}{\sqrt{x^2+y^2}} + \frac{y}{\sqrt{x^2+y^2}}i$ (otherwise return nothing at all).

Write a function `generate_complex_monte_carlo` which takes as argument a positive integer N corresponding to the number of candidate vectors and returns a NumPy array of size $n \leq N$, where each element is a complex generated by the above strategy. *Warning:* n is random and therefore cannot be determined in advance.

Repeat the previous question and display the angle/argument distribution of the complexes generated by this strategy. Does this strategy generate unit vectors uniformly?

d. We now ask how many candidate complexes must be generated on average to accept n of them, which is equivalent to determining how many complexes are accepted on average from the N candidates. The law of large numbers means that the ratio converges to the probability that a candidate vector will be accepted. This probability is equal to the ratio of the inclusion area π (the unit circle) to the total area 4 (the square $[-1,1]^2$), *i.e.*, $\frac{\pi}{4}$. Compare the ratio $\frac{n}{N}$ to $\frac{\pi}{4}$.

e. We now consider the general case $n \geq 2$. The strategy considered is the same as in question *c)*:

1. Generate a vector $x = (x_1, \dots, x_n)$, where each x_k is randomly and independently generated according to the uniform distribution on $[-1,1]$;
1. If $\|x\|_2 \leq 1$, return the unit vector $\frac{x}{\|x\|_2}$.

The volume of the unit ball in $\mathbb{R}^n$ is given by:

$$V_n = \frac{\pi^{n/2}}{\Gamma(\frac{n}{2}+1)}$$

and the volume of the cube $[-1,1]^n$ is 2^n. Plot the probability of a vector being accepted in step 2 of this strategy for $n \in \{2, \dots, 20\}$. Do you think this strategy is effective for large n values?

Hint

We can use `scipy.special.gamma` for the Γ function and the function `scatter` of Matplotlib to graphically display the values of a sequence with an adapted scale.

f. The following strategy can be shown to generate random vectors on $\mathbb{R}^n$:

1. Generate a vector $x = (x_1, \dots, x_n)$, where each x_k is generated randomly and independently according to the reduced centered normal distribution: $x_k \sim \mathcal{N}(0,1)$;
2. Return the unit vector $\frac{x}{\|x\|_2}$.

Verify for $n = 2$ that this strategy does indeed randomly generate unit vectors according to a uniform distribution by displaying the angle distribution (representing the vectors as complex numbers) for $N = 10^6$ vectors.

Hint

You can use the function `numpy.random.standard_normal`.

Exercise 10.6 - Percolation !!

The goal is to study a percolation model in a porous medium. The medium is modeled by a random matrix of Booleans that determines which sites can be invaded by water and which are impermeable. A matrix percolates if there is a water path from the top row to the bottom row. In the examples in Figure 10.1, the entries of a matrix that can be percolated are colored and the entries that are actually filled with water are in blue. The first matrix does not percolate while the second does.

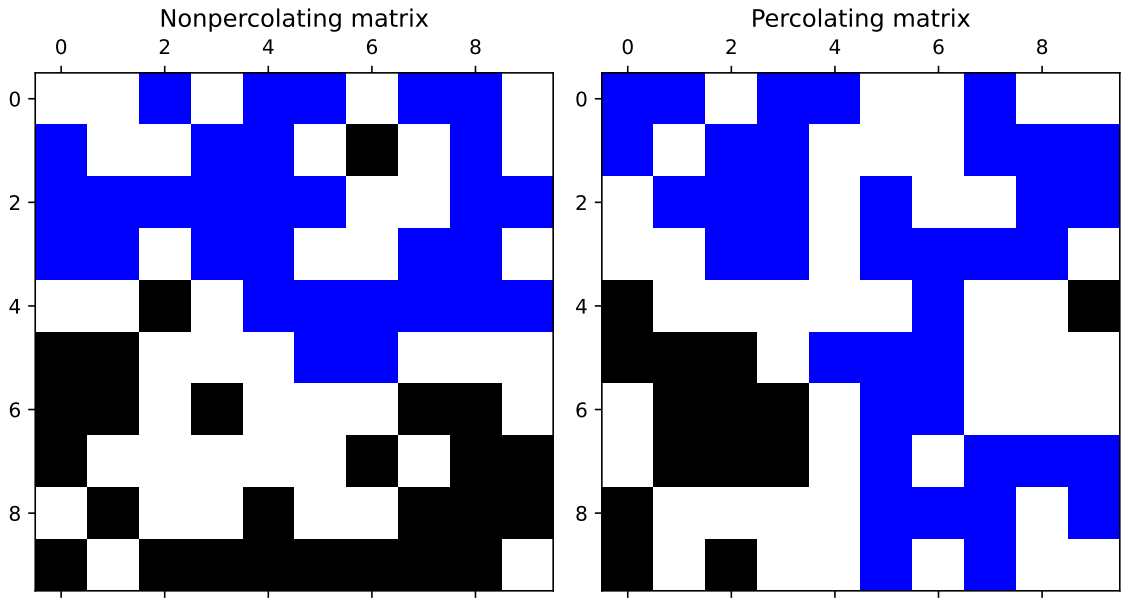


Figure 10.1: The matrix on the left does not percolate while the one on the right does.

a. Write a function `generate(n,p)` that generates a matrix of Booleans of size $n \times n$ such that each entry has probability p of being right and $1 - p$ of being wrong.

Hint

The `random.binomial` function of NumPy can be useful.

b. Define a function `fill(isopen)` that for a given Boolean matrix returns another Boolean matrix with the entries invaded by water.

Hint

Define a Boolean matrix `isfull` to store whether an input is filled with water or not, and then define a recursive function `flow(isopen, isfull, i, j)` to invade all possible inputs from (i, j).

c. Using Matplotlib, represent the filling of different randomly generated matrices.

d. Define a function `percolate(isopen)` to determine whether a Boolean matrix is percolating or not.

e. !! Calculate the time needed to determine if a matrix of size 50×50 with $p = 0.9$ is percolating or not. Read the documentation of the Numba module to reduce the calculation time by compiling one of the functions: https://numba.pydata.org/.

Hint

The function that is most used is the recursive function, so it is the one that should be optimized when compiling it.

f. By doing statistics, determine the probability that a Boolean random matrix of size $n \times n$ with probability p will percolate. Study this probability as a function of p and n.

Hint

Plot this percolation probability as a function of p for different values of n.

g. !!! The statistics performed in the previous point are a typical example of calculations that can be easily executed in parallel, because each case is independent of the others. Parallelize the previous algorithm in such a way as to use all the cores of your processor, for example, with the help of the module `mpi4py`.

Hint

Using Jupyter Lab to do parallel computing is quite complex to implement, it is better to use the command line to run a script in parallel, for example, for four cores: `mpirun -n 4 script.py`. Note that Open MPI or MPICH must be installed on the computer.

