Introduction to Python for Data Analysis

How to manipulate and represent data with python

Romain Madar - DU Data scientist, PFA Master, IMAPP Master

Laboratoire de Physique de Clermont Auvergne (LPCA CNRS/IN2P3), UCA – FRANCE

Contact: romain.madar@clermont.in2p3.fr

July 10, 2024

Contents

Preamble

These notes contain the material for a python lecture proposed for the master PFA, for the Data scientist University Diploma (DU), and the master iMAAP, hosted at Université Clermont-Auvergne (UCA). Basic python knowledge is not required but would be very valuable. However, it is better to know about some basic mathemtatics like simple vectorial operation or statistics.

data scientist university degree proposed at Université Clermont-Auvergne (UCA). No prerequisite knowledge is assumed but being familiar with one progamming language might be useful. It is better to know about some basic mathemtatics like simple vectorial operation or statistics. All the material of this lecture can be found in this github repository.

This lecture is only a support to help you doing things *yourself*. As any other language, you must practice it if you want to progress. If you don't write and test code on your own, this lecture is close to be useless. I am available for any questions or general feedback on this lecture, so feel free to contact me: romain.madar@clermont.in2p3.fr.

General scope of the lecture

Python offers a rapidly evolving ecosystem to perform data analysis and it is both out of scope and hopeless to be extensive in this lecture. The main goal is therefore to make people familiar with the basic of python and data analysis tools in order to *make them able to extend their knowledge on themself*. Object oriented programming is not presented in this lecture. Pratical exercises are also available to provide few working examples as a starting point.

What this lecture is? A *basic* and *practical* introduction to python together with some of the most important data analysis tools namely `numpy`, `matplotlib` and `pandas`.

What this lecture isn't? Neither a formal introduction to python, nor a extensive demonstration of all features available in the tools mentioned above.

Content of the lecture

There are a lot of information in this lecture. In order to help you to focus on important aspect, each chapter start with a list of expected skills that you should take away, ranked with three levels: *basic*, *medium*, *expert*.

1. Introduction to Python. This first section is dedicated to basic object type and operation in python. Fonctions will also be described but object oriented programming will not be covered – *online notebook*

2. Introduction to numpy. Differences between usual python objects and numpy objects will be introduced – *online notebook*

3. Three tools to know. This section gives a glimpse of `matplotlib`, `pandas` and `scipy` packages allowing powerful data analysis – *online notebook*

4. Multidimensional data manipulation. Non-trivial operation for multidimensional data using the full power of numpy. Most of these operation can be performed with existing tools but it is intructive to do it once with native numpy – *online notebook*

5. Introduction to image processing. Very first steps of image processing (definition, plotting, operation) including basic filters application (noising, sharpen, border detection) – *online notebook*

Other practical examples. Depending on the remaining time (and the people taste), we can go through different topics among the following ones. Some of them can be also used as a project performed by students.

- Fourier analysis
- Principal component analysis (PCA)
- Random Forest regression
- Gaussian processes

How to get prepared

1. Get familiar with python. I would recommand two links: w3school tutorial (both basic and complete) and https://www.learnpython.org (code can be ran directly within your web browser).

2. Install python with anaconda. In order to run python on your own machine, you should install it. I would recommand anaconda for this, which also includes jupyter-notebook.

3. Install git. This is a versioning software which can be installed following these instructions. This whole repository can be *cloned* using `git clone https://github.com/rmadar/lecture-python` command.

4. Get familiar with notebooks. This represents a nice environement combining codes, notes and plots. This is very powerful to learn something and play with it. You can checkout this video or this post.

Chapter 1

Practical introduction to Python

> **Skills to take away**
>
> - *Basic*: int/float/str, list/dictionary, indexing/slicing, loops, functions, reading/writing files
> - *Medium*: docstring, comprehension, zip()/enumerate(), sorting dictionary
> - *Expert*: packing/unpacking, parsing file with correct casting, basic plotting

1.1 General Information

Python can be installed using Anaconda. Jupyter Notebook (also included with Anaconda) is probably the easiest way to follow this lecture and make your own notes. The goal of this first chapter is to provide a very quick introduction to the basics, but practice is mandatory to get comfortable with Python objects and syntax. You can practice using a web browser only at LearnPython.org. A more complete tutorial (though not interactive) can be found at W3Schools Python Tutorials. I recommend following the last tutorial up to the "Arrays" section.

In Python, there is one instruction per line. Variable assignment is done with =, and indentation is used to group instructions together under a loop or a condition block; there are no brackets as in C++. Comments (uninterpreted text) start with #. Importation of external modules or functions can be done in three different ways: `import module`, `import module as m`, or `from module import this_function`.

In the following example, the result of the command will be printed so that people can check that the computer is doing what is expected. The instruction `print(x)` will print the content of `x`. If several variables are printed, it is convenient to use the `print('x={} and y={}'.format(x, y))` syntax, which will print `x` and `y` in bracket fields with one command, even if they have different types.

Note: For Python versions greater than 3.6, we can also use *f-strings* which simplify the print commands. This works as follows:

```
print(f'x={x} and y={y}')
```

where the `x` and `y` in brackets are actual Python variables.

1.2 Object types

Numbers: There are three types of numbers: int, float, and complex. The usual operations (+, -, *, /) are available. In addition, there is also `a**b` (which means a^b), `a // b`, and `a % b` (which are the results of integer divisions - see example below).

```
# Basic numbers and operations
a = 2
b = 3.14
print(a+b)
print(a**b)
```

```
5.140000000000001
8.815240927012887
```

```
# Complex numbers and power
a = 1j
a**2
```

```
(-1+0j)
```

```
# Integer division example (// and % operators)
a , b = 10, 4
divisor, rest = a//b, a%b
print('{} = {}x{} + {}'.format(a, divisor, b, rest))
```

```
10 = 2x4 + 2
```

Strings: Strings allow for the manipulation of words, sentences, or even text with specific methods. Strings are also Python lists, and list methods work with them as well (see below). Common and useful string manipulations include counting the number of letters with `len(word)` or manipulating a collection of words using `sentence.split(' ')`. Many methods exist, which can be explored by typing `help(str)` in a Python terminal or a Jupyter Notebook.

```
w1 = 'hello'
print(w1, len(w1), w1[3])
```

```
hello 5 l
```

```
# Summing two strings is possible (all other operators don't work)
blank, w2 = ' ', 'world'
sentence = w1 + blank + w2
print(sentence)
```

```
hello world
```

```
# Multiplying a string by an integer is also possible
repetition = sentence*3
print(repetition)
```

```
hello worldhello worldhello world
```

```
# Get a list of words from a sentence (cf. below for list objects)
s = 'It is rainy today'
list_words = s.split(' ')
print(list_words)
```

```
['It', 'is', 'rainy', 'today']
```

```
# Loop over the words and get the number of letters
for w in s.split(' '):
    print(w, len(w))
```

```
It 2
is 2
rainy 5
today 5
```

1.3 Object Collections

There are four types of collections in Python, each with its own characteristics:

- **List**
- **Dictionary**
- **Tuple**
- **Set**

The most commonly used collections are lists and dictionaries. Sets are unordered, and tuples cannot be modified. Common methods for collections include:

- Number of items: `len(x)`
- Loop over items with `for element in x:`
- Check if an item is in the collection: `element in x`

