Scientific computing with Python

How to apply the principles "Simple is better than complex" and "Complex is better than complicated" to scientific computing

Stefano Piemonte

Contents

Part One: Python and programming basics

1. Python as a simple calculator

> Python is an experiment in how much freedom programmers need. Too much freedom and nobody can read another's code; too little and expressiveness is endangered.
>
> Guido van Rossum

In this chapter the first basic instructions and mathematical tools of python will be explained. The chapter will provide also an understanding of how the programming language is structured.

1.1 Variables and basic mathematical expressions

The basic numerical types of Python are **int**, **float** and **complex**, representing integers, real and complex numbers, respectively. Variables can store the result of any mathematical expressions, and in general variables are a reference to an object, something more abstract and general than a simple number. As python is **dynamically typed**, no keywords or instructions are required to specify what is type of the data that a variable is going to store.

```
a = 3+5-9**2 # an integer expression
b = 4./5. # a float number
c = 4.2-7.6j # A complex number
```

The operator ** is used to denote the exponentiation, while a number followed by the letter "j" is interpreted as the imaginary part of a complex number. Comments are introduced by the character "#" and everything following "#" until a new line is considered a comment.

More on ... 1.1.1 — Python and integer division. In many programming languages, including C and C++, the division between two integers returns an integer, so that "1/2" is simply zero. Such behavior introduces many errors in beginner's (and also not beginner's ...) code. The same occurred in Python up to the version 2.x, while in Python 3 it has been decided that the division between two integers returns a float, so that "1/2" is "0.5". The decision was quite difficult, coming after a "a major flamewar on c.l.py" [1] and negative criticisms from senior programmers: *You're pissing off a fairly significant fraction of the language's existing constituency in hopes of attracting some non-programmers to the language who may not come anyway for various other reasons* [2].

In python 3 there is a new division operator "//" to mean the division between integers as in the older versions of python:

```
#a is equal to 1.5
a = 3/2
#a is equal to 1
a = 3//2
```

Given that in daily life programmers develop code in multiple languages, the recommendation is to always use "1./2." to mean the division returning a float.

Floating point numbers, due to their limited precision, cannot represent all real numbers. Rather they are able to represent only a subset of rational numbers. In python, even simple numbers are objects, meaning that they can perform many more operations other than additions, multiplications, etc. For instance, we can see the rational number represented by a float using

```
a = 4./5.

print(a.as_integer_ratio())
```

The built-in function **print** is used to print to the shell variables and strings given as arguments. The syntax **variable_name.function_name**() allows to call a "function acting on a variable" (see chapter 5). The function "as_integer_ratio" acts on the floating point number pointed by the variable "a", and computes the numerator and the denominator of the rational number represented when computing the ratio 4/5 in floating point arithmetics. Note that the output of the function above is not simply (4,5), due to the limitation of the class float in representing real numbers.

Exercise 1.1 Explain the behavior of integer ratios of the variables in the script below. Are they different or equal? Why?

```
a = 5./3.
b = 11111.258 + 5./3. - 11111.258
c = 0.058 + 5./3. - 0.058
d = 5./3. + 11111.258 - 11111.258
e = 11111.258 - 11111.258 + 5./3.
```

```
print(a.as_integer_ratio())
print(b.as_integer_ratio())
print(c.as_integer_ratio())
print(d.as_integer_ratio())
print(e.as_integer_ratio())
```

What happens if you multiply or divide the number 11111.258 by ten thousands or by ten millions? ■

More on ... 1.1.2 — The dir command. It is possible to inspect the functions that can be called on a variable using the command **dir**.

```
a = 4./5.

print(dir(a))
```

Among many functions with a cryptic name, we can see that we can test whether a float is integer with the function **is_integer**.

Variables can be set also to be equal to any expression involving other variables

```
l1 = 4
l2 = 5
rectangle_area = l1*l2
```

1.2 The modules math and cmath

Python is plenty of built-in modules implementing many functions ranging from mathematics, networking and operating system utilities to more complex tools for iterators and multithreading. There are two modules implementing basic mathematical functions, **math** and **cmath**, for real and complex numbers, respectively. Functions, classes and variables defined in a module are loaded using the keyword **import**

```
import math

print(math.sqrt(2.))
print(math.sin(math.pi/2.))
print(math.pi)
```

The square root **sqrt**, as many other functions, has a different behavior if called from the module **math** or **cmath**

```
import math
import cmath

print(cmath.sqrt(-4.))
print(math.sqrt(-4.))
```

For a negative input number, in the second case an exception **ValueError** is raised, while in the first case the output is "2j". Depending on the context, it might be useful to load cmath instead of math. Note however that an exception raised is not necessary a bad behavior. If the code is supposed to compute the side of a square given the area, an imaginary output for a negative input area would be weird, an exception instead can be handled providing an error message (see Section 5.7).

Exercise 1.2 Remember that the python library has already many useful tools to solve actual problem. Can you fly if you import the module **antigravity**?

```
import antigravity
```

(don't execute the code above if you are working outside!). ■

The module name can be also specified by the user

```
import math as m

print(m.pi)
```

or left unspecified

```
from math import *

print(pi)
```

However, in the latter situation, variables with conflicting names are overwritten, therefore it is recommended to always provide a name when importing a module.

More on ... 1.2.1 — Finding good names for variables. Python is indeed a dynamical programming language, therefore it is possible to reassign variables to functions and numbers and vice-versa. Such behavior might result in unexpected errors

```
from math import *

#We compute the square root of four
sqrt = sqrt(4)
print(sqrt)
#... but now sqrt is 2 and not a function anymore!
# and we cannot call:
b = sqrt(8)
```

It therefore always important to choose names describing at best the variable and the content stored in it, avoiding simple names used already by python built-in functions and modules. Descriptive names will improve also the readability of the code:

```
from math import *

sqrt_of_four = sqrt(4)
```

```
print(sqrt_of_four)
```

It is possible to import only selected features using the keyword **from**, useful especially for big modules.

```
from math import sqrt

print(sqrt(4.))
```

Exercise 1.3 A complex number can be viewed geometrically as a point in a two dimensional space, the so-called Argand diagram. In this diagram, the x- and y-coordinates represent the real and imaginary part of a complex number. The distance of a point in Argand diagram from the origin is the absolute value of the corresponding complex number, while the angle with the x-axis is the phase. Rotations of angle θ of points in the two-dimensional space are represented simply by multiplication of the corresponding complex number by $\exp(i\theta)$.

Compute the distance from the origin of the point (7, 3) *without using the square root function*. Using complex number arithmetics of the module **cmath**, perform a counter-clockwise rotation of the point (7, 3) around the origin of 30 degrees in the two-dimensional space. ■

1.3 Boolean expressions

Boolean expressions are typically introduced to check whether a given condition is fulfilled. The keywords **and**, **not** and **or** can test the simultaneous or alternative validity of multiple expressions. Their use is relatively straightforward.

```
a = 5.
b = 9

print(a > b)
print(a == b)
print(a < b and a >= 3)
print(a < b and not b < 11)
print(a == 5 or b == 0)
```

There is also the operator ^ to test if *only one* out of two conditions is **True** (exclusive or, also called xor)

```
print((a == 5) ^ (b == 0))
```

The priority of the operators requires in this case additional parenthesis for the two conditions.

More on ... 1.3.1 — The "is" operator. In Python variables are references to objects, therefore there exists an operator **is** to check if two variables are pointing to the same object or not. It is logically different from the operator "==", which checks if two variables are equal. Consider for instance the following code

```
print(1 is 3/3)
print(1 == 3/3)
```

It prints "False" and then "True", because the division between two integers is a real number and 1.0 *is not* 1, although 1.0 *is equal* to 1. The behavior of the "is" operator is sometimes weird, the code below

```
print(3 is 3)
print(2**8 is 2**8)
print(2**128 == 2**128)
print(2**128 is 2**128)
```

if executed prints "True", "True", "True" and "False". In the last case, python computes the power expression 2^{128} twice and allocates two objects to store the results, so that they are not the same. We can also check the identity of an object by the built-in function "id", which returns an identifier, an integer, for any python object

```
print(id(2**128) == id(2**128))
```

The "is" operator, as any other comparison operator, can be chained. The result can be puzzling, as "True is not True" *is* "False" but "True is not True is False" is "False" again, because the above expression is interpreted as "(True is not True) and (True is False)"

Exercise 1.4 Explain why the following expression

```
5 == 4 is not True
```

is False, despite the fact that $5 \neq 4$. ∎

1.4 If-else

Typical algorithms require to execute certain instructions depending on whether a condition is fulfilled or not. Consider for instance a simple script that checks whether a given number is even or odd:

```
num = int(input("Give me an integer number: "))

if num % 2 == 0:
      print("The number", num, "is even!")
else:
      print("The number", num, "is odd!")
```

In the first line, we use the function **input** to ask to the user to input an integer number, using a string delimited by double quotes "". The output of **input** is also string, that is converted to an integer by the function **int**. If the input provided is not an integer, an exception will be raised (see section 5.7). An integer is even if the remainder of the division (operator %) by two is zero. The condition is checked

after the **if** keyword, closed by the ":". The instructions **print** that are executed if the condition is true is followed by an indentation, which is mandatory as python does not use curly brackets or any other delimiter to create separate instruction blocks, contrary to many other programming languages. Finally the keyword **else** distinguishes the block of instructions which are executed if the previous condition was false.

It is also possible to test multiple conditions in sequence, using the keyword **elif**. The following script illustrates the use in the test if a year is in the past, future or is the current year.

```
import time

year = int(input("Give me a year: "))
current_year = time.localtime().tm_year

if year > current_year:
      print("The year", year, "is in the future!")
elif year == current_year:
      print("We are in", year, "!")
else:
      print("The year", year, "is in the past!")
      if year < 0:
            print("It is even BC!")
```

The module time **time** is used to get the current year, by calling the function **localtime** and extracting the record **tm_year** from the output using the dot "." twice. As in the last block, additional nested **if** can be inserted in any sub-block.

Notice that the python interpreter will stop as soon as a true condition is found in an **if-elif-else** block and the corresponding instructions are executed, without checking the next remaining conditions.

Exercise 1.5 Write a program that asks to the user a number between 1 and 6 and compares it to a random number generated by the function random.randint(from,to) from the module **random**. The output should be for example:

```
I have just generated a random number between 1 and 6.
Could you guess it? Try : 4
Wrong, it was 5 !
```

The program should check if the number provided by the user is in the interval required. ■

An **if-else** condition can be also expressed in a single instruction.

```
num = int(input("Give me an integer number: "))
is_even = True if num % 2 == 0 else False
```

1.5 While loops

If-else blocks are already a powerful tool to implement complex algorithms. However it is frequently needed that certain instructions are executed not just only once, but *many times* until a condition holds. Computers, unlike humans, express their maximal computational power when performing many repeated operations. The required **loop** controlled by a boolean condition is introduced by the keyword **while**.

Consider for instance an approximation of the exponential function given by the formula

$$\exp(x) = 1 + x + \frac{1}{2}x^2 + \frac{1}{6}x^3 + \cdots = \sum_{n=0}^{N} \frac{1}{n!}x^n \tag{1.1}$$

The sum for the first three or four terms can be written in all terms explicitly, but it would be impractical if the sum runs up to ten or more terms. Therefore we start an index n and the result from zero, and we run a **while** loop up to N adding to result each term $\frac{1}{n!}x^n$.

```
import math

N = 10
x = 1.2
n = 0
result_exp = 0

while n < N:
      result_exp += (x**n)/math.factorial(n)
      n += 1

print("exp(1.2) is approximately ", result_exp)
```

The operator "+=" in the while loop is also equivalent to

```
      result_exp = result_exp + (x**n)/math.factorial(n)
```

The execution of loops can be also interrupted by the instruction **break** and combined with **else**. The following simple script checks whether a given number is prime and is illustrating the use of **break** and **else** in a **while** loop.

```
num = int(input("Give me an integer larger than one: "))

if num > 1:
      i = 2
      while i <= num//2:
            if num % i == 0:
                  print(num, "is not prime!")
                  break
            i += 1
      else:
            print(num, "is prime!")
else:
```

```
        print(num, "is not larger than one!")
```

After checking that the number provided by the user is indeed larger than one, an index variable i is initialized to two. The while loop runs until the index i is smaller than or equal to the half of the number provided, as no larger integers would divide exactly "num". The **if** condition checks whether the number is divisible by the index i. If this is the case, the number is not prime, and there is no reason to continue further, therefore we can use the instruction **break** to interrupt the **while** loop. Otherwise, the index is incremented by one and the next number is considered. The **else** condition is executed if the previous while loop did not break before the end, i.e. if the number is prime.

More on ... 1.5.1 — Spaghetti code. The name *loop* is coming from old programming languages where the first sum computing the exponential would have been implemented by the instruction **goto** and labels in a way like

```
x = 1.2
n = 0
result_exp = 0
sum:   result_exp += (x**n)/math.factorial(n)
if n < 10:
      n += 1
      goto sum
```

The execution of the above code jumps to the label "sum", creating effectively a loop which is interrupted only when $n = 10$. However in more complex situations, the "goto" instruction creates fragmented code which is difficult to read and debug (the execution can jump to a label at any time), leading to the so called "spaghetti code". For this reason the instruction **goto** is deprecated and not implement in modern programming language, such as python.

Exercise 1.6 Write a program that prints the first hundred numbers of the Fibonacci sequence F_n, where each number is the sum of the previous two $F_n = F_{n-1} + F_{n-2}$, starting from $F_1 = 1$ and $F_2 = 1$. ∎

Exercise 1.7 Using the module **turtle** and a **while** loop, write a program that draws a regular heptagon.

Possible useful functions:

1. **turtle.forward**(x): to move forward of x.
2. **turtle.right**(x): to turn right of x, an angle between 0 and 360.
3. **turtle.left**(x): to turn left of x, an angle between 0 and 360.
4. **turtle.speed**(x): to control the speed of the simulation.
5. **turtle.mainloop**(): to stop the execution of the script at the end of the simulation.

Read the manual Turtle graphics for more information. ∎

Exercise 1.8 The St. Petersburg paradox is referred to a game which allows to win theoretically an infinite amount of money after a long sequence of games:

"Petrus in altum proiicit nummum, idque donec in Terram delapsus notatam semel ostenderit frontem: si vero id primo contingat iattu, tenetur Paulo dare ducatum unum, si secundo duos, si tertio quatuer, si quarto octo, et sic porro duplicando quouis iactu ducatorum numerum. Quaeritur fors Pauli?"

"Peter tosses a coin and continues to do so until it should land "heads" when it comes to the ground. He agrees to give Paul one ducat if he gets "heads" on the very first throw, two ducats if he gets it on the second, four if on the third, eight if on the fourth, and so on, so that with each additional throw the number of ducats he must pay is doubled. Suppose we seek to determine the value of Paul's expectation. [4]"

Write a program that asks to the user how much he/she is willing to bet for such a game and runs it. The output should be for example:

```
I am going to toss a coin many times.
You are going to win 2**k if no head appears after k throws.
How many ducats do you want to bet? 10
Bad, a head came out after 2 throws.
You get only 4 ducats and you have lost 6 ducats.
```

Run the script many times and ask yourself how much are you ready to pay to play the game. ■

1.6 Input-output to a file

Instead of writing the output to the terminal, it is possible to store informations to a file. A file must be opened by using the function **open**, and we must specify whether we intend to use the file for input or output.

```
#We open a file to write some
#content in it (mode "w")
f = open("my_file.txt","w")
print("File",f.name,"open in mode", f.mode)

#A newline is introduced by \n
f.write("Hello world\nfrom python!")
f.write("!!!!")
#close the file
f.close()

#Open a file in read mode ("r")
f = open("my_file.txt","r")
print(f.read())
f.close()
```

If the file already exists, it is overwritten by the mode "w" (write). The mode "a" ("append") does not overwrite the content and the new text is appended to the file.

More on ... 1.6.1 — with statement. The **with ... as** statement allows to define a block where an open file is handled, without the need of explicitly closing it at the end.

```
with open("my_file.txt","w") as f:
      f.write("Hello world\nfrom python!")
      f.write("!!!!")

print(f.closed) # True
```

Exercise 1.9 Modify the code above in such a way that the file is written if and only if it does not exists. You can check whether a file exists by using the functions of the module **os.path**. ■

2. Containers and basic data structures

> Representing an object or structured data using (potentially nested) dictionaries with string keys (instead of a user-defined class) is a common pattern in Python programs.
>
> PEP 589 [12]

Python is plenty of structures which allows to store data in an efficient way. The decision of the best data structure to be used depends on the data themselves and on the algorithms working on them. We will see in this chapter **string**, **list**, **set**, **tuple** and **dictionary**, which are the most commonly used structures of python.

2.1 Strings

Strings can be expressed in two different and equivalent ways, using single or double quotes

```
a = "python_test"
a = 'python_test'
```

A single character or a part of a string can be extracted as

```
a = "python_test"
b = a[1] # b = "y"
b = a[1:] # b = "ython_test"
b = a[1:3] # b = "yt"
b = a[:4] # b = "pyth"
b = a[1:-2] # b = "ython_te"
```

Note that in python, all indexes start from zero. It is also possible to create a **slice** object to store to a variable the indexes and extract afterwards a part of a string.

```
a = "python_test"
b = slice(1,7,2) # From 1 to 7 with a step equal to 2
print(a[b]) # yhn
#Also equivalent to
print(a[1:7:2])
```

The length of a string is accessible by the built-in function **len**.

```
a = "python_test"

print(len(a))
```

Curly brackets can be used to introduce a variable which can be expanded and substituted inside a string, for instance

```
user_name = "root"
age = 27

a = "Name {name}, age {age}".format(name= user_name, age=age)

print(a)# Name root, age 27
```

and are useful in to format and store the output inside a string. In the newer version of python, the above code can also be written as

```
user_name = "root"
age = 27

a = f"Name {user_name}, age {age}"

print(a)# Name root, age 27
```

and the variables in curly brackets are expanded using the already defined variables names.

Two or more strings can be concatenated using the "+" sum operator.

```
noun = "house"
article = "the"
color = "red"
space = " "

my_sentence = article + space + color + space + noun
print(my_sentence)
```

Note that the sum operator does not add a whitespace by default, unlike the function **print**.

A string is **immutable**, therefore we cannot change a single character of a string using item assignment.

```
a = "house"
a[2] = "r"#TypeError: 'str' object
          # does not support item assignment!
```

We can however use the sum operator together with the slice operators above to create a new string.

```
a = "house"
b = a[:2] + "r" + a[3:]
```

2.2 Lists

A **list** is the standard container of python objects, which do not need to be of the same type. It is introduced by the squared brackets "[]".

```
a_list = [19, 3, 4, 5, 12]
an_another_list = ["home", 3, 4+8j, -5.1, [12, 3]]
```

It is possible to extract a single or multiple elements of a list using the same syntax of the class **str**. It is also possible to **append** or **insert** new elements to a list.

```
a_list = [11, 3, 4, 5, 12]
a_list[0] = 19
a.append(5)# a is now [19, 3, 4, 5, 12, 5]
a.insert(2,1)# insert "2" in position "1"
# a is now [19, 2, 3, 4, 5, 12, 5]
```

As for strings, it is possible to concatenate two lists using the operator "+".

```
a_list = [19, 3, 4, 5, 12]
b_list = [2,3]
c_list = a_list + b_list
#c_list is now [19, 3, 4, 5, 12, 2, 3]
```

An element of a list can be deleted using the keyword **del**.

```
a_list = [19, 3, 4, 5, 12]
del a_list[2]
# Now a list is [19, 3, 5, 12]
```

More on ... 2.2.1 — The del command and the with statement. The **del** command can be used to delete from the memory of the python interpreter any objects, not only entries of a list. Python has an efficient automatic memory allocation, so that it is not required to allocate explicitly the memory when creating a list or any other object. However, in certain circumstances, for instance when dealing with very big objects that occupy a large amount of memory, it can be useful to call **del** to delete an object that is not needed anymore.

2.3 Tuples

A **tuple** is closely related and quite similar to a list, and it is introduced by commas and by the standard brackets "()". Unlike a list, a **tuple** is immutable, therefore its entries cannot be modified.

```
a_tuple = (19, 3, 4, 5, 12)
an_another_tuple = ("home", 3, 4+8j, -5.1)
```

A **tuple** is used in place of a list to ensure for instance that no entries are added or removed during the later execution of the program or when calling a function giving a tuple as argument. Of course, the tuple itself is immutable, but its objects pointed to can be modified if they are mutable themselves.

```
a_tuple = (19, [3, 4], 6, 12)
a_tuple[1].append(5)
#a_tuple is now (19, [3, 4, 5], 6, 12)
a_tuple[0] = 5# TypeError: 'tuple' object
       # does not support item assignment
```

More on ... 2.3.1 — Mutable vs immutable. A list, unlike a string or a tuple, is a **mutable** object, so that the functions **append** or **insert** do modify the object pointed by the variable from which they are called. Python allocates new memory each time a list, or any other mutable object, is created. The different behavior can be tested using the **is** operator.

```
a_list = [1,2]
b_list = [3,4]
c_list = [1,2,3,4]
print(a_list + b_list is c_list)#False
print(a_list + b_list == c_list)#True

print("" is "")#True
print([] is [])#False
print(() is ())#True
```

In the fourth line we used the operator "==" to check if two lists are equal. In the last two lines, the "[]" or """ syntax is used to create an empty list or string, respectively.

A tuple consisting of a single entry can be created by adding a comma "," before the last bracket.

```
a_tuple = (4,)
```

It is also possible to omit the brackets, except in the case of ambiguities.

```
a_tuple = 4,
```

A possible ambiguity can occur in the case of a function call. For example, if you want to call the function **sorted** to sort a tuple, the parenthesis are mandatory, otherwise all numbers would be interpreted as input parameters.

```
print(sorted((4,2,5)))#[2, 4, 5]
print(sorted(4,2,5))# Meaningless
# 4, 2, 5 are interpreted as three input
#parameters for sorted
```

2.4 Sets

Sets are used to define a container close to the mathematical abstraction of set, where the order of the elements is irrelevant. A set is introduced by curly brackets. For two sets, it is possible to compute the union and the difference using the function **union** and operator difference "-" respectively.

```
a_set = {4,6,8}
b_set = {4,9,7}
union_set = a_set.union(b_set) # 4,6,7,8
difference_set = a_set - b_set # 6,8
intersection_set = a_set.intersection(b_set)# 4
```

By definition, every element in a set appears only once. It is not possible to access to the elements of a set using an index, as for a set it is not defined the order and which element is the first, the second, and so on. A set can contain only *immutable* objects.

2.5 Dictionaries

A **dictionary** is an associative container, in which elements are accessed by **keys**. As simple example, a dictionary can store the age of the students of a class.

```
students_age = {"John": 22, "Paul": 21, "Lisa": 23}
print(students_age["John"])
```

Dictionaries are a very important tools to store structures with multiple records.

```
main_text = {"text": "The sky is blue", "color": "red",
             "font_size": 13, "bold": False}
```

A key can be any hashable object, such strings or integers.

Dictionaries are powerful in particular when the algorithm to be executed depends on a run-time choice. Consider a simple program computing the value of a function chosen by user.

```
import math

function_dict = {"cos": math.cos, "sin": math.sin,
          "tan": math.tan}

function_name = input("Write the name of "
                      "a simple trigonometric function:")
```

```
x = float(input("Write the input value:"))

if function_name not in function_dict.keys():
      print(function_name, "not implemented!")
else:
      print(function_name, "(", x, ") is",
      function_dict[function_name](x))
```

The dictionary is created associating to each name the corresponding function. The list of keys of a dictionary is accessible by the method **keys**(), if the function is not found an error message is printed. In the last line, "function_dict[function_name]" returns the function selected which is evaluated as "function_dict[function_name](x)". Compared to static **if-else**, a dictionary is easy to read and to maintain, and other possible functions can be added to "function_dict" any time also in different parts of the code.

2.6 The in operator

The **in** operator can be used in case that we need to search if an element exists in a set of data.

```
a_set = {4,6,8}
a_list = [2,1,6,7]
print(4 in a_set)# True
print(11 in a_list)# False
```

The **in** operator can be also used for dictionaries, where it checks whether a given key is present.

2.7 Conversions

Lists, sets, tuples and dictionaries can be converted among each other using the functions **list**, **set**, **tuple**, **dict**, which effectively are the constructors of the relative data structures.

```
a_set = {11,6,8}
a_list = [2,21,7,2]
print(set(a_list))# {2, 21, 7}
print(list(a_set))
print(tuple(a_list))

print(dict([[1,2],[2,3]]))
```

Notice that a dictionary requires two entries per element, one for the key and the other one for the value.

2.8 For-loops

Elements of the data structures considered in this chapter can be iterated using **for**-loops.

```
a_list = [4,5,9]

for elem in a_list:
      print(elem)
```

A **for**-loop executes all the instructions in the given indentation block for each element in the container specified after the **in** keyword. The elements are accessible in the indentation block from the variable name specified after the **for** keyword. In general, a for-loop can run not only on containers, such as lists and tuples, but on more general structures called generators and iterators (see 7.2).

For-loops are an efficient alternative to while loops. The first program considered in section 1.5 computing an approximation of the exponential can be rewritten as:

```
import math

N = 10
x = 1.2
result_exp = 0

for n in range(N):
      result_exp += (x**n)/math.factorial(n)

print("exp(1.2) is approximately ", result_exp)
```

The function **range** generates an iterator starting from zero running until N with step 1.

> **More on ... 2.8.1 — Range.** The function **range**(start,stop,s) was creating directly a list of integers in python 2, starting from "start" and ending to "stop" with step "s". In python 3, for efficiency reasons, the range function returns an iterator, without creating and storing a possibly very long list in memory for just running an index in a for-loop (see 7.2).

> **Exercise 2.1** Write a program that computes the factorial of a positive integer number provided by the user. It should use the for-loop to compute for example $5! = 5\times4\times3\times2\times1$. Compare the result with the function **math.gamma**(*number*+1). ■

For-loops can be combined with **break** and **else** statements, to interrupt the for-loop and to execute some final instructions if the for-loop did run till the end. The following code searches whether there is an even number in a list of integers, demonstrating the use of **break** and **else**.

```
a_list = [3,11,4,5,9]
```

```
for elem in a_list:
      if elem % 2 == 0:
            print("There is an even number in the list")
            break
else:
      print("There are no even numbers in the list")
```

A for loop running on a dictionary will iterate over keys.

```
figure_edges = {"triangle": 3, "circle": 0, "square": 4}

for figure in figure_edges:
      print(figure, "has", figure_edges[figure], "edges")
```

It is also possible to run both on keys and values using the method **items**.

```
figure_edges = {"triangle": 3, "circle": 0, "square": 4}

for figure, edges in figure_edges.items():
      print(figure, "has", edges, "edges")
```

The comma after the keyword **for** instructs the python interpreter to unpack two values from the container at each iteration.

More on ... 2.8.2 — Unpacking. Understanding unpacking and multiple assignment is a must for writing elegant python code. Unpacking is used to extract elements from tuples and lists and assign them to variables.

```
two_integers = [1, 2]

a, b = two_integers
```

In the code above the variables "a" and "b" are assigned to the two elements of the list "two_integers". The comma operator, even without brackets, is creating a tuple whose element are set to be equal to the one of the list "two_integers".

Unpacking is useful also for swapping variables.

```
a, b = 7, 5

a, b = b, a
#a is now 5, b is 7
```

Exercise 2.2 Use **for**-loops to generate the list of the prime numbers up to 1000 implementing the sieve of Eratosthenes. Starting from the set of the first 1000 integers, remove first all multiples of two (four, six, eight, etc), then all remaining multiples of three (nine, fifteen, etc), then all remaining multiples of five (twenty-five, one-hundred-twenty-five, etc) and so on until all multiples are scanned to the end. ∎

2.9 For-loops with reversed, enumerate and zip

A for-loop can run over a sequence such as list or a tuple in the reverse order using the built-in function **reversed**.

```
for i in reversed(range(10)):
      print("Count-down", i)
```

The function **enumerate** allows to run an index together with the elements of a container in a for-loop, while **sorted** allows to scan the sorted elements of a sequence.

```
a_list = [3,4,1,6,12]

for i, elem in enumerate(a_list):
      print("Element", i, "of a_list is", elem)

for elem in sorted(a_list):
      print(elem)
```

Finally, it is possible to run a combined for-loop for two or more lists using the function **zip**.

```
a_list = [3,4,1,6,12]
b_list = [6,7,8,11,2]

for a, b in zip(a_list, b_list):
      print(a, b)
```

More tools for running for-loops can be found in the module **itertools**.

Exercise 2.3 Write a program that computes the number of *unique* names without repetitions of the following list:

"Christian Mathias Johannes Benedikt Lukas Markus Andre Felix Maria Christian Georg Luca Niklas Pauline Simon Teresa Robert Muriel Corinna Christina Johannes Daniel Patrick Dominik Fabian Florian Urs Benedikt Christoph Caspar Alexander Thomas Birgit Leonard Joachim Carina Lisa Daniel Christina Sabrina Lea Verena Denise Marlene Vincent Maximilian Daniel Niklas Martin Maximilian Enrico Michael Barbara Jakob Luis Ronja Elena Alex Stefano Andriy Valentina Katharina Eva Veronika Kenneth Rudolf Elisabeth Christoph Laurin Daniel Johannes Simon Florian Maximilian Patrick Sebastian Fabian Anastasia Stefan Stefan Jonathan Nico Juliane Niklas Martino Jonas Markus Maximilian Adrian Stefan Jannis Verena Emily Simon Alexander Kilian Tina Viola"

and prints them sorted accordingly to the number of their occurrences and alphabetically. The output could be for instance:

```
There are 74 unique names:
4 Daniel
4 Maximilian
3 Johannes
3 Simon
3 Stefan
....
```

Suggestions: Try to solve the problem using **set** and **dictionaries**, then try to use the **Counter** class (see documentation here). *Do not* create by hand a list of names from the above text, search in the string method documentation if there is a function that can do it. ■

2.10 List comprehension

List comprehension allows to create lists in clear and concise manner. The list of the squares of all even integers up to 100 can be constructed as

```
squared_even_integers = []
for i in range(0,100,2):
      squared_even_integers.append(i**2)
```

or, using list comprehension, as

```
squared_even_integers = [i**2 for i in range(0,100,2)]
```

As an another example, it is possible to easily define a list from the sum of the elements of two lists.

```
a_list = [3,4,1,6,12]
b_list = [6,7,8,11,2]

sum_list = [a+b for a, b in zip(a_list, b_list)]
```

It is also possible to include a condition such that only certain elements are included in the list. The same list of the example above can be defined by an additional **if** to check whether the number is even or not, instead of restricting the range function in step of 2 starting from zero.

```
squared_even_integers = [i**2 for i in range(100)
      if i % 2 == 0]
```

List comprehension can be combined to create also complex lists, such as the one of all prime numbers up to 100. A number is prime if it is divisible only by one and by itself. The list of divisors of a number is easily created as

```
i = 50
divisors = [j for j in range(1,i+1) if i % j == 0]
```

and a number is prime if the length of the list of divisors (function **len**) is equal to two.

```
prime_numbers = [i for i in range(2,100)
      if len([j for j in range(1,i+1) if i % j == 0]) == 2]
```

It is however recommended to avoid the abuse of list comprehension, as the code might easily become obscure and difficult to read and to maintain. "Readability counts." is an important principle of "The Zen of Python" [3].

Exercise 2.4 Use list comprehension to create a list of N entries where, starting from zero, each entry n is equal to the minimal number of bits required to store the integer number n. The output should be for instance:

```
[1, 1, 2, 2, 3, 3, 3, 3, 4, 4, ...]
```

Test the code for $N = 200$ and $N = 1000$. ■

Exercise 2.5 Use list comprehension to create a list of N entries where, starting from zero, each entry n is equal to the number of different digits of n itself (for instance equal to 3 for 1315 or 2 for 1212). Test the code for $N = 300$ and $N = 1000$. ■

The syntax of list comprehension can be easily extended to set and dictionaries.

```
#Set of the first ten integers numbers squared
print({i**2 for i in range(10)})
#Dictionary of the first ten integers numbers squared
print({i: i**2 for i in range(10)})
```

Exercise 2.6 Cellular automata are discrete models defined by deterministic rules evolving a K-dimensional state vector v from the time t to $t+1$. The entries of the vector are called cells, and can have different states. The simplest one-dimensional cellular automata, often called elementary automata, is described by a vector of length N where each entry is either zero or one, corresponding to a cell being in the states "dead" or "alive", or "on" and "off". A new vector v_{t+1} is generated from the previous v_t accordingly to a rule that takes into account for each cell only its nearest neighbors, i.e. the new $v_{t+1}[i]$-entry depends on the old entries $v_t[i+1]$, $v_t[i]$ and $v_t[i-1]$. Periodic boundary conditions are typically chosen: the left neighbor $0-1$ of the cell zero corresponds to the last entry $N-1$, while the entry N corresponds to the first entry. As the model and the vector entries are discrete, the number of all possible elementary cellular automata is finite and equal to $2^8 = 256$. Although simple and deterministic, cellular automata can exhibit an apparently random chaotic behavior [6, 7, 8].

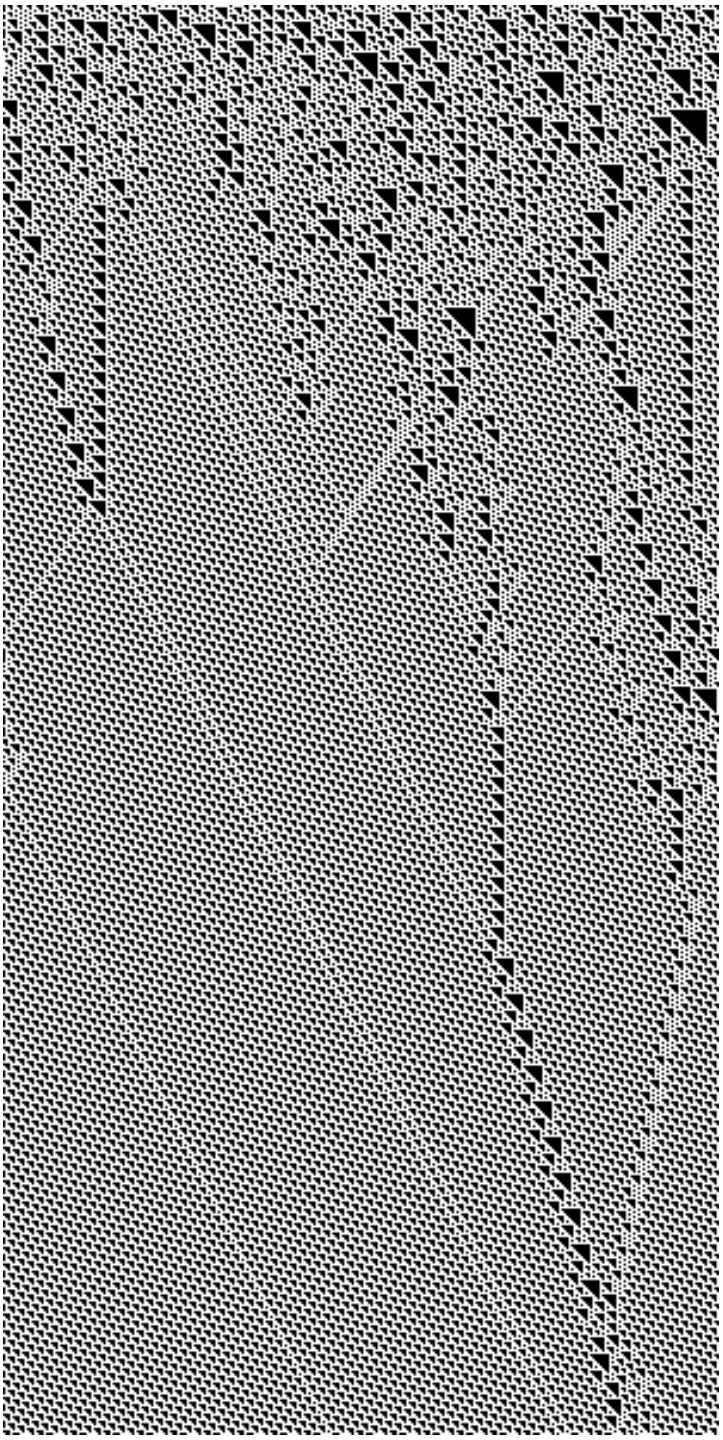


Figure 2.1: Cellular automata evolved for 600 steps from a random initial state accordingly to the Rule 110.

The most interesting rule is the Rule 110:

$$\begin{array}{ll} 1,1,1 \to 0 & 1,1,0 \to 1 \\ 1,0,1 \to 1 & 1,0,0 \to 0 \\ 0,1,1 \to 1 & 0,1,0 \to 1 \\ 0,0,1 \to 1 & 0,0,0 \to 0 \end{array}$$

written using the notation $v_t[i-1], v_t[i], v_t[i+1] \to v_{t+1}[i]$ (for instance, in the first case, if the entry i of the vector at time t and its left and right neighbors are all one, then the entry i of the vector at $t+1$ is zero). The corresponding cellular automata has been proven to be Turing-complete [9].