11 Differential equations

The goal is to introduce the basic methods for solving first-order ordinary differential equations of the type:

$$\dot{x}(t) = f(t, x(t)), \qquad x(0) = x_0,$$

where $f : \mathbb{R}^+ \times \mathbb{R}^n \to \mathbb{R}^n$ is a smooth enough function and $x_0 \in \mathbb{R}^n$ is an initial data. Note that higher-order ordinary differential equations can be put in the previous first-order form.

Concepts covered

- Euler's methods
- Runge-Kutta methods
- nonlinear partial differential equation
- finite differences
- adaptive methods

Exercise 11.1 - Euler's methods

The simplest idea to approximate an ordinary differential equation is to discretize time with a step h and approximate the time derivative on each interval of length h. There are two simple ways to approximate the time derivative. The first is the forward finite difference approximation:

$$\dot{x}(t) \approx \frac{x(t+h) - x(t)}{h},$$

the second, the backward finite difference:

$$\dot{x}(t) \approx \frac{x(t) - x(t-h)}{h}.$$

The unknowns being the evaluations of the solution x at times $t_i = ih$ for $i \geq 0$, *i.e.*, $x_i = x(t_i)$. The differential equation can thus be approximated using forward finite differences by:

$$\frac{x_{i+1} - x_i}{t_{i+1} - t_i} = f(t_i, x_i),$$

which gives the explicit Euler formula:

$$x_{i+1} = x_i + (t_{i+1} - t_i) f(t_i, x_i)\,.$$

With the backward finite difference approximation, we obtain the implicit Euler method (also called backward Euler method):

$$x_i = x_{i-1} + (t_i - t_{i-1}) f(t_i, x_i)\,.$$

On the one hand, the explicit Euler formula allows to compute directly all x_i by recurrence knowing x_0. On the other hand, the implicit Euler formula requires at each time step the resolution of a nonlinear equation for x_i, for example, with the Newton's method.

a. Write a function `euler_explicit(f,x0,t)` that given an initial data `x0` returns the values $x_0, x_1, \dots, x_m$ computed with the explicit Euler method at times $(t_i)_{i=0}^m$ represented by the vector `t`.

b. Use the explicit Euler method to solve the differential equation:

$$\dot{x}(t) + x(t) = \sin(t)\,, \quad x(0) = 1\,,$$

for $t \in [0, 10]$. Compare the results with the exact solution:

$$x(t) = \frac{1}{2}\big(\sin(t) - \cos(t) + 3e^{-t}\big)\,,$$

for different time discretizations.

c. Solve the previous problem with the implicit Euler method.

> **Hint**
>
> Since the previous equation is linear, we can actually make the implicit Euler method explicit by solving the implicit equation by hand.

d. !! Define a function `euler_implicit(f, Dxf, x0, t)` implementing the implicit Euler method for nonlinear equations. Note that to solve the nonlinear problem with Newton's method, the derivative of f according to x is required.

> **Hint**
>
> It is also possible to use the root finding algorithm `optimize.fsolve` from SciPy that does not require to know the derivative of f.

e. ! Use the previous methods to find an approximate solution of the system:

$$\begin{aligned} \dot{x}(t) + \cos(y(t)) &= \sin(t)\,, & x(0) &= 1\,, \\ \dot{y}(t) + \cos(x(t)) &= 0\,, & y(0) &= 0\,. \end{aligned}$$

Exercise 11.2 - Runge-Kutta methods

The purpose of this exercise is to introduce a class of methods more accurate than Euler's methods for solving ordinary differential equations. Instead of doing a first-order approximation in h the idea is to do a higher-order approximation of the derivative.

The basic idea is to construct a sequence x_i giving an approximation of the solution of $\dot{x}(t) = f(t,x)$ at time t_i for $i \in \mathbb{N}$. This sequence is defined by:

$$x_{i+1} = x_i + M(t_i, x_i, t_{i+1} - t_i)\,,$$

for a certain function M called method. For example, for the explicit Euler method, the function M is given by:

$$M(t,x,h) = hf(t,x)\,.$$

A Runge-Kutta method of order two is given by:

$$M(t,x,h) = hf\Big(t + \frac{h}{2}, x + \frac{h}{2}f(t,x)\Big)\,.$$

A Runge-Kutta method of order four is given by:

$$M(t,x,h) = \frac{h}{6}\big(k_1 + 2k_2 + 2k_3 + k_4\big)\,,$$

where

$$\begin{aligned} k_1 &= f(t,x)\,, \\ k_2 &= f\Big(t + \frac{h}{2}, x + \frac{h}{2}k_1\Big)\,, \\ k_3 &= f\Big(t + \frac{h}{2}, x + \frac{h}{2}k_2\Big)\,, \\ k_4 &= f(t+h, x + hk_3)\,. \end{aligned}$$

Note that more generally, a Runge-Kutta method of order s is given by:

$$M(t,x,h) = h\sum_{i=1}^{s} b_i k_i\,,$$

where

$$\begin{aligned}
k_1 &= f(t,x)\,,\\
k_2 &= f(t+c_2h, x+ha_{21}k_1)\,,\\
k_3 &= f(t+c_3h, x+h(a_{31}k_1+a_{32}k_2))\,,\\
&\;\;\vdots\\
k_s &= f(t+c_sh, x+h(a_{s1}k_1+a_{s2}k_2+\cdots+a_{s,s-1}k_{s-1}))\,.
\end{aligned}$$

The coefficients a_{ij} (for $1 \le j < i \le s$), c_i (for $2 \le i \le s$), and b_i (for $1 \le i \le s$), are often represented in a Butcher table:

$$\begin{array}{c|cccc}
0 & & & & \\
c_2 & a_{21} & & & \\
c_3 & a_{31} & a_{32} & & \\
\vdots & \vdots & & \ddots & \\
c_s & a_{s1} & a_{s2} & \cdots & a_{s,s-1} \\
\hline
 & b_1 & b_2 & \cdots & b_{s-1} \quad b_s
\end{array}$$

For example, the Butcher array from the previous method of order two is:

$$\begin{array}{c|cc}
0 & & \\
\frac{1}{2} & \frac{1}{2} & \\
\hline
 & 0 & 1
\end{array}$$

and that of the fourth-order method:

$$\begin{array}{c|cccc}
0 & & & & \\
\frac{1}{2} & \frac{1}{2} & & & \\
\frac{1}{2} & 0 & \frac{1}{2} & & \\
1 & 0 & 0 & 1 & \\
\hline
 & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6}
\end{array}$$

a. Define a function `integrate(f, x0, t, M)` which for a given list of times $(t_i)_{i=0}^N$ returns the corresponding values $x_0, x_1, \dots, x_N$ with method M.

b. Implement the functions `M(f,t,x,h)` for the explicit Euler method and the Runge-Kutta method of order two. Compare the two methods.

c. Implement the function `M(f,t,x,h)` for the Runge-Kutta method of order four. Compare with the second-order method.