Lists: Lists are one of the most frequently used collection objects in Python because they are the next-to-simplest level after individual variables. A Python list is a list of objects with possibly different types. You can search, loop, and count with lists. You can also add two lists or multiply a list by an integer, which performs concatenation or duplication (these concepts will be important for NumPy arrays). Indexing elements is a convenient way to access specific information. You can access the i^{th} element with `my_list[i]` or retrieve

a sub-list with `my_list[i:j]`. You can also take only one element every `n` with `my_list[i:j:n]` (more precisely, this takes elements of index $i + p \times n$ until j, with $p = 0, 1, 2, ...$). With this syntax, reversing the order of the list is easy: `reverted_list = my_list[::-1]`, where empty variables are default values (namely `0` and `len(my_list)`).

```
# Defining a list and access basic information
my_list1 = [1, 3, 4, 'banana']
print('Second element is {}'.format(my_list1[1]))
print('Number of elements: {}'.format(len(my_list1)))
print('Is \'banana\' in the list? {}'.format('banana' in my_list1))
```

```
Second element is 3
Number of elements: 4
Is 'banana' in the list? True
```

```
# Sum of two lists
my_list2 = ['string', 1+3j, [100, 1000]]
my_list = my_list1 + my_list2
print(my_list)
```

```
[1, 3, 4, 'banana', 'string', (1+3j), [100, 1000]]
```

```
# List multiplied by an integer
my_list = my_list*2
print(my_list)
```

```
[1, 3, 4, 'banana', 'string', (1+3j), [100, 1000], 1, 3, 4, 'banana', 'string', (1+3j), [100,
1000]]
```

```
# Looping over list element and print the type of seven first elements in the reversed order.
for element in my_list[6:0:-1]:
    print('{} is {}'.format(element, type(element)))
```

```
[100, 1000] is <class 'list'>
(1+3j) is <class 'complex'>
string is <class 'str'>
banana is <class 'str'>
4 is <class 'int'>
3 is <class 'int'>
```

Sets and Tuples

Tuples and sets are modified versions of Python lists. Tuples are ordered but cannot be modified (no assignment, no addition), while sets are not ordered but can be modified. In this context, "order" means indexing (so `x[i:j:n]` syntax, among others). Searching or looping over elements works in the same way as for lists.

```
# Tuple
t = (1, 3, 7)
print(t)

# Access the third element
print(t[2])

# Try to modify the second element - using the 'try - except' synthax
try:
    t[1] = 'hello'
except TypeError:
    print('Impossible to change the value of a tuple')
```

```
(1, 3, 7)
7
Impossible to change the value of a tuple
```

Sets can be modified with methods like `s.add(x)` or `s.update([x, y])`.

```
# Set
s = {'apple', 'banana', 'orange'}
print(s)

# Add one element
s.add('pineapple')
print(s)

# Add a list
s.update(['pear', 'prune'])
print(s)
```

```
{'orange', 'banana', 'apple'}
{'pineapple', 'orange', 'banana', 'apple'}
{'orange', 'pineapple', 'apple', 'pear', 'prune', 'banana'}
```

```
# Try to access the second element - using the 'try - except' synthax
try:
    print(s[1])
except TypeError:
    print('Impossible to access element via indexing')
```

```
Impossible to access element via indexing
```

Dictionaries: Various object types are important for manipulating and organizing data. The most common one is the dictionary, which works with pairs of (`key`, `value`). The `key` must be a non-modifiable object, typically a string or an integer, but it cannot be a list. This is a powerful concept for storing different types of information within the same object. One can easily loop, search, modify a given key's value, or even add a new key quite easily.

```
# dictionary
person = {'name': 'Charles', 'age': 78, 'size': 173, 'gender': 'M'}
print(person)
```

```
{'name': 'Charles', 'age': 78, 'size': 173, 'gender': 'M'}
```

```
# Accessing value using the key
template = '{} ({}) is {} years old and is {} cm'
print(template.format(person['name'], person['gender'], person['age'], person['size']))
```

```
Charles (M) is 78 years old and is 173 cm
```

```
# Adding a key and its value
person['eyes'] = 'blue'
print(person)
```

```
{'name': 'Charles', 'age': 78, 'size': 173, 'gender': 'M', 'eyes': 'blue'}
```

```
# Test if a key is present
print('name' in person)
print('brand' in person)
```

```
True
False
```

1.4 Loops

Loops are at the core of programming, especially for data analysis-oriented tasks. There are two ways of repeating an instruction several times: the `for` loop and the `while` loop. Several instructions are common to both loops, such as `continue` (skip the instruction after and switch to the next element) or `break` (stop the loop), but the use cases for these two ways are different.

For Loops: For data analysis, these are among the most commonly used loops. However, as we will see in the introduction to NumPy, for loops should not be used for heavy computations in Python. For loops are relevant for small (~1000) data samples (and computations). We'll come back to this point in the lecture. Below, a few examples are provided.

```
# Compute sum(i^2) for i from 0 to 9
x = 0
for i in range(0, 10):
    x += i**2
print(x)
```

```
285
```

```
# Loop over fruit via a set and print only ones with a 'p'
for fruit in s:
    if 'p' in fruit:
        print(fruit)
```

```
pineapple
apple
pear
prune
```

There are several ways to loop over dictionaries, depending on how you want to access the information. You can access information by keys, values, or both. Below, examples of each are provided.

```
# Loop over keys for a dictionary and access the value of each
for properties in person:
    value = person[properties]
    print('{}: {}'.format(properties, value))
```

```
name: Charles
age: 78
size: 173
gender: M
eyes: blue
```

```
# Loop over dictionary values only
for v in person.values():
    print(v)
```

```
Charles
78
173
M
blue
```

```
# Loop over both keys and values directly
for key, value in person.items():
    print('{}: {}'.format(key, value))
```

```
name: Charles
age: 78
size: 173
gender: M
eyes: blue
```

Tip: How to Sort a Dictionary?

It can be noted that dictionaries are natively *not ordered*. This means that you cannot access an item with its index since there is no index. However, it can be convenient to sort dictionary keys according to their values using the general Python function `sorted(collection, key=function)` and a type of object called `OrderedDict` from the `collections` module, as explained below.

```
# Define a dictionary
students_marks = {
    'Jean': 12,
    'Chloe': 17,
    'Olivier': 8,
    'Helene': 10
    }

# Print the key, values items
for name, mark in students_marks.items():
    print(name, mark)
```

```
Jean 12
Chloe 17
Olivier 8
Helene 10
```

The `sorted()` function works on any type of collection that can be looped over (called an *iterable*). It requires the collection and a function that returns a key on which to sort for each object in the collection. This key can be, for example, a letter or a number. Let's try both by defining a function that retrieves either the mark or the first letter of the name from a dictionary item `(name, mark)`.

```
# Order it by increasing marks
def get_mark(item):
    return item[1]

# Or by the first letter of the name
def get_name(item):
    return item[0][0]

# Testing with an item
item_test = ('Jacques', 12)
print('{} has a mark of {} and {} as 1st letter'.format(item_test, get_mark(item_test),
↪  get_name(item_test)))
```

```
('Jacques', 12) has a mark of 12 and J as 1st letter
```

We can now apply the `sorted()` function to the collection of items in the initial dictionary. This will return a collection of sorted items that can later be converted into a dictionary. The last step depends on the Python version. In Python 2.5, one has to use `OrderedDict` from the `collections` module, while it's not needed in Python 3. The version of Python can be dynamically checked with `sys.version_info` from the `sys` module.

```
# Get all items and sort them by increasing mark
all_items = students_marks.items()
items_sorted_by_marks = sorted(all_items, key=get_mark)
items_sorted_by_names = sorted(all_items, key=get_name)
```

```
# Check the version of python
import sys
version = sys.version_info[0]
isPython2 = version == 2
```

```
# Final conversion of collection of items into a dictionary
if isPython2:
    from collections import OrderedDict
    marks_sorted_dict = OrderedDict(items_sorted_by_marks)
    names_sorted_dict = OrderedDict(items_sorted_by_names)

else:
    marks_sorted_dict = {k: v for k, v in items_sorted_by_marks}
    names_sorted_dict = {k: v for k, v in items_sorted_by_names}
```

```
# Check the mark sorted results:
for k, v in marks_sorted_dict.items():
    print(k, v)
```

```
Olivier 8
Helene 10
Jean 12
Chloe 17
```

```
# Check the name sorted results:
for k, v in names_sorted_dict.items():
    print(k, v)
```

```
Chloe 17
Helene 10
Jean 12
Olivier 8
```

While Loops

While loops are a bit less commonly used in practice, but they are described here for completeness. The idea behind a while loop is to repeat a given instruction until a certain condition is met.

```
# Cast (ie change type) the set s into a list
my_list = list(s)
```

```
# Remove item one by one until there are no items left.
while len(my_list)>0:
    my_list.pop()
    print(my_list)
```

```
['orange', 'pineapple', 'apple', 'pear', 'prune']
['orange', 'pineapple', 'apple', 'pear']
['orange', 'pineapple', 'apple']
['orange', 'pineapple']
['orange']
[]
```