Implement a script that evolves accordingly to the Rule 110 for 600 steps an initial state with 300 entries randomly one or zero. *Do not* use if-elif-else block to specify the Rule 110, but try to implement the rule in terms of a list or a dictionary. For instance you can convert the three cells to a string and use it as a key for a dictionary, or you can note that $v_t[i+1]$, $v_t[i]$ and $v_t[i-1]$ are representing in order an integer number from zero to seven written in binary form which can be used as index for a list.

Finally, use the Python Image Library **PIL** to store the output of the evolution of the cellular automata to a black/white picture of 300×600 pixels. Consider for guidance the demonstration code below, creating a simple square picture of a cross.

```
from PIL import Image

#create a new image 300*300
#Mode "1", black/white
pict = Image.new("1",size=(300,300))

#Fill the picture
for i in range(300):
      for j in range(300):
            if i == j or i + j - 299 == 0:
                  #Fill the pixel i,j color black (0)
                  #PIL indexing is column-major
                  pict.putpixel((j,i),0)
            else:
                  #Fill the pixel i,j color white (1)
                  pict.putpixel((j,i),1)

#Save the picture
pict.save("cross.png")
```

A picture showing a possible output of the cellular automata is shown in Fig. 2.1.

■

3. Functions

> On two occasions I have been asked, "Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?" I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.
>
> Charles Babbage

A function is defined mathematically as a relation between two sets X and Y, which associates to every element of the domain X a *single* element of the codomain Y. *Functional programming* is a programming paradigm in which the program is specified in terms of functions that are evaluated, rather than as a list of statements and instructions executed in sequence (*imperative programming*). The advantage of functional programming is that complex tasks can be subdivided in smaller functions which perform simpler operations. Smaller functions are easier to test and can be reused many times in different parts of the code and in many different contests.

3.1 Definition and use of functions

In python functions are defined by the keyword **def** as

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

The input of the above functions are the variables "a" and "b", the statement **return** specifies the output, in this case the sum $a+b$. The function "add" can be called as

```
c = add(4,9)
```

where the arguments "a" and "b" are specified by position, being "a" set to 4 and "b" to 9. It is also possible to specify the arguments of a function by their keyword, in such a case the order is arbitrary

```
c = add(b = 9, a = 4)
```

Finally, it is also possible to mix arguments specified by position and keywords, provided that no further positional arguments are given after the first keyword argument. So the function call

```
c = add(4, b = 9)
```

is perfectly legal, but

```
c = add(a = 4, 9)
```

does not work: "SyntaxError: positional argument follows keyword argument".

In python it is not directly possible to define a function which is executed only if the arguments are of a certain type, unlike C or C++. The **duck typing** programming philosophy requires to the variables in the code, and for instance to the arguments of a function, only the definition of certain needed operations, independent from the actual type of the variables. In the example above, the sum can be performed between two real, integer or complex numbers. Even more interestingly, the above function would work if "a" and "b" are both strings or lists, and then "add" would mean the string concatenation or the union of two lists. "If it walks like a duck and it quacks like a duck, then it must be a duck", so the above code will work as long as the ducks "a" and "b" can be added.

> **More on ... 3.1.1 — Strong versus weak typing.** Although in python a variable is not bound to a type, python is still a **strongly typed language**, in the sense that variables cannot be implicitly converted to unrelated types. For instance, it is not possible to call the function add using a string and a number as arguments
>
> ```
> add(4, "9")
> ```
>
> as a "TypeError" exception will be raised when python will attempt to perform the sum of a string with an integer. "Explicit is better than implicit." is the second entry of the Python Zen [3], so that we must use an explicit conversion
>
> ```
> add(4, int("9"))
> ```
>
> to instruct python that we want to perform the addition of two integers. It is however possible to call
>
> ```
> add(4, 9.3)
> ```
>
> since for built-in numerics types an implicit conversion to the "wider" type is allowed, being float wider than int and complex wider than float and int.

A function can contain multiple instructions, all the one included in an indentation block. If the execution of all instructions of a function ends before before that any **return** statement is reached, the function returns **None** by default.

A **docstring** containing a documentation can be introduced immediately after the ":" following the function declaration name and parameters [5]. The docstring is accessible later by the user calling the built-in function **help**.

```
def add(a, b):
        """Returns the addition of a and b"""
        return a + b

help(add)
```

Exercise 3.1 Write a function that returns the area of a circle given the radius. Write also a documentation, describing what the function is supposed to compute. ■

Exercise 3.2 Use the function **dis** of module **dis** on the function of the previous exercise to get a readable form of the compiled bytecode generated by the python interpreter. Do you understand the output? ■

Exercise 3.3 Can you define a function "quadratic_roots(a,b,c)" that returns the two roots

$$r_{+,-} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{3.1}$$

of a quadratic polynomial $ax^2 + bx + c$? Try to implement the function, assuming $a \neq 0$, and think to what data structure would you use to store and return the result. ■

3.2 Are arguments passed by value or by reference in Python?

The answer is somehow "both", due to the way variables are handled in python. Consider for instance the following code

```
def plus_two(a):
        a += 2
        return a

b = 9
print(plus_two(b))
print(b)
```

If executed, it prints "11" and "9", although the variable "b" is passed as arguments to the function plus_two and modified in it. However the code

```
def append_two(a):
        a.append(2)
        return a

b = [4,5,6]
```

```
print(append_two(b))
print(b)
```

prints "[4,5,6,2]" twice. The reason for the different behavior is that variables in python are only references to objects. In the first case, an integer is immutable and after the instruction "a += 2", "a" is pointing to the int object "11", while the object "9" pointed by "b" is left unchanged. In the second case, a list is mutable, and the instruction "a.append(2)" modifies the original object pointed by "b" outside the function append_two.

Exercise 3.4 Consider the following code:

```
def shout(message):
      message = message.upper()
      print(message)

message = "stop it!"
shout(message)
print(message)
```

The function **upper** returns a string where all characters, excluding symbols and numbers, are in upper case. What message will be printed at the end? A string in lowercase or uppercase? Motivate your answer. Try to answer the question before running the code. ■

3.3 Default value for function arguments

Arguments can have default values, that are used when the function is called without explicitly specifying the value of the argument. Default values are useful to specify options, for instance

```
import math

def derivative(f, x, epsilon=10e-9)
      return (f(x+epsilon)-f(x-epsilon))/(2*epsilon)

print(derivative(math.cos,math.pi))
print(derivative(math.cos,math.pi,10e-11))
```

The above script prints a numerical approximation of the first derivative of the cosine at $x = \pi$, first using a step ϵ equal to the default value 10^{-9}, and then equal to 10^{-11}. Note that in python functions themselves, as any other objects, can be given as arguments of an another function.

Exercise 3.5 Can you provide as input parameter of a function a module? ■

3.4 Variadic arguments

It is possible to define functions accepting an arbitrary number of arguments. Such functions are called **variadic**. The syntax is

```
def add(*args):
      result = 0
      for elem in args:
            result += elem
      return result

add(8,9,10,12,15,3)
numbers = (8,2,3,1,1,5)
add(*numbers)
```

Now the function "add" will accept an arbitrary number of positional arguments, stored in the tuple args, performing the sum of all of them in the loop. In the last line, the * operator is used to *unpack* the tuple "numbers" using its content as parameters for the function "add".

It is also possible to accept an arbitrary number of keyword arguments, in such case the keyword arguments are given as a dictionary, storing the result

```
def add(*args, **kwargs):
      result = 0
      for elem in args:
            result += elem
      for value in kwargs.values():
            result += value
      return result

add(8,9,10,12,r=15,u=3)
```

Finally, arguments given by name can be mixed with *args and *kwargs

```
def my_function(a, *args, b, **kwargs):
      pass
```

In such a case, the argument "b" must be specified by keyword when calling "my_function".

More on ... 3.4.1 — The operators * and **. The operators * and ** can be used also to call a function using directly the element a sequence or a dictionary as input parameters, or also to easily join lists and dictionaries.

```
a_list = [8, 9, 10, 11]
print(a_list)# [8, 9, 10, 11]
print(*a_list)# 8 9 10 11

b_list = [2,3,5]
c_list = [*a_list, *b_list]
```

```
a_dictionary = {"Name": "Stefano", "Age": None}
b_dictionary = {"University": "Regensburg"}
c_dictionary = {**a_dictionary, **b_dictionary}
```

3.5 Scope of variables inside a function

Variables set and defined inside a function are unreachable outside the function in the global scope. Global variables are accessible inside a function

```
b = 5

def my_function():
      #b is accessible inside my_function
      print(b)
      a = 5

#a is undefined outside my_function
my_function()
```

The behavior of the above script would have been different if "b" would have been modified inside "my_function". The script

```
b = 5

def my_function():
      b = 6
      #b is now a local variable
      print(b)
      a = 5

my_function()
print(b)
```

prints 5 and not 6, since now b is a local function variable. Global variables can be modified inside a function if so specified by the **global** keyword

```
b = 5

def my_function():
      global b
      #b is now a global variable
      b = 6

my_function()
print(b)#it prints 6
```

Global parameters can be useful in certain circumstances, however it is strongly recommended to do not abuse of them. For instance, testing and debugging a

function using global variables is more challenging, as the function does not depend only on the input parameters but also on the variables declared as global.

3.6 Lambda expressions

Lambda expressions allow to define a small function of single expression in just a single line

```
add = lambda x, y: x + y
```

The input parameters are "x" and "y", the output is the sum $x+y$. Lambda expressions can be useful for instance to provide a function as argument to an another function. Using the derivative example above

```
def derivative(f, x, epsilon=10e-9)
      return (f(x+epsilon)-f(x-epsilon))/(2*epsilon)

print(derivative(lambda x: x**3,2))
```

we can easily compute the derivative of the function x^3 at $x=2$ without the need of defining a separate function "cube".

Exercise 3.6 The class **list** has a member function **sort** for sorting a list, which can have an optional argument called **key** to specify the function that should be used when defining the sorting order. For instance, to sort by decreasing order we can use:

```
a_list = [2,5,1,-5,6]

a_list.sort()
print(a_list)# [-5, 1, 2, 5, 6]

a_list.sort(key = lambda x: -x)
print(a_list)# [6, 5, 2, 1, -5]
```

Use a **lambda** expression to sort a list of two-element lists accordingly to the second entry. For instance, for the list [[1,9],[-2,1],[3,5]], the sorted output should be [[-2,1],[3,5],[1,9]].

What happens if you try to sort the list [[8,9],[7,9],[12,3],[7,11]]? Can you sort the list in increasing order accordingly to the second and then the first entry to get [[12, 3], [7, 9], [8, 9], [7, 11]]? ■

4. Recursion

An algorithm must be seen to be believed.

Donald Knuth, *Fundamental Algorithms*

In this chapter we analyze some basic algorithms based on **recursion** and on the idea that complex problems can be solved once that a strategy to divide them in smaller and easier subproblems is found.

4.1 The importance of learning algorithms

Carl was an undisciplined nine years old student, and his teacher wanted to punish him by giving a complex, long and tedious homework. The task was compute the sum of the first hundred integers. If Carl were living in the python age, he could have computed the sum using a simple script.

```
sum_up_100 = 0

for i in range(1,101):
      sum_up_100 += i

print(sum_up_100)
```

Unfortunately, Carl was born in the XVIII century, and he could not use any computer. However, he notices that the sum

$$1+2+3+4+5+\cdots+99+100 \tag{4.1}$$

can be rewritten as the sum of fifty times 101

$$(1+100)+(99+2)+(98+3)+\cdots = 101+101+101+\ldots \tag{4.2}$$

so that the sum is simply

$$\sum_{n=1}^{100} n = \frac{100(100+1)}{2} = 5050\,. \tag{4.3}$$

Carl solves the homework in a minute. One day, Carl Friedrich Gauss will be a famous mathematician, but today he is free to go back to play outside.

4.2 What is an algorithm?

The anecdote of the young Carl Friedrich Gauss explains the importance of thinking carefully to the solution of the problem before starting to write any code. Finding an optimal solution to a given mathematical (or even non-mathematical) problem requires a solid background and a long experience in logical thinking. Further experience is required to translate an algorithm, i.e. the formal solution to a problem, to a code in a given programming language, such as python. What has been discussed so far are the basic elements of the python *language*, but a language is worthless without being able to communicate consistently (to a computer or to an interpreter) the instructions that should be executed to produce the desired output from the given input.

The formal definition of what is an algorithm depends of course on the instructions allowed for its execution; in general terms an algorithm is specified by the input (possibly empty) and the expected output that has to be computed in a finite number of steps. An algorithm is "optimal" or "not optimal" depending on many factors, such as the running time required to finalize the output or the possibility to extend the algorithm to find solutions of similar problems with a slightly different input. Finding an optimal algorithm is an "art" that cannot be learnt in few days. The best starting point to introduce yourself in "the art of computer programming" is to study some basic algorithm, their logical structure, and pros and cons. Learning new algorithms will help and improve significantly the ability of logical thinking. In many cases, many different algorithm share the same structure that can be adapted or modified to solve a large class of problems. An example of such algorithm structure is recursion.

4.3 Recursive functions

A function which returns a call to itself is *recursive*. Recursive functions provide an elegant way to define an algorithm to solve certain problems. The classical example is definition of the factorial. Using iteration and a for-loop, we would write a function that multiplies all integers up to n.

```
def factorial(n):
      result = n
      for i in range(2,n):
            result = result*i
      return result
```

We know however that the definition of the factorial is recursive, i.e.

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 1 = n \times (n-1)!. \tag{4.4}$$

Therefore we can rewrite the factorial function using only a single multiplication and a recursive call to the factorial itself.

```
def factorial(n):
      if n == 1:
            return n
      else:
            return n*factorial(n-1)
```

First, we check the terminating instruction, which will be called at the end of the computation. The terminating instruction is stating simply that the factorial of one is just one. Otherwise we just return n multiplied by $(n-1)!$, and from $(n-1)!$, the python interpreter will recursively evaluate the factorials all the way down until one is reached.

Exercise 4.1 A friend asks you to play chess. If you lose you would have to put one cent on the first square of the chessboard, two on the second, four on the third, eight on the fourth, and so on. Would you accept the game? Using *recursion*, write a program that computes how many euro are placed on the chessboard after n squares.

After looking at the amount of money that you would lose for an 8×8 square checkerboard, would you still consider your friend *a friend*? ■

Exercise 4.2 The Fibonacci sequence is defined such that the next element is the sum of the previous two, therefore it is possible to compute a Fibonacci number in terms of a recursive function. Using recursion write a program and a function that computes a Fibonacci number $F(n)$ with n provided by the user. ■

4.4 Divide et impera

"Divide et impera" is an algorithm design paradigm that combines recursion by first *dividing* the problem into two or more smaller subproblems which can be easily solved independently, and then by *combining* them to provide the final output. The Latin locution *divide et impera* comes from politics and military, where it is used as a strategy to defeat and rule by dividing the enemy in smaller groups or entities.

Finding optimal strategy to divide a problem in small subproblems is in general a nontrivial task. In the anecdote reported in the beginning of this chapter, Carl Friedrich Gauss could compute the sum of the first one hundred integers by dividing them in many possible different ways. However, there is only a unique optimal division of the sum in smaller sub-sums so that the same term appears fifty times, yielding to a trivial global sum.

4.5 Maximal entry in a list

A simple problem that can be solved by divide et impera is the search of the maximum in a list of numbers. The iterative version of the algorithm would start a guess from assuming that the maximum is the first entry, and then scan each element, replacing the guess of the maximum with the current entry if so required.

```
def my_max(input_list):
      max_elem = input_list[0]
      for elem in max_elem[1:]:
            if elem > max_elem:
                  max_elem = elem
      return max_elem
```

The divide et impera algorithm starts instead from the observation that it is trivial to find the maximum in a list of simply one item. A longer list can be divided recursively in two smaller lists, and the maximum of the original list can be searched among the two maxima *of the smaller sublists*.

```
def my_max(input_list):
      if len(input_list) == 1:
            return input_list[0]
      else:
            #We use the division returning an integer
            half_ln = len(input_list)//2
            #to divide the original list in two ...
            max_list_1 = my_max(input_list[:half_ln])
            max_list_2 = my_max(input_list[half_ln:])
            # ... et impera
            if max_list_1 > max_list_2:
                  return max_list_1
            else:
                  return max_list_2
```

The execution of the above function will generate effectively a **binary tree**, at each recursion step the list is divided in two and the maximum of the two sublists maxima is returned to the upper node, see Fig. 4.1.

More on ... 4.5.1 — Object creation and efficiency. In the code above, the list is literally split in two parts. When splitting a list, python creates a new object, an operation that requires a certain amount of memory and computing time. For efficiency reasons, it would be beneficial to work directly with only the original list, and use indexes instead of subdividing when using recursion.

```
def my_max(input_list):
      return _my_max(input_list, 0, len(input_list))

def _my_max(input_list, left, right):
      if right-left == 1:
```

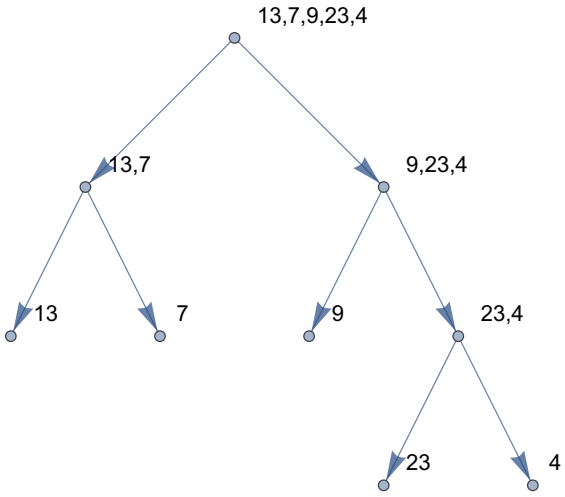


Figure 4.1: Illustration of the expansion tree generated by recursion by my_max algorithm. Each node represents a function call with the list of integers as input.

```
            return input_list[left]
      else:
            #We use the division returning an integer
            half_ln = left + (right-left)//2
            #to divide the original list in two ...
            max_list_1 = _my_max(input_list, left,\
                  half_ln)
            max_list_2 = _my_max(input_list, half_ln,\
                  right)
            # ... et impera
            if max_list_1 > max_list_2:
                  return max_list_1
            else:
                  return max_list_2

print(my_max([13,7,9,23,4]))
```

However, for clarity and simplicity reasons, in the following examples such improvement will not be considered: "Premature optimization is the root of all evils".

4.6 Binary search

Assume that you want to search a number in a list already **sorted**. The most simple algorithm would scan each entry one by one until the end or until a match is found.

```
def simple_search(sorted_list, to_search):
      for elem in sorted_list:
            if elem == to_search:
                  return True
```

```
      return False

sorted_list = [2,9,34,45,87,99,105,167,197]
print(simple_search(sorted_list, 77))
```

The running time of such simple algorithm scales in the worst case scenario as $O(n)$, i.e. the execution time is proportional to length of the list n. It is possible to write a more efficient algorithm exploiting the fact that the numbers in the list are already sorted.

The idea is to divide the list in two halves. If the number we are looking for is larger than the middle element, then we know for sure that it cannot be in the first half list (the opposite would be true if the number we are looking for is smaller than the middle element). Using recursion, we can successively divide the list in two parts until we find a list of a single element, where it is trivial to know whether an element is present or not.

```
def binary_search(sorted_list, to_search):
      if len(sorted_list) == 1:
            return sorted_list[0] == to_search
      else:
            half = len(sorted_list)//2
            if to_search < sorted_list[half]:
                  return binary_search(
                        sorted_list[:half], to_search)
            else:
                  return binary_search(
                        sorted_list[half:], to_search)

sorted_list = [2,9,34,45,87,99,105,167,197]
print(binary_search(sorted_list, 77))
```

The binary search algorithm divides the original list in two parts at each recursion iteration, after k iterations the length of the original is reduced by a factor $n/2^k$. The maximum number of iterations scales as $O(\log_2 n)$, which is a better scaling compared to the linear $O(n)$ of the simple iterative search algorithm.

If the list is not sorted, the above algorithm does not work. However, if many searches have to be executed on a given list, it could be beneficial to sort the list at the beginning and then run the faster binary search algorithm.

Exercise 4.3 Discuss the efficiency and the scaling of the binary search algorithm in case that the sorted list is divided in two, but, if possible, two thirds of the elements are left in the first half, and the remaining one third is given to the second half. Confirm your thoughts by writing and logging the corresponding searching algorithm. ∎

Exercise 4.4 Write a function that sorts a list of integer (or real) numbers of arbitrary length by increasing absolute value using recursion, implementing the

algorithm merge-sort.

The function should run top-down. The function should *first* recursively call itself by dividing the original list in two up to the point where the sublist can be sorted straightforwardly, i.e. until the list has one element, like

```
lst = [5]
```

which is already trivially sorted.

Then, the function should merge the two sorted sublists. Given two sorted lists of any length, the merged sorted list can be constructed easily. Consider for instance the two sorted lists

```
left = [2,5,6,8,10]
right = [4,5,6,7,8]
```

Define the empty list

```
result = []
```

The first element of the list "result", could be either the first element of the list "left" or the first element of the list "right". In fact, it cannot be the second or the third element of "left" or "right", given that they are both already sorted! After a simple comparison, we set

```
result = [2]
left = [5,6,8,10]
```

i.e. we call the function **pop**(0) on the list "left", since the first entry of "left" is smaller than the first entry of "right". By the same reasoning (both "left" and "right" are still sorted), we move further to

```
result = [2,4]
right = [5,6,7,8]
```

and then to

```
result = [2,4,5]
left = [6,8,10]
```

(both the "left" and "right" lists are starting now from the element 5, so that we take for instance from "left"). We continue until both "left" and "right" are empty. Finally we are ready to return the list "result"!

Try the function on the list

```
a = [5, 4, 21, 11, 1, 17, 20, 2, 3, 7, 8, 22, 6,
10, 13, 14, 18, 15, 16, 19, 12, 9, 0]
```

■

Exercise 4.5 Following the same strategy of binary search, write a function "insert(sorted_lst, elem)" that inserts a new entry "elem" to an already sorted list of integers by increasing value in such a way that after the insertion the list is still sorted. Note that scanning and accessing all elements of the sorted list is

not required, even in the worst case scenario where the element to insert is larger than all the entries of the sorted list. ■

4.7 Finding the longest common substring

An another problem that can be solved by recursion is finding the longest common substring (LCS) of two strings. For instance, the LCS of the strings "CCBAC" and "ACBA" is "CBA". Such a problem appears when comparing text, or in genetics when comparing DNA sequences.

The starting point is to realize that if one of the two strings is empty, then the longest common substring is the empty string. If not, there are two alternatives. Either the initial character of the two strings matches, and there is the possibility that such initial character is the beginning a common substring. Start a result variable to be the empty string, add to the result the starting character. Call recursively the same function using the remaining part of the string and the result obtained to find the full longest common substring. If the initial character is different, there are are again two possibilities. Either the largest common substring has been already previously found, or it could be in the remaining substrings.

Before considering the implementation, let us follow the algorithm on the two strings "CCBAC" and "ACBA". We start from an empty string representing the minimal possible LCS. As the two initial characters, "C" and "A" mismatch, the LCS is either between the substrings "CBAC" and "ACBA", or "CCBAC" and "CBA". In the first case, the first letter mismatches again, therefore we move to consider the cases "BAC" and "ACBA", or "CBAC" and "CBA". At this point the algorithm finds recursively three matching letters "CBA", which is the longest common substring, before ending in an empty string. The graph generated by all comparisons is however quite long, see Fig. 4.2.

The implementation of the algorithm explained above is quite straightforward.

```
def lcs(A, B, result = ""):
      #First we define the terminating
      #condition for recursion
      if len(A) == 0 or len(B) == 0:
            return result
      #If the first two characters are equal ...
      elif A[0] == B[0]:
            #it could be that these characters
            #are part of the LCS, and we add them
            #to the previous result when
            #calling recursively lcs on the
            #remaining strings
            return lcs(A[1:],B[1:], result+A[0])
      else:
            #if they are different,
            #the LCS could be in the remaining
            #part of the lists A and B
```

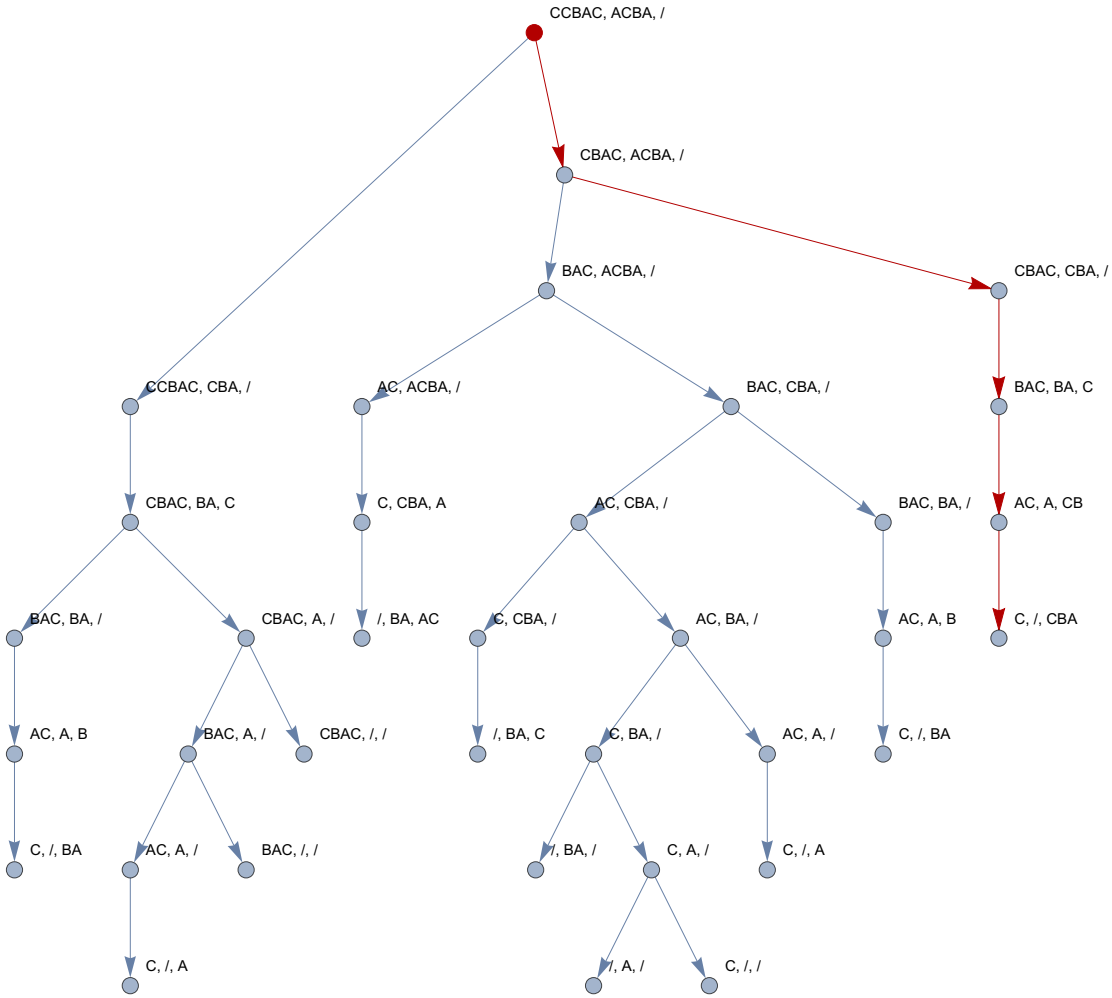


Figure 4.2: Illustration of the expansion tree generated by recursion by the LCS algorithm from the two strings "CCBAC" and "ACBA". Each node represents a function call, and the three strings are the arguments of the function itself ("/" represents an empty string). Each function call leads to one of two further recursive calls, depending on whether the first character of the two strings matches or not. The sequence of function calls that actually finds the longest common substring is highlighted in red.

```
        #we can safely exclude from the LCS
        #the first character from the list A
        left = lcs(A[1:],B)
        #or from the list B
        right = lcs(A,B[1:])
        #Now, or the lcs was found in left
        if (len(left) >= len(right) and
              len(left) >= len(result)):
              return left
        #or in right
        elif (len(right) > len(left) and
              len(right) > len(result)):
              return right
        #or it has been already previously
        #found ...
        else:
              return result

print(lcs("CCBAC","ACBA"))#CBA
```

Compared to the two previous examples computing the maximum and performing a binary search, we have added to the function call an optional input parameter called "result" initialized to an empty string to store the common substring that has been previously computed. The problem of finding the longest common substring has been again divided in smaller subtasks, that can be easily solved. However this time the original strings are not divided in two, but incrementally subdivided by removing a character at each recursion step.

It is interesting to add a **print** statement just at the beginning of the function "lcs" to monitor how recursion breaks the original strings.

```
def lcs(A, B, result = ""):
      print("LCS between", A, "and", B,
            "starting from", result)
      if len(A) == 0 or len(B) == 0:
            ...
```

The output will show all string comparisons being evaluated. If the "lcs" function is called on two long strings such as "the black horse was tired" and "you saw an horse on a green field" you will see a very long output, signaling an exponential explosion of the function calls given by recursion, see Fig. 4.3. Indeed, in the worst case scenario, when the two string are completely different and have no character in common, the number of function calls scales as $O(2^n)$ being n the length of the strings. From the output, you will also notice that many strings are compared over and over again starting from the same result. Caching and storing the results of the function calls, so that they can be returned immediately instead of being recomputed over and over again, is a possible solution to the exponential growth of the expansion tree given by recursion (see 6.4).

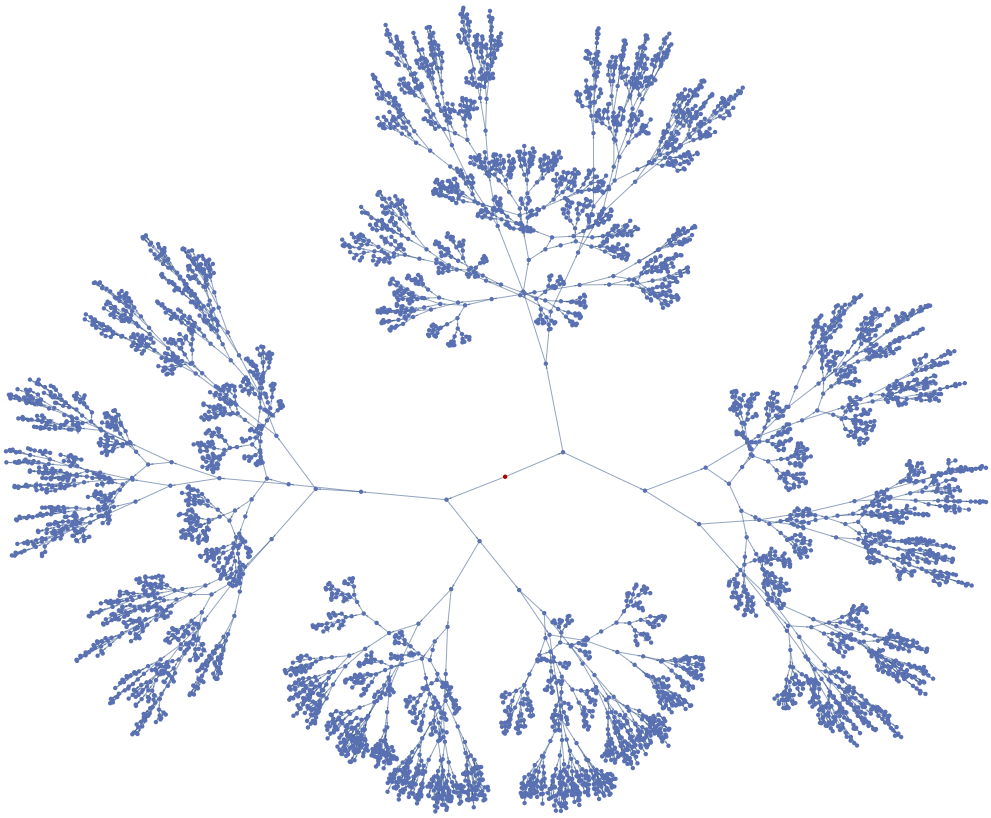


Figure 4.3: Illustration of the expansion tree generated by recursion by the LCS algorithm after 12 recursion iterations starting from the strings "the black horse was tired" and "you saw an horse on a green field". The red point in the middle is the starting point, called *root* in graph theory. The tree has 6189 edges.

More on ... 4.7.1 — Ellipsis. In the example above, in order to avoid repeating the whole code, the ellipsis "..." has been used. The same convention will be use in the rest of the book. Note that the ellipsis "..." are a valid syntax of python, as they correspond to the object Ellipsis, introduced to mean typically "not implemented" or just "something missing".

Exercise 4.6 Using recursion, write an algorithm to find the longest common prefix of a sequence of strings. For instance, given the list ["sunflowers", "sunny", "sunglasses"], the output should be "sun". ∎

Exercise 4.7 Using recursion, write a function that returns the shortest distance between two consecutive occurrences of the same string in a sentence (or None if the string does not occur twice). Test your function on the paragraph "Per me si va ne la città dolente, per me si va ne l'etterno dolore, per me si va tra la perduta gente. Giustizia mosse il mio alto fattore; fecemi la divina podestate, la somma sapienza e 'l primo amore. Dinanzi a me non fuor cose create se non etterne, e io etterno duro. Lasciate ogne speranza, voi ch'intrate." from "Divina Commedia" using the string "per me" (shortest distance 28 characters) and "etterno" (shortest distance 200 characters). ∎

4.8 Finding the shortest path connecting two nodes of a network

Consider a network of N nodes, representing for instance cities, connected by directed links with a given positive weight, for instance highways and distances in kilometers, respectively. A common problem to be solved is to find the shortest path connecting two nodes so that the sum of the weights along the path, called cost, is minimized.

The problem can be solved by recursion. First follow all links from the starting node, and recursively continue to explore all nodes following the links of the network until we reach the target node. While exploring the network, collect the sum of the weights of the links, and, after the scan of all possible paths is complete, return back at each recursion level the one with minimum cost. The algorithm should run at most for a number of recursion steps equal to the number of nodes of the network, otherwise if you continue to follow for more steps the links you will eventually reach the target node only with paths with higher cost containing loops.

The links in the network can be stored for instance using a dictionary, where tuples of source and destination pair are used as key and the weight as value.

```
def shortest_path(net,source,destination):
    #Total number of nodes
    n_nodes = max(max(key) for key in net)

    #Inner function defined for convenience
    #Origin, destination, net are defined
    #in the outer scope
    def _shortest_path(path,cost,k):
        #If we are scanning more links than
```

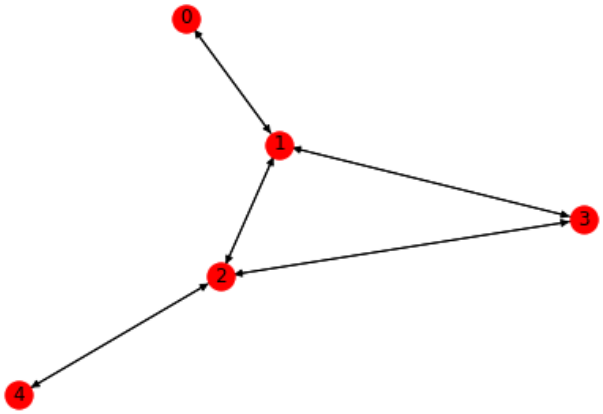


Figure 4.4: A network connecting five nodes. If all links have a weight 1 with the exception of the link connecting the node 1 to the node 2, which has a weight 5, the path with minimum cost between the node 0 and 4 is $0 \to 1 \to 3 \to 2 \to 4$.

```
#there are in the network
#and we did not reach the target
#we return None
if k == 0:
      return None

#Otherwise we continue to explore the network
paths = []

for s, d in net:
      #If the link starts from the current node
      if s == path[-1]:
            w = net[(path[-1], d)]

            #If we can reach the target we append
            #the path and the corresponding cost
            if d == destination:
                  paths.append((path+[d],
                        cost + w))
            else:
                  #Recursion!
                  mpath = _shortest_path(
                        path+[d],
                        cost+w,
                        k-1)
                  #If we have found a path
                  if mpath:
                        #We append it to a list
                        paths.append(mpath)
```

```
            if paths:
                  #If we have found some paths
                  #We return the one with the minimum
                  #cost
                  return min(paths, key=lambda x: x[1])
            else:
                  return None

      #Return the inner function
      #In the beginning the cost is zero and the
      #path must start from source
      return _shortest_path([source],0,n_nodes)

network = {(0,1): 1, (1,2): 5, (1,3): 1, (3,2): 1, (2,4): 1}
shortest_path(network,0,4)
```

In the example above, the recursive function is defined inside the function "shortest_path", in order to wrap and hide to the user the initialization required for instance to compute the number of nodes in the network. The network shown in the example above is represented in Fig. 4.4.