Exercise 11.3 - Movement of a planet

The goal is to simulate the two-dimensional motion of a planet orbiting around a fixed star. The star is supposed to be fixed at the origin and the position of the planet in the plane is described by the vector $x \in \mathbb{R}^2$. The star is supposed to interact with the planet with the potential:

$$V(x) = \frac{1}{\alpha}|x|^\alpha ,$$

for a certain $\alpha \in \mathbb{R}$, where $|x|$ denotes the euclidean norm of the vector x. Note that the gravitational potential corresponds to $\alpha = -1$. The equation of the planet in this force field is given by:

$$\ddot{x} = -\nabla V(x) = -x|x|^{\alpha-2} .$$

a. Rewrite the second-order differential equation as a first-order differential equation for x and $p = \dot{x}$.

b. Implement the function `f(t,xp)` corresponding to the equation found in the previous point.

c. Using the Runge-Kutta method of fourth-order, solve the differential equation for different initial data and different values of α and plot the $x(t)$ trajectories in the plane. Interpret the results and explain in particular why the cases $\alpha = -1$ and $\alpha = 2$ are different from the others.

Exercise 11.4 - Lorenz attractor

The Lorenz model is a system of three coupled differential equations of the form

$$\begin{aligned} \dot{x} &= \sigma(y - x) , \\ \dot{y} &= x(\rho - z) - y , \\ \dot{z} &= xy - \beta z , \end{aligned}$$

where rho, σ, β are three real parameters. This is a very simplified model of coupling between the atmosphere and the ocean proposed in 1963 by Edward Lorenz.

a. Write mathematically the expression of the function $f : \mathbb{R} \times \mathbb{R}^3$ allowing to put the Lorenz system in the form

$$\dot{x} = f(t, x)\,,$$

where x is the vector (x, y, z). Implement a function `f(t,x,rho,sigma,beta)` corresponding to the function f.

b. Write a function `plot_lorenz(rho,sigma,beta)` which for given parameters ρ, σ, β, plots the trajectory $(x(t), z(t))$ for $t \in [0, 20]$ from the initial data $x_0 = (x_0, y_0, z_0) = (1, 1, 1)$. Use, for example, the Runge-Kutta method of order four with a time step $\Delta t = 0.001$. Test with $\sigma = 10$, $\beta = 8/3$, and the values $\rho = 10, 15, 20, 25$, and describe what is observed.

c. Using SymPy, determine the stationary solutions according to the parameters ρ, σ, β, *i.e.*, the solutions of $f(t, x) = 0$ for all $t > 0$. Interpret the previous graphs in the light of this.

Exercise 11.5 - Cubic wave equation !!

The goal is to solve numerically the nonlinear wave equation on $\mathbb{R}$:

$$-\frac{\partial^2 u}{\partial t^2} + \frac{\partial^2 u}{\partial x^2} = u^3\,, \qquad u(0, \cdot) = u_0\,, \qquad \frac{\partial u}{\partial t}(0, \cdot) = v_0\,,$$

for $u : \mathbb{R}^+ \times \mathbb{R} \to \mathbb{R}$ with $u_0, v_0 : \mathbb{R} \to \mathbb{R}$ two given functions.

> **Note**
>
> The properties of this seemingly simple equation are very poorly understood mathematically, see the following research article for more details: doi:10.2140/apde.2012.5.411.

a. Rewrite the previous equation as two first-order equations in time for u and $v = \frac{\partial u}{\partial t}$.

b. By approximating the second derivative in space by finite differences as in Exercise 9.4 show that the equation can be approximated as follows:

$$\begin{aligned} \frac{\partial u_n}{\partial t} &= v_n\,, & u_n(0) &= u_0(x_n)\,, \\ \frac{\partial v_n}{\partial t} &= \frac{u_{n-1} - 2u_n + u_{n+1}}{h^2} - u_n^3\,, & v_n(0) &= v_0(x_n)\,, \end{aligned}$$

where $(x_n)_{n=0}^N$ denotes $N + 1$ evenly spaced points from h in the interval $[-L, L]$ and $u_n(t) = u(t, x_n)$ and $v_n(t) = v(t, x_n)$. For the conditions at the boundary of the domain, *i.e.*, when $n = 0$ or $n = N$, we take:

$$\frac{\partial v_0}{\partial t} = 0\,, \qquad \frac{\partial v_N}{\partial t} = 0\,.$$

c. Determine the function $f : \mathbb{R}^{2N+2} \to \mathbb{R}^{2N+2}$ allowing to put the previous approximation in the form $\dot{u} = f(t, u)$ for $u = (u, v)$ and implement this function.

d. Solve the differential equation given by $\dot{u} = f(t, u)$, for example, with the fourth-order Runge-Kutta method. A good choice of parameters is $L = 100$, $N = 1\,000$ and for the initial data $u_0(x) = e^{-x^2}$ and $v_0(x) = 0$. The speed of propagation of the wave is one and, therefore, after a time greater than L, the wave leaves the box $[-L, L]$ and no longer corresponds to a good approximation of the initial equation.

e. Using the `animation` module of Matplotlib, make a video showing the evolution of the wave as a function of time.

> **Hint**
>
> Use, for example, the `FFMpegWriter` function.

Exercise 11.6 - Bogacki-Shampine methods !!!

By combining two Runge-Kutta methods of different orders (for example, (2,3) or (4,5)), one will obtain an empirical estimate of the error over a time step. Using this error estimate, it is possible to adapt the time step, either by increasing or decreasing it, and thus adapt to the equation.

For a Runge-Kutta method of order s, an internal method of lower order (usually $s - 1$) is given by:

$$M^*(t, x, h) = h \sum_{i=1}^{s} b_i^* k_i\,,$$

where the k_i are identical to those of the s order method. An estimate of the error is then given by:

$$E(t, x, h) = M(t, x, h) - M^*(t, x, h) = h \sum_{i=1}^{s} (b_i - b_i^*) k_i\,.$$

Such a method is given by an extended Butcher table:

$$
\begin{array}{c|ccccc}
0 & & & & & \\
c_2 & a_{21} & & & & \\
c_3 & a_{31} & a_{32} & & & \\
\vdots & \vdots & & \ddots & & \\
c_s & a_{s1} & a_{s2} & \cdots & a_{s,s-1} & \\
\hline
 & b_1 & b_2 & \cdots & b_{s-1} & b_s \\
 & b_1^* & b_2^* & \cdots & b_{s-1}^* & b_s^*
\end{array}
$$

a. Implement the Bogacki-Shampine method of order (4,5). The original article is available at doi:10.1016/0898-1221(96)00141-1.

Hint

The coefficients of the Butcher tables are implemented in a `nodepy` module so the documentation is available here. The name of the method in this package is “BS5”.