1.5 Few Python synthax tips

Comprehension: This is the action of building a collection with one line of code. The comprehension syntax works for all collections, with conditions, or even nested loops (loops within loops). Below, a few examples are provided.

```
# List
list_squares = [i**2 for i in range(1, 10)]
print(list_squares)

# dictionary
dict_squares = {i:i**2 for i in list_squares[0:5]}
print(dict_squares)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
{1: 1, 4: 16, 9: 81, 16: 256, 25: 625}
```

```
# Comprehension list with a condition (e.g. keep only even numbers)
list_even = [i for i in range(0, 10) if i%2==0]
print(list_even)
```

```
[0, 2, 4, 6, 8]
```

```
# Comprehension with nested loops
sum_integers = [i*10+j for i in range(0,5) for j in range(0, 5)]
print(sum_integers)
```

```
[0, 1, 2, 3, 4, 10, 11, 12, 13, 14, 20, 21, 22, 23, 24, 30, 31, 32, 33, 34, 40, 41, 42, 43,
44]
```

Looping with `enumerate` and `zip`:

The keyword `enumerate` returns a counter together with the element in the loop. This is useful if you need to count the number of iterations of the loop. You can achieve this without `enumerate`, but you would need to add two lines (initialization of the counter and incrementation).

```
# Position of each word in a sentence
sentence = 'I would like to analyse this sentence in term of word position'
words = sentence.split(' ')
for i, w in enumerate(words):
    print(w.ljust(10) + ':  ' + str(i))
```

```
I         :  0
would     :  1
like      :  2
to        :  3
analyse   :  4
this      :  5
sentence  :  6
in        :  7
term      :  8
of        :  9
word      :  10
position  :  11
```

The `zip(list1, list2)` syntax allows you to form pairs using elements from each list at the *same position*. This is convenient for associating objects stored in different collections in a quick and readable way. If `list1` and `list2` do not have the same size, the minimum of the two lengths is taken. Furthermore, the `zip()` command can take more than two collections and return a group of elements with a size equal to the number of collections.

```
# Associate fruits and colors
fruits = ['banana', 'orange', 'pineapple', 'pear', 'prune']
colors = ['yellow', 'orange', 'brown', 'green', 'purple']
for f, c in zip(fruits, colors):
    print('{} is {}'.format(f, c))
```

```
banana is yellow
orange is orange
pineapple is brown
pear is green
prune is purple
```

```
# Using zip() with three lists
l1, l2, l3 = range(0, 10), range(0, 100, 10), range(0, 1000, 100)
for i, j, k in zip(l1, l2, l3):
    print(i, j, k, i+j+k)
```

```
0 0 0 0
```

```
1 10 100 111
2 20 200 222
3 30 300 333
4 40 400 444
5 50 500 555
6 60 600 666
7 70 700 777
8 80 800 888
9 90 900 999
```

1.6 Functions

Functions are defined as a set of instructions encapsulated into one object. This is particularly convenient when one has to perform the same list of instructions several times. A good guideline to know when to write a function could be:

> If you copy-paste the same piece of code more than two times, then make a function.

Definition: A function takes some arguments, performs some instructions, and returns a result. The syntax to define and call a function is shown below. In Python, a function must be defined *before* being used (unlike C++ where it can be different, as soon as the function is declared). This makes the concept of a *package* quite relevant to wrap up several functions into a Python file that can be imported into the main code.

```
# Definition syntax
def function(argument):
    result = argument * 3
    return result

# Call syntax
function(2)
```

```
6
```

The type of the argument is not fixed (since it is a general feature of Python), so the same instruction will be interpreted differently depending on the type. The following example demonstrates the different results of the function above for two argument types.

```
# Print the result for two types of arguements
print('function(10) = {}'.format(function(10)))
print('function(\'ouh\') = {}'.format(function('ouh')))
```

```
function(10) = 30
function('ouh') = ouhouhouh
```

```
def person_printer(p):
    template = '{} ({}) is {} years old and is {} cm'
    print(template.format(p['name'], p['gender'], p['age'], p['size']))
    return


def grow_old(p, n_years):

    # 1. copy the dictionary person (otherwise p *will* be modified)
    res = p.copy()

    # 2. Compute the new age and size
    new_age = res['age'] + n_years
    new_size = res['size'] - n_years*0.13

    # 3. Assign the new age/size to the result
    res['age'] = new_age
    res['size'] = new_size

    # 4. Return the result
    return res
```

```
# Print before growing old
person_printer(person)

# Growing old
old_guy = grow_old(person, 10)

# Print after growing old
person_printer(old_guy)
```

```
Charles (M) is 78 years old and is 173 cm
Charles (M) is 88 years old and is 171.7 cm
```

Docstring:

A docstring offers the possibility to document your code in a proper way, which is quite useful for others (and for you when you will reuse code after several years). It is then a good practice to do so, even if it takes time. This documentation can be accessed using the command `help(function)` or by using the keyboard shortcut `Shift+Tab` in Jupyter Notebook (when the cursor is after the opening parenthesis of the function). The syntax to add a docstring is `'''My documentation'''` at the very beginning of the function.

```
def grow_old(p, n_years):
    '''
    Take a person dictionary and update the age and the size to make the person older.

    Parameters
    ----------
```

```
    p: dictionary
        Person object as defined earlier in the code, with at least 'age' (year)
        and 'size' (cm) keys, to get older.
    n_years: integer
        Number of years to be added to the age of the person.

    Returns
    -------
    person: dictionary
        Person object as defined earlier in the code with age and size updated as
        age  -> age + n_years
        size -> size - n_years * 0.13
    '''

    # 1. copy the dictionary person (otherwise p will be modified, which might problematic)
    res = p.copy()

    # 2. Compute the new age and size
    new_age = res['age'] + n_years
    new_size = res['size'] - n_years*0.13

    # 3. Assign the new age/size to the result
    res['age'] = new_age
    res['size'] = new_size

    # 4. Return the result
    return res
```

```
help(grow_old)
```

```
Help on function grow_old in module __main__:

grow_old(p, n_years)
    Take a person dictionary and update the age and the size to make the person older.

    Parameters
    -----
    p: dictionary
        Person object as defined earlier in the code, with at least 'age' (year)
        and 'size' (cm) keys, to get old.
    n_years: integer
        Number of years to be added to the age of the person.

    Return
    ---
    person: dictionary
       Person object as defined earlier in the code with age and size updated as
         age  -> age+n_years
         size -> size-n_years*0.13
```

There are several ways to organize the docstring, and the example above is based on the NumPy docstring style. Note that docstrings can also be added to a module (in practice, a Python file) to document the content, goal, and usage of that module.

Arbitrary number of arguments: `*args` **and** `**kwargs`

The examples above are relatively simple, and generally, a function takes several arguments. Sometimes it is even convenient to have an unfixed number of arguments, so that the function is more flexible as the code grows. Python offers a nice way to define such a function thanks to the *packing* and *unpacking* notion, which is described right below.