For complex and large networks the above recursive algorithm does not scale well, as we are scanning all possible paths with length up to the number of nodes of the network. An interesting observation is that it is pointless to continue to explore all paths starting from a node if a previous recursive iteration has been already able to reach the same node at a cost that is smaller than the current cost. We therefore create a list storing the cost required to reach a given node of the network, initialized to infinity in the beginning for all nodes, with the exception of the cost to reach the starting node set to zero. The recursive implementation below is effectively a slight modification of the Dijkstra's algorithm [10].

```
def shortest_path(net,source,destination):
      #The number of nodes
      n_nodes = max(max(key) for key in network)

      #We keep a list of the minimal cost
      #required to reach a city which is updated
      #meanwhile the network is scanned
      minimal_cost = [math.inf for i in range(n_nodes+1)]
      #The cost to reach the source is zero
      minimal_cost[source] = 0

      def _shortest_path(path, cost):
            paths = []

            for s, d in net:
                  if s == path[-1]:
```

```
                        w = net[(path[-1], d)]

                        if d == destination:
                              paths.append(
                                    (path+[d],cost+w)
                                    )

                        #We carry on with recursion
                        #if and only if the
                        #cost to reach the current node
                        #is smaller than the one stored
                        elif minimal_cost[d] > cost+w:
                              minimal_cost[d] = cost+w
                              mpath = _shortest_path(
                                    path+[d], cost+w)
                              if mpath:
                                    paths.append(mpath)
            if paths:
                  return min(paths, key=lambda x: x[1])
            else:
                  return None
      return _shortest_path([source],0)

network = {(0,1): 1, (1,2): 5, (1,3): 1, (3,2): 1, (2,4): 1}
shortest_path(network,0,4)
```

As in the previous example, the inner recursive function is wrapped inside the outer function which defines the number of nodes and initializes the cost list, hiding these details when calling the function "shortest_path".

Exercise 4.8 Is it possible to modify the code above to return the longest *simple* path in a network between two nodes? ■

Exercise 4.9 You are planning a bicycle tour on the Alps and you have selected a nice path around the mountains. You have the altitude data expressed in meters for every kilometer of your path and you want to compute the longest difference in meters ΔH that you will need to climb up without a break from a downhill or a plateau. Using recursion, write a function that takes as input the altitude list and returns ΔH.

For instance, given the list [800, 900, 400, 1100, 1200, 800, 1400, 1200, 1800] the output should be 800 meters, corresponding to the uninterrupted uphill from 400 to 1200 meters, while for the list [800, 900, 400, 1100, 1200, 800, 1400, 1500, 1800] the output should be 1000 meters, corresponding to the uninterrupted uphill from 800 to 1800 meters. ■

Exercise 4.10 Recursion is a powerful tool to generate simply and efficiently all combinations or solutions to given a problem, such as all possible paths in a network connecting two nodes. To fully understand this concept, write a recursive function that returns all possible permutations of the elements of a list. ■

4.9 Top-down parser

A more complex problem that can be solved by divide et impera is input validation to check whether an algebraic expression involving integers is valid or invalid. For instance, the expression "2+4*(9-2)" is valid, while "2*(5+)-(/6" is invalid.

The starting point is again the idea that parsing the validity of a string consisting of just a single integer number is quite a simple task. Binary operations can be handled by splitting recursively the string in two and testing the validity of the resulting substrings for each possible combination. For instance the string "4+(5+7)" can be divided in "4" and "(5+7)", or as "4+(5" and "7)". The validity of each substring is recursively checked, and only the splitting combination corresponding to the strings "4" and "(5+7)" will return a valid expression. If, after exhausting all possible combinations, none of them have been able to identify a valid expression, the whole expression is invalid.

Parentheses are handled by recursion in the same fashion. Given that parentheses define the evaluation order, they are parsed first in the beginning before any other possible expression is checked. If the input string is of the form "(expression)", the inner "expression" could be a valid expression that can be checked by recursion. If not, as in the case "4+(5+7)", we continue down to check whether the whole expression can be represented as a binary operation between two operands. There is also the possibility that the whole expression cannot be represented as "(expression)", although it starts and ends with a parenthesis. For instance, the inner expression for "(1+2)*(3*4)", "1+2)*(3*4", is invalid. Therefore, if "(expression)" returns invalid, we still need to check for binary operations.

It could be also interesting to evaluate an expression instead of simply checking whether it is valid or not. In such a case, an additional care is required as multiplications and divisions have a higher priority compared to sum and subtractions. In the expression "5+4*3", "4*3" must be executed first, while executing first "5+4" would provide the incorrect result. In such a case, we first check and evaluate the operators with lower priority, leaving last multiplication and divisions. When a valid the expression is evaluated, the results will be collected starting from the latest function calls, which will be the one for multiplication and divisions which have higher priority.

More on ... 4.9.1 — The eval function. The built-in function **eval** allows to evaluate and check the validity of a string of python code.

```
result = eval("12+4*(19*2)")
```

However the function **eval** is strongly not recommended, especially when dealing with untrusted input. The user could inject easily malicious code, for instance

> by loading the functions of the module **os** to try to erase all documents from the hard disk.

In the implementation of the top-down parser below, additional print statements are added so that the recursive execution of the algorithm can be tracked down.

```
import re

def check_expression(string):
      #we remove first redudant white spaces
      return parenthesis(string.replace("\ ",""))

#We check first whether the input
#is in the form "(expression)"
def parenthesis(string):
      if len(string) == 0:
            return False
      if string[0] == "(" and string[-1] == ")":
            print("Parsing a parenthesis of", string[1:-1])
            #If so we try to see if expression is valid
            #or we try with a simple bynary operation
            return parenthesis(string[1:-1]) or \
                  sum_binary(string)
      #if no parenthesis are there we move down
      else:
            return sum_binary(string)


def sum_binary(string):
      for i in range(len(string)):
            #For all possible ways of splitting the string
            if string[i] == "+" or string[i] == "-":
                  print("Parsing operation", string[i],\
                        "between", string[:i], "and",\
                        string[i+1:])
                  #checks if the input is in the form
                  #expression +/- expression
                  if parenthesis(string[:i]) and \
                        parenthesis(string[i+1:]):
                        return True
      #If none of the above returned a valid expression
      #we try with a multiplication/division
      return multiplication_binary(string)

def multiplication_binary(string):
      for i in range(len(string)):
            #For all possible ways of splitting the string
            if string[i] == "*" or string[i] == "/":
```

```
                print("Parsing operation", string[i],\
                      "between", string[:i], "and",\
                      string[i+1:])
                #checks if the input is in the form
                #expression +/- expression
                if parenthesis(string[:i]) and\
                      parenthesis(string[i+1:]):
                      return True
        #If none of the above returned a valid expression
        #we try with a unary minus
        return unary_minus(string)

def unary_minus(string):
        #we check if the input is in the form
        # - expression
        if string[0] == "-" :
              print("Parsing unary minus of", string[1:])
              return parenthesis(string[1:])
        else:
              #if not we check for a simple number
              return term(string)

def term(string):
        #If the string consists of only a digit
        if string.isdigit():
              #It is a number!
              return True
        else:
              #Otherwise we have exhausted
              #all possible options
              return False

if __name__ == "__main__":
        print(check_expression("(3+4)"))

        print(check_expression("(3+4)*(5*7"))

        print(check_expression("(-3+4)*(5/7)"))

        print(check_expression("(-3+4)*(5/(7+5))"))
```

More complex operations and syntaxes can be handled following the same logic, after considering carefully the order in which the various expressions are checked.

More on ... 4.9.2 — if __name__ == "__main__". The python interpreter defines many global variables, which are typically named with double underscores.

"___name___" is set to be equal to the string "___main___" if the script is running as the main script. There is the possibility to load a script as a **module**, in order to reuse the functions defined in the script for a new different program, in which case "___name___" is set to be equal to the script name. Running the test examples would not be required when the script is executed as a module, for this reason the if instruction has been used to prevent the executions of the tests if ___name___ is not "___main___".

Exercise 4.11 Modify the algorithm above to extend the correctness check to expressions involving also floating point numbers written as "5.56". ■

II Part Two: Advanced python

5. Introduction to OOP

> The joy of coding Python should be in seeing short, concise, readable classes that express a lot of action in a small amount of clear code – not in reams of trivial code that bores the reader to death.
>
> Guido van Rossum

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of class, which is an abstraction representing data (called **properties** or **data members**) and functions acting on the data (called **methods** or **member functions**). Python is an object oriented programming language, so that even the basic numerical types, such as **int** and **float**, are classes themselves. Their property is the corresponding number stored and their methods are the functions that allow to check if a float is an integer or how it can be represented as a rational number. More interestingly, even simple functions are classes themselves, with their instructions and their arguments as attributes.

In the terminology of the object-oriented programming languages, an **object** pointed by a variable is an instance of a **class**. The number "5.5" is an object, an instance of the class "float". The class float is therefore an abstraction representing a floating point number and the methods that allows to perform operations such as additions, multiplications, divisions, etc.

Contrary to simple functional programming, OOP allows to extend further code reuse and helps programmers in organizing and structuring better their code. The core idea is to decouple the various part of the code which are addressing different problems by implementing different classes.

5.1 Definition of a class

A class in python is introduced by the keyword **class**. A typical class will define an **initializer** method called "__init__" to set the attributes of the class itself.

```
import math

class Square:
      def __init__(self, length):
            self.length = length

      def area(self):
            return self.length**2

      def diagonal(self):
            return math.sqrt(2)*self.length


my_square = Square(5)
print(my_square.area())
```

The initializer of the class in the example above takes two arguments. The first one is **self**, a reference to the current instance of the class, i.e. a "Square" object. The second argument is the length of the square. The length variable is stored as **attribute** inside the initializer, using the reference to the object **self**.

The "area" and "diagonal" **methods** compute the area and the diagonal of the square pointed by "self". Methods and attributes of an object are accessible using the dot ".". The explicit reference to the object **self** is omitted when creating the object "my_square" and when calling the method "area". The python interpreter translates automatically the last call to the equivalent form:

```
print(Square.area(my_square))
```

where it is explicitly written that we want to call the method "area" of the class "Square" using the object "my_square" as argument.

Although the example considered is quite simple, it shows already some interesting feature of object oriented programming. If you were to develop a library to manage two dimensional graphics for a game engine, you might need to compute the area and the length diagonal of many different figures. A simple function computing the area of a generic two-dimensional figure would first need to understand the kind of figure (for instance with an if-elif block for each two-dimensional figure required for the graphics), then load the relevant data like the length of the sides from some structure (like a list or better a dictionary), and finally compute the area. Classes allow implement the same code in a cleaner way. Each class is abstracting one and only one single figure by defining all relevant methods and attributes, such as area, diagonal and length.

Exercise 5.1 Write a class Sphere, storing as attributes the radius and the position of the center. The class should implement the methods "surface", "volume" and "is_intersecting" to return the area of the surface, the volume, and to test whether a sphere is intersecting with an another sphere, respectively. What is the type that the member function "is_intersecting" should return? ■

5.2 Private data members

Python does not have the concept of **private** attributes and methods inaccessible and invisible from outside the class. The convention is to name with an initial underscore a variable which is intended to be private, i.e. that should be modified only by the class methods. As an example, consider the following alternative to the class Square.

```
class Square:
      def __init__(self, length):
            self._length = length
            self._area = length**2

      def area(self):
            return self._area

      def set_length(self, length):
            self._length = length
            self._area = length**2

my_square = Square(5)
my_square._length = 9
print(my_square.area())# prints 25, not 81!

my_square.set_length(9)
print(my_square.area())# prints 81!
```

In the class above, the area is precomputed and stored in the initializer, so that we do not need to compute it again and again each time that the method "area" is called. The problem is when we change a private variable directly without calling the method "set_length", violating the intended use of the class. The area property is not updated, and the subsequent call to the method "area" will provide the incorrect result.

More on ... 5.2.1 — Double underscores. Double underscores are reserved typically for methods and attributes used to define special class functions, such as the initializer of an object. Variables stored with double underscores are **obfuscated**, meaning that their name is changed so that they are not directly accessible from outside the class (although they are still accessible once the obfuscated name is known, for instance by using the command **dir**).

5.3 Static and object attributes

Classes can have something similar to static attributes shared among all instances, defined in the class outside the initializer. Further, single objects can have their own unique attributes, defined dynamically after the object is created and initialized.

```
class Square:
      sides = 4

      ...

a = Square(4.5)
b = Square(5.9)

print(b.sides)  # Prints 4, sides is a shared class variable

# Also accessible as:
print(Square.sides)

Square.sides = 5
print(a.sides)  # Prints 5, sides is a shared class variable


a.color = "red" # Create a new attribute for the object a
print(b.color)  # AttributeError: 'Square' object has no
                # attribute 'color'

print(a.__dict__) # {"length": 4.5, "color": red}
```

Object attributes are accessible using the property "__dict__", a dictionary containing the values and the names of the data properties as keys.

More on ... 5.3.1 — Classes and type. In python there is no intrinsic difference between a class and an object. Classes themselves are instances of the metatype **type**.

```
a = [4,5,6]
type(a)     # <class 'list'>
type(list)  # <class 'type'>
```

Static attributes are therefore "class attributes".

Classes can be dynamically modified as any other object. It is possible to add methods directly to a class, not only to instances of the class.

```
def area(obj):
      return obj.length**2

class Square:
      def __init__(self, length):
```

```
            self.length = length

Square.area = area
Square.sides = 4

sq = Square(9)
print(sq.area())
print(sq.sides)
```

5.4 Operator overloading

Operator overloading allows to extend the basic arithmetic and non-arithmetic operations, such as the sum "+", to classes. In the python standard library, the meaning of the operator "+" is not only restricted to additions of numbers, but it is also extended to strings and lists to mean list and string concatenation. Many operators can be overloaded, not only the basic binary operators. Operator overloading allows to implement complex mathematical structures, such as vectors in Euclidean spaces and matrices.

```
class Vector2D:
      def __init__(self, x, y):
            self.x = x
            self.y = y

      def norm(self):
            return self.x**2 + self.y**2

      #Called when checking equality
      def __eq__(self, second):
            return (self.x == second.x and
                  self.y == second.y)

      #Called when performing an addition
      def __add__(self, second):
            return Vector2D(self.x + second.x,
                  self.y + second.y)

      #Called when performing a difference
      def __sub__(self, second):
            return Vector2D(self.x - second.x,
                  self.y - second.y)

      #Called when getting an element from []
      def __getitem__(self, index):
            if index == 0:
                  return self.x
```

```
            else:
                  return self.y

      #Called when setting an element from []
      def __setitem__(self, index, value):
            if index == 0:
                  self.x = value
            else:
                  self.y = value

      #Called when converting to a string
      def __str__(self):
            return "("+str(self.x)+","+str(self.y)+")"

a = Vector2D(4,5)
b = Vector2D(-3,1)
print(a+b)
print(a-b)
print(a.norm())
b[1]=4
print(b[1])
```

The addition and the subtraction return a new "Vector2D" object which stores the result of the corresponding operation. The method "___str___" returns a string to print a two dimensional vector. The methods "___setitem___" and "___getitem___" are called when setting and getting an element using the square bracket.

Exercise 5.2 Write a class **PolarComplex** that stores a complex number in polar coordinates using the data members ρ and ϕ as $\rho \exp(i\phi)$, with member functions:

1. **___abs___** to return the absolute value
2. **___mul___** to multiply an another PolarComplex number and return the result
3. **___truediv___** to divide by an another PolarComplex number and return the result
4. **___str___** to return a string representing the PolarComplex number

5.5 Inheritance

Inheritance is a central concept in OOP programming languages, based on the idea that it is possible to create specialized classes starting from classes abstracting more general types. A standard example is represented by the classes Student, Musician, Teacher, where we want to avoid to repeat all the code that manages the basic information about a person, such as name and age.

```
class Person:
      def __init__(self, name, age):
            self.age = age
```

```
            self.name = name

      def print_info(self):
            print("I am", self.name, "and I am",
                  self.age, "years old")

class Student(Person):
      def __init__(self, name, age, university):
            super().__init__(name, age)
            self.university = university

      def print_info(self):
            super().print_info()
            print("I study in", self.university)

class Musician(Person):
      def __init__(self, name, age, instrument):
            super().__init__(name, age)
            self.instrument = instrument

      def print_info(self):
            super().print_info()
            print("I play the", self.instrument)

class Teacher(Person):
      def __init__(self, name, age, subject):
            super().__init__(name, age)
            self.subject = subject

      def print_info(self):
            super().print_info()
            print("I teach", self.subject)

lukas = Student("Lukas", 23, "London")
lukas.print_info()
```

The **base** class is called "Person", that handles the name and the age. The **derived** classes are Student, Musician, Teacher. In each of the derived classes, the function of the base class, such as the constructor, are available through the built-in function **super**. The function "print_info" is **overridden**, the function defined in the base class is replaced by the one of the derived class. When we print out the information of a student, the basic informations are already handled by the class Person, printed as "super().print_info()" and set as "super().__init__(name, age)", so that we do not need to write three times the same code.

An another advantage of inheritance, apart from code reuse, is that we can extend the base class, and by doing so we are extending all derived classes at once. For instance we can add a function to the class Person that prints when a person was

born.

```
import time

class Person:
      ...

      def born(self):
            current_year = time.localtime().tm_year
            print("I was born in", current_year-self.age)
...

lukas = Student("Lukas", 23, "London")
lukas.born()
```

In this way, all classes derived from Person have inherited the function "born". On the other hand, for the same reason, a certain care is required when modifying the base class, as a change might influence and break all derived classes (issue known as "fragile base class problem").

Inheritance is a powerful tool to extend existing classes. For example we can implement an extension of the built-in class **list** to handle the division as a tool intended for breaking a list in n parts.

```
class MyList(list):
      def __init__(self, *args, **kwargs):
            super().__init__(*args, **kwargs)

      def __floordiv__(self, n):
            ll = len(self)
            return tuple((MyList(self[i:i + ll//n])
                  for i in range(0,ll,ll//n)))

      def __getitem__(self, elem):
            if not isinstance(elem, slice):
                  return super().__getitem__(elem)
            else:
                  return MyList(super().__getitem__(elem))

test = MyList([3,2,4,5])
print(test//2)
print(test//4)
```

The "__getitem__" method has been overridden, so that when we slice our list, we return again an instance of the class MyList.

Exercise 5.3 Extend the build-in class **dict** to implement a home-made class "defaultdict", so that if a key is accessed but not present in a dictionary, it is automatically set to a default value. ∎

5.6 Functors

Functors are classes that can be called as a function. As their behavior depends on their properties and not only on their input parameters, strictly mathematically they are not functions. Any class can behave as a function if a method "__call__" is defined. The advantage of functors is that they can store a state between each call, for instance for logging or to store options.

```
import math

class NumericalDerivative:
      def __init__(self, func, epsilon):
            self.eps = epsilon
            self.func = func
      def __call__(self, x):
            return (self.func(x + self.eps)
                  - self.func(x - self.eps))/(2*self.eps)

dsin = NumericalDerivative(math.sin,10e-9)
print(dsin(math.pi))
```

The functor created by "NumericalDerivative" behaves effectively as a function taking one argument and computing the numerical derivative of the function stored using the discrete step "self.epsilon", which is an option to be chosen. The function "dsin" can be given to any algorithm that requires a function, allowing for instance to find the roots of the numerical derivative using bisection as described in Section 10.1.1.

Exercise 5.4 Define a functor to compute an approximation of the Riemann zeta function defined from the infinite sum

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}, \tag{5.1}$$

which is convergent for $\mathrm{Re}(s) > 1$. The sum should be truncated to N, with the integer N specified by the user and stored as data member. Compute the value of $\zeta(s)$ for $s = 1.15$ and $N = 10000000$ and compare the result with the function **zeta** of the module **scipy.special**. ∎

Along the same idea, functors can also implement easily partial evaluation.

```
class PartialLeftEvaluation:
      def __init__(self, fun, argument):
            self.argument = argument
            self.fun = fun

      def __call__(self, *args, **kwargs):
            return self.fun(self.argument,*args,**kwargs)

def multiply(a,b):
      return a*b
```

```
two_times = PartialLeftEvaluation(multiply,2)
print(two_times(4))
```

In the example above, the left argument and the function are stored inside the class "PartialLeftEvaluation". The functor transforms a function taking n into one taking $n-1$ positional arguments. The function "multiply" takes two arguments, while the functor "two_times" only one.

Exercise 5.5 Write a functor "Cache" that stores a function f of one variable and implements a method "___call___" that calls and stores the output of the function f, so that if the function is called twice with the same argument, the stored output is returned without calling again f.

Test your functor on a recursive function implementation of the Fibonacci numbers. ■

5.7 Exceptions

Exceptions are used to handle errors. An exception in python is a class derived from **BaseException**. Exceptions are handled using the **try** and **except** keyword.

```
try:
      num = int(input("Give me an integer number"))
      print("The inverse of the number is", 1/num)
except ValueError:
      print("Invalid integer number!")
except ZeroDivisionError:
      print("The number zero has no inverse")
else:
      print("No exception were raised")
```

The python interpreter will execute the block of instructions followed by the **try** keyword. If an exception is raised, for instance because the user inserted an invalid integer, all exceptions listed by the keyword **except** are tested until the one raised is found and the corresponding instructions are executed. The **else** instruction block is executed if no exceptions have been raised.

In old programming languages, an integer input parameter was used to store an error code during the execution of functions, resulting in situations complex to manage compared to the exception handled by python, where an exception interrupts the execution of the code if not captured. In addition, if an exception is not captured, the python interpreter will show the full stack of function call that lead to an exception together with the line of code where the exception has been raised, allowing an easy debugging and understanding of the error.

It is also possible to **raise** an exception when an error occurs, providing a message to the user.

```
class Square:
      def __init__(self, length):
```

```
            if length < 0:
                  raise ValueError("The length of a\
                        Square cannot be negative!")
            self._length = length

test = Square(-5.6)
```

Customized exceptions can be defined by inheritance from the class **BaseException**.

Exercise 5.6 The example Vector2D considered in Sec. 5.4 is only restricted to two dimensions. Can you write a class to handle an n dimensional vector? Include in the extension also the scalar product and the element-wise multiplication of two vectors. What should it happen if the user tries, for instance, to sum two vectors with different dimensions? ■

6. Decorators

> For people who have a passing acquaintance with algebra (or even basic arithmetic) or have used at least one other programming language, much of Python is intuitive. Very few people will have had any experience with the decorator concept before encountering it in Python. There's just no strong preexisting meme that captures the concept.
>
> PEP 318 [11]

Decorators are a more advanced tool based on the idea that a function can return an another function; possibility open by the dynamical nature of python, as there is no real difference between a function and any other type. This observation allows to encapsulate, control and even modify the behavior of functions and classes from outside.

6.1 Function that returns a function

In python, unlike C and C++, there are no obstacles preventing a function from returning an another function.

```
import math
import random

def random_function_provider():
      functions = [math.cos, math.tan, math.exp, math.sin]
      return random.choice(functions)
```

```
random_f = random_function_provider()
print("The evaluation of a random function, chosen",
      "between cos, tan, exp, sin, of 0 and \pi/2 is",
      random_f(0.), "and", random_f(math.pi/2))
name = input("Can you guess the function?")
if name == random_f.__name__:
      print("Correct!")
else:
      print("Incorrect, the function was",
            random_f.__name__)
```

The function "random_function_provider" uses the **choice** function from the module **random** to return randomly a function between cos, tan, exp, sin. The returned function can be stored in the variable "random_f", which now behaves effectively as a standard function (indeed, it *is* a function), and can be called as in the last line before the input. The class function has a property "___name___" which stores the name of the function.

The example above might appear quite useless, however, following the same intuition, we can easily code a function to provide "symbolic" analytical differentiation.

```
import math

def derivative(f):
      deriv = {math.sin: math.cos,
               math.cos: (lambda x: - math.sin(x)),
               math.tan: (lambda x: 1./math.cos(x)**2)}
      if f in deriv:
            return deriv[f]
      else:
            raise NotImplementedError

df = derivative(math.cos)
print("The derivative of cos(x) at x=0 is ", df(0))
```

Notice that we have used a dictionary to define a map between a function and its derivative, instead of using a difficult to maintain static if-elif-else block of instructions.

6.2 A simple decorator

We first describe a simple decorator that squares the output of a function to illustrate the logic behind this powerful but at beginning obscure concept.

```
def squared(func):
      def inner(*args,**kwargs):
            return func(*args,**kwargs)**2
```

```
      return inner

def difference(x,y):
      return x-y

wrapped_difference = squared(difference)
difference = wrapped_difference
print(difference(6,9))
```

When the function "squared" is called, the inner function is returned without being executed. The variable "wrapped_difference" is now a function, that, when called, returns the square of the output of the function that was provided as input in the beginning. The function "difference" can be now set to be equal to "wrapped_difference". As final result, the function "difference" changed behavior without changing the definition of function itself: this mechanism is the core of decorators.

6.3 Counter of calls

Assume that you want to monitor how many time a given function is called, a crucial information for instance to understand which part of the code should be optimized.

Changing the code of the function itself would be an impractical possibility (think to how to monitor how many times the built-in function **max** is called). A better idea is to develop a function to "wrap" the target function whose calls should be monitored. Such a function will print a message each time the wrapped function is called before actually calling it.

```
def counter_of_calls(func):
      counter = 0

      def inner(*args, **kwargs):
            nonlocal counter
            counter += 1
            print("Function", func.__name__, "called",
                  counter, "times")

            return func(*args,**kwargs)

      return inner

wrapped_max = counter_of_calls(max)
max = wrapped_max
max(8,21,3)
```

The function "counter_of_calls" is called **decorator**, defining and returning an another function called "inner". The core of the decorator is in the "inner" function, which returns the call to the function given as argument, before printing the number of calls:

- When the function "counter_of_calls" is executed giving "max" as input parameter, a counter variable is initialized to zero, and the "inner" function is returned.
- In the main code, "wrapped_max" is now a function, which was defined as "inner" inside "counter_of_calls". The function "wrapped_max" can accept an arbitrary number of arguments.
- When executed, "wrapped_max" updates the counter, it prints the number of calls, and then it returns back the result of "func", which is exactly the function "max" which was provide as argument in the beginning. The keyword **nonlocal** instructs the interpreter that the counter variable is not a local function variable, but that it is defined and set in the outer scope.
- The statement "max = wrapped_max" sets the original "max" function to be equal to its wrapped version, so that each time that the function max is called, before computing the max, the python interpreter will execute, update and prints the counter of calls log.

When possible, a decorator can be explicitly called using the "@" and the function name just before the wrapped function definition.

```
...

@counter_of_calls
def add(a,b):
      return a+b
```

The above code is effectively equal to

```
...

def add(a,b):
      return a+b

add = counter_of_calls(add)
```

Exercise 6.1 Write a decorator to measure the execution time of a function. The printed output of the decorated function should be:

```
The function <name function> has been executed in 3.1 secs.
```

Use the decorator to see how long does it take to execute the function **fibonacci**(34) defined using recursion. ∎

6.4 Caching a function

A cache decorator is a standard solution when calling repetitively function long to be evaluated with the same arguments, it can also solve the problems of recursion encountered in chapter 4. The idea is to use a decorator to store in a dictionary the input and the output of a function, in such a way that if a function is called again with the same input, the output can be immediately returned.

```
def cache(func):
      call_map = dict()

      def inner(*args):
            #We first check if we have not already
            #computed the function
            if args not in call_map:
                  #If not we compute and cache it!
                  call_map[args] = func(*args)
            #Then we return the precomputed output
            return call_map[args]

      return inner

@cache
def fibonacci(n):
      if n == 1 or n == 2:
            return 1
      else:
            return fibonacci(n-1)+fibonacci(n-2)
```

The recursive definition of the Fibonacci number F_n would require to compute many times the same number. In fact, F_n requires F_{n-1} and F_{n-2}, but F_{n-1} requires also F_{n-2}, which would be recomputed if it were not stored by the decorator.

Exercise 6.2 Compare the computing time of the fibonacci function above with and without decorator for $n = 30$. Do you notice a difference? Estimate the number of function calls in both cases. ∎

A full cache decorator handling also keyword arguments and unhashable arguments whose value cannot be used as key in a dictionary requires some more code. Such cache decorator is however already available under the name "lru_cache" of the module **functools**.

6.5 Arguments to decorators

It is possible to iterate further the logic of decorators to allow arguments to be given as input parameter to a decorator.

```
import time

def wait(secs):
      def decorator(f):
            def inner(*args,**kwargs):
                  time.sleep(secs)
                  return f(*args,**kwargs)

            return inner
```

```
        return decorator

@wait(8)
def add(a,b):
        return a + b

add(8,9)
```

The first time the "wait" function is called giving the number of seconds as input and the actual decorator as output. The decorator acts on "sum" and returns the decorated function as usual, which wait before executing the function itself.

6.6 Decorating a class

Class can be also decorated, for instance to add standard methods and it can be useful in the context of metaprogramming, i.e. when implementing programs and algorithms intended to modify other programs, or, in this case, classes.

```
def log_properties(cls):
        class Inner(cls):
                def __init__(self, *args, **kwargs):
                        super().__init__(*args, **kwargs)
                        for attr in self.__dict__:
                                print(attr, "initialized to",
                                        self.__dict__[attr])

                def __setattr__(self, attr, value):
                        print("setting", attr, "to", value)
                        self.__dict__[attr] = value

        return Inner

@log_properties
class Square:
        def __init__(self, length):
                self.length = length

a_square = Square(5)
a_square.length = 9
```

In the example above, we use the method "___setattr___" to log the attributes of the objects. Each time an attribute is set to a new value, a message will be printed.

Exercise 6.3 Modify the decorator above to log also each time an attribute of an object is accessed. ∎

6.7 Annotations

Annotations are declared after each argument of a function and stored in the attribute "__annotations__". The returned value is annotated after the "arrow" "->" before the colon. Annotations can take any value.

```
def area_of_square(length: float) -> float:
      return length**2

print(area_of_square.__annotations__)
print(area_of_square(9))
print(area_of_square(9.5))
```

Notice that the annotation does not force the input parameter to be a float. The meaning of annotations is therefore a guidance given to the user to be interpreted depending on the context.

Annotations can be combined with decorators. For instance we can ensure that the returned type is indeed the one declared in the annotations.

```
def force_return_type(f):
      def inner(*args,**kwargs):
            return f.__annotations__["return"](f(*args,
                  **kwargs))

      return inner

@force_return_type
def area_of_square(length: float) -> float:
      return length**2

print(area_of_square(9))    #Prints 81.0
print(area_of_square(9.5))
```

From the annotations, we take the value of the key "return", in the example the class float, used to try instantiate and convert the output of the function to decorate. Following the same idea, we can also check the type of the input parameters.

Annotations can be used also to provide a documentation on the expected parameters.

```
import inspect

def log_and_print_documentation(f):
      def inner(*args,**kwargs):
            print("Calling function", f.__name__)
            binded_args = inspect.signature(f).\
                  bind(*args,**kwargs).arguments
            for arg in binded_args:
                  print(arg, f.__annotations__[arg],
                        "set to", binded_args[arg])
            res = f(*args,**kwargs)
```

```
            print("Returning", res, "as",
                  f.__annotations__["return"])
            return res

      return inner

@log_and_print_documentation
def area_of_rectangle(a: "the first side", b:
      "the second side") -> "the area of a rectangle":
      return a*b

area_of_rectangle(4,5)
#a the first side set to 4
#b the second side set to 5
#Returning 20 as the area of a rectangle
```

The **inspect** module is used to get the signature of the function to decorate and **bind** the arguments of the call to iterate over all of them, including the positional one, by keyword.

Exercise 6.4 Write a decorator **check_range** to check the range of the input value of a function, the range being provided with annotations. The decorator should for instance work like

```
@check_range
def f(a: (0, 9), b: (3,12)):
   return a+b
```

so that if the input variables "a" and "b" are outside the range specified, such as by calling f(5,13), an exception ValueError is raised. ■

7. Python design patterns

> Design patterns should not be applied indiscriminately. Often they achieve flexibility and variability by introducing additional levels of indirection, and that can complicate a design and/or cost you some performance. A design pattern should only be applied when the flexibility it affords is actually needed.
>
> Erich Gamma, *Design Patterns: Elements of Reusable Object-Oriented Software*

Design patterns are reusable solutions to common recurring problem in the context of software design. Patterns address typical problems in object oriented programming languages, and allow to find an elegant and clear solution that can be easily adapted when considering the actual implementation. In this chapter we will see three basic examples of design patterns adapted for python.

7.1 Singleton pattern

The singleton pattern allows to define a class which is ensured to have a single and unique instance. A singleton can handle operations and a state across the execution of different part of the code. In python, a singleton can be implemented by overriding the method "___new___", that effectively controls the creation of an objection (to do not be confused with "___init___", the initializer of an already created object).

```
class Singleton:
      __instance = None
```

```
        def __new__(cls, *args, **kwarg):
                if Singleton.__instance is None:
                        Singleton.__instance = \
                                object.__new__(cls)
                return Singleton.__instance
        def __init__(self, a, b):
                self.a = a
                self.b = b

first = Singleton(2,3)
print(dir(first))
print(first.a)
second = Singleton(4,5)
print(first.a)
first.b = 5
print(second.b)
```

In the example above, the value of "first.a" is changed after the call of the initializer for "second", as no new instances are created after "first". The "___new___" method checks if the instance already exists, if not it creates it by calling the "___new___" of the corresponding base class, in this case **object**.

> **More on ... 7.1.1 — object.** In python 3, every class inherits from the base class **object** automatically (while in python 2 it was not the case).
>
> ```
> ...
>
> print(isinstance(Singleton,object))
> ```
>
> The built-in function **isinstance** can be used to check whether a class is an instance of an another class.

A singleton class behaves effectively as a sort of global variable, whose utility should be carefully considered. A standard use of a singleton object is to provide a logger that can be accessed and store information during the execution of the whole program.

```
class Logger:
        __instance = None
        def __new__(cls, *args, **kwarg):
                if Logger.__instance is None:
                        Logger.__instance = object.__new__(cls)
                return Logger.__instance

        def __init__(self):
                #Try to see if the logger string
                #already exists
                try:
                        self.message += ""
```

```
            except AttributeError:
                  #If not, we initialize it
                  self.message =
                        "Initializing a new logger ...\n"

      def log(self, msg):
            self.message += msg
            self.message += "\n"

      def __str__(self):
            return self.message


logger1 = Logger()
logger1.log("First test of log")
logger2 = Logger()
logger2.log("Second test of log")
print(logger2)
```

7.2 Iterators and generators

Generators and iterators are design pattern that allows to disentangle the part of the code acting on elements of a container from the part of the code required to scan the elements of the container itself.

In python, **generators** are a flexible way to create custom iterators. Class generators implement the method "__iter__" and are distinguished by the keyword **yield**. Generators can be used in **for**-loops.

```
class EvenNumbers:
      def __init__(self, limit):
            self.limit = limit
            self.i = 0

      #called when testing "in"
      def __contains__(self, element):
            return (element % 2 == 0
                  and element < self.limit)

      #called in for-loops
      def __iter__(self):
            self.i = 0
            while i < self.limit:
                  yield self.i
                  self.i += 2

      def __len__(self):
```

```
            return self.limit//2 + 1

print(5 in EvenNumbers(100))
print(len(EvenNumbers(100))
for i in EvenNumbers(100):
      print(i)
```

When the python interpreter encounters the keyword **yield**, it returns and corresponding value and save the function execution point. When called again, the evaluation of the function continues from the last call without starting from the beginning.