12 Data science

The methodology of data science is to use available data (usally a large amount) to answers questions. Dealing with large number of data is not very practical with Python's default data structures or NumPy. First, the Pandas module specially designed for data analysis will be presented. The Pandas documentation is available here.

To load the Pandas module, it is usual to proceed as follows:

```
import pandas as pd
```

Next, real data will be analyzed by making statistics on the proportion of numbers beginning with a certain digit, as well as determining trends. Finally, three aspects of machine learning will be examined, with handwritten digit recognition, automatic differentiation, and the use of a neural network.

Concepts covered

- data import and analysis
- use of Pandas
- Benford's law
- least-squares methods
- image classification
- automatic differentiation
- neural networks

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

Exercise 12.1 - Introduction to Pandas

Creation. In Pandas, a data table is a `DataFrame` and constructing a table can be done manually from a dictionary, for example:

```
df = pd.DataFrame(
    {
        "Name": [
            "Braund, Mr. Owen Harris",
            "Allen, Mr. William Henry",
            "Bonnell, Miss. Elizabeth",
```

```
        ],
        "Age": [22, 35, 58],
        "Sex": ["male", "male", "female"],
    }
)
```

In Jupyter Lab, it suffices to execute the cell:

```
df
```

	Name	Age	Sex
0	Braund, Mr. Owen Harris	22	male
1	Allen, Mr. William Henry	35	male
2	Bonnell, Miss. Elizabeth	58	female

to display the table. One can see here, it consists of three columns (with given labels) and three rows that are labeled by integers by default.

Columns extraction. One single column of a `DataFrame` is called a `Series` and can be extracted easily:

```
df["Age"]
```

```
0    22
1    35
2    58
Name: Age, dtype: int64
```

On such a column, statistics can be done in a simple way, for exemple, to determine the oldest people and the mean age:

```
df["Age"].max(), df["Age"].mean()
```

```
(np.int64(58), np.float64(38.333333333333336))
```

Two columns can also be extracted:

```
df[["Age", "Sex"]]
```

	Age	Sex
0	22	male
1	35	male
2	58	female

New column creation. One can also add additional data, for example, a new column with 100 over the ages:

```
df["Age inv"] = 100/df["Age"]
df
```

	Name	Age	Sex	Age inv
0	Braund, Mr. Owen Harris	22	male	4.545455
1	Allen, Mr. William Henry	35	male	2.857143
2	Bonnell, Miss. Elizabeth	58	female	1.724138

We note that such operation are elementwise so there is no need for a loop, in the same spirt as NumPy, even for more complex logic:

```
def myfunction(l):
    if l["Sex"] == "male":
        return l["Age"]+5
    else:
        return l["Age"]+2
df["Age new"] = df.apply(myfunction, axis=1)
```

Rows extraction. Selecting rows satisfying some criterion is very important and quite simple, for example, people being more than 30 years old:

```
df[df["Age"]>30]
```

	Name	Age	Sex	Age inv	Age new
1	Allen, Mr. William Henry	35	male	2.857143	40
2	Bonnell, Miss. Elizabeth	58	female	1.724138	60

or female being more than 30 years old:

```
df[(df["Age"]>30) & (df["Sex"]=="female")]
```

	Name	Age	Sex	Age inv	Age new
2	Bonnell, Miss. Elizabeth	58	female	1.724138	60

This concept is very similar to NumPy indexing. For a more complex selection, we could use the following syntax:

```
value=3
df.query('`Age inv`>@value and Sex=="male"')
```

	Name	Age	Sex	Age inv	Age new
0	Braund, Mr. Owen Harris	22	male	4.545455	27

Slicing. Similar to NumPy, it is possible to use slicing to select part ot the table:

```
df.loc[0:1, "Sex":"Age new"]
```

	Sex	Age inv	Age new
0	male	4.545455	27
1	male	2.857143	40

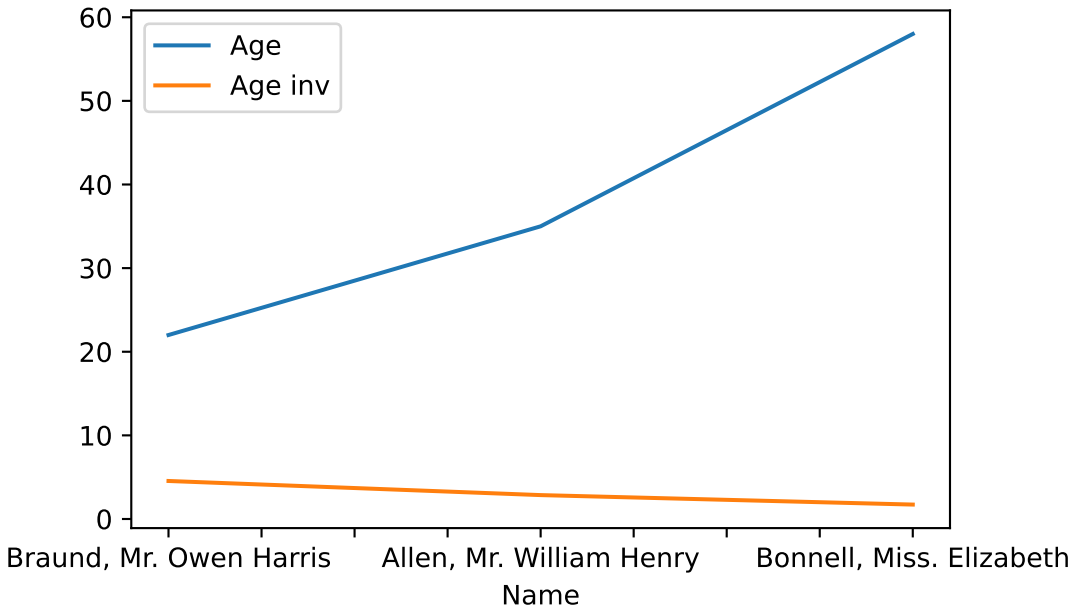
Note that the end points are included, unlike the standard Python method. However, here is slicing without endpoints:

```
df.iloc[0:2,3:5]
```

	Age inv	Age new
0	4.545455	27
1	2.857143	40

Plotting. Finally, data can be represented graphically in various ways. The simplest way is to do:

```
df.plot(x="Name", y=["Age","Age inv"])
```



a. Download the World Bank's education data in CSV format at the following address: https://data.worldbank.org/topic/education.

b. By inspecting the previous files, extract the data required to create a two-column table, the first containing country codes, the second country names.

> **Hint**
>
> Use the `read_csv` function to read a CSV file with Pandas.

c. Looking at the documentation of the `rename` and `set_index` functions, rename the column labels to `code` and `country`, then set the country code as the row index. The aim is that the name of the country can then be determined from its code with the function `loc["FRA"]`.

d. Determine the code for Zimbabwe from the table above.

e. Determine the code associated with the proportion of unemployed and the proportion of people with at least a master's degree from the data downloaded above.