Aside: Packing and Unpacking

In short, this is the possibility to convert a list into a series of objects (unpacking) or vice versa (packing). This way of writing collections makes code development very concise and fast, especially when calling functions with several arguments in a nice way. This also allows you to define functions with an arbitrary number of arguments, as already mentioned. The following dummy function is used to illustrate the concept of packing/unpacking with both a list and a dictionary.

```
# Test function
def mean(a, b, c):
    return (a+b+c)/3.
```

It is possible to use a list of three numbers to specify the argument values of the `mean(a, b, c)` function, using the unpacking syntax for a list: `*list`. This is demonstrated below:

```
# Packing & unpacking with a list (or a tuple): *list
my_numbers = [10, 12, 15]
mean(*my_numbers)
```

```
12.333333333333334
```

It is also sometimes convenient to call the arguments by their names (mostly to make the code more readable). These types of arguments are called *keyword arguments* and can be packed/unpacked into a dictionary. Each argument name is a key of this dictionary, and the value is the value passed to the function. The unpacking is done with `**dict`.

```
# Packing & unpacking with a dictionary: **dict
my_numbers = {'a': 10, 'b': 12, 'c': 15}
mean(**my_numbers)
```

```
12.333333333333334
```

Coming back to the initial motivation, i.e., having an arbitrary number of arguments, it is possible to define such a function as follows - which in this case just prints the number and the list of arguments:

```
# Function definition with *args
def test_function(*args):
    print('There are {} arguments: '.format(len(args)))
    for a in args:
        print('  -> {}'.format(a))
    print('')
    return
```

```
# Test with different numbers/types of arguments
test_function()
test_function('hoho')
test_function('hoho', 3)
test_function('hoho', 3, [1,'banana'], {'mood': 'happy', 'state': 'holidays'})
```

```
There are 0 arguments:

There are 1 arguments:
  -> hoho

There are 2 arguments:
  -> hoho
  -> 3

There are 4 arguments:
  -> hoho
  -> 3
  -> [1, 'banana']
  -> {'mood': 'happy', 'state': 'holidays'}
```

```
# Function definition with kwargs
def test_function(**kwargs):
    print('There are {} arguments: '.format(len(kwargs)))
    for k, v in kwargs.items():
        print(' {}={}'.format(k, v))
    print('')
    return
```

```
test_function()
test_function(x='hoho')
test_function(word='hoho', multiplicity=3)
test_function(a='hoho', N=3, shopping=[1,'banana'], feeling={'mood': 'happy', 'state':
↪  'holidays'})
```

```
There are 0 arguments:

There are 1 arguments:
 x=hoho
```

```
There are 2 arguments:
 word=hoho
 multiplicity=3

There are 4 arguments:
 a=hoho
 N=3
 shopping=[1, 'banana']
 feeling={'mood': 'happy', 'state': 'holidays'}
```

This can be used to declare an argument in a very readable and concise way. This might be helpful for some cosmetic arguments of plots that can be common to several plots (but not all). We'll see some concrete examples later in the lecture. In the meanwhile, here is the equivalent of the last call from the code above:

```
# Pack all keyword arguments in a dictionary first
my_args = {
           'a': 'hoho',
           'N': 3,
           'shopping': [1,'banana'],
           'feeling': {'mood': 'happy', 'state': 'holidays'}
          }

# Then call the function
test_function(**my_args)
```

```
There are 4 arguments:
 a=hoho
 N=3
 shopping=[1, 'banana']
 feeling={'mood': 'happy', 'state': 'holidays'}
```

1.7 File manipulation

File handling is quite important since it enables interaction between your code and input/output data (called I/O). There are several features related to file handling in Python, and this short section just gives a few basic practices.

Open/close a file

Python has native methods to open and close files. While closing a file doesn't allow for many variations, the opening can be done in different modes, depending on whether we want to read, write, or append to the opened file. The basic syntax is:

```
# Open
f = open('my_file_name.txt', option)
```

```
# Close
f.close()
```

where `option` is a one-letter string that can be: - `r` (read, default): to just read the file - `a` (append): to add content at the end of an existing file - `w` (write): to write content to a file (it creates a new file) - `x` (create): to create a new file

Write a file.

```
# Text to be written (can be one line string 'my text' or multiple lines string - docstring)
text = '''Gervaise avait attendu Lantier jusqu'à deux heures du matin. Puis,
toute frissonnante d'être restée en camisole à l'air vif de la fenêtre,
elle s'était assoupie, jetée en travers du lit, fiévreuse, les joues
trempées de larmes.
'''

# Open in write mode
f = open('test.txt', 'w')

# Write string
f.write(text)

# Close
f.close()
```

Read a File: The following example loads the previous file and loops over each line to analyze its content. One can note several issues: first, one sentence can be on two lines, and second, each end of the line contains a `\n` (which is an invisible character meaning "go to the next line"). There is a method to clean a line, called `line.strip()`, which removes all spaces and invisible characters, unless specified otherwise - see `help(str.strip)`.

```
# Open the file in read mode
f = open('test.txt', 'r')

# Loop over the lines
for line in f:

    # Print a header to make the ouput clearer
    print('\n\n{}'.format('='*50))

    # Print the line as it is given
    print('This line: {}'.format(line))

    # Split by '.' to isolate sentence
    sentences = line.split('.')
    print('This line has {} sentences: '.format(len(sentences)))

    # Split each sentence by ' ' to isolate words
```

```
    for i,s in enumerate(sentences):
        sclean = s.strip()
        words = sclean.split(' ')
        print(' - sentence {}: {}'.format(i, words))

f.close()
```

```
==================================================
This line: Gervaise avait attendu Lantier jusqu'à deux heures du matin. Puis,

This line has 2 sentences:
 - sentence 0: ['Gervaise', 'avait', 'attendu', 'Lantier', 'jusqu'à', 'deux', 'heures', 'du',
'matin']
 - sentence 1: ['Puis,']


==================================================
This line: toute frissonnante d'être restée en camisole à l'air vif de la fenêtre,

This line has 1 sentences:
 - sentence 0: ['toute', 'frissonnante', 'd'être', 'restée', 'en', 'camisole', 'à', 'l'air',
'vif', 'de', 'la', 'fenêtre,']


==================================================
This line: elle s'était assoupie, jetée en travers du lit, fiévreuse, les joues

This line has 1 sentences:
 - sentence 0: ['elle', 's'était', 'assoupie,', 'jetée', 'en', 'travers', 'du', 'lit,',
'fiévreuse,', 'les', 'joues']


==================================================
This line: trempées de larmes.

This line has 2 sentences:
 - sentence 0: ['trempées', 'de', 'larmes']
 - sentence 1: ['']
```

Rewrite a modified version of a file into a new file. It can be quite convenient to modify an existing file to correct a systematic mistake automatically or simply perform more complex operations. The example below shows how to remove all the "e" characters from the text below and write it to a new file.

```
# Open the in/out files
f_i = open('test.txt', 'r')
f_o = open('test_without_e.txt', 'w')

# Loop over line, remove all "e" for each, and write the result in the output file
```

```
for line in f_i:
    line_without_e = line.replace('e', '') # replace "e" by nothing
    f_o.write(line_without_e)

# Close all files
f_i.close()
f_o.close()
```

```
# Open in read mode and check the result
f = open('test_without_e.txt', 'r')
print(f.read())
```

```
Grvais avait attndu Lantir jusqu'à dux hurs du matin. Puis,
tout frissonnant d'êtr rsté n camisol à l'air vif d la fnêtr,
ll s'était assoupi, jté n travrs du lit, fiévrus, ls jous
trmpés d larms.
```

Read a CSV File to Get Data:

This use case is quite important because it allows converting a file with data into variables accessible in the code (for some computations, plotting, etc.). One of the most basic formats to store data is called *CSV* (for comma-separated values), which can be imported/exported from any spreadsheet software (like Excel). This format is not necessarily appropriate for large datasets but is quite useful in a large number of situations, and one must be able to manipulate it easily, as shown in the example below.

Creation of a CSV File on the Fly Using a Docstring

```
# Data taken from kaggle: https://www.kaggle.com/jolasa/waves-measuring-buoys-data-mooloolaba
data_csv_format = '''index,date,height,heightMax,period,energy,direction,temperature
1,01/01/2017 00:00,-99.9,-99.9,-99.9,-99.9,-99.9,-99.9
2,01/01/2017 00:30,0.875,1.39,4.421,4.506,-99.9,-99.9
3,01/01/2017 01:00,0.763,1.15,4.52,5.513,49,25.65
4,01/01/2017 01:30,0.77,1.41,4.582,5.647,75,25.5
5,01/01/2017 02:00,0.747,1.16,4.515,5.083,91,25.45
6,01/01/2017 02:30,0.718,1.61,4.614,6.181,68,25.45
7,01/01/2017 03:00,0.707,1.34,4.568,4.705,73,25.5
8,01/01/2017 03:30,0.729,1.21,4.786,4.484,63,25.5
9,01/01/2017 04:00,0.733,1.2,4.897,5.042,68,25.5
10,01/01/2017 04:30,0.711,1.29,5.019,8.439,66,25.5
11,01/01/2017 05:00,0.698,1.11,4.867,4.584,64,25.55
12,01/01/2017 05:30,0.686,1.14,4.755,5.211,56,25.55
13,01/01/2017 06:00,0.721,1.12,4.843,5.813,67,25.5
14,01/01/2017 06:30,0.679,1.22,4.948,4.71,81,25.45
15,01/01/2017 07:00,0.66,1.08,5.068,5.353,90,25.45
16,01/01/2017 07:30,0.662,1.18,5.263,7.436,67,25.4
17,01/01/2017 08:00,0.653,1.21,5.007,6.001,90,25.45
18,01/01/2017 08:30,0.665,1.17,4.952,6.414,90,25.55
19,01/01/2017 09:00,0.684,1.55,5.022,6.691,88,25.6
```

```
20,01/01/2017 09:30,0.679,1.09,4.926,6.804,88,25.65
21,01/01/2017 10:00,0.667,1.12,4.928,6.641,122,25.75
22,01/01/2017 10:30,0.688,1.13,4.808,5.958,91,25.7
23,01/01/2017 11:00,0.644,0.99,4.559,6.691,92,25.9
'''

# Create csv file using these data
f = open('wave_data.csv', 'w')
f.write(data_csv_format)
f.close()
```