A generator can be converted to an **iterator**, whose elements can be accessed successively one after the other using the function **next**.

```
...

evenNumberIterator = iter(EvenNumbers(100))
print(next(evenNumberIterator))
print(next(evenNumberIterator))
print(next(evenNumberIterator))
```

Exercise 7.1 Write a generator for the Fibonacci sequence running up to a given F_n by defining a class with a member function **___iter___**. Add also to the same class the possibility to check if a number **is** in the given sequence (function **___contains___**), and the possibility to access randomly the element k of the sequence (function **___getitem___**). If k is larger than n, an **IndexError** exception should be raised. ∎

7.3 Decorator pattern: wrappers

Wrappers are frequently a better alternative to class inheritance when extending or modifying the behavior of a given class. In the language of design pattern, the pattern described is known as "decorator" (to do not be confused with the python decorators explained in the previous chapter). The decorator pattern extends the functionality of a class without using inheritance, allowing at the same for example to disable certain functionalities which are not requested.

```
#First we define a class-decorator for later convenience
class AttributesHiderDecorator:
      def __init__(self, class_to_be_wrapped,
                   methods_to_hide = []):
            self.class_to_be_wrapped = class_to_be_wrapped
            self.methods_to_hide = methods_to_hide

      #The decorator returns an instance to the actual class
      def __call__(self,*args,**kwargs):
            return AttributesHider(
```

```
                self.class_to_be_wrapped(*args,**kwargs),
                self.methods_to_hide)

#Wrapper class that hides some of the methods
#of the wrapped class
class AttributesHider:
      def __init__(self, obj, methods_to_hide):
            self.wrapped_object = obj
            self.methods_to_hide = methods_to_hide

      def __getattr__(self, name):
            #When we want to get an attribute
            #We try to get it from the wrapped
            #object
            result = getattr(self.wrapped_object,name)
            #But if the attribute is hidden,
            #we do not return it!
            if (name in self.methods_to_hide):
                  raise AttributeError(
                        "Attribute {} is hidden!"
                        .format(name))
            return result

      def __setattr__(self, attr, value):
            #Check first on the wrapper class
            #attributes to avoid infinite
            #recursion
            if (attr == "wrapped_object" or
                  attr == "methods_to_hide"):
                  #and use the __dict__ for
                  #this purpose
                  self.__dict__[attr] = value
            else:
                  #If the attribute is hidden
                  #We do not set it!
                  if name in self.method_to_hide:
                        raise AttributeError(
                              "Attribute {} is hidden!"
                              .format(name))
                  setattr(self.__dict__['wrapped_object'],
                        attr, value)

      def __delattr__(self, attr):
            delattr(self.__dict__['wrapped_object'], attr)
```

```
class Person:
      def __init__(self, name, age):
            self.name = name
            self.age = age

      def get_age(self):
            return self.age

      def get_name(self):
            return self.name

SecretIdentityPerson = AttributesHiderDecorator(Person,
      ["get_name","name"])
superman = SecretIdentityPerson("Clark Kent", 32)
print("Superman is", superman.get_age(), "years old.")
print("The name of Superman is", superman.get_name())
print("The name of Superman is", superman.name)
```

The example above works effectively in the "opposite" way of inheritance, as we have created a new wrapped class "SecretIdentityPerson" that has an hidden method and an hidden property, "get_name" and "name", respectively. When we try to print the name of superman, we will encounter an "AttributeError". We can also add new methods to the wrapper class to extend the functionalities of the wrapped class, allowing therefore more flexibility with respect to standard inheritance.

8. Graphic User Interfaces

Python has a built-in module based on the library **Tkinter** to develop Graphic User Interfaces (GUI), allowing a simpler and more advanced interaction between the user and python applications compared to command line scripts.

8.1 Messageboxes

Messageboxes are the simplest graphic tool to interact with the user provided by the submodule **messagebox** of **tkinter**.

```
import tkinter.messagebox as mbox

mbox.showinfo("Tk", "Hello World!")
mbox.showerror("Error", "Fatal error!")
```

A messagebox can ask a question to the user and returns the answer.

```
import tkinter.messagebox as mbox

yes_clicked = mbox.askyesno(
  "Yes or no?",
  "Is Berlin in Germany?")

if yes_clicked:
  mbox.showinfo(
    "Correct!",
    "Berlin is the capital of Germany.")
else:
  mbox.showinfo(
```

```
    "Wrong!",
    "You should improve your knowledge of geography.")
```

Exercise 8.1 Using the module **tkinter.messagebox**, write a simple program that waits four seconds and then displays an information message. The message should be "All documents have been successfully deleted from your hard disk.". Given that the script could really run some functions from the module **os**, how safe is to run a python code that you have not carefully read? Do you have a solution to prevent such a situation? If so, the program should wait other ten seconds before displaying an another message "It was a joke, don't worry! However the next time I would suggest to ... *your suggestion*". ■

8.2 Graphic applications and events

The development of advanced graphic applications requires to first define the **main window** that is going to be filled with **widgets**. After creating the widgets, we need instruct the python interpreter and tkinter on what to do in response of a certain events, such as a key pressed or the click of a button of the mouse. The function **bind** associates to each event of a Widget a **callback** function. At the end, we call the function **tk.mainloop** to run the window application until the user decides to close it.

```
import tkinter as tk
import tkinter.messagebox as mbox

#First create the main window
window = tk.Tk()
window.title("Buttons")

#Create the first button
button1 = tk.Button(master=window,text="Click me!")
#Insert it in the main window
button1.pack()

#Do the same for the second
#and third button
button2 = tk.Button(master=window,
  text="Click me using the right button of your mouse!")
button2.pack()

button3 = tk.Button(master=window,
  text="Try to click me!")
button3.pack()

#Define three callback functions
def callback1(event):
      #The event class contains the position of the mouse
```

```
        #relative to the widget
        mbox.showinfo("Button clicked",
          "Mouse position {} {}".format(event.x, event.y))

def callback2(event):
        mbox.showinfo("Button right-clicked",
          "Mouse position {} {}".format(event.x, event.y))

def callback3(event):
        mbox.showinfo("Mouse over the button",
          "Mouse position {} {}".format(event.x, event.y))

#Bind the first callback to the left click
#of the mouse on the first button
button1.bind("<ButtonRelease>", callback1)
#Bind the second callback to the right click
#of the mouse on the second button
button2.bind("<Button-3>", callback2)
#Bind the third callback to the mouse
#entering on the third button
button3.bind("<Enter>", callback3)

tk.mainloop()
```

In addition to the coordinates of the mouse, keyboard events store also the character pressed in the data member **char**. If you want to know the complete list of events available for a widget, consult the manual and the function bind of the corresponding widget.

Inheritance is the easiest way to customize a widget. In the example below, the CounterButton class stores as data member the number of times a button has been clicked, and update the text accordingly.

```
import tkinter as tk

class CounterButton(tk.Button):
        def __init__(self, *args, **kwargs):
                super().__init__(*args, **kwargs)
                self.n = 0
                #bind the click event to the self.click callback
                self.bind("<ButtonRelease>", self.click)

        def click(self, event):
                #change the text of the button by calling
                #the method configure
                self.configure(text =
                        "Button clicked {} times".format(self.n))
                self.n += 1
```

```
#First create the main window
window = tk.Tk()
window.title("Custom button")

button = CounterButton(master=window,text="Click me!")
button.pack()

tk.mainloop()
```

8.3 Checkbuttons and radiobuttons

Radiobuttons are used to select one of many options, while checkbuttons are used to allow multiple choices. The status of a radiobutton is stored inside a **tk.VarInt** variable, which can be used to retrieve which option has been select outside of the Radiobutton class.

```
import tkinter as tk

#First create the main window
window = tk.Tk()
window.title("Choices")


#Create a label to show a text
label = tk.Label(master=window, text="Select some option")
label.grid(row=0, column=0, columnspan=2)

#Create the Radiobutton
option = tk.IntVar()

r_button1 = tk.Radiobutton(master=window,
      text="First option", variable=option, value=1)
r_button1.grid(row=1, column=0)

r_button2 = tk.Radiobutton(master=window,
      text="Second option", variable=option, value=2)
r_button2.grid(row=2, column=0)

#Create the Checkbutton, each button requires a variable
extra1 = tk.BooleanVar()
extra2 = tk.BooleanVar()

c_button1 = tk.Checkbutton(master=window,
```

```
      text="First extra option", variable=extra1)
c_button1.grid(row=1, column=1)

c_button2 = tk.Checkbutton(master=window,
      text="Second extra option", variable=extra2)
c_button2.grid(row=2, column=1)

#Callback function to show the select options
def callback(event):
      text = "Selected option {}".format(option.get())
      if extra1.get() == 1:
            text = text + ", and extra option 1"
      if extra2.get() == 1:
            text = text + ", and extra option 2"
      label.configure(text=text)

#Bind all buttons to the callback
c_button1.bind("<ButtonRelease>", callback)
c_button2.bind("<ButtonRelease>", callback)
r_button1.bind("<ButtonRelease>", callback)
r_button2.bind("<ButtonRelease>", callback)

tk.mainloop()
```

The grid layout in the example above is organized like a table, where widgets are placed in corresponding rows and columns. A widget can also occupy multiple rows and multiple columns, by using the option **columnspan** and **rowspan** of the function **grid**.

Exercise 8.2 Using the module **tkinter**, create a simple GUI to ask to the user which one is the best programming language among "java", "python", "C", "C++", "fortran" and "perl". The user should be able to select only one option. If "python" is selected, a message "Excellent choice!" should be shown, otherwise the message should be "I respect your opinion, but I totally disagree with you, programming in xxx is horrible!" (xxx in the message should be replaced with the name of the programming language chosen by the user). ■

Exercise 8.3 Using the module **tkinter**, create a GUI to order a Pizza. The user should be able to select one or more ingredients ("pineapples", "mushrooms", "prosciutto", "onions", "sardellen", ...), possible extras (extra cheese, extra tomato sauce, basilicum etc) and a field free for comments of the user. The user should click on the button "Order" to order the pizza. If pineapples are chosen as ingredients, an error message "Pineapples for a pizza are totally wrong, please retry!" should be shown, otherwise the message should show the ingredients, the extras selected and the comment of the user. ■

8.4 A simple text manager application

Typical GUI applications are developed following the principles of object oriented programming. A common choice is to implement the main application inheriting from the class **tk.Frame**, which is container widget for organizing and grouping other widgets. In the example below, this option is demonstrated by implementing a very basic text editor application. It is also shown how to develop menubars for the main window.

```
import tkinter as tk
import tkinter.filedialog as filedialog
import tkinter.messagebox as mbox

class TextEditor(tk.Frame):
      def __init__(self, *args, **kwargs):
            super().__init__(*args, **kwargs)
            #First create the menu bar
            self.menubar = tk.Menu()

            #Fill in the file menu bar
            self.file_menubar = tk.Menu(self)
            self.file_menubar.add_command(label="Save",
                  command=self.save)
            self.file_menubar.add_command(label="Exit",
                  command=self.quit)
            self.menubar.add_cascade(label="File",
                  menu=self.file_menubar)
            self.master.configure(menu = self.menubar)

            #Finally create the text widget
            self.text = tk.Text(self, height=50, width=50)
            self.text.pack()

            self.pack()

      def save(self):
            f = filedialog.asksaveasfile(mode='w',
                  defaultextension=".txt")
            #If the user did not close the save
            #window by "cancel"
            if f is not None:
                  #Save the text
                  f.write(str(self.text.get(1., tk.END)))
                  f.close()

      def quit(self):
            q = mbox.askyesno(
                  "Exit?",
```

```
                "Do you really want to quit?")
            if q == 1:
                super().quit()

window = tk.Tk()
window.title("Text editor")

app = TextEditor(master = window)

tk.mainloop()
```

Exercise 8.4 Create a class that implements a GUI interface to ask multiple choice questions to the user. The initializer should take as input the question, a list of the possible answers, and an index of the correct one. A message should inform the user whether the selected answer is correct or not.

Use the class to create a quiz with the following questions:

- "Ratisbona" is
  1. an Italian name for a pizza
  2. the Italian name of Regensburg
  3. an Italian cheese
- Germany has a population of roughly
  1. 83 million
  2. 67 million
  3. 101 million
- Santa Claus lives in
  1. Finland
  2. Bayern
  3. Nowhere, he does not exists.

■

III Part Three: Numerical python

9. Numpy and matplotlib

Plotting data is a basic but still underrated skill in science and data science. A convincing plot might be a crucial ingredient for a scientific publication, or might be the most important element to convince the reader about a theory or a simple statement. In this chapter we describe the library **matplotlib**, and we will also introduce the first basic ingredients of the library **numpy**.

9.1 Numpy and arrays

Numpy is one of the most important numerical library implemented for python. Numpy has many functionalities, ranging from linear algebra to Fourier Transforms. In this chapter we will widely use the class **numpy.array** to store arrays of **homogeneous** data. Numpy arrays can be initialized from lists.

```
import numpy as np

a = np.array([8,9,5])
b = np.array([5.6,2.1,-3.3])
c = np.array([[9,1,2],[2,3,4]])
print(a)
print(b)
print(c)
print(a[1])
print(c[2])
print(c[:,1])
```

Numpy arrays have one, two or more dimensions, and they typically contain integers or floats. Multiple elements across many indexes of two or more dimensional arrays

can be accessed using commas, in the line of the example above we are extracting from each entry (the first ":") the second element (the second "1" after the comma).

Numpy arrays can be element-wise summed, subtracted, multiplied, divided; multiplied by a constant and mathematical expressions can computed on all entries if they are called from the module **numpy**.

```
import numpy as np

a = np.array([-1.2,-9.18,5.1])
b = np.array([1.6,3.1,-3.7])
print(a+b)
print(a-b)
print(a*b)
print(a/b)
print(5*b)
print(np.cos(a))
print(np.exp(a))
```

Numpy arrays are therefore an efficient alternative to lists and for-loops when working with arrays of homogeneous type.

Arrays set to zero or to one can be created using the function **zeros** and **ones**.

```
import numpy as np

a = np.zeros(shape=10) # One-dimensional array with
                       # ten elements

b = np.ones(shape=(2,3)) # Two-dimensional array, with
                         # 3*2 elements in total
```

An important advantage of numpy is its low level implementation that allows to run very fast vector operations. In the example below we try to perform the sum of the entries of a very large array if their value is smaller than three.

```
import numpy as np
import time

v = np.random.uniform(-5,5,size=1000000)

start_time = time.time()
np_sum = np.sum(v[v < 3])
end_time = time.time()
print(end_time-start_time)

start_time = time.time()
my_sum = 0
for i in range(1000000):
    if v[i] < 3:
        my_sum += v[i]
```

```
end_time = time.time()
print(end_time-start_time)

print(np_sum, my_sum)
```

For the numpy directive, "v < 3" is an array of booleans depending on whether the corresponding entry of the array "v" is smaller than three or not, while "v[v < 3]" is an array containing only the entries fulfilling the condition "v < 3". Depending on the architecture, numpy in the code above might be a factor 100 times faster than the simple python for-loop.

Exercise 9.1 Considering the performance test above, what happens if the size of the array is ten thousands or ten millions? ■

9.2 Plotting a function

The module **matplotlib.pyplot** is meant to produce from simple to high quality complex plots. The starting point is to create an numpy.array to store the coordinates and ordinates, given as input to the function plot.

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

#The x and y coordinates
x = np.linspace(-2,2,1000)
y = np.exp(x)

#plot the function exp(x), the plt.plot function
#returns a list containing a line
#that we can give to the legend function
p1,  = plt.plot(x,y,label="exp(x)",color="red")
plt.xlabel("x")
plt.ylabel("y")

#add the legend
plt.legend(handles=[p1])

#Save and show the plot
plt.savefig("exp_plot.png")
plt.show()
```

The function **np.linspace** is used to create an array of thousand evenly separated floating numbers between -2 and 2.

More on ... 9.2.1 — Matplotlib. The preparation a plot using matplotlib directives can follow two possible ways. As described above, the plotting instructions can given one after the other directly using the corresponding functions of **mat-**

plotlib.pyplot and the final is shown at the end. Alternatively, it is possible to get an instance of an **Axis** object, to populate it with the curves to plot, set the relative options, and finally show the plot. In the latter way, it is possible to manage multiple plots simultaneously, as demonstrated in the example below.

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

#The x and y coordinates
x = np.linspace(-2,2,1000)
y1 = np.exp(x)
y2 = np.cos(x)

#Prepare two plots (1*2) for the two
#functions

#Get the first axis (last parameter = 1)
ax1 = plt.subplot(1,2,1)
#Get the second axis (last parameter = 2)
ax2 = plt.subplot(1,2,2)

ax1.yaxis.set_label_text("exp(x)")
ax2.yaxis.set_label_text("cos(x)")

ax1.xaxis.set_label_text("x")
ax2.xaxis.set_label_text("x")

ax1.plot(x,y1)
ax2.plot(x,y2)

#Adjust the spacings between the two plots
plt.subplots_adjust(wspace=0.4,left=0.1,top=0.9,bottom=0.1)
plt.show()
```

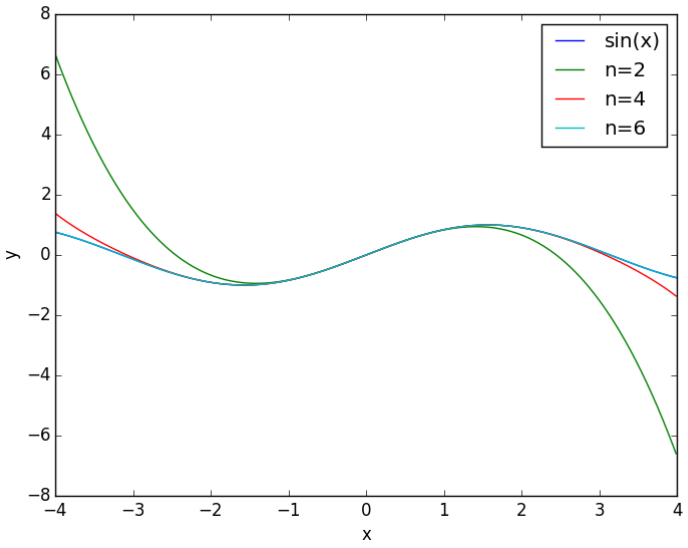
Exercise 9.2 The Taylor approximation of a continuous function $f(x)$ in the neighborhood of a point x_0 is given by the formula

$$f(x) = f(x_0) + \sum_{n=1}^{N} f^n(x_0)\frac{x^n}{n!}, \tag{9.1}$$

the sum can run up to n as long as the n-th derivative $f^n(x_0)$ evaluated at the point x_0 exists. For instance, the Taylor expansion of $\sin(x)$ around $x = 0$ is

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!} \tag{9.2}$$

Using the module **matplotlib.pyplot**, prepare a plot that shows the Taylor approximation of $\sin(x)$ between -4 and 4 including the first two, four, and six terms of the infinite series above. The plot should look approximatively as below.



9.3 Scatter plots

Unlike plots of continuous function of a single variable, two-dimensional scatter plots are intended to show points corresponding to discrete series of the form $\{(x_1, y_1), \ldots, (x_n, y_n)\}$. Suppose we want to compare the Gross Domestic Product (GDP) of Germany, France, Spain and Italy starting from 1995. In this case, the x_n is a discrete time variable, the year, while y_n corresponds to the time series of the GDP of a country. Scientific dataset are available in many different formats, and XML is a quite common choice.

More on ... 9.3.1 — XML. XML (eXtensible Markup Language) allows to store data to a file without loosing their structure. Assume for instance that you want to store the information about a class of students. Storing name, surname, and matriculation number one after the other for all student would be the simplest possibility. Assume however you want add also the information about the teacher of the class, and maybe later on the marks for the final exam. Assume the mark is a number, then it could be easily confused with the matriculation number, and we would need some special character to distinguish the two. Each time new information are added, the interface for reading and writing the student class would have to change, and the old code would become incompatible with the new format.

XML helps in solving this problem. A typical XML file would look like:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<class>
        <teacher>
                <name>Stefano</name>
        </teacher>
        <student>
                <name>Alexander</name>
                <matr_number>1234</matr_number>
        </student>
        <student>
                <name>Sasa</name>
                <matr_number>2341</matr_number>
        </student>
        <student>
                <name>Alessia</name>
                <matr_number>3412</matr_number>
        </student>
</class>
```

In the beginning the XML version is written together with the character encoding. The whole information is stored in a tree structure, the root being “class”. From the root, we can go down to the "student" to reach the “name” and “matr_number” information in the leafs. Each tag is delimited by “<” and “>” as “<tag_name>” and must be closed as “</tag_name>”.

Python offers several libraries to handle with XML files. To create the file above we can use the library **xml.etree**.

```
#To handle trees we can use xml
import xml.etree.cElementTree as xtree

#We first need a root element
root = xtree.Element("class")

#Then we add a student as subelement
student = xtree.SubElement(root,"student")
#And also his name as subelement of student
xtree.SubElement(student,"name").text = "Alexander"
#and his matriculation number
xtree.SubElement(student,"matr_number").text = "1234"

#The same for the other students
student = xtree.SubElement(root,"student")
xtree.SubElement(student,"name").text = "Sasa"
xtree.SubElement(student,"matr_number").text = "2341"

student = xtree.SubElement(root,"student")
xtree.SubElement(student,"name").text = "Alessia"
```

```
xtree.SubElement(student,"matr_number").text = "3412"

#Now for the teacher
teacher = xtree.SubElement(root,"teacher")
xtree.SubElement(teacher,"name").text = "Stefano"

#We create now the tree
xmltree = xtree.ElementTree(root)
#And we store it in xml
xmltree.write("python_class.xml")
```

In the same way, we can read the information stored in a xml file using the function **parse**. In the example above, we just scan for the name of all students, and we do not need to care whether one of the name in file correspond to the teacher, as the name of the teacher is stored in a different tag.

```
import xml.etree.cElementTree as xtree

root = xtree.parse("python_class.xml").getroot()

#We can iterate for each element in the tree
#staring from the root
for child in root.iter("student"):
      #Finding the information for the name and
      #print it!
      print("Student name", child.find("name").text)
```

The GDP data are available from the World Bank Data in the XML. We use the module **urllib.request** to download the data directly from within python and the module **os** to check if the XML datafile is already present. After collecting the data, we plot the GDP time series using the function **scatter**.

```
import xml.etree.ElementTree
import matplotlib.pyplot as plt
import os
import urllib.request
import zipfile
import numpy as np

url_gdp = "http://api.worldbank.org/v2/en/indicator/NY.GDP\
.MKTP.KN?downloadformat=xml"
xml_gdp = "./API_NY.GDP.MKTP.KN_DS2_en_xml_v2_10230884.xml"
xml_zip = "API_NY.GDP.MKTP.KN_DS2_en_xml_v2_10230884.zip"

def getLCUGDP(country, year_start=0):
      #If the XML file is not there
      if not os.path.isfile(xml_gdp):
```

```
            #Download it from the worldbank database
            urllib.request.urlretrieve(url_gdp, xml_zip)
            #Unzip the file
            zip_ref = zipfile.ZipFile(xml_zip, 'r')
            zip_ref.extractall("./")
            zip_ref.close()

      #get the root
      root = xml.etree.ElementTree.parse(xml_gdp).getroot()

      gdp = []

      #read the GDP data of country
      for data in root.findall("./data/record"):
            #We access to the subfield using an index
            #Index starts from 1!
            data_country = data.find("field[1]").text
            data_year = data.find("field[3]").text
            data_gdp = data.find("field[4]").text
            #Search for the country and the year requested
            if (data_country == country
                  and int(data_year) > year_start):
                  try:
                        yr = int(data_year)
                        dr = float(data_gdp)
                        gdp.append([yr,dr])
                  #For some country and for some year
                  #the GDP might be missing
                  except ValueError:
                        pass
      return np.array(gdp)


handles = []
for country in "Germany","Italy","Spain","France":
      gdp = getLCUGDP(country,1995)
      p = plt.scatter(gdp[:,0],gdp[:,1],label = country)
      handles.append(p)

plt.xlabel("year")
plt.ylabel("GDP")
plt.legend(handles=handles)
plt.show()
```

Exercise 9.3 Prepare a scatter plot of the yearly growth in percentage of the nominal and real GDP for France, Germany, Italy, Spain, United Kingdom, USA and Greece for the last thirty years using the data of the World Bank database. The growth in percentage is defined as

$$g(y) = 100 \times \frac{\mathrm{GDP}(y) - \mathrm{GDP}(y-1)}{\mathrm{GDP}(y-1)}. \tag{9.3}$$

Use the function **savefig** to save the scatter plot to a file. ■

9.4 Combining matplotlib and LaTeX

Frequently complex formulas are required when preparing a plot. Matplotlib can be combined with the markup language LaTeX to provide a better handling of mathematical formulas. In the example below we enable LaTeX by setting the global parameters **rc** [13]. LaTeX formulas are inserted using the symbol "$".

```
import numpy as np

#Uncomment to plot directly to file
#import matplotlib as mpl
#mpl.use('Agg')

import matplotlib.pyplot as plt
import matplotlib.ticker as plticker
import math

#Preambles required to have a Latex style fonts
from matplotlib import rc
rc('font',**{'family':'sans-serif',
      'sans-serif':['Helvetica'],'size':22})
rc('text', usetex=True)

#Take the figure and the subplots
fig, ax = plt.subplots()

ax.plot(np.arange(0.,8*math.pi,0.01),
      [math.cos(x) for x in  np.arange(0.,8*math.pi,0.01)])
ax.scatter([math.pi*k/2. for k in range(1,16,2)],
      [0 for k in range(1,16,2)],color='r')
ax.scatter([math.pi*k for k in range(0,9,2)],
      [1 for k in range(0,9,2)],color='y')
ax.scatter([math.pi*k for k in range(1,8,2)],
      [-1 for k in range(1,8,2)],color='c')

ax.set_xlim(0.,8*math.pi)
ax.set_ylim(-1.1,1.1)
```

```
ax.yaxis.set_label_text("$\cos(x)$")
ax.xaxis.set_label_text("$x$")

def f(x, pos):
      return '${x}\pi$'.format(x=int(x/math.pi))

ax.xaxis.set_major_locator(
      plticker.FixedLocator([math.pi*k for k in range(1,9)]))
ax.xaxis.set_major_formatter(
      plticker.FuncFormatter(f))

fig.subplots_adjust(wspace=0.45,hspace=0.45,left=0.15,
      top=0.9,bottom=0.15)

fig.savefig("cos_plot")

plt.show()
```

The module **matplotlib** is relatively flexible and has several options to change the appearance of the plot. In the example above, the x-axes has been changed by the class **matplotlib.ticker**. The two member functions **set_major_locator** and **set_major_formatter** from **matplotlib.axes.Axes** are called in order to show useful ticks for trigonometric functions for the x-axis. In general it is not useful to remember by memory all possible functions offered by matplotlib; it is smarter to understand the examples provided in the manual [14] in order to use and to adapt them when needed. The output of the script is shown in Fig. 9.1.

Exercise 9.4 Prepare a plot of the function $\tan(x)$ similar to the one in Fig. 9.1 in the x-range -4π and 4π. Red points should mark the position of the zeros on the x-axis of $\tan(x)$, while dashed red vertical lines should highlight the position of the asymptotes. Modify the FuncFormatter function so that π is shown instead of 1π for the first tick in the x-axis. ■

9.5 Animation plots

Several pictures can be combined together to produce an animated figure, such as a GIF. The interface to produce animated plots is **animation.FuncAnimation**, which requires to specify a function to update the plot between each frame. In the example below we prepare an animated GIF that shows a stationary wave, whose oscillations leave some node points fixed.

```
import numpy as np

#Uncomment to plot directly to file
#import matplotlib as mpl
#mpl.use('Agg')
```

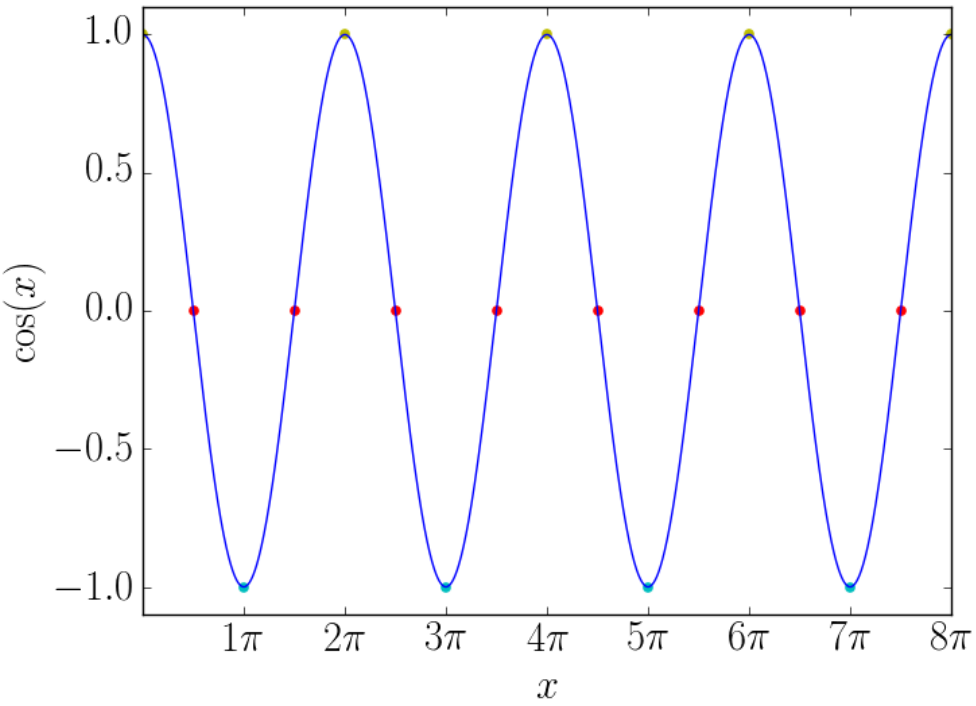


Figure 9.1: Plot of the cos(x) function, with zeros, maxima and minima highlighted.

```
import matplotlib.pyplot as plt

import matplotlib.animation as animation

#Preambles required to have a Latex style fonts
from matplotlib import rc
rc('font',**{'family':'sans-serif',
      'sans-serif':['Helvetica'],'size':16})
rc('text', usetex=True)

xp = np.arange(0,7*np.pi,0.001)
yp = np.cos(xp)

#Take the figure and the subplots
fig, ax = plt.subplots()

#Y axes label
ax.yaxis.set_label_text("$\sin(x)\cos(t)$")
#X axes label
ax.xaxis.set_label_text("$x$")

#Plot the nodes
```

```
ax.scatter(np.arange(0,8*np.pi,np.pi),
      0*np.arange(0,8*np.pi,np.pi),color="blue")

#Get the line of the wave plot
ln, = plt.plot(xp,yp,c="r")

def update(t):
      yp = np.cos(t)*np.sin(xp)
      ln.set_data(xp,yp)

      #Set the title
      ax.set_title('$t={}$'.format(t))

#Center the plots
plt.subplots_adjust(wspace=0.45,hspace=0.45,
      left=0.15,top=0.9,bottom=0.15)

ani = animation.FuncAnimation(fig, update,
      #The t interval for the animation
      frames=np.linspace(0, 4*np.pi, 128),
      repeat_delay=2000)

plt.show()

#save the plot as a GIF using the library
#imagemagick
ani.save('stationary_wave.gif', dpi=300, writer='imagemagick')
```

In the example above, we update the position of the points used to plot the function $\sin(x)\cos(t)$ at each frame.

10. Scipy and numerical algorithms

> Science is what we understand well enough to explain to a computer. Art is everything else we do.
>
> Donald Knuth, $A = B$

The library **scipy** contains many relevant numerical algorithms that can be used out of the box to compute for instance the zeros and the maxima of a function. The study of the ideas behind the numerical algorithms is interesting not only *per se*, but also to find the best algorithm for the problem to be solved and to understand why some algorithm might fail in certain circumstances.

10.1 Root finding

Finding the zeros of a function is a standard numerical problem solved by many different algorithms. Excluding special functions and lower order polynomials, the exact computation of all zeros of a function is in general impossible. The problem to solve is how to find an approximation of a zero of a function.

10.1.1 Bisection

If a continuous function $f(x)$ has a single zero in the interval $[a,b]$, the function crosses the x-axis going either upward, from negative to positive values, or downward, going from positive to negative values. In both cases, the product $f(a)f(b)$ is negative, a condition used as a starting point to find the zero of a function by the bisection algorithm. The algorithm runs recursively, by splitting the interval $[a,b]$ in two equal parts $[a,c]$ and $[c,b]$, with the midpoint c set to

$$c = \frac{a+b}{2}. \tag{10.1}$$

Now, if $f(c)f(b)$ is negative, the root of f is between c and b, otherwise $f(a)f(c)$ must be negative, and the root is between a and c. In either cases, we have been able to restrict the interval where the root is located, and we can start again the same procedure from the interval $[a,c]$ or $[c,b]$. Eventually, by repeating the bisection many times, we will converge to a small interval representing a good approximation of the zero of f. The convergence criteria might be the absolute length of the interval $[a,b]$, or the absolute difference from zero of the function f at the a and b, as in the implementation below.

```
import math
import numpy as np

def bisection(f,a,b,tolerance=0.00001):
      #We check the convergence
      difference_from_zero =
            min(math.fabs(f(a)),math.fabs(f(b)))
      #until the
      while difference_from_zero > tolerance:
            #Midpoint
            central = (a+b)/2.
            #If we spot the zero exactly
            if f(central) == 0.:
                  return central
            #Otherwise we proceed
            #with bisection
            elif f(central)*f(a) < 0:
                  b = central
            else:
                  a = central
            #check again the convergence
            difference_from_zero =
                  min(math.fabs(f(a)),math.fabs(f(b)))
      #Return the closest point to zero
      if math.fabs(f(a)) > math.fabs(f(b)):
            return b
      else:
            return a

if __name__ == "__main__":
      def f(x):
            return x**2-3.*(x**3)+5.*x-1.

      print(bisection(f,1,2))
```

Exercise 10.1 Does the bisection method work if there are an even number of zeros in the interval $[a,b]$? And if the numbers of zeros is odd? ∎

10.1.2 Scipy root finders

The library **scipy** has many root finder already implemented for function of one or many variables, such as bisection and Newton method. The bisection algorithm converges always to a root of f, but it has a linear convergence to the desired accuracy. The Newton method is based on a linear first order approximation of the f around a point x_0,

$$f(x) = f(x_0) + (x - x_0) \left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_{x=x_0} = f(x_0) + (x - x_0) f'(x_0)\,, \tag{10.2}$$

used to find an approximation of the root from the crossing of a straight line with the x-axis

$$f(x_0) + (x - x_0) f'(x_0) = 0\,. \tag{10.3}$$

After solving the above equation for x

$$x = x_0 - \frac{f(x_0)}{f'(x_0)}\,, \tag{10.4}$$

we get a new guess point $x_0 \equiv x$ for the position of the root of f. The Newton algorithm converges quadratically to the root, faster than the bisection method. It requires only one starting point and the knowledge of the first derivative, however it is not guaranteed to converge, as shown in the example below.

```
import scipy.optimize

scipy.optimize.newton(
      #The function, which is zero for x=0
      lambda x: x*math.exp(-x**2),
      #The starting point
      x0=0.65,
      #The first derivative
      fprime=lambda x: math.exp(-x**2)
            - 2*(x**2)*math.exp(-x**2))
```

Exercise 10.2 Implement your own function for finding the root of a function using the Newton method. Apply the method to the function and the starting point of the example above for 20 steps and plot the linear approximation of Eq. 10.2 together with f for each iteration. Do you understand why the Newton method fails to converge? ■

Exercise 10.3 Considering the example above, find the interval around the origin where the Newton method converges to the zero $x = 0$. ■

10.2 Finding the minimum of a function

Finding the minimum of a function is a quite challenging task, at least as complicated as finding the roots. In addition, it is frequently impossible to ensure that the

minimum that has been found is the *global* minimum, as many iterative algorithm converges only to the nearest local minimum. Many algorithms for finding the minimum of a function use the first derivative, or the gradient, for moving toward the minimum of a function of a single, or many variables, as a hiker that tries to go down step by step as fast as he can to reach the valley from the top of the mountain in a foggy day. In fact, a minimum, by definition, is a point where the first derivative or the gradient of the function vanishes.

10.2.1 Newton-CG

The Newton method can be extended to locate the minima of a function, if we consider a second order approximation of the form

$$f(x) = f(x_0) + (x - x_0) f'(x_0) + \frac{1}{2}(x - x_0)^2 f''(x_0)\,, \tag{10.5}$$

describing effectively a parabola in the two-dimensional Cartesian plane. An extreme point of the approximation above is reached when the first derivative is zero, i.e. when

$$f'(x_0) + (x - x_0) f''(x_0) = 0\,. \tag{10.6}$$

Solving in terms of x,

$$x = x_0 - \frac{f'(x_0)}{f''(x_0)}\,. \tag{10.7}$$

we find a better approximation for the extreme point $x_0 \equiv x$ that can be used to iteratively find a local maximum or minimum of a function. Notice that the algorithm can converge to a minimum if $f''(x_0) > 0$, otherwise it will move toward a maximum. If the second derivative is not known, different algorithms can be employed to approximate it from the actual behavior of the function itself.

```
import scipy.optimize

result = scipy.optimize.minimize(
       #The function to minimize
       lambda x: x*math.exp(-x*x),
       #A starting guess point
       -0.3,
       #The jacobian, or first derivative
       jac=lambda x: math.exp(-x*x)-2*(x**2)*math.exp(-x*x),
       #The second derivative is approximated by the
       #algorithm, or it can be specified as
       #hess=lambda x:
       #  -8*x*math.exp(-x*x) + 4*(x**3)*math.exp(-x*x)
       method="Newton-CG")

print("The minimum is reached for x=", result.x,
       "with f=", result.fun)
```

10.2.2 Simulated annealing

There are also stochastic algorithms aiming to find the global minimum of a function, where the domain of the function is randomly sampled in search of a better minimum.