> **Hint**
>
> Search for the "Unemployment" and "Master" strings in the file `Metadata_Indicator...`.

f. Plot the evolution of the unemployment rate in France.

> **Hint**
>
> The file `API...` contains a header and really only starts at line 5; use the `skiprows=4` option to ignore the header.

g. Write a function `time_plot(code, countries)` to plot the time evolution of the `code` indicator for the countries in the `countries` list. Test it on the unemployment rate and the proportion of people with at least a master's degree for different countries.

Exercise 12.2 - Benford's law

Benford's law predicts that statistically in a list of given numbers, the probability that one of these numbers begins with the digit 1 is greater than the probability that it begins with the digit 9. More precisely, Benford's law predicts that the probability that a number begins with the digit d is:

$$p_d = \log_{10}\left(1 + \frac{1}{d}\right),$$

where $\log_{10}$ is the logarithm in base 10. It is possible to verify that Benford's law is the only one that remains invariant by change of units, *i.e.*, by multiplying the numbers of the list by a constant, the previous probabilities remain unchanged.

a. Write a function `firstdigit(n)` which for a given number `n` returns its first digit and a function `occurrences(lst)` which returns the number of occurrences of the first digits in `lst`.

> **Hint**
>
> Make the `occurrences` function work even if the list contains zeros by ignoring them.

b. Check if Benford's law seems to be satisfied for the sequence of numbers $(2^n)_{n\in\mathbb{N}}$ by comparing the empirical histogram with Benford's law.

c. Check if Benford's law seems to be satisfied for the sequence of numbers $(3n+1)_{n\in\mathbb{N}}$.

d. By going to the INSEE website at the address: https://www.insee.fr/fr/statistiques/7631680, download the file in CSV format containing the French population data by sex and age grouped (POP1A). Import this data to have the population by postal code, sex, and age group.

> **Hint**
>
> Documentation on how to read files is available here.

e. Determine if the list of all populations by postal code, sex, and age follows Benford's law.

f. Sum the previous data to obtain the list of populations by postal code and determine if it follows Benford's law.

g. Repeat the previous two questions, but using the POP1B population file with ungrouped ages and using Pandas.

h. !! By going to the INSEE website or elsewhere download your favorite data set and test if it follows Benford's law.

> **Hint**
>
> Use, for example, the detailed French government accounts available here.

Exercise 12.3 - Least squares method

a. Reuse the INSEE data on the French population by sex and age POP1B. Write a function `pop(code)` that returns, as a list or NumPy vector, the population by age, without distinction between men and women, in the municipality with postal code `code`. Use this function to determine the total number of people living in postal code 75102.

b. Write a function `plot_pop(code)` to plot the population fractions (normalized by the total population of the municipality) as a function of age, without distinction between men and women, in the municipality with postal code `code`. Test for municipalities with postal codes 13201 and 75102.

For a given municipality, if we denote by $p(a)$ the fraction of the population of age a, we look for the coefficients r_0, r_1, r_2 such that the law:

$$p(a) = r_0 + r_1 a + r_2 a^2 \,,$$

is best satisfied for ages $a \geq 25$. To do this, we solve the least squares problem:

$$\min_{r \in \mathbb{R}^3} \|Xr - p\|^2 \,,$$

where by noting the vector of ages $a = (25, 26, \ldots, 100)$, X is the matrix of size 76×3 such that $X_{i,1} = 1, X_{i,2} = a_i, X_{i,3} = a_i^2$, and p is the vector of populations by age $p_i = p(a_i)$. The solution to this problem is $r = (r_0, r_1, r_2) \in \mathbb{R}^3$. This solution satisfies the equation:

$$X^\mathsf{T} X r = X^\mathsf{T} p \,,$$

where X^T is the transpose of X.

c. For the municipality with postalcode 13201, form the matrix X and the vector p, then determine the solution r.

d. For municipalities with postal codes 13201 and 75102, plot the theoretical curve $r_0 + r_1 a + r_2 a^2$ as a function of age over the data.

Exercise 12.4 - Handwritten number recognition

The aim of this exercise is to classify images of handwritten numbers, *i.e.*, to recognize handwritten numbers. This is a simple example of machine learning and one of the first industrial applications for automatic reading of cheques or postal codes.

The scanned handwritten digits dataset comes from the UCI ML repository. There are two ways to import this dataset. If the scikit-learn package (import name `sklearn`) is installed, simply run:

```
from sklearn.datasets import load_digits
digits = load_digits()
X, y = digits.data, digits.target
```

If the `sklearn` module is unavailable, the following commands can be used instead to load data:

```
import urllib.request, gzip, io
# download link
url =
 ↪  "https://raw.githubusercontent.com/scikit-learn/scikit-learn/main/sklearn/datasets/dat
# download gz file
file = urllib.request.urlopen(url)
# extracts the gz file
file = gzip.GzipFile(fileobj=io.BytesIO(file.read()))
# import txt file
digits = np.loadtxt(file, delimiter =',')
# extract images and labels
X, y = digits[:,:-1], digits[:,-1]
```

In both cases, `X` is a NumPy array containing numerous examples of handwritten digitized digits in an 8×8 pixel image stored as an array of 64 integers stored as floats. The `y` variable contains the integer between 0 and 9 corresponding to the digitized digit. This is referred to as *label*.

a. Determine the dimensions of `X` and `y` and deduce the number of examples contained in the database.

b. Display the data contained in `X` associated with the index `idx=12`. This is the 12th line of the `X` table, starting the numbering at zero.

c. Using NumPy function `reshape` and Matplotlib function `imshow`, display the image with index `idx=12`. You can use the `cmap='gray'` argument in the `imshow` call to display the result in grayscale. Which digit is encoded in this way?

For each digit class (from 0 to 9), the idea is to calculate its centroid, *i.e.*, the "average" representation of a class.

d. Define the `X` and `y` sub-tables corresponding to all digitized 0 digits.

e. For all the zeros from the previous question, calculate the mean value for each pixel, to define the "average zero".

f. For all the digits from 0 to 9, plot the associated average image on the same line as shown in Figure 12.1.

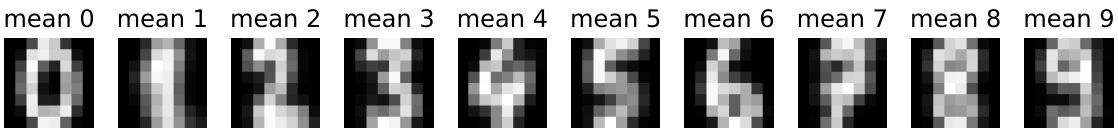


Figure 12.1: Averages of digitized figures.

Hint

Use Matplotlib's `subplot` function.