Reading the CSV file and storing values in Python objects:

In this example, we'll see how to store all information about the wave in a list of dictionaries.

```
# Open the file in read mode
f = open('wave_data.csv', 'r')

# Get the first line (calling the readline() method once) to extract the feature names.
features = f.readline().strip().split(',')
print(features)

# Loop over lines and store the information
data = []
for l in f:
    values = l.strip().split(',')
    data_single_wave = {var: val for var, val in zip(features, values)}
    data.append(data_single_wave)
```

```
['index', 'date', 'height', 'heightMax', 'period', 'energy', 'direction', 'temperature']
```

```
# helper function for a nice printing
def print_wave(w):
    tmp = 'Wave {} ({}) had a heigh of {}m with a temperature of {} degree'
    print(tmp.format(w['index'], w['date'], w['height'], w['temperature']))

# Print the first 5 waves
for wave in data[:5]:
    print_wave(wave)
```

```
Wave 1 (01/01/2017 00:00) had a heigh of -99.9m with a temperature of -99.9 degree
Wave 2 (01/01/2017 00:30) had a heigh of 0.875m with a temperature of -99.9 degree
Wave 3 (01/01/2017 01:00) had a heigh of 0.763m with a temperature of 25.65 degree
Wave 4 (01/01/2017 01:30) had a heigh of 0.77m with a temperature of 25.5 degree
Wave 5 (01/01/2017 02:00) had a heigh of 0.747m with a temperature of 25.45 degree
```

At this stage, the problem is that the type of objects stored is a string and not numbers, so no computation can be made. Typically, the following code will crash because the division between strings is not defined:

```
# Compute the average height
heights = [w['height'] for w in data]
average = sum(heights) / len(heights)
```

One has to cast (i.e., change the type) the objects stored in the dictionaries. They can all be cast as floats, but not the date. The index also makes more sense as an integer. So the following can work:

```
# Manage string to time object conversion
def str_to_time(date_str):
    import datetime
    return datetime.datetime.strptime(date_str, '%d/%m/%Y %H:%M')

# Container with properly converted data
DATA = []

# Loop over waves and make the proper conversion depending on the feature name
for w in data:
    wgood = w.copy()
    for k in w:
        if k == 'index':   # Cast string into an integer
            wgood[k] = int(w[k])
        elif k == 'date':  # cast string into a datetime object
            wgood[k] = str_to_time(w[k])
        else:              # cast string into a float
            wgood[k] = float(w[k])
    DATA.append(wgood)
```

Computing the average height of waves now works, but it gives a quite strange result:

```
# Compute the average height
heights = [w['height'] for w in DATA]
average = sum(heights)/len(heights)
print('Averaged waveheight is {:.1f} m'.format(average))
```

```
Averaged waveheight is -3.7 m
```

This is due to the first row, which has all values at -99. Removing it (using indexing techniques) gives a more sensible result:

```
heights = [w['height'] for w in DATA[1:]]
average = sum(heights)/len(heights)
print('Averaged waveheight is {:.1f} m'.format(average))
```

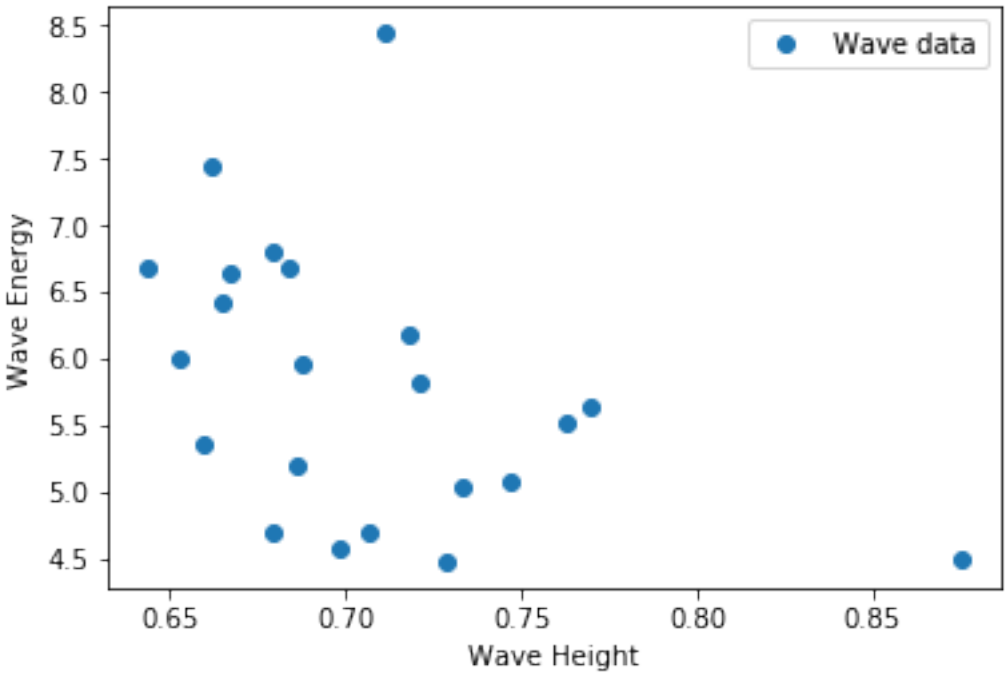
```
Averaged waveheight is 0.7 m
```

1.8 Plotting Data: The Very First Step

Graphical representation of data is a key element of data analysis. It allows us to gain intuition and generate ideas or simply perform visual checks. You might encounter the terminology "Exploratory Data Analysis (EDA)" in the literature, which corresponds to plotting data in all possible ways to *extract exploitable information* from the data. EDA is an entire field that we will not cover in this lecture. Instead, we will provide some basic examples that will serve as a starting point to expand your knowledge. The standard library to produce plots is `matplotlib`, but there are many other tools that we will not introduce, such as `seaborn`, `bokeh` for browser-interactive plots, `cartopy` for geographic data analysis/plots, etc.

```
# Prepare data to plot: wave height v.s. wave energy (removing point with -99 values)
height = [w['height'] for w in DATA if w['height']>-99]
energy = [w['energy'] for w in DATA if w['energy']>-99]
```

```
# Import the key part of the package: pyplot
import matplotlib.pyplot as plt

# Display matplotlib output in the notebook
%matplotlib inline

# Call the simplest function to plot x vs y
plt.plot(height, energy, linewidth=0, marker='o', label='Wave data')

# Set x and y axis axis labels
plt.xlabel('Wave Height')
plt.ylabel('Wave Energy')

# Adding a legend based on label keyword
plt.legend();
```


Wave data
Wave Energy
Wave Height
8.5
8.0
7.5
7.0
6.5
6.0
5.5
5.0
4.5
0.65
0.70
0.75
0.80
0.85

Chapter 2

Basic introduction to NumPy

> **Skills to take away**
>
> - **Basic**: n-dimensional arrays (dimension, shape, size), organization of data along axes, element-wise operations
> - **Medium**: n-dimensional slicing, fancy indexing, basic broadcasting (no new axis), linspace/arange
> - **Expert**: general broadcasting, n-dimensional random arrays (`np.random`)

2.1 Motivations

Why NumPy? NumPy stands for *Numerical Python* and is highly optimized (and therefore fast) for computations in Python. NumPy is one of the core packages on which many others are based, such as SciPy (for *Scientific Python*), Matplotlib, or Pandas. Many other scientific tools are also based on NumPy, which justifies having at least a basic understanding of how it works. However, one might also ask, *why Python* in the first place?