```
import scipy.optimize

result = scipy.optimize.dual_annealing(
        #The function to minimize
        lambda x: x*math.exp(-x*x),
        #A 1D window to locate the minimum
        ((-10,10),))

print("The minimum is reached for x=", result.x,
        "with f=", result.fun)
```

The algorithm is called "annealing" as it simulates the temperature treatment used in metallurgy. In the beginning, the temperature is high and the algorithm proceeds with strong random fluctuation to explore the domain of f, then, as the temperature lowers, the algorithm starts to fluctuate only around the minimum.

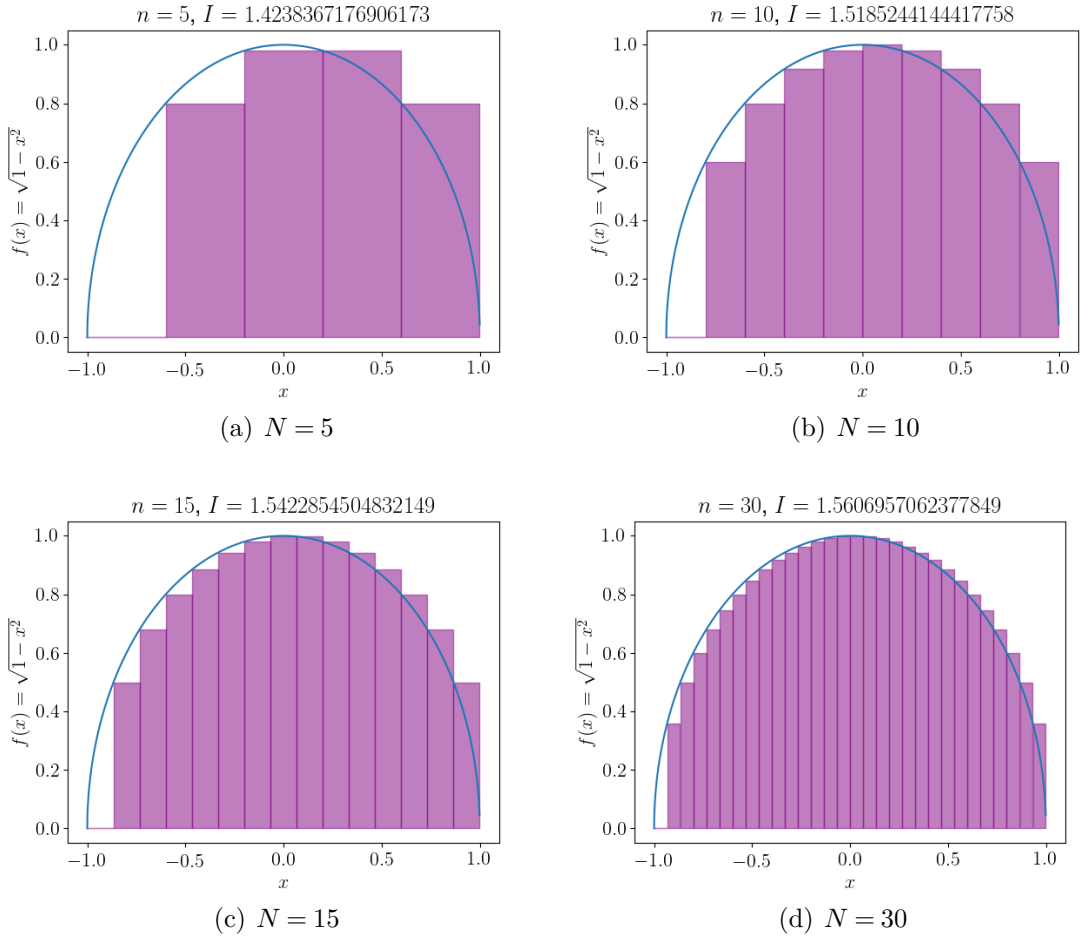
10.3 Integrals

A definite integral of a function $f(x)$ between a and b is defined as the limit $N \to \infty$ of the Riemann sum

$$\int_a^b f(x)\,\mathrm{d}x = \lim_{N\to\infty} \sum_{i=0}^{N} \frac{b-a}{N} f\left(a + i\frac{b-a}{N}\right). \tag{10.8}$$

The geometrical meaning of the summands in the limit above is the signed area of a rectangle of base $\frac{b-a}{N}$ and height $f\left(a+i\frac{b-a}{N}\right)$. For instance, the integral of $\sqrt{1-x^2}$ between -1 and 1 converges to $\pi/2$, the area of a semicircle of unit radius. The script below creates an animated plot demonstrating the convergence of the limit of the Riemann sum.

```
import numpy

#Uncomment to plot directly to file
#import matplotlib as mpl
#mpl.use('Agg')

import matplotlib.pyplot as plt
import math

import matplotlib.animation as animation

#Preambles required to have a Latex style fonts
from matplotlib import rc
```



(a) $N = 5$ (b) $N = 10$

(c) $N = 15$ (d) $N = 30$

Figure 10.1: Convergence of the Riemann sum to the integral $\int_{-1}^{1} \sqrt{1-x^2}\,\mathrm{d}x$.

```
rc('font',**{'family':'sans-serif',
      'sans-serif':['Helvetica'],'size':16})
rc('text', usetex=True)

xp = numpy.arange(-1,1,0.001)
yp = np.sqrt(1-xp*xp)

#Take the figure and the subplots
fig, ax = plt.subplots()

#function to compute the integral
def integrate(n):
      if n == 0:
            return 0
      integral = 0.
      epsilon = 2./n
      for i in range(n):
            xi = -1.+2.*i/n
```

```
            integral = (integral +
                  np.sqrt(1.-xi*xi)*epsilon)
      return integral

def update(n):
      #we need to modify completely,
      #so we clear the original plots
      global ax

      fig.delaxes(ax)

      ax = fig.add_subplot(1, 1, 1)

      #Plot the function
      ax.plot(xp, yp)

      for i in range(n):
            #Rectangle approximation of the
            #integrated function
            xap = numpy.arange(-1.+i*2./n,
                  -1.+(i+1)*2./n, 0.001)
            yap = [np.sqrt(1-(-1.+i*2./n)*(-1.+i*2./n))
                  for x in range(len(xap))]

            #Fill the space between the function
            #and the x-axes
            ax.fill_between(xap, 0, yap,
                  color='purple', alpha=0.5)


      #Y axes label
      ax.yaxis.set_label_text("$f(x) = \sqrt{1-x^2}$")
      #X axes label
      ax.xaxis.set_label_text("$x$")

      #Set the title
      ax.set_title('$n='+str(n)+'$, $I='+
            str(integrate(n))+'$')

#Center the plots
plt.subplots_adjust(wspace=0.45,hspace=0.45,
      left=0.15,top=0.9,bottom=0.15)

ani = animation.FuncAnimation(fig, update, interval=500,
      repeat_delay=2000, frames=50)
```

```
plt.show()

#save the plot as a GIF using the library
#imagemagick
ani.save('integrate.gif', dpi=300, writer='imagemagick')
```

In the example above, we clear the figure and we plot again everything from scratch, as the structure of the plot changes for each iteration. Some frame of the animated plot is shown in Fig. 10.1.

The Riemann sum converges as $O(1/N)$ to the limit $N \to \infty$, a faster convergence can be achieved if trapezoids or higher order curves are used in place of rectangles when approximating the whole area represented by an integral. The module **scipy.integrate** has a function called **trapz** and **simps** implementing the trapezoidal and Simpson's rule.

```
import scipy.integrate
import numpy as np

#We divide the integration intervals
#in 2000 subintervals
x = np.arange(-1,1,0.001)
fx = np.sqrt(1-x**2)
print("pi/2 is approximately", scipy.integrate.trapz(fx,x))
print("pi/2 is approximately", scipy.integrate.simps(fx,x))
```

10.4 Solution of an ordinary differential equation

Differential equations are basic ingredient of physics, chemistry, biology and mathematics. The solution of a first order differential equation requires to find the unknown function $y(t)$ that fulfills

$$\frac{\mathrm{d}y(t)}{\mathrm{d}t} = f(y(t))\,. \tag{10.9}$$

As typical example, differential equations describe a dynamical system, whose state variable y depends on the time t. The differential equation is complemented by an initial condition of the form

$$y(t=0) = y_0\,. \tag{10.10}$$

In general, differential equations cannot be solved analytically, and only a numerical approximation of the exact solution can be computed.

To better clarify the concept, let us start from the very simple differential equation with $f(y(t)) = -\alpha y(t)$, i.e.

$$\frac{\mathrm{d}y(t)}{\mathrm{d}t} = -\alpha y(t)\,. \tag{10.11}$$

being α a constant. Let the initial condition be $y(0) = K$. The solution of this differential equation is

$$y(t) = K\exp(-\alpha t)\,, \tag{10.12}$$

as it can be easily proven by substitution. If α is positive, then $y(t)$ decreases exponentially. The process described is followed by many nuclear and chemical reactions, and $y(t)$ is the number of atoms or chemical molecules that has not yet decayed or reacted. If α is negative, then $y(t)$ increases exponentially. In economics, under a constant inflation, the prices increases every year at a constant rate and follow an exponential law.

Exercise 10.4 Consider for instance $\alpha = -0.05$, $\alpha = -0.10$, and $\alpha = -0.15$, with $K = 100$, for which t does y double? ∎

A simple numerical solution of a first order ordinary differential equation can be constructed by iteration if the first derivative is approximated to first order (Euler scheme) as

$$\frac{\mathrm{d}\,y(t)}{\mathrm{d}\,t} \simeq \frac{y(t+\epsilon)-y(t)}{\epsilon}\,. \tag{10.13}$$

Inserting the expression above into the differential equation

$$\frac{y(t+\epsilon)-y(t)}{\epsilon} = f(y(t))\,, \tag{10.14}$$

we solve for $y(t+\epsilon)$

$$y(t+\epsilon) = y(t)+\epsilon f(y(t))\,. \tag{10.15}$$

The above equation is an iterative relation that can be used to compute $y(t)$ starting from $y(0)$

$$y_{n+1} = y_n + \epsilon f(y_n)\,, \tag{10.16}$$

where we have defined $y_n \equiv y(n\epsilon)$. The parameter ϵ is an integration step-size which can be tuned to be small until the desired accuracy is reached.

Exercise 10.5 The Euler scheme provides a simple numerical approximation of the solution of a differential equation. For instance the sequence defined from

$$y_{n+1} = (1-\alpha\epsilon)y_n \tag{10.17}$$

describes a percentage reduction of $100\times(\alpha\epsilon)\%$ at each iteration (for instance a reduction of the number of radioactive atoms due to decays). The sequence goes in the limit $\epsilon \to 0$ to $y(t) = K\exp(-\alpha t)$, the solution of the differential equation

$$\frac{d}{dt}y(t) = -\alpha y(t)\,. \tag{10.18}$$

Plot the difference between the exact and the numerical solution for $\epsilon = 2.$, $1.$, and 0.5, from $t = 0$ to $t = 10$ for $\alpha = 0.15$. The staring point is $y_0 = 100$, i.e. $K = 100$.

What happens if the sequence would be defined as

$$y_{n+1} = \left(1-\alpha\epsilon+\frac{\alpha^2\epsilon^2}{2}\right)y_n\,? \tag{10.19}$$

Plot the again the same differences to the function $y(t) = K\exp(-\alpha t)$. ∎

Typical ordinary differential equations describing physical systems are not of first order, but of second order. The Newton's second law of mechanics

$$\vec{F} = m\vec{a}, \tag{10.20}$$

connects the force to the acceleration, that is the second derivative of the position with respect to time. However, in terms of the velocity, the second law can be rewritten as a system of first-order differential equations

$$\frac{\mathrm{d}\vec{v}}{\mathrm{d}t} = \frac{1}{m}\vec{F} \tag{10.21}$$

$$\frac{\mathrm{d}\vec{x}}{\mathrm{d}t} = \vec{v}. \tag{10.22}$$

The most fundamental physical system is the harmonic oscillator, describing the oscillations of a mass attached to a spring. Consider a one-dimensional problem, a body of mass $m = 1$ attached to a spring located at $x = 0$, the force is proportional to the distance from the origin

$$F = -kx\,. \tag{10.23}$$

The corresponding system of differential equations is

$$\frac{\mathrm{d}v}{\mathrm{d}t} = -kx \tag{10.24}$$

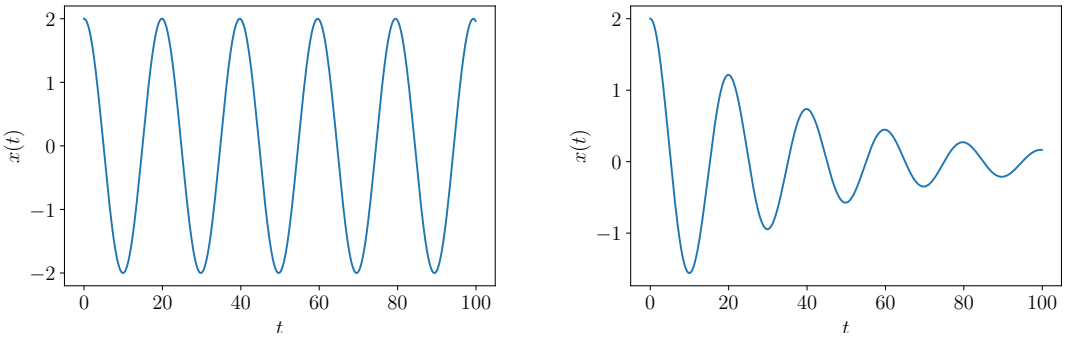
$$\frac{\mathrm{d}x}{\mathrm{d}t} = v\,, \tag{10.25}$$

that can be integrated using the module **scipy.integrate**. The velocity $v(t)$ and the position $x(t)$ are saved as a single array state y, with $y[0] = x$ and $y[1] = v$

```
import scipy.integrate
import functools
import numpy as np
import matplotlib.pyplot as plt

#The save the state
def harmonic_oscillator(t, y, kappa):
      return np.array([
            #The position evolves with the velocity
            y[1],
            #The velocity evolves with the force
            -kappa*y[0]])

#We use functools to store the constant k
f = functools.partial(harmonic_oscillator, kappa=0.1)
sol = scipy.integrate.solve_ivp(
      #The function to integrate
      f,
      #The time interval
      [0, 100],
```

```
        #The initial value, we start from x=2
        #and zero velocity
        [2,0],
        #We store the output every 0.1
        t_eval=np.arange(0,100,0.1))

#We plot the position as
#a function of the time
plt.plot(sol.t, sol.y[0])
plt.xlabel("$t$")
plt.ylabel("$x(t)$")
plt.show()
```

The resulting plot is shown in Fig. 10.2(a), as expected the position of body oscillates between the plus and minus the initial position.



(a) No friction (b) With friction

Figure 10.2: Plot of the solution of the simple (left) and damped (right) harmonic oscillator.

Exercise 10.6 Starting from the example above, consider the damped harmonic oscillator with an external periodic force, i.e.

$$F = -kx - cv + F_0 \cos(\omega t)\,. \tag{10.26}$$

If $F_0 = 0$, the friction term, proportional to minus the velocity, has the effect to slow down the motion of the body. The oscillation becomes smaller and smaller, as shown in Fig. 10.2(b) for $c = 0.1$.

A more interesting situation occurs when the external forced oscillations are turned on. Solve the corresponding dynamical system for $F_0 = 0.1$, $\omega = 0.1$, 0.2, 0.3, 0.316, 0.4, 0.5 and 0.6. Try to follow the behavior of the oscillations for with and without friction, $c = 0.1$ and $c = 0$, respectively. Do you notice a difference? ■

10.5 Curve fitting

Differential equations describe the deterministic evolution of a system starting from a given initial condition. The input is the model, i.e. the differential equation, and the output is the evolution of the state variables as a function of time. However, frequently the problem might be reversed. What we want to understand is the model that explains *at best* the experimentally observed behavior of a variable as a function of time, a problem known as fitting.

For instance, as input we consider the Gross Domestic Product of USA in the last forty years from the World Bank Database, that has been already plotted in Sec. 9.3. The problem to solve is understanding what is the function that describes at best the growth of the USA GDP of the last forty years. The two simplest *ansatz* could be a linear growth

$$y(t;K,\alpha) = \alpha t + K\,, \tag{10.27}$$

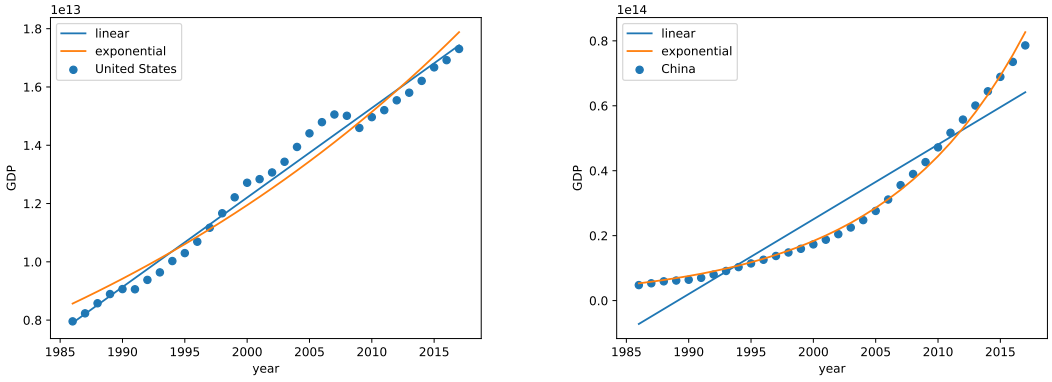
or an exponential growth

$$y(t;K,\alpha) = K\exp(\alpha t)\,. \tag{10.28}$$

In both cases, we want to fit the two unknown parameters K and α. To this end, we search the minimum of the function $\chi(K,\alpha)$, that is the sum of the squared difference from $y(t)$ to the observed data

$$\chi(K,\alpha) = \sum_t (y(t;\alpha,K) - \mathrm{GDP}_{\mathrm{USA}}(t))^2\,. \tag{10.29}$$

In some cases, the function $y(t;\alpha,K)$ will be smaller than the observed data $\mathrm{GDP}_{\mathrm{USA}}(t)$, and the difference $y(t;\alpha,K) - \mathrm{GDP}_{\mathrm{USA}}(t)$ is negative; in some other cases larger, and the difference $y(t;\alpha,K) - \mathrm{GDP}_{\mathrm{USA}}(t)$ is positive. The square $(y(t;\alpha,K) - \mathrm{GDP}_{\mathrm{USA}}(t))^2$ will always be positive and provides a measure that we can minimize of how far the function $y(t;\alpha,K)$ is from our data.



(a) USA (b) China

Figure 10.3: Plot of the linear and exponential fit of the GDP in local constant current of China and United States.

The function **curve_fit** from the module **scipy.optimize** provides a routine to fit the GDP data of United States, that requires a starting guess point for the fitting

parameters, other than the data and the function to fit. For comparison we also fit the GDP of China, using the function of Sec. 9.3.

```
...


from scipy.optimize import curve_fit

def linear(t,alpha,K):
      return alpha*(t-1985) + K

def exponential(t, alpha, K):
      #We normalize the time,
      #so that we avoid fitting large
      #numbers in the exponent
      return K*np.exp((t-1985)*alpha)


#for each country
for country in "United States","China":
      handles = []

      gdp = getLCUGDP(country,1985)

      #For each fitting function
      for f in linear, exponential:
            #Fit the function
            #The starting point for K can be the first entry
            #of the series.
            #For alpha we need a better guess
            alpha_starting = \
                  {linear: 1e5, exponential: 0.01}[f]
            popt, pcov = curve_fit(f, gdp[:,0], gdp[:,1],
                  p0=[alpha_starting,gdp[0,1]],
                  #We also set a large
                  #maximal number of iterations
                  maxfev = 30000)
            print("Fitting parameters for", country,
                  "and fitting function", f.__name__, *popt)

            #Plot the fitted function
            p, = plt.plot(gdp[:,0],f(gdp[:,0],*popt),
                  label=f.__name__)
            handles.append(p)
      p = plt.scatter(gdp[:,0],gdp[:,1],label = country)
      handles.append(p)
```

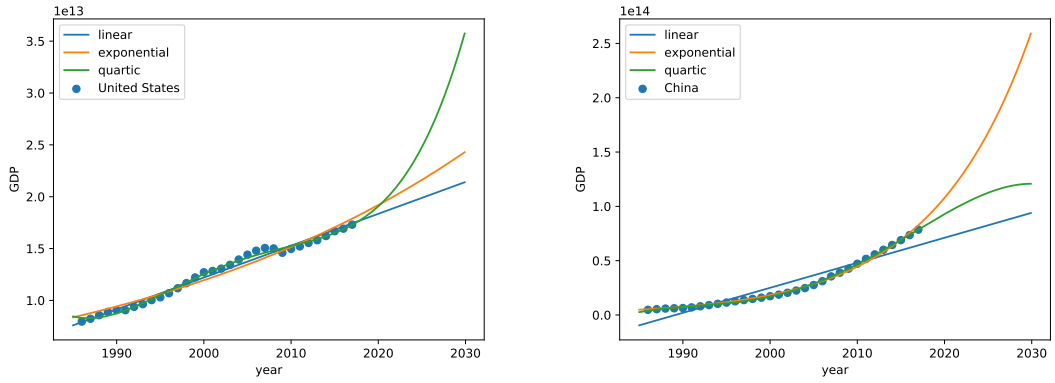
```
plt.xlabel("year")
plt.ylabel("GDP")
plt.legend(handles=handles)
plt.show()
```

The output of the script above is shown in Fig. 10.3. In the case of China, the exponential function seems to describe the growth of the GDP better, with a yearly percentage increase of approximately 8%, while for United States both functions seems to work fine.

The coefficient of determination is a measure of the goodness of the fit that can help to judge analytically the quality of a fit. The coefficient of determination of the best fit function f, also called R-coefficient

$$R = 1 - \frac{\sum_n (y_n - f(t_n))^2}{\sum_n (y_n - \langle y \rangle)^2}, \tag{10.30}$$

is exactly equal to one if $f(t)$ interpolates perfectly the data. Otherwise R compares the magnitude of the square of the differences of y_n from the fit to the standard deviation of y_n itself ($\langle y \rangle$ is the mean value of y_n). Smaller and smaller values of R indicates bad fit. If R is equal to zero, the fitting function has deviations from our data that are as large as the standard deviation. It means that even a simple constant function $f(t) = \langle y \rangle$ would give the same error difference, i.e. f has not been able to capture the dynamics of our data. For non-linear fits, R might be negative.



(a) USA (b) China

Figure 10.4: Plot of the linear, quartic and exponential fit of the GDP in local constant current of China and United States, together with an extrapolation-prediction up to 2030.

There are an infinite number of possible fitting functions, however models with few parameters are frequently preferred. Complicated functions with many parameters, such as higher order polynomials, tend to over-fit the data and are poorly able to extrapolate the trend in regions where data are missing. The problem is demonstrated in Fig. 10.4, where we compare a linear, exponential and quartic

$$y(t; K, \alpha, \beta, \gamma, \delta) = \alpha t + K + \beta t^2 + \gamma t^3 + \delta t^4 \tag{10.31}$$

functions. The quartic fit form is able to capture better the fluctuations of the GDP of United States during the subprime crisis. However, the quadratic form predicts almost a doubling of the output of the United States economy and a significant slowing down of the growth of China in the next ten years. Of course, nothing is impossible, but such scenario seems quite unlikely. Over-fitting is one of the reason why complex macro-economical models have proven to be more unreliable than the 10-days weather forecasts.

Exercise 10.7 Using the function **scipy.optimize.curve_fit** and data from the World Bank, fit the population from 1960 to today of Germany, France, Italy, USA, Angola and China to the functions:

- $K\exp(\alpha t)$: the fitting parameters are α and K
- $at+b$: the fitting parameters are a and b

A fit should be performed separately for each country. Plot the fitting functions together with the population data. (A for-loop could be the fastest solution.)

Which of the two functions does fit the population data better? Compute the coefficient of determination as a measure of the goodness of the fit. ■

11. Statistics

In this chapter we introduce the basic ingredients for the statistical analysis of datasets, namely the mean, the median, and the standard deviation.

11.1 Mean and median of set of data

The mean value, denoted as $\langle X \rangle$ or μ, of a set of data $\{X_1, X_2, \ldots, X_N\}$ is defined as

$$\mu = \frac{1}{N} \sum_n X_n \tag{11.1}$$

The median, to do not be confused with the mean, is the middle value X dividing the greater and lesser halves of a data set. If the dataset is sorted, then the median is equal either to $X_{(N+1)/2}$, for N odd, or to $(X_{N/2-1} + X_{N/2+1})/2$, for N even.

The median, contrary to the mean value, is less sensitive to outliers in the dataset. The difference can be clearly seen in the following example, where we use the functions **mean** and **median** of the module **numpy**.

```
import numpy as np

dataset_A = [3,4,5,6,7]
dataset_B = [3,4,5,6,37]

print("Mean and median for dataset A:",
      np.mean(dataset_A),# mean is equal to 5
      np.median(dataset_A))# median is equal to 5
print("Mean and median for dataset B:",
      np.mean(dataset_B),# mean is equal to 11
      np.median(dataset_B))# median is equal to 5
```

The functions **mean** and **median** are also present in the module **statistics**.

Depending on the context, the median might be more informative than the mean. The function **percentile** is the generalization of the median, returning the value of X dividing a dataset in p and $(1-p)$ percent. By definition, the 50 percentile is equal to the median.

```
import numpy as np

dataset_A = [3,4,5,6,7]

print("50\% percentile", np.percentile(dataset_A,50))# 5
print("75\% percentile", np.percentile(dataset_A,75))# 6
```

11.1.1 Wages of men and women in California

As concrete example for understanding median and mean, we will consider the Public Use Microdata Sample (PUMS) of the American community survey (ACS) to estimate the difference in the wages earned by women and men in California. The dataset is publicly available and each PUMS entry encodes the response to the survey of an individual.

The dataset is issued in the **csv** format, storing basically a very large table, which can be loaded using the module **csv**. First, we need to retrieve and unpack the dataset from internet.

```
import urllib
import os
import urllib.request
import zipfile

#Url to download the dataset
url_pop = "https://www2.census.gov/programs-surveys/acs/"
      "data/pums/2017/1-Year/csv_pca.zip"
#Zip file
zip_file = "csv_pca.zip"

#If the zipfile is not found
if not os.path.isfile(zip_file):
      #Download it from the internet
      urllib.request.urlretrieve(url_pop, zip_file)
      #Unzip the file
      zip_ref = zipfile.ZipFile(zip_file, 'r')
      zip_ref.extractall("./")
      zip_ref.close()
```

As the survey does not sample the whole population, weights are associated to each individual record, expressing the number of people an individual entry represents [15]. Weights are required in order to recover the correct statistics for the number of people with a given age/race/etc [16]. Consider for instance the class where ten

students are 18 years old, two 19 years old and seven 17 years old. We can compute the mean age either by considering the raw list

```
np.mean([17,17,17,17,17,17,17,
      18,18,18,18,18,18,18,18,18,18,19,19])
```

or by observing that the mean age is effectively given by

$$\frac{1}{19}\sum_{i=0}^{19}\text{age}_i = \frac{1}{19}(17\times 7+18\times 10+19\times 2) = \frac{1}{19}\sum_{\text{age}=17}^{20}\text{num\_students(age)}\times\text{age}\,. \tag{11.2}$$

Therefore, to compute the mean value, we include in formula 11.1 the weights in the sum as

$$\text{Mean wage} = \frac{1}{W}\sum_{i=0}^{N} w_i \text{wage}_i \tag{11.3}$$

where the normalization N is replaced by the sum of all weights W

$$W = \sum_{i=0}^{N} w_i\,. \tag{11.4}$$

We need to run two sums to compute the weighted mean, the first over the weights and the second over the weights times the wages.

```
...

import csv

wages_female = 0
total_female = 0

wages_male = 0
total_male = 0

with open("psam_p06.csv") as f:
      #Load the data
      reader = csv.DictReader(f, delimiter=',')
      #For each row
      for row in reader:
            #We first need to check if the individual
            #has a wage or not
            wage = row["WAGP"]
            if wage != "" and int(wage) != 0:
                  #If so, we separate the data by sex
                  #The weight is stored in
                  #the field "PWGTP"
                  weight = int(row["PWGTP"])
                  wage = int(wage)
                  if row["SEX"] == "1":
```

```
                        wages_male += weight*wage
                        total_male += weight
                  else:
                        wages_female += weight*wage
                        total_female += weight

# approx 63000 dollars
print("Men average wage", wages_male/total_male)
# approx 45000 dollars
print("Female average wage", wages_female/total_female)
```

To compute the median, we need to store all individual wages and weights to a list. We must be careful when considering weights, that are counting the number of people represented by an individual. Consider again the class where ten students are 18 years old, two 19 years old and seven 17 years old. We can compute the median age either by considering the raw list

```
np.median([17,17,17,17,17,17,17,
      18,18,18,18,18,18,18,18,18,18,19,19])
```

or by observing that the median age is reached when the sum of the number of students with a given age is equal or above the total number of students (in the example above $7+10 > 19/2$, therefore the median age is 18). The median of the sorted dataset is defined therefore as the entry for which the sum of all previous weights is equal to the half of the sum of all weights.

```
#We use a list to store all the weight and wages
wages_female = []
wages_male = []

with open("psam_p06.csv") as f:
      reader = csv.DictReader(f, delimiter=',')
      for row in reader:
            if row["WAGP"] != "" and int(row["WAGP"]) != 0:
                  if row["SEX"] == "1":
                        wages_male.append(
                              #First the wage
                              [int(row["WAGP"]),
                              #Then the weight
                              int(row["PWGTP"])])
                  else:
                        wages_female.append(
                              #First the wage
                              [int(row["WAGP"]),
                              #Then the weight
                              int(row["PWGTP"])])
      #At the end we convert the output
      #to a numpy array
```

```
        wages_male = np.array(wages_male)
        wages_female = np.array(wages_female)

def compute_median(data_with_weights):
        #The median sum
        median_sum = np.sum(data_with_weights[:,1])/2

        #We try to see for which element
        #we reach the median sum of the weights
        mm = 0

        #We run the loop on the sorted list
        #by the data to compute the median
        for data, weight in sorted(data_with_weights,
                key=lambda x: x[0]):
                if mm + weight > median_sum:
                        #If we are above the median
                        #we return the corresponding
                        #data
                        return data
                else:
                        mm += weight

# approx 40000 dollars
print("Men median wage", compute_median(wages_male))
# approx 30000 dollars
print("Female median wage", compute_median(wages_female))
```

As you can see, the median wages are lower than the mean wages, due to the presence of large outliers in the dataset. We can directly see the difference if we plot the wage distributions.

```
...

import matplotlib.pyplot as plt
#Preambles required to have a Latex style fonts
from matplotlib import rc
rc('font',**{'family':'serif',
        'serif':['Computer Modern'],'size':22})
rc('text', usetex=True)

#Histograms with weights
plt.hist(wages_male[:,0],weights=wages_male[:,1],
        #We use a transparency (alpha=0.5) to show
        #overlapping bars
        label="Male wages", alpha=0.5, bins=30)
plt.hist(wages_female[:,0],weights=wages_female[:,1],
```

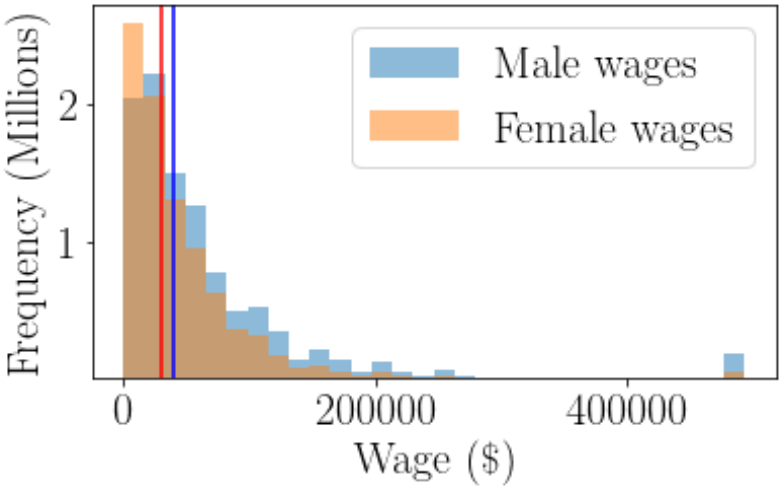


Figure 11.1: Distribution of the wages in California for male and females. The vertical blue and red lines represent the median wage for male and female, respectively. The distribution is quite asymmetrical, with more than four millions workers in the first two bins for each class, and the mean wages are affected by outliers.

```
      label="Female wages", alpha=0.5, bins=30)
plt.axvline(x=compute_median(wages_male), color="blue")
plt.axvline(x=compute_median(wages_female), color="red")
plt.xlabel("Wage (\$)")
plt.ylabel("Distribution (Millions)")
plt.yticks([1000000,2000000],[1,2])

plt.legend()
plt.tight_layout()
plt.show()
```

The output of the above script is shown in Fig. 11.1.

Exercise 11.1 Starting from the example above, compute the median income for males and females in California as a function of the age, considering employees from 20 up to 70 years old. Fit the male and female median income to a linear function $f(t) = a + bt$ and to a function $f(t) = a + b/t$ and discuss which function is fitting better the data. ∎

11.2 Standard deviation

The standard deviation, denoted as σ, is defined as

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{n=1}^{N}(x_n - \mu)^2}\,. \tag{11.5}$$

The standard deviation is effectively a measure of the spread of the distribution of the elements x_n of the dataset, a larger σ meaning a larger spread of the x_n around the mean value μ. The standard deviation is zero if and only if all x_n are equal to μ. For large N, the square of the standard deviation, called variance, is effectively equal to the mean value of the the square of elements of the dataset minus their mean value. Therefore we have

$$\sigma^2 \simeq \langle (X - \langle X \rangle)^2 \rangle , \tag{11.6}$$

where $(X - \langle X \rangle)^2$ is the dataset $\{(X_1 - \langle X \rangle)^2, (X_2 - \langle X \rangle)^2, \ldots, (X_N - \langle X \rangle)^2\}$.

The standard deviation can be computed using the function **std** of the module **numpy**.

```
import numpy as np

dataset_A = [3,4,5,6,7]
dataset_B = [3,4,5,6,37]

print(np.std(dataset_A))
print(np.std(dataset_B))
```

In the example above, the standard deviation is larger for the dataset B, due to the larger spread of the data around the mean value.

11.2.1 Working hours per week in California

Starting from the example of the previous sections, we now compute the mean value and the standard deviation of the number of working hours per week in California. In order to take into account the weight, as done in the previous section for the mean, we consider the weighted standard deviation

$$\sigma = \sqrt{\frac{1}{W} \sum_{n=0}^{N} w_n (x_n - \mu)^2} \tag{11.7}$$

where μ is defined as the weighted mean accordingly to Eq. 11.3 with wages replaced by working hours.

After loading the PUMS data, we can collect the working hours per week stored in the field "WKHP".

```
...

#We use a list to store all the weight and wages
working_hours = []

with open("psam_p06.csv") as f:
      reader = csv.DictReader(f, delimiter=',')
      for row in reader:
            #Take the working hours per week
            #if they are there
```

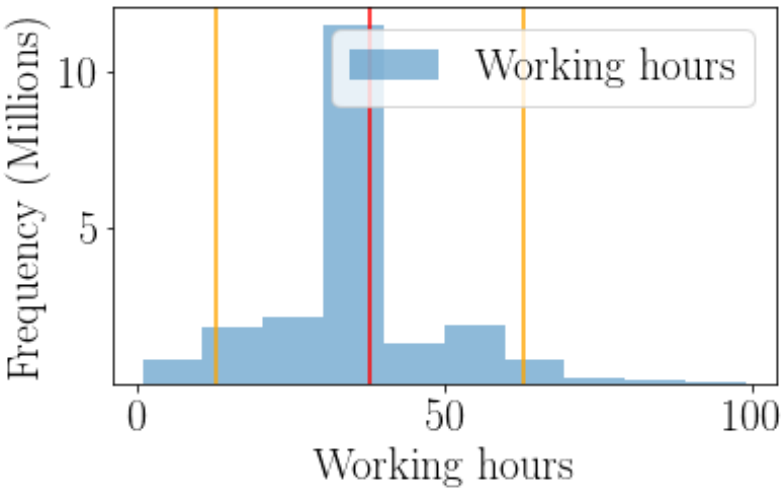


Figure 11.2: Distribution of the working hours per week in California. The vertical red line is the mean value, while the two orange lines represent the interval $\mu \pm 2\sigma$.