Finally, we will implement our own classifier: for a new digitized digit image, we will predict the class whose average digit is closest. To do this, we divide our dataset into two parts of similar size: the first part will serve as training data (`X_train` and `y_train`) and the second part will serve as test data (`X_test` and `y_test`).

g. Define variables: `X_train`, `y_train`, `X_test` and `y_test`.

h. For each digit in the training set, calculate the centroids (*i.e.*, the average digits) of the classes from 0 to 9. Note the variable containing the set of averages `centroids_train`.

i. Write a function which, given a number in the test set (`X_test`), returns the centroid of the nearest `centroids_train` in the Euclidean norm.

j. Finally, evaluate whether the digit thus obtained corresponds to the true digit using `y_test` and deduce an estimate of the percentage of correct predictions on the test set.

Exercise 12.5 - Automatic differentiation !

The aim of this exercise is to introduce one of the fundamental building blocks of machine learning: automatic differentiation. This is a technique for calculating derivatives or gradients of Python functions in a way that is virtually transparent to the user. Given a certain Python function `f(x)`, the aim of automatic differentiation is to make it virtually as easy for the user to evaluate the derivative of `f` at a point `x=1` as it is to do `f(1)`, even if the function f is complicated. This is the fundamental building block enabling machine learning to learn parameters by optimizing complicated nonlinear functions.

The idea behind automatic differentiation is to use the derivation rule for compound functions and knowledge of the derivatives of basic functions. In fact, a Python function is "just" a composition of basic functions (or instructions).

For sake of simplicity, we are only interested here in the composition of functions on $\mathbb{R}$. For any $i \in \{0, 1, \dots, n\}$, let $f_i : \mathbb{R}$ be an elementary function whose derivative is known. Consider the composition of the first $i \leq n$ functions:

$$F_i = f_i \circ f_{i-1} \circ \cdots \circ f_1 \circ f_0 \,.$$

The derivative is given by the composition rule (or chain rule):

$$F_i' = (f_i \circ F_{i-1})' = (f_i' \circ F_{i-1}) \cdot F_{i-1}' \,.$$

This gives a recursive way of calculating F_n' with the anchor $F_0' = f_0'$. To calculate the value of $F_n(x)$ for a given x, Pyhon will intrinsically calculate successively $F_0(x)$, then $F_1(x)$, $F_2(x)$, up to $F_n(x)$. Automatic differentiation consists of evaluating $F_0'(x)$, $F_1'(x)$ up to $F_n'(x)$ at the same time or later.

Note that automatic differentiation is neither a numerical approximation nor symbolic calculation. In fact, the value of the derivative is determined exactly (to machine precision), which is not the case with numerical approximation:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h},$$

with some $h > 0$ small. It is not symbolic calculation either, as there are no symbols in automatic differentiation, only real numbers. Automatic differentiation calculates the value of the derivative at a given point, whereas symbolic differentiation does this for any symbol.

a. The first step is to define the derivatives of the elementary functions. To do this, build the tuples $\tilde{f} = (f, f')$ manually for the elementary functions sin, cos, op : $x \mapsto -x$, inv : $x \mapsto x^{-1}$ et square : $x \mapsto x^2$.

b. A composition of functions $F_n = f_n \circ f_{n-1} \circ \cdots \circ f_1 \circ f_0$, will be stored in Python as the list of tuples $[\tilde{f}_0, \tilde{f}_1, \dots, \tilde{f}_n]$. In Python, define the composition corresponding to the function:

$$F(x) = \cos\left(\frac{1}{\sin(-\cos x^2)^2}\right).$$

c. Write a function `eval(list, x)` which, given a list of tuples defining a composition of functions F_n, returns $F_n(x)$. Test on the previous example.

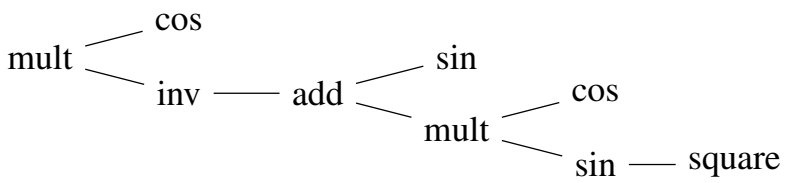
d. Write a function `autodiff(list, x)` which, given a tuple list defining a composition of functions F_n, returns $F_n(x)$ and $F_n'(x)$. Test again on the same example.

The previous approach only allows the composition of functions of one variable, which is very limiting, as sum and multiplication are functions of two variables. The idea is to be able to consider more complicated functions as well, for example:

$$G(x) = \frac{\cos(x)}{\sin(x) + \cos(x)\sin(x^2)}.$$

e. As before, implement the sum function add : $x, y \mapsto x + y$ and the multiplication function mult : $x, y \mapsto xy$ in tuple form.

f. The previous function G can no longer be represented in Python as a list of compositions, as it has the structure of a graph:



We choose to store it in Python as a list of lists, with the first element of each list being the function to be applied, and the arguments being the following children. For the example G above:

```
myG = [mult, [cos], [inv, [add, [sin], [mult, [cos], [sin, [square]]]]]]
```

Write the previous simple composition F in this new form, as well as the function:

$$H(x) = \frac{\cos(x^2) + \sin x}{\cos x + 2\sin(x^{-1})}\,.$$

g. Write a function `eval(list,x)` which evaluate an expression in the form of a list of lists described above at `x`. Test on functions F, G, and H.

h. !! Write a function `autodiff(list,x)` which, in addition to returning the evaluation of the function at `x`, also returns the evaluation of its derivative at `x`.

i. !! The previous implementation is not very usable concretely. In practice, automatic differentiation is coded by overloading basic operations in order to be relatively transparent to the user. Using the package JAX or PyTorch, determine the derivative of the following function by automatic differentiation at `x=0.4`:

```
def f(x):
    for i in range(50):
        if x>0.5:
            x = 3.7*x*(1-x)
        else:
            x = 3*x*(1-x)
    return x
```

Finally, compare with numerical differentiation.

> **Hint**
>
> The documentation on automatic differentiation with JAX is here and with PyTorch here.

Exercise 12.6 - Neural network !