Why Python? Depending on your preferences and purposes, Python can be a very good option (or not - this language has pros and cons, like any other). In any case, many tools are available in Python that cover a broad spectrum of applications, from machine learning to web design or string processing. Learning Python is definitely a good investment for general-purpose applications.

2.2 The core objects: arrays

The core of NumPy is the numpy array. These objects allow for efficient computations over large datasets in a concise way from a language point of view, and very fast from a processing time point of view. The price to pay is giving up explicit *for* loops. This leads to a somewhat counterintuitive logic - at first.

2.2.1 Main differences to usual Python lists

The first point is to differentiate a NumPy array from a Python list, as they don't behave in the same way. Let's define two python lists and their equivalent NumPy arrays.

```
import numpy as np
l1, l2 = [1, 2, 3], [3, 4, 5]
a1, a2 = np.array([1, 2, 3]), np.array([3, 4, 5])
print(l1, l2)
```

```
[1, 2, 3] [3, 4, 5]
```

First of all, all mathematical operations act element-wise in a NumPy array. For a Python list, addition acts as concatenation of the lists, and multiplication by a scalar acts as replication of the lists.

```
# obj1+obj2
print('python lists: {}'.format(l1+l2))
print('numpy arrays: {}'.format(a1+a2))
```

```
python lists: [1, 2, 3, 3, 4, 5]
numpy arrays: [4 6 8]
```

```
# obj*3
print('python list: {}'.format(l1*3))
print('numpy array: {}'.format(a1*3))
```

```
python list: [1, 2, 3, 1, 2, 3, 1, 2, 3]
numpy array: [3 6 9]
```

One other important difference is the way to access elements of an array, known as slicing and indexing. Here, the behavior of Python lists and NumPy arrays is similar, except that NumPy arrays support a few additional features, such as indexing by an array of integers (which does not work for Python lists). Use cases for such indexing will be extensively illustrated in the next chapters.

```
# Indexing with an integer: obj[1]
print('python list: {}'.format(l1[1]))
print('numpy array: {}'.format(a1[1]))
```

```
python list: 2
numpy array: 2
```

```
# Indexing with a slicing: obj[slice(1,3)]
print('python list: {}'.format(l1[slice(1,3)]))
print('numpy array: {}'.format(l1[slice(1,3)]))
```

```
python list: [2, 3]
numpy array: [2, 3]
```

```
# Indexing with a list of integers: obj[[0,2]]
print('python list: IMPOSSIBLE')
print('numpy array: {}'.format(a1[[0,2]]))
```

```
python list: IMPOSSIBLE
numpy array: [1 3]
```

2.2.2 Memory management in Python and NumPy

When a list is created in Python (and NumPy), you might want to copy it and modify the copy to have both the unmodified original version and the modified copied version. However, this doesn't work as expected: the original list will be modified too. If you are not sure what is happening, you can try it out. Here is an example:

```
# Create a list
l1 = [1, 2, 3]

# Getting a copy and modifying it
l2 = l1
l2[1] = 10

# Print the two lists : both are modified
print(f'l1={l1}, l2={l2}')
```

```
l1=[1, 10, 3], l2=[1, 10, 3]
```

```
# Create an array
a1 = np.array([1, 2, 3])

# Getting a copy and modifying it
a2 = a1
a2[1] = 10

# Print the two arrays: both are modified
print(f'a1={a1}, a2={a2}')
```

```
a1=[ 1 10  3], a2=[ 1 10  3]
```

Explanation: Python works with memory addresses (called *pointers* in C). This means that `l1` and `l2` don't contain the data, but instead, they both contain the addresses in the computer memory to which the data are stored. Since the addresses of `l1` and `l2` are the same, they point to the same data, and any modification in `l2` will be seen in `l1`.

How to avoid this?

You must make a copy of the object. In NumPy, you can simply use the `a.copy()` command. In pure Python, there is a package called `copy` which can either make a copy or a `deepcopy()`. The difference is explained just after this example.

```
import copy
l1 = [1, 2, 3]
l2 = copy.copy(l1)
l2[1] = 10
print(f'l1={l1}, l2={l2}')
```

```
l1=[1, 2, 3], l2=[1, 10, 3]
```

```
a1 = np.array([1, 2, 3])
a2 = a1.copy()
a2[1] = 10
print(f'a1={a1}, a2={a2}')
```

```
a1=[1 2 3], a2=[ 1 10  3]
```

To understand the difference between `copy` and `deepcopy`, one needs to perform some tests on nested lists, i.e., a list of lists. `deepcopy` allows the copy to be propagated to all the nested lists.

```
# This work with a copy
l1 = [[1, 2], [3, 4], [5, 6]]
l2 = copy.copy(l1)
l2[1] = 3
print(f'l1={l1}, l2={l2}')
```

```
l1=[[1, 2], [3, 4], [5, 6]], l2=[[1, 2], 3, [5, 6]]
```

```
# When modifying the most inner list, it doesn't work anymore:
l1 = [[1, 2], [3, 4], [5, 6]]
l2 = copy.copy(l1)
l2[1][0] = 10
print(f'l1={l1}, l2={l2}')
```

```
l1=[[1, 2], [10, 4], [5, 6]], l2=[[1, 2], [10, 4], [5, 6]]
```

```
# Using deepcopy() and modifying the most inner list:
l1 = [[1, 2], [3, 4], [5, 6]]
l2 = copy.deepcopy(l1)
l2[1][0] = 10
print(f'l1={l1}, l2={l2}')
```

```
l1=[[1, 2], [3, 4], [5, 6]], l2=[[1, 2], [10, 4], [5, 6]]
```

2.2.3 Main characteristics of an array

The strength of NumPy arrays is their multidimensionality. This enables the description of complex datasets using a single NumPy array, on which operations can be performed. In NumPy, dimensions are also referred to as *axes*. For example, a set of 2 positions in space $\vec{r}_i$ can be represented as a 2D NumPy array, with the first axis representing the points $i = 1$ or $i = 2$, and the second axis representing the coordinates (x, y, z). There are a few attributes that describe multidimensional arrays:

- `a.dtype`: the type of data contained in the array
- `a.shape`: the number of elements along each dimension (or axis)
- `a.size`: the total number of elements (the product of the entries in `a.shape`)
- `a.ndim`: the number of dimensions (or axes)

```
points = np.array([[ 0,  1, 2],
                   [ 3,  4, 5]])

print('a.dtype = {}'.format(points.dtype))
print('a.shape = {}'.format(points.shape))
print('a.size  = {}'.format(points.size))
print('a.ndim  = {}'.format(points.ndim))
```

```
a.dtype = int64
a.shape = (2, 3)
a.size  = 6
a.ndim  = 2
```

2.3 The three key features of NumPy

2.3.1 Vectorization

Vectorization is a way to perform computations on NumPy arrays without using explicit loops, which can be very slow in Python. The idea behind vectorization is to perform a given operation element-wise on the array itself. An example is provided below to demonstrate the computation of the inverse of 100000 numbers using both explicit loops and vectorization.

```
a = np.random.randint(low=1, high=100, size=100000)

def explicit_loop_for_inverse(array):
    res = []
    for a in array:
        res.append(1./a)
    return np.array(res)
```

```
# Using explicit loop
%timeit explicit_loop_for_inverse(a)
```

```
167 ms ± 12.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
# Using list comprehension
%timeit [1./x for x in a]
```

```
151 ms ± 1.63 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
# Using vectorization
%timeit 1./a
```

```
106 µs ± 185 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

The suppression of explicit *for* loops is probably the most unfamiliar aspect of NumPy - according to me - and deserves a bit of practice. In the end, lines of code become relatively short, but one needs to properly think about how to implement a given computation in a *pythonic way*.

Many standard functions are implemented in a vectorized way; they are called *universal functions*, or `ufunc`. A few examples are given below, but the full description can be found in the NumPy documentation.