```
            if row["WKHP"] != "":
                  working_hours.append(
                        [int(row["WKHP"]),
                        int(row["PWGTP"])])
      working_hours = np.array(working_hours)

norm = np.sum(working_hours[:,1])
mean = np.sum(working_hours[:,0]*working_hours[:,1])/norm
std = np.sqrt(np.sum(working_hours[:,1]*
      (working_hours[:,0]-mean)**2)/norm)
#Approximately 38 hours
print("Mean working hours per week:", mean)
#Approximately 13 hours
print("Standard deviation:", std)
```

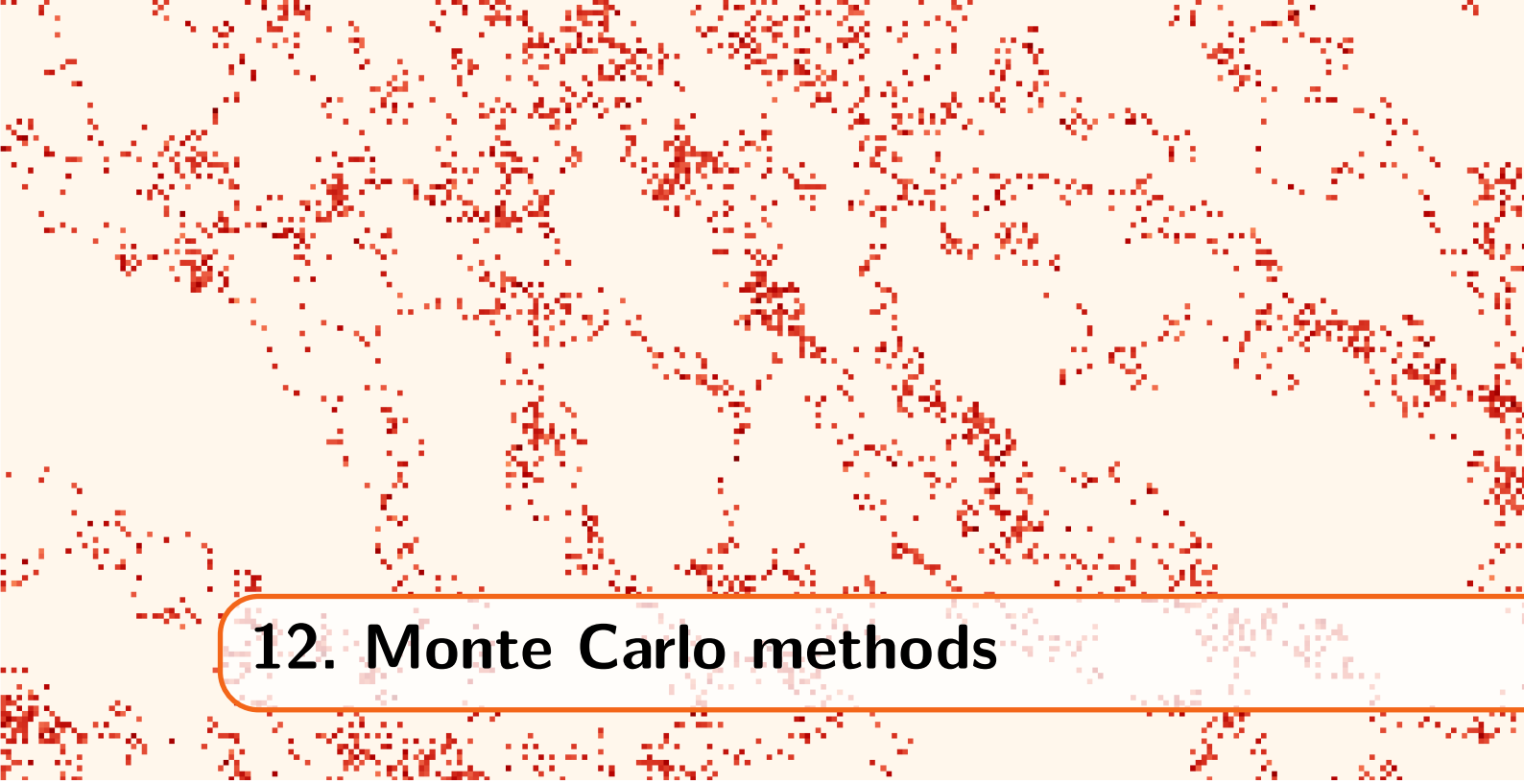
The Chebyshev's inequality states that at least a fraction equal to $1-\frac{1}{k^2}$ of the entire dataset is included between $\mu - k\sigma$ and $\mu + k\sigma$ [17], for $k > 1$. For $k = 2$, the Chebyshev's inequality states that at least 75% of the entries in the dataset is larger than $\mu - 2\sigma$ and smaller than $\mu + 2\sigma$. In our example, at least 75% of the employee in California works between 12 and 64 hours. In the script below we check the Chebyshev's inequality.

```
...

#Approximately 93 percent > 75 percent
print(np.sum([weight for hours, weight in working_hours
      if hours > mean-2*std and hours < mean+2*std])/norm)
```

```
plt.hist(working_hours[:,0],weights=working_hours[:,1],
      label="Working hours", alpha=0.5, bins=10)
plt.ylabel("Frequency")
plt.axvline(x=mean, color="red")
plt.axvline(x=mean-2*std, color="orange")
plt.axvline(x=mean+2*std, color="orange")
plt.xlabel("Working hours")
plt.ylabel("Frequency (Millions)")
plt.yticks([10000000,5000000],[10,5])

plt.legend()
plt.tight_layout()
plt.show()
```

The output of the script is shown in Fig. 11.2.

Exercise 11.2 Starting from the example above, estimate the mean, the median and the standard deviation of the age (field "AGEP") of the people living in California. ∎



12. Monte Carlo methods

> Instead of choosing configurations randomly, then weighting them with exp(-E/kT), we choose configurations with a probability exp(-E/kT) and weight them evenly.
>
> Nicholas Metropolis *et al.* [18]

The name "Monte Carlo methods" stands for a wide class of algorithms using random numbers and random sampling to compute for instance integrals or simulate stochastic and quantum models. In this chapter we will learn the basic Monte Carlo algorithms and we will use Monte Carlo methods to simulate the evolution of the price of a stock or a commodity.

12.1 Random numbers and integrals

Random numbers can be used to compute integrals. Considering Fig. 12.1, we are interested in the integral of

$$f(x) = x\exp(-x^2) + 1 \tag{12.1}$$

between -1 and 1. If we generate random points inside the square box $-1 < x < 1$ and $0 < y < 2$ (the blue plus the red area), the probability P_{in} that a point is in the red area is equal to

$$P_{\text{in}} = \frac{A_{\text{red}}}{A_{\text{blue}} + A_{\text{red}}} = \frac{1}{4}\int_{-1}^{1} (x\exp(-x^2) + 1)\,\mathrm{d}\,x\,, \tag{12.2}$$

since $A_{\text{blue}} + A_{\text{red}} = 2 \times 2 = 4$ and A_{red} is equal to the integral by definition. An approximation of the integral can be therefore computed by generating N coordinates

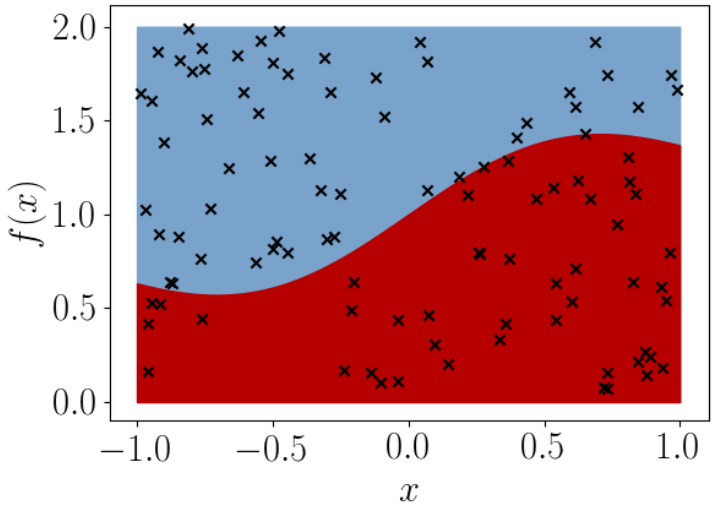


Figure 12.1: The integral of $f(x)$ between -1 and 1 is equal to four times the ratio of the red area over the blue plus the red area. The same integral is approximately equal to the ratio of the number of random points inside the red area over the total number of points.

(x, y), where both x and y are random uniform numbers between 0 and 2. The integral reads

$$\int_{-1}^{1} f(x)\mathrm{d}x = 4\frac{N_{\text{in}}}{N}, \tag{12.3}$$

where N_{in} is the number of points that occurred to be in the red area (i.e. the points (x, y) such that $y \leq f(x)$).

The script below computes a Monte Carlo approximation of the integral 12.3 using $N = 100$, 1000, 10000 and 100000 points.

```
import numpy as np

#The function to integrate
def f(x):
      return np.exp(-x**2)*x + 1

for NN in 100000,10000,1000,100:
      inside = 0
      #The x coordinates of random points
      random_points_x = np.random.uniform(-1,1,size=NN)
      #The y coordinates of random points
      random_points_y = np.random.uniform(0,2,size=NN)

      #f(x)
      f_x = f(random_points_x)

      #Count the numbers of points inside
```

```
        #The condition is random_points_y < f_x
        #False is converted to zero in the sum
        #True to one
        number_of_points_inside = \
              np.sum(random_points_y < f_x)

        print("The integral is", 4*number_of_points_inside/NN,
              " vs 2 for N=", NN, "random points")

import matplotlib.pyplot as plt
#Preambles required to have a Latex style fonts
from matplotlib import rc
rc('font',**{'family':'sans-serif',
      'sans-serif':['Helvetica'],'size':22})
rc('text', usetex=True)

x = np.arange(-1,1,0.001)

#Plot the blue area
plt.fill_between(x, 0, 2*np.ones(shape=len(x)),
      color='#7AA3CC')
#Plot the red area on top
plt.fill_between(x, 0, f(x), color='#b60000')
#Plot the random points
plt.scatter(random_points_x, random_points_y,
      marker="x",c="black")
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
plt.tight_layout()
plt.show()
```

In the script above we use the features of **numpy** explained in Sec. 9.1 to simplify the calculation, avoiding to write explicitly many loops.

Exercise 12.1 The integral

$$\int_{-1}^{1} \sqrt{1-x^2}\,\mathrm{d}x = \frac{\pi}{2} \tag{12.4}$$

can be approximated by Monte Carlo integration generating random points in the box $(-1,1)\times(0,1)$. Compute an approximation of π using 10000000 random points. How does the Monte Carlo approximation converge to π? Plot the error of the approximation of π using only the first 10000, 20000, 30000, 40000, and so on, randomly generated points. ∎

The convergence of Monte Carlo integration is asymptotically $O(1/\sqrt{N})$, which is rather slow compared to standard non-stochastic algorithms, based on piecewise approximations of the function to integrate seen in Sec. 12.1.

Exercise 12.2 Compare the integral approximation of the previous exercise to the "trapezoidal rule" by preparing the same convergence plot using the same number N of evaluations of the function $\sqrt{1-x^2}$. ∎

The most important feature of the Monte Carlo integration is that the convergence is independent from the number of dimensions of the integral. In contrast, for a scalar d-dimensional function, the "trapezoidal rule" would require to approximate the integral as the volume of hyper-trapezoids equally spaced on a grid of dimension N^d to keep the order of the error constant. Already for $d \approx 10$ and $N = 100$, the trapezoidal estimate of the integral would require the prohibitively expensive evaluation of the function f for 100^{10} points, while Monte Carlo integration could get a reasonable estimate already with only $n \approx 1000$ points.

12.2 Metropolis algorithm

The Metropolis algorithm has been introduced in 1953 to study the equation of state of a classical gas of rigid spheres [18]. Later, many important applications of the same idea have been found in spin system, classical and quantum fluid mechanics, statistical field theory and quantum mechanics.

The Metropolis algorithm is able to sample a generic probability distribution $p(x)$, i.e. it generates a sequence of random variables $\{x_0, x_1, \ldots, x_n\}$ distributed accordingly to a given probability function $p(x)$. Generating random numbers sampled accordingly to a distribution is a common problem in many fields even outside physics, such as economics and mathematics. In economics, a simulation of the random behavior of the price of a stock might require sampling from distributions with complex shapes.

More on ... 12.2.1 — Probability functions. A probability function can be any function as long as it is always positive

$$p(x) \geq 0 \qquad \forall x \in \mathcal{D} \tag{12.5}$$

and its integral over the domain of definition $\mathcal{D}$ is one

$$\int_{\mathcal{D}} f(x)\,\mathrm{d}x = 1\,. \tag{12.6}$$

If the latter condition is not fulfilled, and if the above integral is not infinite, $p(x)$ can be normalized by dividing it by the constant $\mathcal{N} = \int_{\mathcal{D}} f(x)\,\mathrm{d}x$.

Classical example of probability function are the Gaussian bell function over all the real axis

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{x^2}{2\sigma^2}\right), \tag{12.7}$$

or the uniform constant function over $x \in [0,1]$

$$p(x) = \begin{cases} 1 & 0 \leq x \leq 1 \\ 0 & \text{otherwise} \end{cases}. \tag{12.8}$$

The meaning of a probability function is that the probability $P(a,b)$ for a number sampled from $p(x)$ to be in the interval $[a,b]$ is

$$P(a,b) = \int_a^b p(x)\,\mathrm{d}x\,. \tag{12.9}$$

The sequence of random variables $\{x_0, x_1, \ldots, x_n\}$ is constructed iteratively starting from an initial x_0, chosen randomly, that can also be always the same as long as $p(x_0) \neq 0$. At each iteration, a new element is appended to the sequence with a probability that depends only on last element x_n of $\{x_0, x_1, \ldots, x_n\}$. Each iteration requires two steps:

- **Random move**: A random move δ of the last entry x_n is chosen with uniform random probability in the interval $(-\epsilon, \epsilon)$, with ϵ a tunable parameter.
- **Accept/Reject**: The point $x_n + \delta$ is accepted or rejected if a random uniform number in the interval between zero and one is smaller than $p(x+\delta)/p(x_0)$. If accepted, then append to the list $x_n + \delta$, otherwise x_n again.

The above steps are repeated a large number of times and ϵ is tuned such that the acceptance rate is reasonable, not too below 40% or above 90%. The outcome of the Metropolis algorithm is a **Markow chain**.

Roughly speaking, a random move is accepted always if $x_n + \delta$ goes in regions where the probability is higher with respect to x_n. If not, the acceptance of the random move is bound to the ratio $p(x+\delta)/p(x_0)$, which is smaller than one by definition. As $p(x+\delta)$ gets smaller and smaller, i.e. if $x_n + \delta$ goes in regions where the probability is small, there are few chances that a random number between zero and one is smaller than $p(x+\delta)/p(x_0)$, and the random move is most likely rejected. The Metropolis algorithm is able to sample a distribution $p(x)$ correctly in this way.

We can use the Metropolis algorithm to sample 10000 random numbers accordingly to the distribution

$$p(x) = \frac{1}{\mathcal{N}} \exp\left(+5x^2 - x^4\right). \tag{12.10}$$

Note that the normalization factor $\mathcal{N}$ is irrelevant for the Metropolis algorithm, as only ratios of probabilities are required to accept or reject a move. The parameter ϵ is tuned to $\epsilon = 0.5$ so that the acceptance rate is approximately 70%. For generality, we write the algorithm so that we can sample a multivariate distribution of many random variables.

```
import random
import math
import numpy as np

#Accept/Reject rule
def metropolis(ratio):
      if ratio >= 1.:
            return True
      elif ratio > random.uniform(0,1):
            return True
      else:
            return False
```

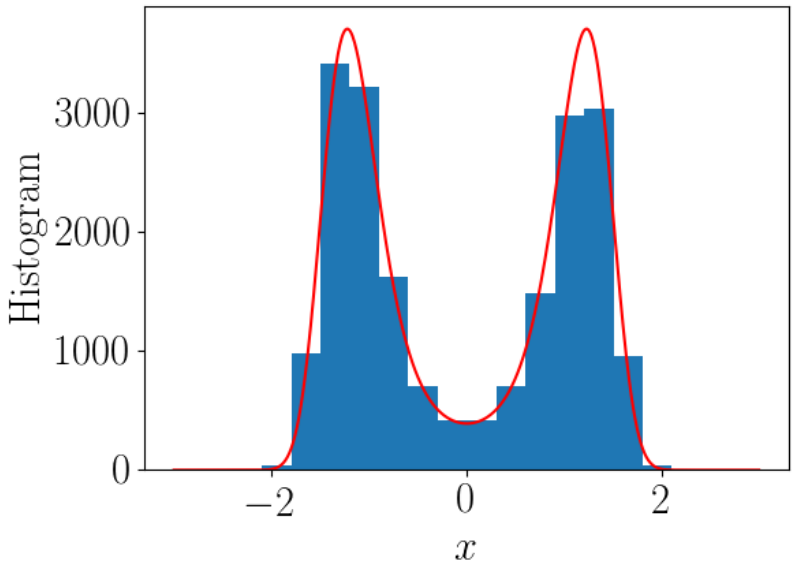


Figure 12.2: Sampled distribution using the Metropolis algorithm of $p(x) \propto \exp(+3x^2 - x^4)$ against the theoretical expectation. The histogram shows the number of sampled random numbers for each bin of size 0.3.

```
#Random move step
def random_move(x, epsilon = 1.):
      return x + np.random.uniform(-epsilon,epsilon,
            size=len(x))

def markov_chain(
      #The probability function
      p,
      #The number of iteration steps
      steps,
      #The starting point
      x0,
      #The epsilon of the random move
      epsilon = 1.):
      #The Markov chain
      chain = [x0]
      accepted = 0
      for i in range(steps):
            trial = random_move(chain[-1], epsilon)
            if metropolis(p(trial)/p(chain[-1])):
                  accepted += 1
                  chain.append(trial)
            else:
```

```
                chain.append(chain[-1])
        print("Acceptance rate=", accepted/steps)
        return np.array(chain)

#The probability function
def p(x):
        return np.exp(np.sum(+3*x**2 - x**4))


random_sampling = markov_chain(p,100100,np.zeros(1),0.5)

import matplotlib.pyplot as plt
#Preambles required to have a Latex style fonts
from matplotlib import rc
rc('font',**{'family':'sans-serif',
        'sans-serif':['Helvetica'],'size':22})
rc('text', usetex=True)

#Plot the histogram of the Markov chain
plt.hist(random_sampling[100::5,0],bins=np.arange(-3,3,0.3))
plt.ylabel("$\\textrm{Histogram}$")
plt.xlabel("$x$")

#Now we plot the expected distribution
#Which is p(x) times the number of samples
#times the bin size (=0.3)
#We also need to determine the normalization
#constant
import scipy.integrate
x = np.arange(-3,3,0.01)
N=scipy.integrate.trapz(np.exp(3*x**2-x**4),x)

plt.plot(x,0.3*(1/N)*(100000/5)*np.exp(3*x**2-x**4),
        color="red")

plt.tight_layout()
plt.show()
```

Note that in a Markov chain every new entry is generated from the previous one. The sampled random numbers are therefore correlated. Assuming an acceptance rate of 80%, there is a 20% chance that the next number in the chain is equal to the previous one, a 0.04% probability for the second next, a 0.8% to the third next, and so on. On the other hand, too large acceptance rates could mean that the random move does not produce a new different and uncorrelated random number. To ensure a more uncorrelated sampling, in the example above we have plotted the histogram of the generated chain skipping every fifth entries. We have also excluded the first hundred elements as thermalization, since the starting point was chosen to

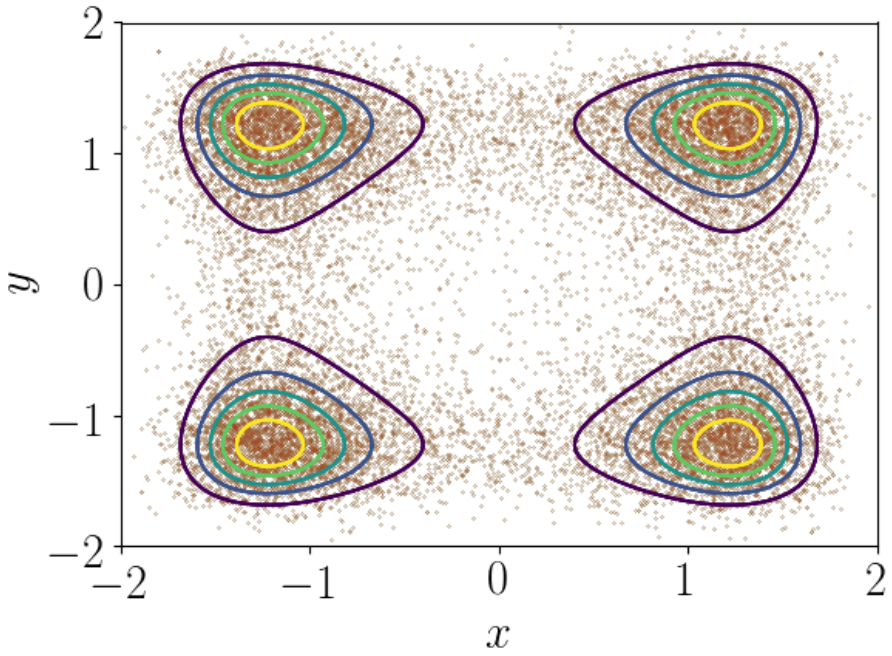


Figure 12.3: Sampled distribution using the Metropolis algorithm of $p(x,y) \propto \exp\{+3(x^2+y^2)-(x^4+y^4)\}$ against the theoretical expectation. The randomly generated points are more concentrated in the regions inside the contours.

be zero and not distributed accordingly to $p(x)$. The output of the above script is in Fig. 12.2.

The above script is written in such a way that we can try to test the sampling of a distribution in two dimensions.

```
...

#Two dimensional sampling
#Only the starting point must be changed
random_sampling = markov_chain(p,100000,np.zeros(2),0.5)

#First we prepare a scatter plot of our data
plt.scatter(random_sampling[100::5,0],
      random_sampling[100::5,1],
      s=0.01)

#Then we prepare a contour plot
#of the expected distribution
x = np.arange(-2.0, 2.0, 0.01)
y = np.arange(-2.0, 2.0, 0.01)

X, Y = np.meshgrid(x, y)
```

```
Z = np.exp(3*(X**2 + Y**2) - (X**4 + Y**4))

plt.contour(X, Y, Z)

plt.xlabel("$x$")
plt.ylabel("$y$")
plt.show()
```

The output of the two dimensional sampling is shown in Fig. 12.3.

12.3 The central limit theorem

The central limit theorem provides an alternative easy method to Metropolis to generate random Gaussian numbers distributed as

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{x^2}{2\sigma^2}\right), \tag{12.11}$$

starting from random numbers which can be distributed in various ways, as long as they have a finite mean and standard deviation.

More on ... 12.3.1 — Mean and standard deviation. The mean μ of a probability distribution $p(x)$ is defined as the integral

$$\mu = \int_{\mathcal{D}} x p(x) \, \mathrm{d}x \,, \tag{12.12}$$

while the variation is

$$\sigma^2 = \int_{\mathcal{D}} (x-\mu)^2 p(x) \, \mathrm{d}x \,. \tag{12.13}$$

The standard deviation σ is the square root of the variation. The above definitions are effectively the generalization to the continuum case of the mean and the standard deviation of a dataset considered in Ch. 11.

The central limit theorem states that the mean S_N, from a sequence of N independent and identically distributed random variables X_n sampled from a distribution with mean μ and variation σ^2, deviates from μ as Gaussian random variable with mean zero and variation σ^2/N in the limit $N \to \infty$.

We can see the central limit theorem at work by choosing a simple random variable which is $+1$ 50% of the times and zero the other 50%. The mean value μ of such a distribution is

$$\mu = 0 \times \frac{1}{2} + 1 \times \frac{1}{2} = 0.5 \,, \tag{12.14}$$

while the variation is

$$\sigma^2 = (0-0.5)^2 \times \frac{1}{2} + (1-0.5)^2 \times \frac{1}{2} = 0.25 \,. \tag{12.15}$$

We therefore expect the mean of N of such random variables to be distributed accordingly to a Gaussian distribution with mean zero and standard deviation

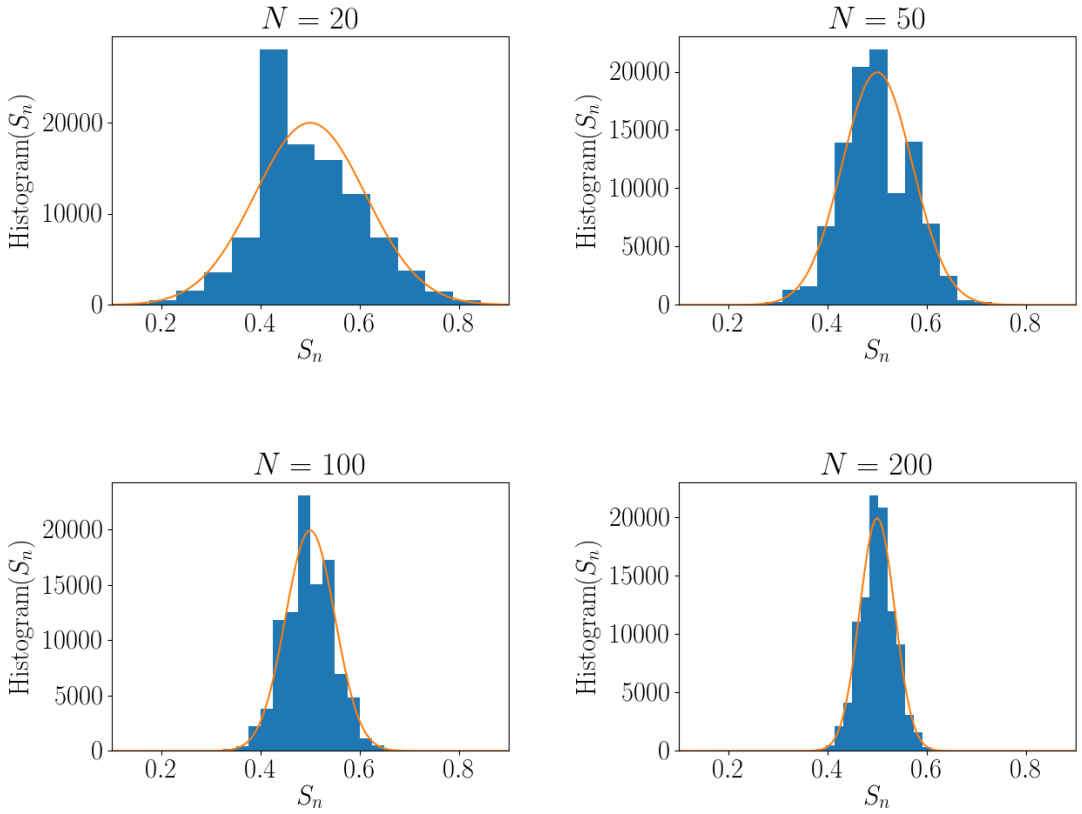


Figure 12.4: Demonstration of the central limit theorem for the distribution of the mean of N random variables that are either zero or one with equal probability. The standard deviation narrows for N larger, and the distribution is better approximated by a Gaussian.

$\sqrt{0.25/N}$ in the large N limit. We need to generate a large number of sequences in order to study the distribution of their mean values.

```
import matplotlib.pyplot as plt
#Preambles required to have a Latex style fonts
from matplotlib import rc
rc('font',**{'family':'sans-serif',
      'sans-serif':['Helvetica'],'size':22})
rc('text', usetex=True)

import numpy as np

#for each sequence length
for N in 20,50,100,200:
      #generate 100000 sequences
      means = np.array(
            #Consider their mean
            [np.mean(
```

```
                np.random.choice([0,1],size=N)
                ) for M in range(100000)])

    #Theoretical sigma
    sigma = np.sqrt(0.25/N)

    print("Mean of S_n", np.mean(means), "versus", 0.5)
    print("Std of S_n", np.std(means), "versus",
          sigma)

    #plot the histogram of the means
    binsize = sigma/2
    plt.hist(means,bins=np.arange(-1,3,binsize))

    #Plot against the Gaussian distribution
    x = np.arange(-1,3,0.005)
    norm_factor = binsize*(100000/
          (np.sqrt(2*np.pi)*sigma))
    plt.plot(x,norm_factor*np.exp(-(x-0.5)**2/
          (2*sigma**2)))

    plt.xlabel("$S_n$")
    plt.ylabel("$\\textrm{Histogram}(S_n)$")
    plt.xlim(0.1,0.9)
    plt.title("$N={}$".format(N))
    plt.tight_layout()
    plt.show()
```

The normalization factor of the Gaussian distribution has to be multiplied by the number of samples and the bin-size in order to compare the theoretical expectation to the raw probability histogram plotted by matplotlib. The output of the script is shown in Fig. 12.4.

As discussed in the introduction of this section, the central limit theorem effectively describes a simple algorithm to sample a Gaussian distribution, by simply averaging over a set of random variables with finite variance and mean. The target variance and mean of the wanted Gaussian can be adjusted by a translation and a rescaling.

Exercise 12.3 Repeat the script above and consider the central limit theorem for a random uniform variable between -1 and 1, and a random variable which can be 5, 7, 11, 19 or 43 with equal probability. Compare the sampled distribution of the mean values of generated random sequences to the theoretical expectation. For which N are you able to reproduce a Gaussian distribution? ■

12.4 Brownian motion

The Brownian motion describes the effective random path followed by particles of dust in the air. In two dimensions, the particle moves at each iteration in a random

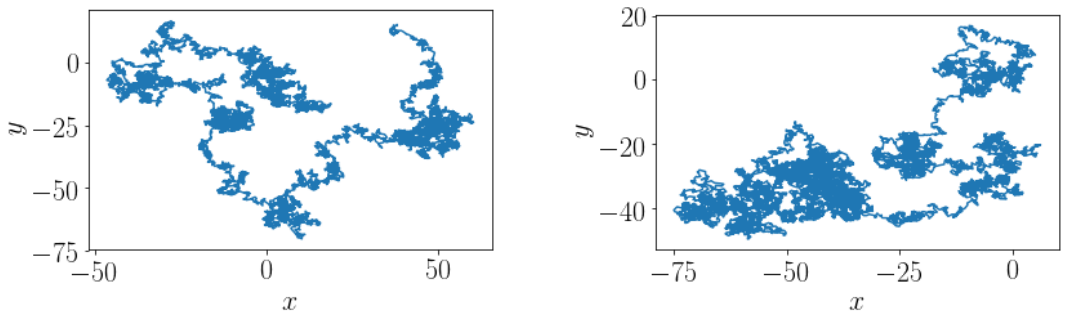


Figure 12.5: Two examples of Brownian motion in two dimensions.

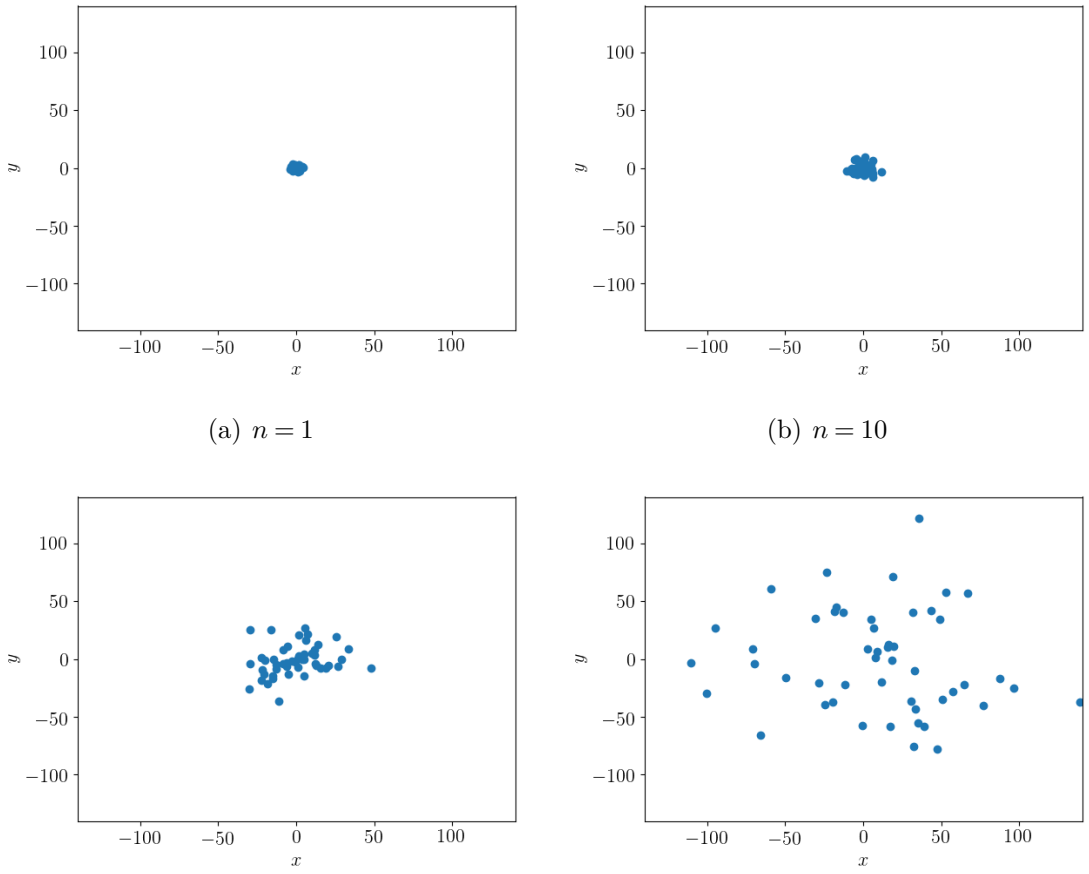
direction by a step

$$x_{n+1} = x_n + g_x(0,\sigma), \tag{12.16}$$
$$y_{n+1} = y_n + g_y(0,\sigma), \tag{12.17}$$

where $g_x(0,\sigma)$ and $g_y(0,\sigma)$ are two independent Gaussian random numbers with mean zero and standard deviation σ. The motion described by the above equation is stochastic, and, given an initial condition, it does not describe a single deterministic path, but rather an entire ensemble of possible trajectories. For this reason, the Brownian motion is also known as random walk.

Simulating a Brownian motion using numpy is relatively straightforward.

```
import matplotlib.pyplot as plt
#Preambles required to have a Latex style fonts
from matplotlib import rc
rc('font',**{'family':'sans-serif',
      'sans-serif':['Helvetica'],'size':22})
rc('text', usetex=True)

import numpy as np

#The Gaussian random increments
#10000 iterations in two dimensions
increments = np.random.normal(0,0.5,size=(10000,2))

#The starting point
x0 = np.array([0,0])

#The Brownian motion
path = x0 + np.cumsum(increments,axis=0)

plt.plot(path[:,0],path[:,1])
plt.xlabel("$x$")
plt.ylabel("$y$")
```



(a) $n = 1$ (b) $n = 10$

(c) $n = 100$ (d) $n = 1000$

Figure 12.6: Evolution of the Brownian motion of fifty particles in two dimension after n iterations.

```
plt.show()
```

The **np.cumsum** function have been used to compute the array of cumulative sum of the random variables generated, implementing effectively the motion of Eq. 12.16. The output of the above script is presented in Fig. 12.5.

12.5 Diffusion processes

The Brownian motion of many particles describes effectively a diffusion process, where the particles are slowly spreading in all directions. Starting from the last script of the previous section, we show below an animation of the Brownian motion of 50 particles.

```
...

import matplotlib.animation as animation

#Take the figure and the subplots
```

```
fig, ax = plt.subplots()

#The Gaussian random increments
#10000 iterations in two dimensions
#Fifty particles
increments = np.random.normal(0,0.5,size=(10000,50,2))

#The starting point
x0 = np.array([0,0])

#The Brownian motion
paths = x0 + np.cumsum(increments,axis=0)

#The plot limit
plot_limit = np.max(np.abs(paths))

#Initial plot
pc = ax.scatter(paths[0,:,0],paths[0,:,1])

def update(n):
      #Update the points plotted
      pc.set_offsets(paths[n])

#Center the plots
plt.xlim(-plot_limit,+plot_limit)
plt.ylim(-plot_limit,+plot_limit)
plt.xlabel("$x$")
plt.ylabel("$y$")
plt.tight_layout()

ani = animation.FuncAnimation(fig, update, interval=10,
      repeat_delay=2000, frames=10000)

plt.show()
```

Some pictures of the above animations are shown in Fig. 12.6.

The Brownian motion is stochastic, each simulation will generate likely a different path. All trajectories can be generated from a Brownian motion as described above. Although *quite unlikely*, a possible path after five iteration could describe a particle jumping from the origin and moving at the edges of a regular pentagon. The crucial aspect of the Brownian motion is that it describes an entire ensemble of trajectories, and there is a probability for each path to occur. In general, computing paths and weighting them with respect to this probability might be challenging. Monte Carlo simulations allows to sample the path space and to compute the correct average of certain quantities, like the mean value of the coordinates of the particles or the mean distance from the origin as a function of time, as shown in the script below.

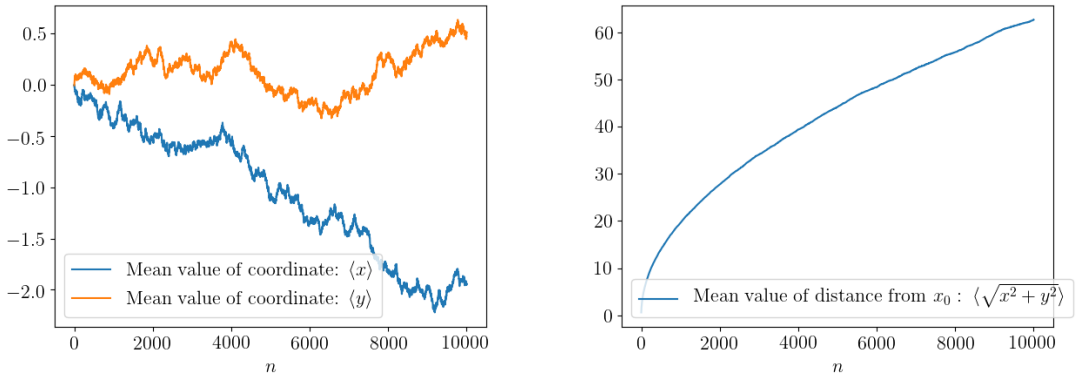
```
...