The aim of this exercise is to introduce the concept of neural network for finding a real function for which only its noisy evaluation is known. More precisely, consider the function $f : [0,1] \to \mathbb{R}$ defined by:

$$f(x) = 1 + \sin(4\cos x)^2\,,$$

and consider the data generated by 500 noisy evaluations of f:

```
rng = np.random.default_rng(123456)
f = lambda x: 0.1 + np.sin(4*np.cos(x))**2
x = rng.random(500)
y = f(x) + rng.normal(0,0.1,500)
```

The aim is to forget that the data were generated in this way and to find a function approximating f from x and y alone. To achieve this, a single-layer neural network will be used, and we speak of learning the function from the data. The principle is to construct the learned function in the form:

$$f_\omega(x) = \frac{1}{n}\sum_{i=0}^{n-1} w_i \sigma(a_i x + b_i)\,,$$

where $\omega = (a, b, w) \in \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n$ are parameters to be determined and σ the sigmoid function:

$$\sigma(x) = \frac{1}{1+e^{-x}}\,.$$

This is a single-layer neural network with n neurons. The principle is to sum n nonlinear functions (the sigmoids) with input weights $a = (a_i)_{i=0}^{n-1}$, biases $b = (b_i)_{i=0}^{n-1}$, and output weights $w = (w_i)_{i=0}^{n-1}$. Parameters are chosen to minimize the following cost function:

$$J(\omega) = \sum_{k=0}^{499} \big(f_\omega(x_k) - y_k\big)^2\,.$$

The strategy for minimizing J on the parameters ω is to perform a gradient descent starting with random values ω_0 of the parameters and then successively defining:

$$\omega_{i+1} = \omega_i - \eta J'(\omega_i)\,,$$

where $J'(\omega)$ is the gradient of $J(\omega)$ with respect to the parameters and $\eta \in (0, 1]$ is a parameter called the learning rate. The idea of the gradient descent algorithm is to move the parameters in the direction of greatest gradient in order to minimize $J(\omega)$. This is known as parameters learning.

a. Plot the data x and y and the function f.

b. Determine the value of the cost function J for the function f:

$$J_f = \sum_{k=0}^{499} \big(f(x_k) - y_k\big)^2\,.$$

c. Define the sigmoid function σ in Python and its derivative σ', and represent the sigmoid graphically.

d. Implement in Python a function `F(x,omega)` corresponding to the function $f_\omega(x)$. Make the function `F(x,omega)` vectorized, *i.e.*, if $x = (x_0, x_1, \dots, x_{k-1}) \in \mathbb{R}^k$, then the function should return $(f_\omega(x_0), f_\omega(x_1), \dots, f_\omega(x_{k-1}))$.

e. Calculate by hand the gradient of $f_\omega(x)$ with respect to ω (and not with respect to x) and implement this gradient in Python, taking care that it is also vectorized.

f. Implement in Python $J(\omega)$ and its gradient $J'(\omega)$.

g. With learning rate $\eta = 0.01$ and four neurons, learn the parameters $\omega \in \mathbb{R}^{12}$ that tend to minimize J. Compare the value of the cost function $J(\omega)$ of the learned function f_ω with the value of the cost function J_f of the function f. Plot the function f_ω as a function of x for these parameters and compare with the function f.

13 Cryptography

Since the Caesar cipher, the cryptographic methods allowing to transmit secret messages evolved following the progress allowing to break them. The Vigenère cipher which is an improvement of the Caesar cipher will be studied and we will see how it is possible to break this encryption method. Then, the RSA encryption method which is one of the most used asymmetric cryptography methods today will be introduced.

Concepts covered

- Vigenère cipher
- greatest common divisor
- text import
- prime and pseudoprime numbers
- Fermat's little theorem
- Euclid's algorithm
- Miller-Rabin algorithm
- optimization by decorator
- asymmetric RSA encryption

Exercise 13.1 - Vigenère cipher

The Vigenère cipher consists in choosing a key formed by a secret word (in capital letters) and to transform it into a vector whose elements are the positions of these letters in the alphabet. For example, "ASECRET" corresponds to (0, 18, 4, 2, 17, 4, 19). To encode a text (in capital letters, without spaces or punctuation) with the "ASECRET" key, you just have to shift the first letter by 0, the second by 18, the third by 4, and so on, and repeat in a loop. The details, especially the historical ones, are available on Wikipedia.

a. Write a function `to_int(s)` that transforms a character into its place in the alphabet and write the inverse function `to_chr(i)`.

Hint

See the documentation of the `ord` and `chr` functions.

b. Write a function `crypt(text, key)` that encrypts `text` with the secret word `key`.

c. Write a function to decipher a text by knowing the key.

Exercise 13.2 - Breaking the Vigenère cipher !

Charles Babbage was the first to break the Vigenère cipher. The idea is that three consecutive letters appearing several times in the cipher text are likely to be the result of encrypting the same letters of the message with the same letters of the key. This is even more likely with a group of four letters. Thus, the spacing between two same groups of cipher letters is a multiple of the key length. For example, if the repetition of one group is spaced 28 letters apart, then the repetition of another is spaced 91 letters apart, the greatest common divisor (GCD) of 28 and 91 is 7. So, it is likely that the key has 7 letters. Then knowing the size of the key, it is enough to base on the fact that the letter “E” is the most common in English. For this strategy to have a chance of success, the size of the text must be large enough.

a. Write a function to calculate the GCD between two numbers. Write another function to calculate the GCD between a list of numbers.

b. Visit the Project Gutenberg website (https://www.gutenberg.org/), choose your favorite English text and download it in “Plain Text” format. Write a function that converts the text to uppercase and strips it of all punctuation and other special characters.

Hint

To convert a text to uppercase (converting accents) and remove all punctuation and other characters, it is possible to use the following function:

```
import unicodedata, re
def convert_upper(text):
    # convert to upper case
    text = text.upper()
    # convert accents
    text = unicodedata.normalize('NFKD', text)
    # delete special characters
    regex = re.compile('[^a-zA-Z]')
    text = regex.sub('', text)
    return text
```

c. Keep about of few thousand characters in the middle of the chosen text and encrypt it with a key. Then, write a function to determine the length of the key by looking at identical strings of three or more characters in the encrypted message.

Hint

First, form a dictionary with as key all occurrences of three letters and as value the positions of the occurrences. Then, determine the list of distances between the occurrences of three letters, then calculate the GCD of these distances. If this GCD is equal to 1 or is too small, then try again but with strings of more than three characters.

d. Write a function to decrypt an encrypted message by returning the key. Try to decrypt the text of your favorite author with this function.

> **Hint**
>
> To find the first letter of the key, it is necessary to calculate the number of occurrences of the 26 letters of the alphabet in the encrypted message that have been encrypted with the first character of the key. In principle, the letter with the maximum occurrence corresponds to the letter "E". It is then enough to do the same thing to find the other letters of the key.

Exercise 13.3 - Generating prime numbers

Most current encryption algorithms are based on the use of large prime numbers. The goal is to write a function to generate prime numbers. The first step is to generate a large random number, *i.e.*, having a certain number of bits. Then, a primality test allows to decide if this number is prime or not. If $\pi(n)$ denotes the number of primes less than or equal to n, then asymptotically $\pi(n) \approx \frac{n}{\ln n}$. For a number less than n drawn at random, the probability that it is prime is about $1/\ln(n)$. For example, to generate a prime number of 1 024 bits (the minimum guaranteeing reasonable security at the moment), *i.e.*, of the order of 2^{1024}, one must try $\ln(2^{1024}) \approx 710$ random numbers before finding one that is prime. Since all even numbers are clearly not prime, it is enough to test an average of 355.

a. Write a program to generate an odd random number of k bits, *i.e.*, between 2^{k-1} and 2^k.