```
a = np.random.randint(low=1, high=100, size=3)
print('a        : {}'.format(a))
print('a^2      : {}'.format(a**2))
print('a/(1-a^a): {}'.format(a/(1-a**a)))
print('cos(a)   : {}'.format(np.cos(a)))
print('exp(a)   : {}'.format(np.exp(a)))
```

```
a        : [49 22 82]
a^2      : [2401  484 6724]
a/(1-a^a): [ 1.91469204e-17 -4.41611606e-18  8.20000000e+01]
cos(a)   : [ 0.30059254 -0.99996083  0.9496777 ]
exp(a)   : [1.90734657e+21 3.58491285e+09 4.09399696e+35]
```

All these ufuncs can work for n-dimensional arrays and can be used in a very flexible way depending on the axis you are referring to. Indeed, the mathematical operation can be performed over a different axis of the array, having a totally different meaning. Let's give a simple concrete example with a 2D array of shape (5,2), i.e. 5 vectors of three coordinates (x, y, z). Much more examples will be discussed in section 2.

```
# Generate 5 vectors (x,y,z)
positions = np.random.randint(low=1, high=100, size=(5, 3))

# Average of the coordinate over the 5 observations
pos_mean = np.mean(positions, axis=0)
```

```
print('mean = {}'.format(pos_mean))

# Distance to the origin sqrt(x^2 + y^2 + z^2) for the 5 observations
distances = np.sqrt(np.sum(positions**2, axis=1))
print('distances = {}'.format(distances))
```

```
mean = [27.4 65.2 59.2]
distances = [ 96.27564593  97.71898485  91.41662868 116.05602096 103.74487939]
```

Note on matrix product: NumPy arrays can be used to describe and manipulate matrices. There is a special way to perform a matrix product instead of an element-wise product. You can use `np.dot(a, b)` (or `a.dot(b)`). Other syntaxes are also possible, such as `a@b` or equivalently `np.matmul(a, b)`. If you are interested in these features, I would recommend reading the `np.dot` documentation in detail. Different syntaxes do not correspond to the same mathematical operation. For example, `np.matmul(a, b)` allows for *broadcasting* in 2×2 matrix product (see later).

```
a = np.array([[1, 1],
              [1, 0]], dtype=int)

b = np.array([[2, 4],
              [1, 1]], dtype=int)

print(np.dot(a, b))
```

```
[[3 5]
 [2 4]]
```

2.3.2 Broadcasting

Broadcasting is a way to compute operations between arrays of different sizes in an implicit (and concise) manner. One concrete example could be translating three positions $\vec{r}_i = (x, y)_i$ by a vector $\vec{d}_0$ simply by adding `points+d0`, where `points.shape=(3,2)` and `d0.shape=(2,)`. A few examples are given below, but more details are given in this documentation.

```
# operation between shape (3) and (1)
a = np.array([1, 2, 3])
b = np.array([5])
print('a+b = \n{}'.format(a+b))
```

```
a+b =
[6 7 8]
```

```
# operation between shape (3) and (1,2)
a = np.array([1, 2, 3])
b = np.array([
```

```
              [4],
              [5],
             ])
print('a+b = \n{}'.format(a+b))
```

```
a+b =
[[5 6 7]
 [6 7 8]]
```

```
# Translating 3 2D vectors by d0=(1,4)
points = np.random.normal(size=(3, 2))
d0 = np.array([1, 4])
print('points:\n {}\n'.format(points))
print('points+d0:\n {}'.format(points+d0))
```

```
points:
 [[-0.87306615  0.2632651 ]
 [ 0.02112935 -0.59555212]
 [-1.15652288 -0.02169556]]

points+d0:
 [[ 0.12693385  4.2632651 ]
 [ 1.02112935  3.40444788]
 [-0.15652288  3.97830444]]
```

Not all shapes can be combined together, and there are *broadcasting rules*, which are (quoting the NumPy documentation):

> When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing dimensions and works its way forward. Two dimensions are compatible when:
>
> 1. They are equal, or
> 2. One of them is 1.

This means that NumPy starts from the most right dimension (i.e., the most internal structure) of the two arrays and checks if they are compatible (either equal or one of them is one). If they aren't, a broadcast error is thrown; if they are, the next axis is checked in the same way. Broadcasting is possible if all dimensions of the two arrays are compatible.

In case two arrays are not immediately "broadcastable," it might be possible to add a new *empty axis* `np.newaxis` to an array to make some of their dimensions compatible and then make the broadcasting possible if the other non-empty dimensions are compatible. Here are two very simple examples:

Example 1: Case that can be fixed by modifying one array

The rightmost dimension is not compatible, but the next one is. This case can be solved by adding an empty axis to the right of `b`.

```
a = np.arange(10).reshape(2,5)
b = np.array([10, 20])
```

```
try:
    res = a+b
    print('Possible for {} and {}:'.format(a.shape, b.shape))
    print('a+b = \n {}'.format(res))
except ValueError:
    print('Impossible for {} and {}'.format(a.shape, b.shape))
```

```
Impossible for (2, 5) and (2,)
```

```
c = b[:, np.newaxis]
try:
    res = a+c
    print('Possible for {} and {}:'.format(a.shape, c.shape))
    print('a+c = \n {}'.format(res))
except ValueError:
    print('Broadcasting for {} and {}'.format(a.shape, c.shape))
```

```
Possible for (2, 5) and (2, 1):
a+c =
 [[10 11 12 13 14]
 [25 26 27 28 29]]
```

Example 2: Case that cannot be fixed by modifying one array.

The rightmost dimension is not compatible, and the next one is also not compatible. This case cannot be solved by adding an empty axis on the right of `b2`, as it will result in an error when comparing the next dimension.

```
b2 = np.array([10, 20, 30])
```

```
c2 = b2[:, np.newaxis]
try:
    res = a+c2
    print('Possible for {} and {}:'.format(a.shape, c2.shape))
    print('a+b = \n {}'.format(res))
except ValueError:
    print('Impossible for {} and {}'.format(a.shape, c2.shape))
```

```
Impossible for (2, 5) and (3, 1)
```

However, we can modify both `a` and `b2` to make all their dimensions compatible. The required shapes in that case would be: - `a_broad -> (1, 2, 5)` where an empty dimension is added to the left. - `b_broad ->`

`(3, 1, 1)` where two empty dimensions are added to the right. - Any mathematical operation will result in an array of shape `(3, 2, 5)`.

An application of this logic to perform a vectorized grid scan search is presented at the end of this chapter.

```
a_broad = a[np.newaxis, :, :]
b_broad = b2[:, np.newaxis, np.newaxis]
c_broad = a_broad + b_broad
print(f'{a_broad.shape} + {b_broad.shape} --> {c_broad.shape}')
```

```
(1, 2, 5) + (3, 1, 1) -> (3, 2, 5)
```

```
print(c_broad)
```

```
[[[10 11 12 13 14]
  [15 16 17 18 19]]

 [[20 21 22 23 24]
  [25 26 27 28 29]]

 [[30 31 32 33 34]
  [35 36 37 38 39]]]
```

2.3.3 Working with sub-arrays: slicing, indexing, and masking (or selection)

As mentioned earlier, *slicing and indexing* are ways to access elements or sub-arrays in a smart way. Python allows slicing with the `slice()` object, but NumPy allows us to push the logic much further with what is called *fancy indexing*. A few examples are given below, and for more details, please have a look at this documentation page.

Rule 1: The syntax is `a[i]` to access the ith element. It is also possible to go to the last element using negative indices: `a[-1]` is the last element.

```
a = np.random.randint(low=1, high=100, size=10)
print('a = {}'.format(a))
print('a[2] = {}'.format(a[2]))
print('a[-1] = {}'.format(a[-1]))
print('a[[1, 2, 5]] = {}'.format(a[[1, 2, 5]]))
```

```
a = [89 30 48 19 39 93 41  5 97 30]
a[2] = 48
a[-1] = 30
a[[1, 2, 5]] = [30 48 93]
```

Rule 2: NumPy also supports arrays of indices. If the index array is multi-dimensional, the returned array will have the same dimensions as the indices array.