#We use 1000 particles to get a better
#convergence to the mean values
increments = np.random.normal(0,0.5,size=(10000,1000,2))

#Plot the mean value of the coordinates
#We are averaging over the particles
#stored in axis=1
p1, = plt.plot(range(len(paths)),
      np.mean(paths[:,:,0],axis=1),
      label="$\\textrm{Mean value of coordinate: }"
      " \\langle x \\rangle$")
p2, = plt.plot(range(len(paths)),
      np.mean(paths[:,:,1],axis=1),
      label="$\\textrm{Mean value of coordinate: }"
      " \\langle y \\rangle$")
plt.xlabel("$n$")
plt.legend(handles=[p1,p2])
plt.show()

#Plot the mean value of the distance
p3, = plt.plot(range(len(paths)),
      np.mean(
            np.sqrt(paths[:,:,0]**2 + paths[:,:,1]**2),
            axis=1),
      label="$\\textrm{Mean value of distance from } x_0:"
      "~\\langle \\sqrt{x^2 + y^2} \\rangle$")
plt.xlabel("$n$")
plt.legend(handles=[p3])
plt.show()
```

As the Brownian motion is basically a sum of Gaussian random variables, unsurprisingly we discover that the mean value of x and y fluctuate around zero, see Fig. 12.5. The cloud of diffusing particles does not move nor translate significantly in average in space in the time (compare the values to the final spread of the particles in Fig. 12.6). However, they spread around in the space, as their average distance increases as $\sqrt{N}$, the square root of the number of iterations, as dictated by the central limit theorem, see Fig 12.5.

Exercise 12.4 Consider the Brownian motion defined by the iteration

$$x_{n+1} = x_n + \cos(\theta), \tag{12.18}$$
$$y_{n+1} = y_n + \sin(\theta), \tag{12.19}$$

starting from the origin $x_0 = (0,0)$, where θ is a random angle, i.e. a random



(a) Coordinates (b) Distance

Figure 12.7: Left) Mean value of the coordinates of 1000 particles as a function of the iteration n. Right) Mean value of the distance from the origin.

uniform variable in the range $[0, 2\pi)$. Repeat the plots considering the average coordinates and distance from the origin for fifty particles for 10000 iterations. How does the random walk change? ■

It is possible to define a continuum version of the Brownian motion, depending on a continuum time t variable rather than on a discrete step n. To this end, we introduce a discretization step ϵ as

$$x(t+\epsilon) = x_n + \sqrt{\epsilon} g_x(0,\sigma)\,, \tag{12.20}$$
$$y(t+\epsilon) = y_n + \sqrt{\epsilon} g_y(0,\sigma)\,. \tag{12.21}$$

The square root of the discretization step might appear strange in the first instance. However, as we are dealing with particles moving on random trajectories, the idea is to construct a continuum limit so that the average properties of the random paths are constant. In order to keep constant the average properties, in particular the average distance, we extrapolate to the limit $\epsilon \to 0$ while rescaling the standard deviation σ as $\sigma \to \sigma/\sqrt{\epsilon}$, or, that is the same, multiplying the random Gaussian variable by $\sqrt{\epsilon}$.

Exercise 12.5 Consider the example shown in the scripts above, where ϵ was set to one and the number of iterations to 10000. How does the mean value of the coordinates and of the distance from the origin change if you consider $\epsilon = 0.5$ and 20000 iterations or $\epsilon = 0.25$ and 40000 iterations? ■

12.6 Log-normal random walk and stock simulations

Brownian motion and random walks can be extended to simulation of the random fluctuations which occur every day in the price of stocks or commodities. As the price of a stock cannot be negative, we cannot simply add at each iteration to the stock price S_n a random Gaussian variable which can be positive or negative, otherwise it might happen, after many iterations, that the price might become negative. We can

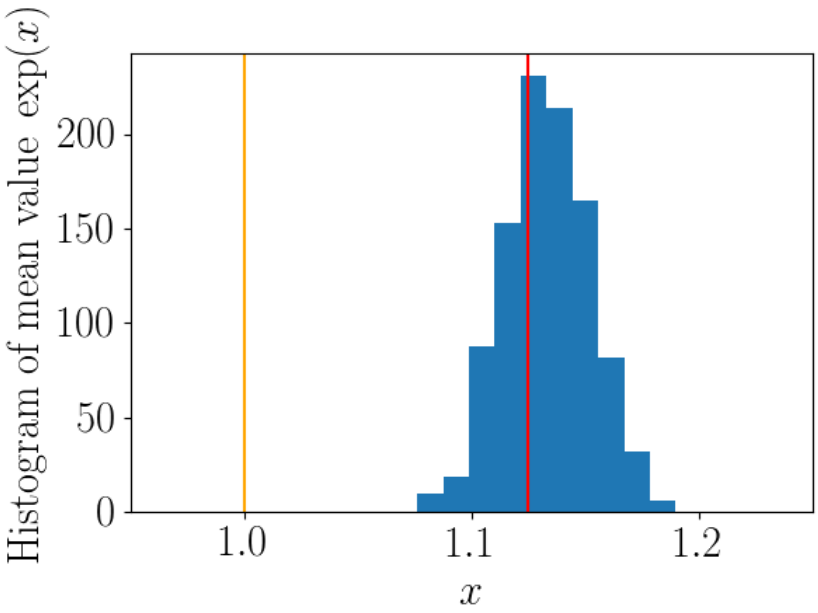


Figure 12.8: Histogram of 1000 mean values of the exponential of 1000 random variables. Compared to the naïve value $x = 1$ (orange line), $x = 1 + \sigma^2/2$ (red line) is a better estimate of the mean.

consider instead the update rule

$$S_{n+1} = \exp\left(g(\mu,\sigma)\right) S_n \,, \tag{12.22}$$

being $g(\mu,\sigma)$ a random Gaussian variable with mean μ and standard deviation σ. The exponential of a number is always positive, and $100 \times \exp\left(g(\mu,\sigma)\right)$ is effectively equal to the percentage change of the price of the stock S_n, as can be seen by dividing both members of Eq. 12.22 by S_n:

$$\frac{S_{n+1}}{S_n} = \exp\left(g(\mu,\sigma)\right). \tag{12.23}$$

Taking the logarithm of the equation above and rearranging the terms

$$\log(S_{n+1}) = \log(S_n) + g(\mu,\sigma) \,, \tag{12.24}$$

we can see that Eq. 12.22 describes a random walk for the logarithm of the price of the stock, called "log-normal random walk". The basic assumption behind this model is that the fluctuations of the price of a stock are proportional to the price of the stock itself.

The standard deviation σ, known as "volatility", is proportional to the size of the fluctuations of the price of the stock. The mean μ of the random Gaussian variable, called drift, is a tunable parameter used to ensure that prices of stocks and commodities grow with time at given rate, for instance the one given by inflation. It can be positive or negative, and not necessary negative drifts correspond to decreasing stock prices, as the random fluctuations might dominate. Indeed, if you consider

the mean value of the exponential of N random variables $g_n(0,\sigma)$ with zero mean expanded to second order, we get

$$\frac{1}{N}\sum_{n=0}^{N}\exp\left(g_n(0,\sigma)\right) \simeq \frac{1}{N}\sum_{n=0}^{N}1+\frac{1}{N}\sum_{n=0}^{N}g_n(0,\sigma)+\frac{1}{2N}\sum_{n=0}^{N}g_n(0,\sigma)^2\,. \tag{12.25}$$

The first term on the right is simply 1, the second term is equal to the mean and it is zero as μ is zero. The lest term is proportional to the variation σ^2, and it is different from zero as it is the sum of the square of many terms that can be positive or negative, but after taking the square are all positive. Therefore we have

$$\frac{1}{N}\sum_{n=0}^{N}\exp\left(g_n(0,\sigma)\right) \simeq 1+\frac{\sigma^2}{2} > 1\,. \tag{12.26}$$

The relation above is an example of the Jensen's inequality [19]. We can directly verify our calculations using numpy and considering an histogram of 1000 mean values of the exponential of 1000 random variables, see Fig. 12.8.

```
import matplotlib.pyplot as plt
#Preambles required to have a Latex style fonts
from matplotlib import rc
rc('font',**{'family':'sans-serif',
      'sans-serif':['Helvetica'],'size':22})
rc('text', usetex=True)

import numpy as np

#1000 mean values of the exponential
#of 1000 random variables
exp_gauss = np.mean(np.exp(
      np.random.normal(0.,0.5,size=(1000,1000))),axis=1)

#Histogram of the exp of a Gaussian variable
plt.hist(exp_gauss)
#Vertical line to mark the approximation of the mean value
plt.axvline(x=1+0.5**2/2,color="red")
#Vertical line to mark the naive mean value
plt.axvline(x=1,color="orange")
plt.xlabel("$x$")
plt.ylabel("$\\textrm{Histogram of mean value }\exp(x)$")
plt.xlim(0.95,1.25)
plt.tight_layout()
plt.show()
```

A simulation of a log-normal random walk is a straightforward generalization of the random walk of the previous section, but, instead of the numpy function **cumsum**, we use the function **cumprod** to compute the cumulative product required by the update rule 12.22.

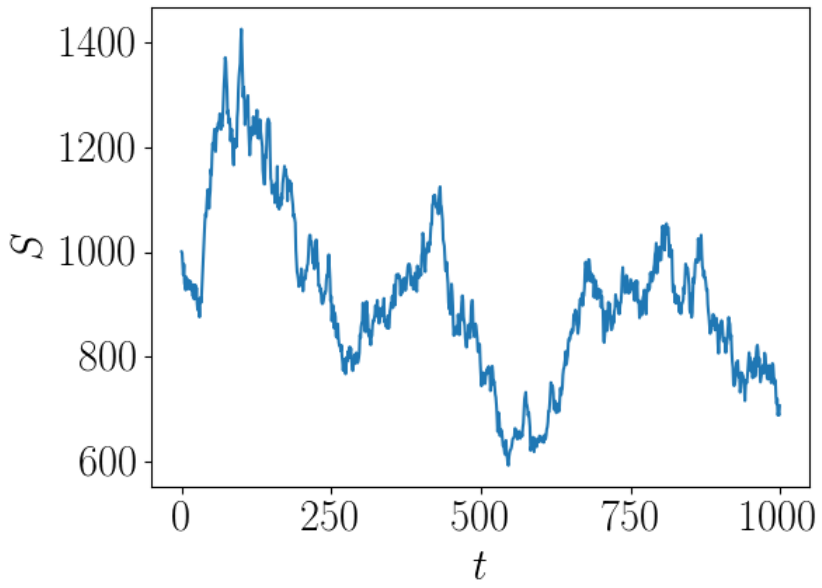


Figure 12.9: Log-normal random walk after 1000 iterations starting from $S_0 = 1000$.

```
import matplotlib.pyplot as plt
#Preambles required to have a Latex style fonts
from matplotlib import rc
rc('font',**{'family':'sans-serif',
      'sans-serif':['Helvetica'],'size':22})
rc('text', usetex=True)

import numpy as np

#The Gaussian random increments
increments = np.exp(np.random.normal(0.,0.02,size=1000))

#The starting point
x0 = 1000

#The log-normal Brownian motion
path = x0*np.cumprod(increments)

plt.plot(range(1000),path)
plt.xlabel("$t$")
plt.ylabel("$S$")
plt.tight_layout()
plt.show()
```

A possible output of the above script is shown in Fig. 12.9.

Exercise 12.6 Consider 1000 simulations of the log-normal random walk starting from $S_0 = 1000$ with $\sigma = 0.02$. Tune the value of μ such that in average, after 1000 steps, the value of S is equal to 1000, i.e. $S_0 = S_{1000}$. ∎

Log-normal random walk are a useful model for the evolution of the price of stock and commodities. However, many empirical studies observed that rare events, where the price jumps in one day significantly above or below 3σ occurs more frequently than predicted by log-normal random walks. Alternative models are based on the Cauchy distribution

$$C(x; x_0, \gamma) = \frac{1}{\pi\gamma} \frac{1}{1 + \left(\frac{x - x_0}{\gamma}\right)^2}, \tag{12.27}$$

that has "fat tails", i.e. $C(x; x_0, \gamma)$ decays for large x with a power law, instead of exponentially as for a Gaussian distribution. Large fluctuations in the logarithm of the price are therefore not suppressed and can occur with quite large probability.

More on ... 12.6.1 — Cauchy distribution. The Cauchy distribution is used to model physical systems which have no intrinsic scale. In fact, the Cauchy distribution has infinite mean and variance, as the integrals

$$\int_{-\infty}^{\infty} x^n C(x; x_0, \gamma)\, \mathrm{d}x \tag{12.28}$$

diverge for $x \geq 1$. As a consequence, the central limit theorem fails to hold for the Cauchy distribution. The Cauchy distribution is an example of Levy flights distributions [20].

In order to perform a stock simulation with a Cauchy distribution, we need to normalize the random variables generated so that the change in the logarithm of S is not too drastic, see Fig. 12.10.

```
...

#The Cauchy increments
increments = np.exp(
      np.random.standard_cauchy(size=1000)/10000)

#The starting point
x0 = 1000

#The log-cauchy Brownian motion
path = x0*np.cumprod(increments)

plt.plot(range(1000),path)
plt.xlabel("$t$")
plt.ylabel("$S$")
plt.tight_layout()
plt.show()
```

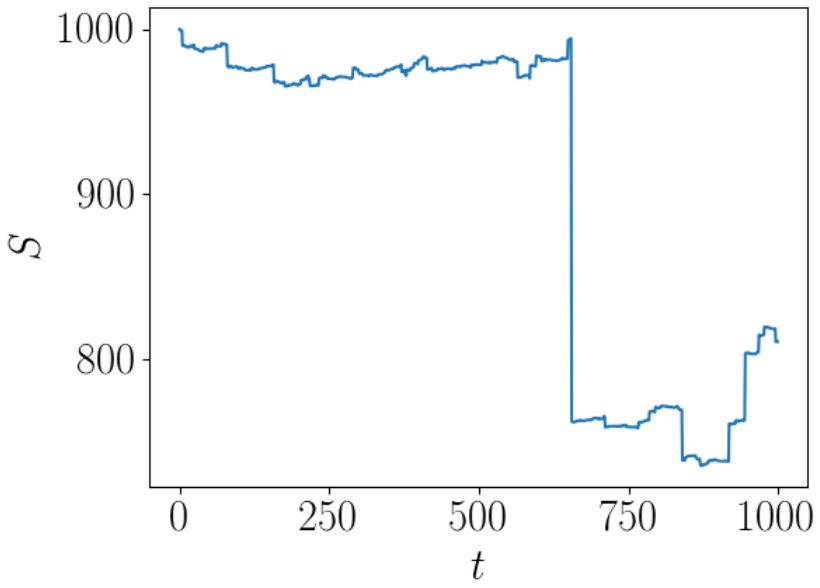


Figure 12.10: Random walk after 1000 iterations starting from $S_0 = 1000$ where the logarithm of S changes accordingly to a Cauchy distributed random variables. Compared to Fig. 12.9, large jumps in S are clearly visible.

Exercise 12.7 Plot a simulation of the random walk in two-dimensions given by the update

$$x_{n+1} = x_n + c_x\,, \tag{12.29}$$
$$y_{n+1} = y_n + c_y\,, \tag{12.30}$$

where c_x and c_y are two independent random standard Cauchy variables. Start from $x_0 = 0$ and $y_0 = 0$. How does the plot differ from Fig. 12.5? ■

The absence of mean and standard deviation of the Cauchy distribution introduces jumps in the price which might be too frequent and too strong. An alternative is to consider a truncated Cauchy distribution, for instance by an hard cut K, proportional to

$$CT(x,K) \propto \begin{cases} 1/(1+x^2) & \text{if } |x| < K \\ 0 & \text{otherwise} \end{cases} \tag{12.31}$$

The cut K has the effect to introduce a maximal possible value when sampling from a Cauchy distribution. We use the Metropolis algorithm and the function defined in Section 12.2 to sample the distribution $CT(x,K)$.

```
def p(x):
      if np.sum(x**2) < 25:
            return 1/(1+np.sum(x**2))
      else:
            return 0
```

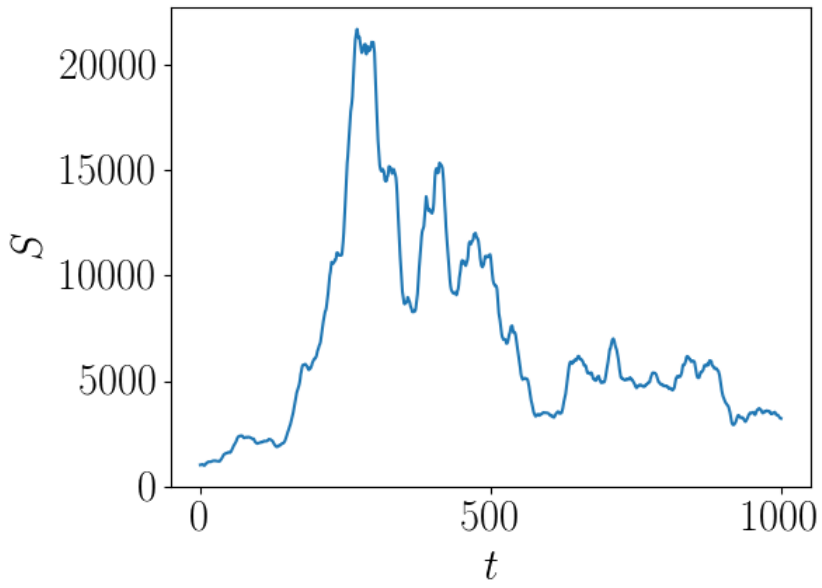


Figure 12.11: Random walk after 1000 iterations starting from $S_0 = 1000$ where the logarithm of S changes accordingly to a truncated Cauchy distributed random variables. Compared to Fig. 12.10, the jumps in S are smoother.

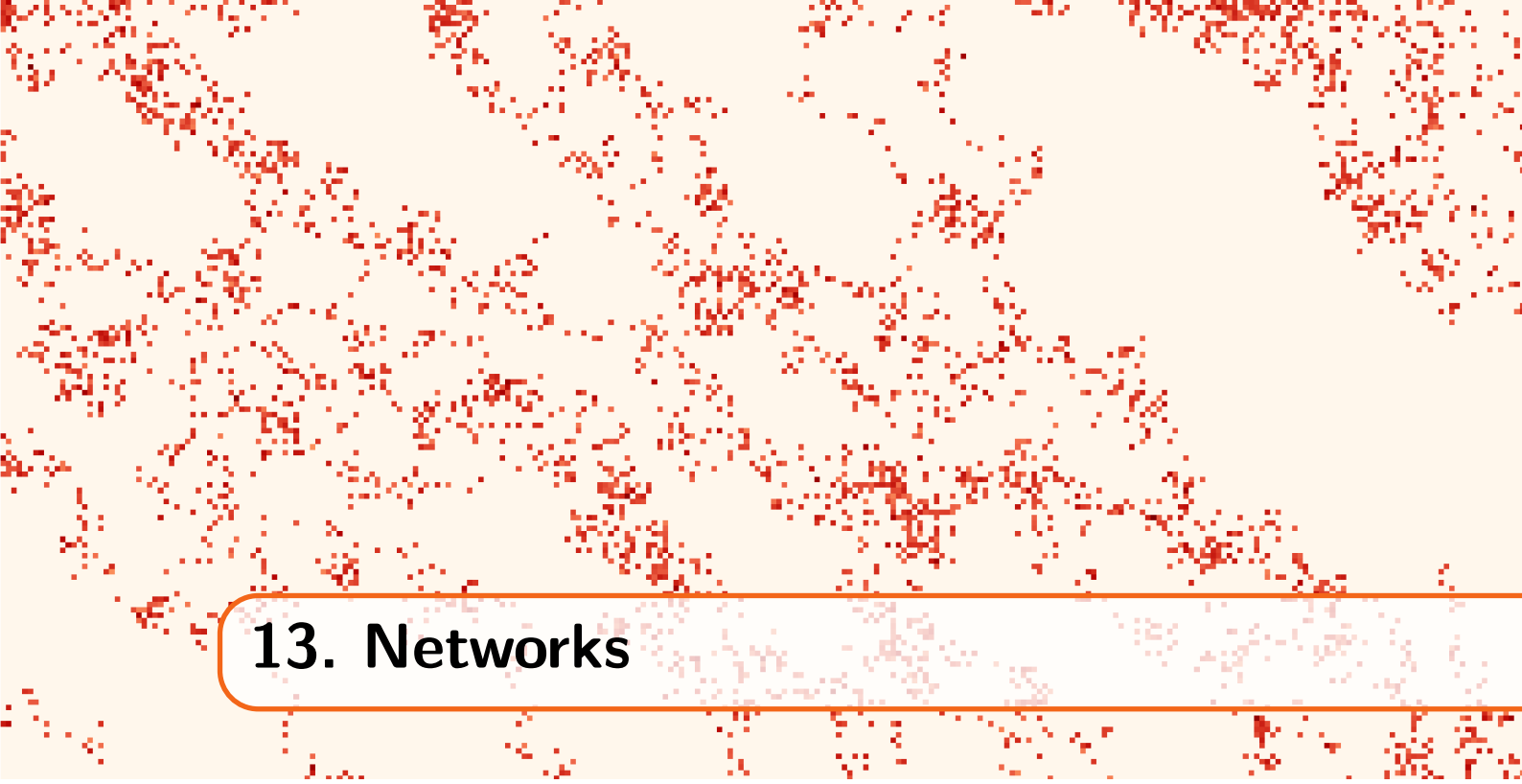
```
#We sample using the metropolis algorithm
random_sampling = markov_chain(p,11000,np.zeros(1),0.6)

#We skip the first 1000 numbers for thermalization
#We consider the increment every tenth random numbers
#to reduce autocorrelation
increments = np.exp(random_sampling[1000::10,0]/100)

#The starting point
x0 = 1000

#The log-truncated-cauchy Brownian motion
path = x0*np.cumprod(increments)

plt.plot(range(len(path)),path)
plt.xlabel("$t$")
plt.ylabel("$S$")
plt.tight_layout()
plt.show()
```

A simulation using truncated Levy flights is shown in Fig. 12.11.

Exercise 12.8 Which simulation does appear to be closer to the price of Bitcoin in dollars for the last five years? The one in Fig. 12.9, Fig. 12.10 or Fig. 12.11? What about the Apple stock? ∎

Exercise 12.9 Monte Carlo simulations are not limited to simulations of physical or financial systems, but can also provide an important tool to develop winning strategies for games.

You are invited to a game TV show. There are five boxes, in one of them there are 2000 dollars, while in the other four there is a carrot in each of them. You choose one box. The presenter knows in which boxes the carrots are, and he opens randomly two other boxes where there are the carrots, without opening the one with the 2000 dollars and the one you have chosen. Now there are three boxes left. The presenter gives you the opportunity to change the box, if you wish. What is the best strategy? Change box or don't change box? Is it indifferent? Reply to question by running 1000 simulations of the game for each possible strategy. ∎



13. Networks

In this chapter we will understand the tools of linear algebra that are required to analyze mathematically the structure of networks. We will also demonstrate an interesting application to networks of eigenvalues and eigenvectors of a matrix using the module **numpy.linalg**.

13.1 Networks and graphs

Graphs are the abstract tool used to represent networks, such as the train connections between cities, or the marriage relationships between the Royal families in the Europe of the XIX century. Graphs are formally a set of N nodes and a set of links between them. Directed graphs contain links (i, j) connecting the node i to the node j, while for undirected graphs all links connect the nodes i to j and vice-versa. For example, directed graph can be used to represent links of the world wide web, where each link represents an hyper-link from the website i to the website j. Undirected graph can be used to represent networks where all connections are bidirectional, such as a highways connections between two cities.

A common mathematical tool to represent networks is the adjacency matrix. The adjacency matrix of a graph $\mathcal{G}$ is a square matrix of size $N \times N$ defined such that

$$A_{ij} = \begin{cases} 1 & \text{if the node } i \text{ is connected with } j \\ 0 & \text{if the node } i \text{ is not connected with } j \end{cases} . \tag{13.1}$$

The adjacency matrix of an undirected graph is a symmetric matrix, $A = A^T$, since

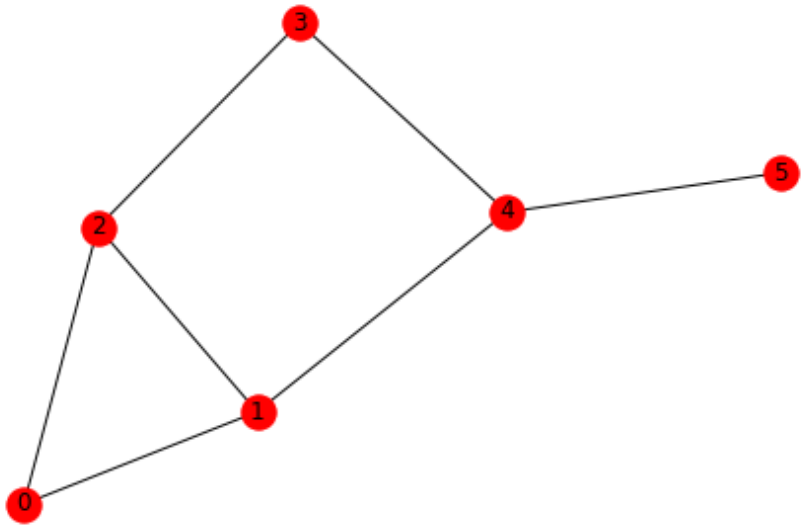


Figure 13.1: A graph describing a network between six nodes.

$A_{ij} = A_{ji}$. For instance, the adjacency matrix of the graph in Figure 13.1, is

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} . \tag{13.2}$$

More on ... 13.1.1 — Plotting networks and graphs. Networks and graphs can be plotted using the module **networkx**.

```
#Networkx is a package
#that allows to plot networks
import networkx as nx

#A undirected graph
G = nx.Graph()

edges = [[0,1],[0,2],[2,3],[1,2],[1,4],[4,5],[4,3]]

G.add_edges_from(edges)

nx.draw(G, with_labels = True)
```

The adjacency matrix provides many informations about the structure and the connectivity of a network. If we consider the square of the adjacency matrix

$$(A^2)_{ij} = \sum_k A_{ik} A_{kj} \,, \tag{13.3}$$

we realize that $(A^2)_{ij}$ is not zero if and only if the node i can be reached from the node j in two steps following the links of the networks. Indeed, in order for $(A^2)_{ij}$ to be non-zero, it must occur that both A_{ik} and A_{kj} are non-zero for at least one k, meaning that the two nodes are connected by the links to k. Further, we see that $(A^2)_{ij}$ counts effectively the numbers of possible paths of length two from i to j. Following the same reasoning, the cube of the adjacency matrix

$$(A^3)_{ij} = \sum_{kl} A_{ik} A_{kl} A_{lj} \,, \tag{13.4}$$

counts effectively the number of paths of length 3 from the node i to j. By induction, the same property holds for the matrix A^n for any integer n larger than one.

For instance, following the network in Figure 13.1, we have

$$A^2 = \begin{pmatrix} 2 & 1 & 1 & 1 & 1 & 0 \\ 1 & 3 & 1 & 2 & 0 & 1 \\ 1 & 1 & 3 & 0 & 2 & 0 \\ 1 & 2 & 0 & 2 & 0 & 1 \\ 1 & 0 & 2 & 0 & 3 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} , \tag{13.5}$$

and

$$A^3 = \begin{pmatrix} 2 & 4 & 4 & 2 & 2 & 1 \\ 4 & 2 & 6 & 1 & 6 & 0 \\ 4 & 6 & 2 & 5 & 1 & 2 \\ 2 & 1 & 5 & 0 & 5 & 0 \\ 2 & 6 & 1 & 5 & 0 & 3 \\ 1 & 0 & 2 & 0 & 3 & 0 \end{pmatrix} , \tag{13.6}$$

and we see that we cannot start from the node 5 and come back to the node 5 using three links, as $(A^3)_{55}$ is zero, while the node 0 is unreachable in two steps from the the nodes 5, as $(A^2)_{05}$ is zero.

More on ... 13.1.2 — The @ operator. In 2014, a new operator "@" has been introduced to mean the matrix-matrix multiplication [21].

```
import numpy as np

matrix = np.array([0,1,2,1]).reshape(2,2)

print(matrix @ matrix)
print(np.linalg.matrix_power(a, 2))
```

Exercise 13.1 Considering the network in Figure 13.1, can you count how many paths of length 7 are there between the link 1 and the link 4? ∎

13.2 Eigenvector centrality

The main problem we want to address in this section is how to determine the most important node in a network. We will try to understand the problem and we will

also try to define "precisely" what is meant by "most important". In this section we will assume that we are dealing with a **connected undirected graph**.

We have already seen that the entries of the matrix A^n counts the number of paths length n connecting the nodes of the network. As the adjacency matrix of an undirected graph is symmetric, we can use the spectral theorem to decompose A as

$$A = O\Lambda O^T \tag{13.7}$$

being Λ a diagonal matrix containing the eigenvalues $\lambda_1, \ldots, \lambda_k$ of A as diagonal entries, while O is an orthonormal matrix where each column corresponds to an eigenvector. An orthonormal matrix fulfill the relation

$$O^T O = I\,, \tag{13.8}$$

where I is the identity matrix, while an eigenvector v_i is a vector such that

$$Av_i = \lambda_i v_i\,. \tag{13.9}$$

We can compute the eigenvalues and eigenvectors of a symmetric matrix using the function **np.linalg.eigh**. In the example below, we demonstrate the spectral theorem for the adjacency matrix of the network in Figure 13.1.

```
import numpy as np

A = np.array([
      0, 1, 1, 0, 0, 0,
      1, 0, 1, 0, 1, 0,
      1, 1, 0, 1, 0, 0,
      0, 0, 1, 0, 1, 0,
      0, 1, 0, 1, 0, 1,
      0, 0, 0, 0, 1, 0]).reshape((6,6))

eigvl, eigvc = np.linalg.eigh(A)

print("Eigenvalues", eigvl)
#Print the eigenvectors stored in rows
print("Eigenvectors", eigvc.T)
print(np.round(eigvc @ np.diag(eigvl) @ eigvc.T))
```

Exercise 13.2 Using the library **numpy.linalg**, compute the eigenvalues and eigenvectors of the symmetric matrix

$$B = \begin{pmatrix} 1 & 5 & 1 \\ 5 & 0 & -3 \\ 1 & -3 & 7 \end{pmatrix}. \tag{13.10}$$

Verify, up to the numerical precision, that

$$I = O^T O \tag{13.11}$$
$$B = O\Lambda O^T, \tag{13.12}$$
$$B^3 = O\Lambda^3 O^T, \tag{13.13}$$
$$Bv_i = \lambda_i v_i, \tag{13.14}$$

where O is the matrix of eigenvectors given by **numpy.linalg.eigh**, O^T its transpose and Λ a diagonal matrix with the eigenvalues as diagonal entries. ■

Using the spectral theorem and the relation 13.8, we can note that

$$A^n = O\Lambda O^T O\Lambda O^T \dots O\Lambda O^T = O\Lambda^n O^T, \tag{13.15}$$

where Λ^n is the diagonal matrix with the eigenvalues λ_i^n as diagonal entries. Explicitly, the matrix A^n can be written in terms of the eigenvalues and eigenvectors as

$$(A^n)_{ij} = \sum_k \lambda_k^n (v_k)_i (v_k)_j, \tag{13.16}$$

being $(v_k)_j$ the j-th entry of the eigenvector k. For very large $n \gg 1$, the dominant contribution in the sum above comes from the largest eigenvalue, say λ_0 (if the eigenvalues are sorted, $\lambda_0^n \gg \lambda_1^n \gg \lambda_2^n \gg \dots$), and we have

$$(A^n)_{ij} = \lambda_0^n (v_0)_i (v_0)_j. \tag{13.17}$$

All entries of the eigenvector v_0 are all positive by the Perron-Frobenius theorem.

The formula above tells that for n large, the connectivity of the network as given by the number of paths of length n connecting the nodes is given by the largest eigenvalue and eigenvector. We also note that the number of paths of length n connecting the node i to some other node j is proportional to $(v_0)_i$. We can define a node to be "important" if there are many paths of length n connecting it to the rest of the network, i.e. the importance of the node i is defined to be proportional to $(v_0)_i$ [22].

For instance, for the network in Figure 13.1, $\lambda_0 \simeq 2.53948$ and

$$v_0 \simeq (0.4011, 0.5163, 0.5023, 0.3583, 0.4076, 0.1605). \tag{13.18}$$

We could have just used the number of incoming links to a node as a measure of its importance, but in this case we would have had three nodes with three incoming links equally ranked (node 1, 2 and 4). The eigenvector corresponding to the largest eigenvalue provides instead a more precise information. For instance we see that the weight given to node 4 is quite small compared to node 1 and 2, which intuitively is meaningful as one of the three connections of node 4 is to the isolated node 5. The largest score is given to node 1, which would be the "most important node", as it is connected to two nodes which are important and have three links (2 and 4), while node 2 is connected only to one important node (node 1) and two less important "peripheral" nodes (node 0 and 3).

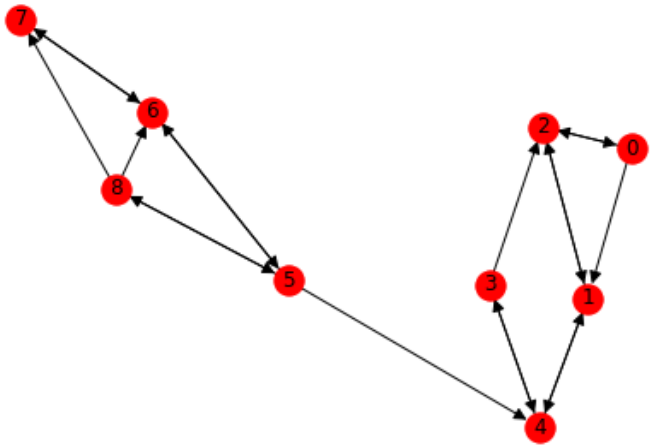


Figure 13.2: A network containing directed links between nine nodes.

Exercise 13.3 Using the module numpy, verify formula 13.16 for the matrix in equation 13.2. ∎

13.3 Random surfers

The limit of eigenvector centrality is the assumption that the adjacency matrix of the network is symmetric, otherwise the spectral decomposition of equation 13.7 does not hold. A symmetric adjacency matrix means that if the node i is connected with the node j, then also the node j is connected to the node i. However, many real world networks have directed links connecting only the source to the destination, and not vice-versa. We can relax the undirected graph restriction if we think in terms of random processes and stochastic matrices. The idea is to do not use the numbers of paths of length n connecting nodes to the rest of the network as a measure of their relative relevance, but use random surfer which explore the network randomly to see in which node they spend most of their time.

Assume we start to explore the network from a node. At each step, either we decide to jump randomly to an another node with probability α, or we follow one of the links of the current node with probability $1-\alpha$. The parameter α is between zero and one ($0 < \alpha < 1$, for instance 0.05). If the current node has no outgoing links, we just jump randomly to an another node, avoiding to be stuck to a single node without any way to get out of it (for instance, avoiding to be trapped to a web-page without any hyper-link). After a large number of steps, we collect the statistics of the nodes visited by the random surfer, and we rank each node accordingly to the number of time it has been visited. The node with the largest number of visits is going to be the most important node in the network.

The implementation of the random surfer motion is quite straightforward. We first define a list of links in the network of the form $[i,j]$, then we collect the outgoing links from each node in such a way that we can explore the network easily by just calling the function **choice** of the module **random**.