> **Hint**
>
> The fastest way to implement this is to use the bit operations explained here.

The simplest way to determine whether a number n is prime is to try to divide it by all the integers $1 < d < n$. There are two reasons for not testing all d between 2 and $n-1$. The first is that it is unnecessary to try even d greater than 2. The second is that there is no point in testing numbers larger than $\sqrt{n}$.

b. Write an algorithm `isprime(n)` to determine if a number is prime or not.

c. Write a function `generate(k,primality)` to generate a random prime of `k` bits with the `primality` test. Test with the primality test `isprime`. Is it reasonable to expect to generate a prime number of 1 024 bits with this algorithm?

Exercise 13.4 - Generating pseudoprime numbers

The previous algorithm for generating primes being unusable for generating large primes, another approach, probabilistic, is advocated. A probabilistic primality test decides that a number is

prime if it is prime with a very high probability. Such a number is called pseudoprime. Thus, a probabilistic test can be wrong and assume that a number is prime when in fact it is not.

The simplest primality test is based on Fermat's little theorem: if n is prime, then $a^{n-1} = 1 \pmod{n}$ for all $1 \leq a \leq n-1$. So, if we find an a such that $a^{n-1} \neq 1 \pmod{n}$, then n is not prime. Fermat's primality test tests N values of a chosen at random and if $a^{n-1} = 1 \pmod{n}$ for these N values, then it declares that n is probably prime. Carmichael numbers are not prime, but satisfy $a^{n-1} = 1 \pmod{n}$ for all a prime with n. The prime Carmichael numbers are 561, 1 105, and 1 729. If n is not a Carmichael number, then the probability that Fermat's test is wrong is 2^{-N}. Choosing, for example, $N = 128$, we get a probability of being wrong of less than 3×10^{-39}.

a. Write a function implementing Fermat's primality test. Use this test to generate random pseudoprimes.

> **Hint**
>
> See the documentation for the `pow` function for a quick implementation. If OpenSSL is installed on your computer, it is easy to check if a number is prime with the command `openssl prime 11`, for example, for 11.

b. ! Improve the speed of the previous algorithm by first testing whether n is divisible by primes less than 1 000 before applying Fermat's test.

Fermat's primality test allows to generate large pseudoprime numbers with a good probability of being right. The main problem comes from the existence of Carmichael numbers which are excluded from this probability. The Miller-Rabin primality test avoids this problem.

c. !! Understand and implement the Miller-Rabin primality test explained in detail on Wikipedia.

Exercise 13.5 - RSA encryption

The RSA algorithm, from the initials of Ronald Rivest, Adi Shamir, and Leonard Adleman who invented it in 1983, is one of the most widely used asymmetric cryptographic algorithms still in use today. Asymmetric encryption allows an encrypted message to be transmitted to Alice without having to first transmit a secret key to Bob who encrypts the message. The creation by Alice of a public key is enough for Bob to encrypt the message and for Alice to decrypt it with his private key. There are three main steps in the RSA algorithm:

Creation of the keys. Alice wanting to receive a secret message chooses two very large prime numbers p and q which she keeps secret. She then computes $n = pq$ and the Euler's totient function $\varphi(n) = (p-1)(q-1)$ which counts the number of integers between 1 and n which are prime with n. Then, she chooses an encryption exponent e that is prime with $\varphi(n)$. The public key of Alice is given by the pair (n, e). Finally, Alice computes the decryption exponent d which is the inverse of e modulo $\varphi(n)$, *i.e.*, such that $ed = 1 \pmod{\varphi(n)}$. The private key of Alice is (p, q, d).

Encryption of the message. To encrypt her message, Bob first transforms it into an integer $M < n$. The encrypted message is then given by:

$$C = M^e \pmod{n}.$$

Decryption of the message. The encrypted message C is then transmitted to Alice. To decrypt it, Alice calculates:

$$M = C^d \pmod{n},$$

which is again the original message.

> **Note**
>
> The prime numbers p and q must be truly random, otherwise it is possible to guess their values. The random numbers generated by the `random` module are generated with the Mersenne Twister algorithm. This algorithm is not considered cryptographically secure in the sense that an observation of about a thousand random numbers generated by this algorithm is sufficient to predict all future iterations. To generate cryptographically secure random numbers one would have to use the secrets module.

a. Show mathematically that the decoded message corresponds to the original message.

> **Hint**
>
> If $a = b \pmod{\varphi(n)}$ and M is prime with n, then $M^a = M^b \pmod{n}$.

b. Given e and $\varphi(n)$, write a function `bezout(e, phi)` to determine d such that $ed = 1 \pmod{\varphi(n)}$.

> **Hint**
>
> Use the generalized Euclid algorithm to determine the GCD g between two numbers a and b and x and y satisfying $ax + by = g$.

c. Write an algorithm `generate_keys(length)` that generates prime numbers p and q such that n has `length` bits, then determines $\varphi(n)$, e, and d, and finally returns the public key (n, e) and the private key (p, q, d).

d.

By choosing to encode each character on 8 bits, a string of length ℓ is written as a list $(a_0, a_1, \ldots, a_\ell)$ with each $0 \leq a_i \leq 255$. This list can be converted into an integer k in the following way:

$$k = \sum_{i=0}^{\ell} a_i 256^i.$$

Write a function `toint` and a function `tostr` allowing respectively to convert a string into this integer and vice versa.

e. Write a function to encrypt a text with a public key and another to decrypt it with the private key. To do this, we must make sure that the text is convertible to an integer less than n, otherwise we must split it into blocks and encrypt them separately.

Exercise 13.6 - Breaking RSA encryption !!!

Here is a public key:

```
(681173384823993924634706123599189498672431584217436911278577378677214308161 93,
 ↪ 857480488977203937958496372891351154528533346142902255835809356706830819 3213)
```

and a message encrypted with this public key:

```
[585964756195065347905137644572870097507827740916505134921296164744425489812 18,
 ↪ 631044417760880427913263809393347832875875146680949667389450856812796669 11085,
 ↪ 285540710278784283552200781062397242699759100959189201533956214022774751 01298,
 ↪ 597415013381133343753644438714453796529060867921756779777325611022428956 63472,
 ↪ 581127503299642996191371492487876120300295275466426173073332930762803149 87312,
 ↪ 357978894808801313614190441062853820242277522454839285319351792653715279 89598,
 ↪ 407206725649039531032575227039893283345874374156992335140269650268952566 21366]
```

a. Decrypt the previous message!

> **Hint**
>
> It is probably necessary to choose a suitable algorithm, for example, using quadratic screens (QS, MPQS, SIQS).