```
import random
```

```
import numpy as np

def random_surf(edges, number_steps, alpha):
      #First we reorganize the links
      number_nodes = max([j for i in edges for j in i])+1
      #by collecting all outgoing link
      #for each node
      links = [[] for i in range(number_nodes)]
      for start, end in edges:
            links[start].append(end)

      #We start from a random node
      current_node = random.randint(0, number_nodes - 1)

      #How many times we have visited each node
      stat = np.zeros(shape=(number_nodes))

      for step in range(number_steps):
            if (len(links[current_node]) == 0
                  or random.uniform(0,1) < alpha):
                  #jump to a node randomly
                  current_node = random.randint(
                        0,
                        number_nodes - 1)
            else:
                  #jump to a node connected
                  #to the one where we are currently
                  current_node = random.choice(
                        links[current_node])

            stat[current_node] += 1

      return stat/number_steps

links = [[0,1],[3,2],[2,1],[1,2],[0,2],[2,0],[3,4],
      [4,3],[5,4],[4,1],[1,4],[5,6],[6,5],[7,6],
      [6,7],[8,6],[8,7],[8,5],[5,8]]

statistics = random_surf(links,
      number_steps=100000,
      alpha=0.05)

print(statistics)
#For instance
#0.106  0.240   0.211  0.091  0.178  0.042  0.066  0.043  0.019
```

The network is depicted in Figure 13.2 for the example running in the script above.

The most important node would be node 1, being visited 24% of the time, followed by node 2. Nodes 5, 6, 7 and 8 are less important, as they are connected to the rest of the network only by the the link going from the node 5 to the node 4.

Exercise 13.4 Compute the relevance weight given by the random surf algorithm of the example in the script above in the case when the link between the node 5 to the node 4 is bidirectional. Try to explain intuitively the result you obtain. ■

13.4 Stochastic matrices

We now introduce a mathematical abstraction useful to handle stochastic processes such as the Brownian motion seen in the previous chapter. The key idea is that the random surfer motion introduced in the previous section is effectively an extension of the Brownian motion to networks. In order to keep the discussion simple, we will introduce the concept of stochastic matrices by a simple example of a frog jumping in a pond.

A frog is jumping randomly between six waterlilies, and sometimes it happens that it finishes in the sixth waterlily, that it likes a lot. Once the frog is in the sixth waterlily, it will jump only with ten percent of probability to the fifth waterlily, otherwise the frog will not move. The stochastic process followed by the frog can be described by the stochastic matrix P

$$P = \begin{pmatrix} 1/6 & 1/6 & 1/6 & 1/6 & 1/6 & 0 \\ 1/6 & 1/6 & 1/6 & 1/6 & 1/6 & 0 \\ 1/6 & 1/6 & 1/6 & 1/6 & 1/6 & 0 \\ 1/6 & 1/6 & 1/6 & 1/6 & 1/6 & 0 \\ 1/6 & 1/6 & 1/6 & 1/6 & 1/6 & 1/10 \\ 1/6 & 1/6 & 1/6 & 1/6 & 1/6 & 9/10 \end{pmatrix} . \tag{13.19}$$

The element P_{ij} gives the probability that the frog jumps from the place j to the place i. Note that the sum of the elements of each column of P is one. Therefore, the sum of the elements of a vector v is unchanged after that P is applied to it

$$w_i = \sum_j P_{ij} v_j \,, \tag{13.20}$$

since

$$\sum_i w_i = \sum_i \sum_j P_{ij} v_j = \sum_j \left(\sum_i P_{ij} \right) v_j = \sum_j v_j \,. \tag{13.21}$$

If the vector v represents a normalized probability distribution, for instance the probability that the frog is in the i-th waterlily, then the normalization is preserved after that P is applied many times to it. Therefore

$$w^n = P^n v \tag{13.22}$$

will be a vector whose entries w_j are the probability that the frog is in the place j after n jumps. The study of the properties of the stochastic matrix is therefore an alternative to direct Monte Carlo simulations of the corresponding stochastic random process.

Exercise 13.5 Compute the probability vector w after four jumps for $n = 4$ starting from the second waterlily

$$v = \{0, 1, 0, 0, 0, 0\}. \tag{13.23}$$

What is the probability that the frog is on the sixth place? ∎

13.5 The Page-Rank matrix of a network

The limit of eigenvector centrality is the assumption that the adjacency matrix of the network is symmetric, otherwise the spectral decomposition of equation 13.7 does not hold. However, we can see that the random surfer motion is effectively similar to jumps of the frog described in the previous section, and we can try to construct the corresponding stochastic matrix P as

$$P_{ij} = \frac{\alpha}{N} + \frac{(1-\alpha)}{L(j)} A_{ij} \tag{13.24}$$

where N is the number of nodes of the network, A_{ij} is the adjacency matrix and $L(j)$ is the number of outbound links from the node j, equal to $\sum_j A_{ij}$. The stochastic matrix above is called Page-Rank matrix, by the name of the corresponding algorithm used by Google [23]. The first term $\frac{\alpha}{N}$ represents the random jump from one node to an another with probability α, while the second term is the jump to one the nodes connected to the current node, normalized to represent a probability.

The code required to compute the Page Rank matrix is simple. We need first to compute the adjacency matrix, and then normalize the columns while being careful to handle the situation when a node does not have outgoing links.

```
def adjacency_matrix(edges):
      number_nodes = np.max(edges)+1
      result = np.zeros(shape=(number_nodes,number_nodes))
      for start, end in edges:
            result[end,start] = 1

      return result

def page_rank_matrix(edges, alpha):
      number_nodes = np.max(edges)+1

      I = np.full([number_nodes, number_nodes], 1.)
      M = adjacency_matrix(edges)

      #Compute the number of outgoing links for each node
      sums = [np.sum(M[:,i]) for i in range(len(M))]

      for i in range(number_nodes):
            for j in range(number_nodes):
                  #normalize the adjacency matrix
                  if (sums[j] != 0):
```

```
                    M[i,j] = M[i,j]/sums[j]
                else:
                    M[i,j] = 1/number_nodes

    PG = (alpha/number_nodes)*I + (1.-alpha)*M
    return PG

edges = [[0,1],[3,2],[2,1],[1,2],[0,2],[2,0],[3,4],
    [4,3],[5,4],[4,1],[1,4],[5,6],[6,5],[7,6],
    [6,7],[8,6],[8,7],[8,5],[5,8]]

print(page_rank_matrix(edges,0.05))
```

Exercise 13.6 Verify that the Page Rank matrix is normalized such that the sum of the columns is equal to one. ∎

Assume now we start to explore the network from the node k following the random surfer algorithm. We start therefore from a vector z that is zero everywhere, except the entry k equal to one, i.e. we are in the node k with 100% probability. For instance, for a network with nine nodes and $k = 2$, we have z equal to

$$z = (0,0,1,0,0,0,0,0,0)\,. \tag{13.25}$$

After n steps, we will land in a node with a probability distribution w_n given by

$$w_n = P^n z\,. \tag{13.26}$$

For a stochastic matrix, the Perron-Frobenius theorem states that, for an irreducible and aperiodic random process, the largest eigenvalue in magnitude is $\lambda_0 = 1$ and the corresponding eigenvector has only positive entries which sums up to one. Therefore, for n very large, the contribution of all other eigenvalues λ_n, $n \neq 0$, becomes smaller and smaller. The dominant contribution is given by the eigenvalue $\lambda_0^n = 1$ and by the corresponding eigenvector, so we have

$$\lim_{n\to\infty} w_n = v_0\,. \tag{13.27}$$

The vector v_0 is equal to the ranking we have computed from the simulation of the random surfers of the previous section. We can verify directly this statement by computing the eigenvalues and eigenvectors using the function **numpy.linalg.eig**.

```
...

P = page_rank_matrix(edges,0.05)

eigv, eigc = np.linalg.eig(P)

#The largest eigenvalue
print(eigv[0])
```

```
#The corresponding eigenvector
#normalized to one
norm = np.sum(eigvc.T[0])
print((eigvc.T[0]/norm).real)

#The output will be like:
#0.107 0.241 0.213 0.089 0.176 0.043 0.066 0.043 0.019
```

As we can see from the script above, we are able to indeed get a similar ranking to the one obtained in the previous section by a direct simulation of the random surf.

13.6 Ranking of the most important physicists from the wikilinks of wikipedia

In this section we apply the Page Rank algorithm to rank by importance the physicists of all times using the wiki-links from their corresponding biographies on wikipedia. The code below might appear complex, but it is performing three simple tasks in sequence. First, after we loading the main modules and defining the hyper-links to wikipedia, we define a function downloading a list of names of physicist from some wikipedia category, such as "German physicists" or "Nobel laureates in Physics".

```
...


import urllib.request
import time
from urllib import parse

import matplotlib
import xml.etree.ElementTree as etree
import re
import math
import pickle
import os
import numpy as np

api_url = "https://en.wikipedia.org/w/api.php?"
index_url = "https://en.wikipedia.org/w/index.php?"
link_url = "action=query&prop=links&pllimit=500&titles="
cat_url = "&action=query&list=categorymembers&cmlimit=500\
&format=xml"

def load_names_from_category(name):
      result = []
      continue_string=""
      while continue_string != "stop":
            url=(api_url+"cmtitle=Category:"+name+
```

```
                cat_url)

            print("Loading url ", url)

            #download the xml data
            data = urllib.request.urlopen(url).read()
            #decode the xml file
            data = data.decode("utf-8")

            parser = etree.XMLParser(encoding="utf-8")
            #we parse the xml file using utf-8 encoding
            root = etree.fromstring(data, parser)

            #for each page
            for elem in root.findall("./query/categorymembers/cm"):
                  #only for pages in the main namaspace
                  if elem.attrib["ns"] == "0":
                        #only if the page is not a list
                        #or a society
                        title = elem.attrib["title"]
                        if (title.count("List")
                              +  title.count("Society")) == 0:
                              result.append(title)

            #load the next results, wikipedia shows
            #500 results at maximum per xml page
            try:
                  continue_elem = root.find("continue")
                  continue_string = ("&plcontinue="+
                        continue_elem.attrib["plcontinue"]+
                        "&continue="+
                        continue_elem.attrib["continue"])
            except:
                  continue_string = "stop"
      return result
```

Then, for each wikipedia article, we define a function loading all wikilinks and we filter out all wikilinks pointing to other wikipedia pages.

```
...

def add_links_from_xml(data_xml,
      source,
      links_between_pages,
      names_without_underscores):

      parser = etree.XMLParser(encoding="utf-8")
```

```
        root = etree.fromstring(data_xml, parser)

        #for each link
        for elem in root.findall("./query/pages/page/links/pl"):
                #For each name
                for j, name in enumerate(names_without_underscores):
                        #Avoid self linking
                        if source != j:
                                #Try to look if there is the name
                                #without underscores in the link
                                if name == elem.attrib["title"]:
                                        #If so, create the link
                                        links_between_pages.append(
                                                [source,j]
                                        )

        #Try to see if there might be more elements
        #from the list
        try:
                continue_elem = root.find("continue")
                continue_string = ("&plcontinue="+
                        continue_elem.attrib["plcontinue"]+
                        "&continue="+
                        continue_elem.attrib["continue"])
        except:
                continue_string = "stop"

        return continue_string

def get_links_between_pages(names_without_underscores):
        #Here we replace the spaces with underscores
        # (for wikipedia links)
        names_with_underscores = [name.replace(" ", "_")
                for name in names_without_underscores]
        #List which will contain the links between the pages
        links_between_pages = []

        #for each element in the list
        for i, name in enumerate(names_with_underscores):
                continue_string=""
                while continue_string != "stop":
                        url = (api_url+link_url
                                +parse.quote_plus(name)
                                +"&format=xml"+continue_string)

                        try:
```

```
                        print("Loading url ", url)
                        #Download the page
                        data = urllib.request.urlopen(url)
                        data = data.read().decode("utf-8")

                        continue_string = add_links_from_xml(
                              data,
                              i,
                              links_between_pages,
                              names_without_underscores)

                  except:
                        print("There has been some",
                          "problem in reading url",url)
                        continue_string = "stop"

            #We want to avoid large number of connections
            #in a short time, otherwise we might overload
            #the wikipedia server
            time.sleep(0.08)
      print("Number of links loaded: ",
            len(links_between_pages))
      return links_between_pages
```

Finally, we compute the Page Rank matrix and we print the ranking of each physicist loaded from the categories.

```
categories = ["Quantum_physicists","Particle_physicists",
      "Russian_physicists", "English_physicists",
      "Italian_physicists", "French_physicists",
      "German_physicists", "Austrian_physicists",
      "Cosmologists", "American_physicists",
      "Nobel_laureates_in_Physics",
      "Winners_of_the_Max_Planck_Medal"]

if not os.path.isfile("links_between_pages.dat"):
      #First we load the list of the physicists
      #we are interested in
      names_without_underscores = []
      for category in categories:
            names_without_underscores += \
                  load_names_from_category(category)

      #list and set will remove duplicated
      #sort will allow to preserve the order of the pages
      names_without_underscores = \
            sorted(list(set(names_without_underscores)))
```

```
        print("Constructing the network from",
              len(names_without_underscores),
              " wikipedia pages")

        #Load the links
        links_between_pages = \
              get_links_between_pages(names_without_underscores)

        with open("links_between_pages.dat","wb") as f_to_save:
              pickle.dump(
                    (links_between_pages,
                    names_without_underscores),
                    f_to_save)
else:
        with open("links_between_pages.dat","rb") as f_to_load:
              links_between_pages, names_without_underscores = \
                    pickle.load(f_to_load)


#Take the eigenvalues and eigenvectors of the adjacency_matrix
#We do consider directions in the links
PG = page_rank_matrix(links_between_pages,0.03)
eigenvalues, eigenvectors = np.linalg.eig(PG)

print("Eigenvalues of the page matrix: ", eigenvalues)

#Take the maximal eigenvector, corresponding to the
#lambda=1 eigenvalue
max_eigenvector_index = \
      np.where(np.abs(eigenvalues.real-1) < 10e-12)
maximal_eigenvector = eigenvectors.T[max_eigenvector_index][0]
print(maximal_eigenvector.shape)

#Ensures that the sum is one to have a ranking of the pages
ranking = maximal_eigenvector/np.sum(maximal_eigenvector)

print("Ranking vector: ", ranking)

print("List of the pages in descendent order: ")
sorted_pages = sorted(zip(names_without_underscores, ranking),
      key=lambda x: x[1])

n = len(sorted_pages)
for name, rank in sorted_pages:
      print(n, name, round(rank,6).real)
```

```
        n = n - 1
```

The output of the above script will provide a ranking like:

- ...
- 7 J. J. Thomson 0.004327
- 6 Enrico Fermi 0.004367
- 5 Max Planck 0.00439
- 4 Marie Curie 0.004423
- 3 Werner Heisenberg 0.004441
- 2 Niels Bohr 0.004523
- 1 Albert Einstein 0.005578

The above result is in agreement with the popular opinion of Albert Einstein being the most important and popular physicist of all the times. Keep however in mind that the score given by Page Rank does not provide any real score of the achievements of a scientist, and it also biased toward physicists died in the last century (for instance Isaac Newton is classified around the 200th place). A reason for such bias might be that there are many articles of less recent important physicists connected for instance to the study of the relativity and to Albert Einstein, while only few researchers is nowadays involved directly in the study of Newton dynamics. Also, we are considering only biographies and not pages related to actual physics topic when constructing the network provided to Page Rank.

A similar idea to the one sketched in this section has been proposed as alternative of the h-index to estimate the impact of the scientific achievements of researchers [24].

Exercise 13.7 How does the ranking change if α is set to 0.10, 0.2 or 0.3? What if more categories are included? What if the wikipedia in German, Spanish or in French are used in place of the English wikipedia? ∎

IV Introduction to machine learning

14. Regression for image classification

Image classification is a standard problem solved by "machine learning" algorithms. Handwritten text and human face recognition, distinguishing pictures of dogs from the one of cats, and object detection are typical example of image classification problems. In this chapter we will explore the basic regression algorithm and the softmax function on the MNIST database (Modified National Institute of Standards and Technology database), which includes 70000 classified pictures of handwritten digits.

14.1 Image recognition as a fitting problem

The MNIST dataset consists of 70000 grayscale pictures of size 28×28 pixels, each of them representing a centered and rescaled image of a single handwritten digit. The shades of gray are stored as an integer between 0 and 256, from white to black, respectively. Each picture is classified by a label from 0 to 9 corresponding to the written number. The MNIST dataset is split in two parts, the first 60000 pictures are intended to be used for "training", while the other 10000 are left for "testing". The aim is to find an algorithm to extract the digit written in each "test" picture, using only the 60000 "training" pictures as input (**supervised learning**).

In mathematical terms, image classification can be represented as a fitting problem, i.e. finding a suitable function taking as input 28×28 integers and giving as output a single integer number from 0 to 9. The function can be found iteratively, using the training dataset to determine an error that can be minimized step by step.

Instead of working with functions handling two indexes, we first flatten each 28×28 picture to a single linear array of 784 entries, constructed by simply putting all pixels for all rows of the picture one next to the other. Explicitly, a pixel in position (i, j) is mapped to the entry $28i + j$. Flattening the input will simplify drastically the implementation and the expression of the formulas for fitting in the

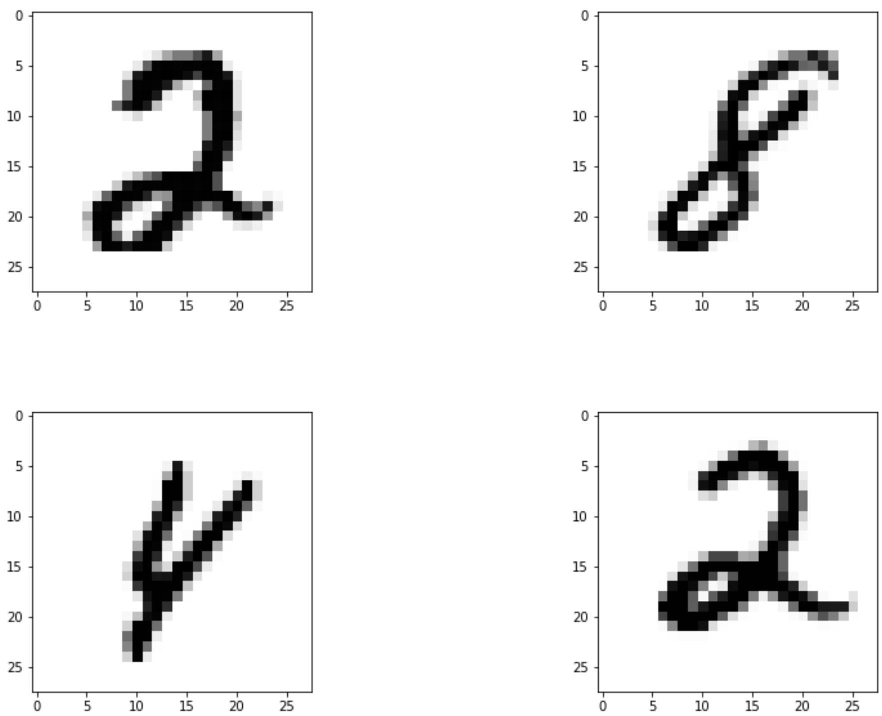


Figure 14.1: Example of handwritten digits of the MNIST database.

$$
\begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 11 & 51 & 113 & 152 & 152 & 193 & 234 & 112 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 41 & 173 & 252 & 253 & 252 & 253 & 252 & 253 & 252 & 41 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 52 & 233 & 254 & 253 & 254 & 253 & 244 & 203 & 254 & 253 & 234 & 30 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 152 & 252 & 253 & 252 & 233 & 111 & 40 & 0 & 172 & 252 & 253 & 50 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 152 & 253 & 254 & 233 & 41 & 0 & 0 & 102 & 254 & 253 & 254 & 50 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 163 & 233 & 252 & 233 & 91 & 0 & 0 & 0 & 61 & 253 & 252 & 253 & 50 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 21 & 61 & 0 & 0 & 0 & 0 & 0 & 0 & 152 & 253 & 254 & 112 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 152 & 252 & 253 & 70 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 152 & 253 & 254 & 50 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 233 & 252 & 233 & 30 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 102 & 254 & 253 & 183 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 183 & 253 & 252 & 20 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 31 & 92 & 152 & 193 & 254 & 253 & 254 & 253 & 254 & 192 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 82 & 233 & 252 & 253 & 252 & 253 & 252 & 253 & 252 & 253 & 192 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 41 & 173 & 253 & 254 & 253 & 224 & 122 & 132 & 253 & 254 & 253 & 254 & 253 & 254 & 91 & 0 & 0 & 31 & 10 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 41 & 243 & 253 & 252 & 192 & 70 & 20 & 0 & 173 & 252 & 253 & 212 & 151 & 192 & 253 & 252 & 183 & 142 & 233 & 50 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 113 & 253 & 254 & 233 & 41 & 0 & 11 & 173 & 254 & 253 & 203 & 61 & 0 & 0 & 123 & 243 & 255 & 233 & 123 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 71 & 252 & 253 & 70 & 0 & 0 & 132 & 252 & 253 & 212 & 61 & 0 & 0 & 0 & 0 & 40 & 50 & 30 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 41 & 243 & 254 & 172 & 31 & 173 & 254 & 253 & 244 & 40 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 122 & 253 & 252 & 233 & 252 & 253 & 252 & 81 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
$$

Figure 14.2: Example of a grayscale encoded handwritten "2" of the MNIST database.

next sections.

14.2 Softmax for classification problems

For classification problems, a single linear or polynomial function to label the input images might be inefficient. A binary function, whose output is a real number in the range between zero and one, can be interpreted as a true/false answer to the question "does the picture belong to the class X?". For instance, in the case of the MNIST dataset, "Is it written in the picture a five?". To fully classify a picture, we need to fit ten functions in total, say f_n with $n \in \{0,1,2,\ldots,9\}$.

A linear model would try to fit from the set of the training pictures a function f_n of the form

$$f_n(x^I) = \sum_j W_n^j x_j^I + b_n \,, \tag{14.1}$$

being x_k^I an individual pixels of an image I, W_n^j 784 real coefficients, and b_n a constant shift. As the output of a linear function is not in the range between zero and one, we apply the function "softmax" to all the f_n as

$$\text{softmax}(f_n) = \frac{\exp(f_n)}{\sum_n \exp(f_n)} \,. \tag{14.2}$$

The exponential gives a positive output, while the sum in the denominator is a normalization ensuring an output not larger than one. As the sum of all f_n is one, each f_n might be interpreted as the probability that the picture represents the digit n.

The fitting algorithm finds the minimum of the squared error-distance, called **loss** function

$$\text{loss} = \sum_{I,n} (f_n(x^I) - y_n^I)^2 \,, \tag{14.3}$$

with respect to the 784×10 linear parameters W_n^j and to the 10 constant shifts b_n. Each train picture is finally classified by finding the n for which $f_n(x^I)$ is maximal. The labels y_n^I are encoded in an array of ten elements, all set to zero except the one corresponding to the number written in the picture. For instance, for the number 3, the corresponding array y_n^I would be

$$y^I = \{0,0,0,1,0,0,0,0,0,0\} \,. \tag{14.4}$$

14.3 Tensorflow for gradient descent

Finding a minimum in a parameter space of $784 \times 10 + 10 = 7850$ dimensions is quite challenging. To this end, many libraries have been developed including efficient routines for gradient descent algorithms, along the lines discussed in Section 10.2. In this section we will use the library **tensorflow** for the softmax regression on the MNIST dataset. Tensorflow is a powerful libraries implementing many functionalities of numpy, and including automatic differentiation tools.

More on ... 14.3.1 — Tensorflow versus numpy. Tensorflow has been developed by the Google Brain team and first publicly released in 2015. Many applications for machine learning are based on tensorflow, although tensorflow can be used for instance as alternative to numpy when dealing with multidimensional arrays. The basic class is **tf.Tensor**, which represents **immutable** multidimensional arrays or even simple numbers.

```
import tensorflow as tf

a_array = tf.random.normal(shape=(1000,))
b_array = tf.random.normal(shape=(1000,))

print(tf.cos(a_array+b_array).numpy())
```

The method **numpy** is used to retrieve the actual content stored in a tensor. **tf.Tensor**s are effectively the equivalent of numpy arrays.

An instance of the class **tf.Variable** represents a mutable object whose value is linked to a **tf.Tensor**. A **tf.Variable** represents a tensor whose value can be changed by running operations on it. The difference is fundamental for instance when instructing the minimizers of tensorflow what are the parameters that should be optimized and what are instead the tensors that are kept constant.

As first step we need to load the MNIST database, and flatten each 28×28 picture to a single array. We also cast the input images from integer to floating numbers, in order to be able to perform later multiplications and additions between homogeneous structures.

```
import tensorflow as tf
import numpy as np

(x_train, y_train), (x_test, y_test) = \
      tf.keras.datasets.mnist.load_data()

x_train = tf.cast(x_train.reshape(60000,784),
      dtype=tf.float32)
x_test = tf.cast(x_test.reshape(10000,784),
      dtype=tf.float32)
```

The labels y_train and y_test are integers between zero and nine, so we need to transform them as in Eq. 14.4 for fitting the training pictures using the softmax function.

```
...

y_train_softmax = np.zeros(shape=(60000,10),dtype=np.float32)
y_test_softmax = np.zeros(shape=(10000,10),dtype=np.float32)

#For each picture, set to 1 only the entry corresponding
#to the number written in it
```

```
for i in range(10):
    y_train_softmax[np.where(y_train == i),i] = 1
    y_test_softmax[np.where(y_test == i),i] = 1

y_train_softmax = tf.convert_to_tensor(y_train_softmax)
y_test_softmax = tf.convert_to_tensor(y_test_softmax)
```

In above code, we use numpy to initialize the y_train and y_test arrays in the softmax format, and at the end we call the function **tf.convert_to_tensor** to get a **tf.Tensor** object that can be used by tensorflow. Working with numpy arrays simplifies the setup code, as **tf.Tensor** objects are immutable.

At this point we are ready to write the function that tensorflow will minimize. Tensorflow requires as input for the minimization routine a function that takes no arguments, while the variables which are going to be minimized are stored as **tf.Variable** objects.

```
...

def setup_loss_function(
      X_train,
      Y_train,
      batch_size = 10000):

    W = tf.Variable(initial_value=tf.ones(shape=(784,10)))
    B = tf.Variable(initial_value=tf.ones(shape=(10)))

    dataset = tf.data.Dataset.from_tensor_slices(
      (X_train,Y_train)
      ).batch(batch_size).repeat(5000)
    batches = iter(dataset)

    def loss():
        X, Y = next(batches)
        return tf.reduce_sum(
            (tf.nn.softmax(tf.matmul(X,W)+B)-Y)**2
        )


    return loss, [W, B]
```

The formula defined in the **loss** function is given in Eq. 14.3

$$\begin{aligned} \text{loss} &= \sum_{j,I,n} (\text{softmax}(W_j^n x_j^I + b_n) - y_n^I)^2 && (14.5) \\ &= \text{reduce\_sum}((\text{softmax}(X.W + B) - Y)^2), && (14.6) \end{aligned}$$

rewritten in terms of the matrix-matrix product between the tensors X and W, performing the product and summing over the index j. The function **tf.reduce_sum**

performs the sum over the open indexes I and n. The entries of the variables W and B are initialized to one, and returned back together with the loss function.

The loss function is not computed over the entire MNIST training dataset, but only on subsets including 10000 elements each. The training dataset is looped over in batches of 10000 elements, given by calling the **next** function on an iterator. The loss function defined on small batches is easier to compute, and the possibility that the minimizer gets stuck on local minima are reduced when batches are used one after the other at each iteration step. In fact, a function depending on a large number of parameters might have many local minima. The global minimum, representing the function that is able to decipher an handwritten digit, must be present on all subsets, regardless on which training images are included in the loss function. Unwanted local minima that do not represent a good decipher function are therefore removed if the batches are changed at each iteration step, and there are better chances that the minimizer will be able to find a good global minimum.

The final step is to setup and run the minimizer for 5000 iterations to fit the linear softmax model.

```
...

loss, variables = \
      setup_loss_function(x_train, y_train_softmax)

opt = tf.keras.optimizers.Adadelta(
      learning_rate=0.25,
      epsilon=0.0005)

for i in range(5000):
    opt.minimize(loss, var_list=variables)
    if i % 20 == 0:
        print("Loss function at iteration", i, ":",
            loss().numpy())

fitted_W, fitted_B = variables

print(tf.reduce_sum(
      tf.cast(
            tf.argmax(
                  tf.nn.softmax(
                        tf.matmul(x_test,fitted_W)
                        +fitted_B),
                  1)==y_test,
            tf.int32
      )
)
```

The **tf.argmax** function looks for the maximal entry of softmax along the second axis, returning the corresponding index that is indeed the number recognized in the

test picture. The cast to an integer, zero for false and one for true, allows to perform the sum and count how many pictures have been correctly identified. Running the script above, the percentage of correctly identified pictures should be around 92%.

Exercise 14.1 What happens if the batch size is reduced to 5000 or 1000 elements? What happens if you change the parameters of the Adadelta minimizer? ■

14.4 Fitting and overfitting

A simple linear function as considered in this chapter on such low resolution images requires to fit 7850 parameters from 60000 data, and is able to reach an accuracy of approximately 92-93%. We can try to improve the accuracy by using a quadratic function

$$f_n(x) = \sum_{ij} (A_n^j x_j + C_n^{ij} x_j x_i) + b_n \tag{14.7}$$

would require $10 \times (784 + 784 \times (784+1)/2 + 1) = 3085050$ parameters to be fitted, a number significantly larger than the available pictures for training. As seen in Section 10.5, overfitting occurs quickly as the number of fitting parameters increases with respect to the number of available fitting points, yielding a poor function that is unable to provide a stable and reliable interpolation or extrapolation of the available training data. Overfitting is a common problem arising from the *curse of dimensionality* given by the large number of input parameters, large with respect to the number of available samples for the fit. In addition, finding the minimum of the loss function in a space with hundreds or thousands parameters is a challenging problem. On the other hand, a simple linear model might perform poorly on complex tasks such as image classifications. In the next chapters we will try to improve our algorithms.

15. Vector spaces and image classification

In this chapter we will explore the k-nearest neighbor algorithm and the Principal Component Analysis (PCA) for the analysis of the MNIST database.

15.1 The geometric nature of image classification

A simpler and widely employed alternative to the softmax fitting is represented by the k-nearest neighbors algorithm. Assume you want to know the specie of a plant you have never seen before. If you have a botanic book on your bookshelf, you can navigate through the pages searching for a photograph which looks *similar* to the plant you see in front of you. We can try to mimic the same strategy to the MNIST database if we are able to define in abstract terms what it is meant by "similar pictures". The idea is to classify each test handwritten digit by returning the label corresponding to the most similar picture in the training database.

First we need to specify a proper definition of similarity between two pictures. To this end, we observe that the flatten array of 784 entries constructed from a 28×28 pixel image can be interpreted as a vector. Two pictures, as vectors, can be added and subtracted, where the sum or the subtraction is performed element-wise, for instance as

$$(x^{I_1} - x^{I_2})_k = x_k^{I_1} - x_k^{I_2} \,. \tag{15.1}$$

A vector space can be extended to a *metric* space, allowing to define the length of vectors, often called norm in linear algebra. For instance, the "Euclidean norm" is equal to the square root of the sum of the square of all entries

$$\text{norm}(v^I) = \sqrt{\sum_k (x_k^I)^2} \,. \tag{15.2}$$

The norm of a vector allows to further define the distance between two vectors x^{I_1}

and x^{I_2} as the norm of their difference

$$\text{distance}(x^{I_1}, x^{I_2}) = \text{norm}(x^{I_1} - x^{I_2}) = \sqrt{\sum_k (x_k^{I_1} - x_k^{I_2})^2}\,. \tag{15.3}$$

The formula above might appear quite abstract, however it is just a simple extension to an arbitrary dimension of the standard distance between two points in the three dimensional Euclidean space

$$\text{distance}(p^1, p^2) = \sqrt{(p_x^1 - p_x^2)^2 + (p_y^1 - p_y^2)^2 + (p_z^1 - p_z^2)^2} \tag{15.4}$$

up to the identification of the coordinates of the points p^1 and p^2 with the entries of a vectors.

More on ... 15.1.1 — Vector spaces and norms. The sum of the squares under the square root is not the only possibility to define a norm in a vector space. An another alternative could be the so-called 1-norm

$$\text{1-norm}(v^I) = \sqrt{\sum_k |v_k^I|}\,, \tag{15.5}$$

using the absolute value in place of the square, or the sup-norm

$$\text{sup-norm}(v^I) = \sqrt{\max_k |v_k^I|}\,. \tag{15.6}$$

The distance between two vectors is the key ingredient that defines the similarity between two pictures. To identify an handwritten digit, we first compute the distance between all training pictures, and then we return simply the label corresponding to the closest training picture with the smallest distance.

```
import tensorflow as tf

(x_train, y_train), (x_test, y_test) = \
      tf.keras.datasets.mnist.load_data()

x_train = tf.cast(x_train.reshape(60000,784),dtype=tf.int32)
x_test = tf.cast(x_test.reshape(10000,784),dtype=tf.int32)

correct = 0

for picture, label in zip(x_test,y_test):
    min_distance_pict = tf.argmin(
      tf.reduce_sum(
            (x_train-picture)**2,
            axis=1)
      )
    if y_train[min_distance_pict] == label:
        correct += 1
```

```
print(correct)
```

In the third and fourth instruction, we flatten and cast the pictures to integers, otherwise the pixels are load as unsigned integers and we would have problems when computing differences. The tf.argmin returns the index of the training picture with the minimal distance to the test picture considered. The index is used to retrieve the identified label in the training database, and later to compare its correctness. The simple code above reaches an accuracy of almost 97%.

15.2 The k-nearest neighbors algorithm

The k-nearest neighbors algorithm is a generalization of the idea explored in the previous section to the case where the k closest vectors are simultaneously considered. If the labels corresponding to the k closest image-vectors are mismatching, the classification can be given by a simple majority vote, i.e. by returning the label that appears more frequently. Alternatively, the vote can be weighted by the distance of the picture to the nearest neighbors.

Exercise 15.1 Modify the code of the previous section to include three and five nearest neighbors. Does the accuracy improve? ■

The k-nearest neighbors algorithm is effective when pictures to classify are clustered, i.e. when the pictures belonging to different classes are concentrated in cluster closed to each other and well separated from each other.

15.3 Principal component analysis

The k-nearest neighbor algorithm requires to compute the distance of the test picture from all training pictures, requiring a huge amount of computational resources, especially for high-resolution images. However, it is worth noting that there is a large redundancy in the information stored in the individual pixels. For instance, the borders of many if not all images are frequently empty white pictures. We need therefore to find an improved representation of the pictures of the training database.

The task we are seeking is compressing the pictures to a smaller size, while preserving at the same time the largest possible information of the original pictures.

Bibliography

[1] PEP 238 – Changing the Division Operator

[2] proposed language change to int/int==float (was: PEP0238 lament)

[3] PEP 20 – The Zen of Python

[4] Daniel Bernoulli, *Exposition of a New Theory on the Measurement of Risk*, Econometrica, Vol. 22, No. 1. (Jan., 1954), pp. 23-36. Original version here.

[5] PEP 257 – Docstring Conventions

[6] S. Wolfram: *Computation theory of cellular automata*, Communications in Mathematical Physics. 96 (1): 15?57 (1984).

[7] S. Wolfram: *Cellular automata as models of complexity*, Nature. 311 (5985): 419?424 (1984).

[8] S. Wolfram: *Statistical mechanics of cellular automata*, Reviews of Modern Physics. 55 (3): 601?644 (1983).

[9] M. Cook: *Universality in Elementary Cellular Automata*, Complex Systems. 15 (1) (2004).

[10] E. W. Dijkstra: *A Note on Two Problems in Connexion with Graph*, Numerische Mathematik 1, 269 - 271 (1959).

[11] PEP 318 – Decorators for Functions and Methods

[12] PEP 589 – TypedDict: Type Hints for Dictionaries with a Fixed Set of Keys

[13] Customizing Matplotlib with style sheets and rcParams

[14] Matplotlib example gallery

[15] U.S. Census Bureau: *A Compass for Understanding and Using American Community Survey Data*, U.S. Department of Commerce (2008).

[16] Public Use Microdata Sample (PUMS), Accuracy of the Data (2017)

[17] P. Tchebichef: *Des valeurs moyennes*, Journal de Mathématiques Pures et Appliquées, 2. 12: 177–184 (1867).

[18] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, E. Teller: *"Equation of State Calculations by Fast Computing Machines"*, Journal of Chemical Physics, Vol. 21, p.1087-1092 (1953)

[19] J. Jensen: *Sur les fonctions convexes et les inégalités entre les valeurs moyennes*, Acta Mathematica. 30 (1): 175?193 (1906)

[20] B. B. Mandelbrot: *The Fractal Geometry of Nature* New York: W. H. Freeman. ISBN 0-7167-1186-9 (1982).

[21] PEP 465 – A dedicated infix operator for matrix multiplication

[22] Philip D. Straffin: *Linear Algebra in Geography: Eigenvectors of Networks*, Mathematics Magazine, Vol. 53, No. 5 (Nov., 1980), pp. 269-276.

[23] S. Brin and L. Page: *The anatomy of a large-scale hypertextual Web search engine*, Computer Networks and ISDN Systems. 30 (1–7): 107–117 (1998).

[24] U. Senanayake, M. Piraveenan, A. Zomaya: *The Pagerank-Index: Going beyond Citation Counts in Quantifying Scientific Impact of Researchers*, PLoS ONE 10(8): e0134794 doi.