```
# Small n-dimensional indices array: 3 arrays of 2 elements
indices = np.arange(6).reshape(3,2)
print('indices =\n {}'.format(indices))
print('a[indices] =\n {}'.format(a[indices]))
```

```
indices =
 [[0 1]
 [2 3]
 [4 5]]
a[indices] =
 [[89 30]
 [48 19]
 [39 93]]
```

```
# Playing with n-dimensional indices array: 2 arrays of (10, 10) arrays
indices_big = np.random.randint(low=0, high=10, size=(2, 3, 2))
print('indices_big =\n {}'.format(indices_big))
print('a[indices_big] =\n {}'.format(a[indices_big]))
```

```
indices_big =
 [[[3 2]
  [0 8]
  [2 4]]

 [[6 3]
  [5 1]
  [3 9]]]
a[indices_big] =
 [[[19 48]
  [89 97]
  [48 39]]

 [[41 19]
  [93 30]
  [19 30]]]
```

Rule 3: There is a smart way to access sub-arrays with the syntax `a[min:max:step]`. In this way, it's, for example, very easy to take every second element (`step=2`), or reverse the order of an array (`step=-1`). This syntax also works for n-dimensional arrays, where each dimension is separated by a comma. An example is given for a 1D array and for a 3D array of shape (5, 2, 3) - which can be considered as 5 observations of 2 positions in space.

```
# 1D array
a = np.random.randint(low=1, high=100, size=10)
print('full array a              = {}'.format(a))
print('from 0 to 1: a[:2]        = {}'.format(a[:2]))
print('from 4 to end: a[4:]      = {}'.format(a[4:]))
```

```
print('reverse order: a[::-1]    = {}'.format(a[::-1]))
print('all even elements: a[::2] = {}'.format(a[::2]))
```

```
full array a              = [10 23 60 57 77 12 80 67 86 60]
from 0 to 1: a[:2]        = [10 23]
from 4 to end: a[4:]      = [77 12 80 67 86 60]
reverse order: a[::-1]    = [60 86 67 80 12 77 57 60 23 10]
all even elements: a[::2] = [10 60 77 80 86]
```

```
# 3D array
a = np.random.randint(low=0, high=100, size=(5, 2, 3))
print('a = \n{}'.format(a))
```

```
a =
[[[99 11 92]
  [73 63 47]]

 [[78  3 16]
  [64 47 58]]

 [[85 81 17]
  [98 66 73]]

 [[55  0 43]
  [ 4 92 61]]

 [[ 8 78 34]
  [77 30 14]]]
```

Let's say one wants to extract only the (x, y) coordinates for the first vector in each of the 5 observations. This is how each axis will be sliced: - First axis (5 observations): `:`, i.e. all observations - Second axis (2 vectors): `1`, i.e. only the second element - Third axis (3 coordinates): `0:2`, i.e. from 0 to $2 - 1 = 1$, so only (x, y)

```
# Taking only the x,y values of the first vector for all observation:
print('a[:, 0, 0:2] =\n {}'.format(a[:, 0, 0:2]))
```

```
a[:, 0, 0:2] =
 [[99 11]
 [78  3]
 [85 81]
 [55  0]
 [ 8 78]]
```

```
# Reverse the order of the 2 vector for each observation:
print('a[:, ::-1, :] = \n{}'.format(a[:, ::-1, :]))
```

```
a[:, ::-1, :] =
[[[73 63 47]
  [99 11 92]]

 [[64 47 58]
  [78  3 16]]

 [[98 66 73]
  [85 81 17]]

 [[ 4 92 61]
  [55  0 43]]

 [[77 30 14]
  [ 8 78 34]]]
```

Rule 4: The last part of indexing is about *masking* an array or, in more common language, *selecting* sub-arrays/elements. This allows you to retrieve only elements that satisfy a given criterion, by exploiting the indexing rules described above. Indeed, a boolean operation applied to an array, such as `a>0`, will directly return an array of boolean values (`True` or `False`) depending on whether the corresponding element satisfies the condition or not.

```
a = np.random.randint(low=-100, high=100, size=(5, 3))
mask = a>0
print('a = \n{}'.format(a))
print('\nmask = \n {}'.format(mask))
```

```
a =
[[-89 -20 -75]
 [ 63  18  47]
 [ 68   9  58]
 [ 92 -59  13]
 [ 65   8 -33]]

mask =
 [[False False False]
 [ True  True  True]
 [ True  True  True]
 [ True False  True]
 [ True  True False]]
```

```
print('\na[mask] = \n {}'.format(a[mask])) # always return 1D array
print('\na*mask = \n {}'.format(a*mask)) # preserves the dimension (False=0)
print('\na[~mask] = \n {}'.format(a[~mask])) # ~mask is the negation of mask
print('\na*~mask  = \n {}'.format(a*~mask)) # working for a product too.
```

```
a[mask] =
 [63 18 47 68  9 58 92 13 65  8]
```

```
a*mask =
 [[ 0  0  0]
 [63 18 47]
 [68  9 58]
 [92  0 13]
 [65  8  0]]

a[~mask] =
 [-89 -20 -75 -59 -33]

a*~mask  =
 [[-89 -20 -75]
 [  0   0   0]
 [  0   0   0]
 [  0 -59   0]
 [  0   0 -33]]
```

Note: The case of boolean arrays as indices has a special treatment in NumPy (since the result is always a 1D array). There is actually a dedicated NumPy object called a *masked array* (cf. documentation) which allows you to keep the whole array but without considering some elements in the computation (e.g. CCD camera with dead pixels). Note, however, that when a boolean array is used in a mathematical operation (such as `a*mask`), `False` is treated as `0` and `True` as `1`:

```
print('a+mask = \n{}'.format(a+mask))
```

```
a+mask =
[[-89 -20 -75]
 [ 64  19  48]
 [ 69  10  59]
 [ 93 -59  14]
 [ 66   9 -33]]
```

This boolean array is also very useful to replace a category of elements with a given value in a very easy, concise, and readable way:

```
a = np.random.randint(low=-100, high=100, size=(5, 3))
print('Before: a=\n{}'.format(a))

a[a<0] = a[a<0]**2
print('\nAfter: a=\n{}'.format(a))
```

```
Before: a=
[[-24  79 -20]
 [-69 -60  50]
 [-75  53 -57]
 [-52 -37  42]
 [  1  72 -36]]
```

```
After: a=
[[ 576   79  400]
 [4761 3600   50]
 [5625   53 3249]
 [2704 1369   42]
 [   1   72 1296]]
```

2.4 Few useful NumPy tips

This short section presents a few handy features to know about NumPy, which can help beginners. For a slightly more complete view of "everyday NumPy," I would recommend taking a look at the *cheat sheet* from DataCamp.

Dummy array initialization.

```
x = np.zeros(shape=(3, 2))                # Only 0
x = np.ones(shape=(3, 2))                 # Only 1
x = np.full(shape=(3, 2), fill_value=10) # Only 10
x = np.eye(2)                             # Create identity matrix (only return 2D array)
```

Create sequence of numbers.

```
# Linear inteveral from 0 to 10
x = np.linspace(0, 10, 10) # 10 numbers between 0 and 1
x = np.arange(0, 10, 1.0)  # One number every 1.0 between 0 and 1
x = np.logspace(0, 10, 10) # 10 numbers between 10**0 and 10**10
```

Shape-based manipulation of arrays.

```
a = np.arange(0, 18).reshape(3, 3, 2)
x = a.ravel()                      # Return a flat array
x = a.reshape(9, 2)                # change the shape
x = a.T                            # transpose array: a.T[i, j, k] = a[k, j, i]
x = np.concatenate([a,a], axis=0) # concatenate arrays along a given axis: shape=(6, 3, 2)
x = np.stack([a, a], axis=0)       # group arrays along a given axis: shape=(2, 3, 3, 2)
```

Compare arrays.

```
# Making dummy arrays for comparisons
a = np.arange(-6, 6).reshape(3, 4)
b = np.abs(a)
c = np.append(b, [[1, 2, 3, 4]], axis=0)

# Print the arrays
print('a = {}\n'.format(a))
```

```
print('b = {}\n'.format(b))
print('c = {}'.format(c))
```

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

b = [[6 5 4 3]
 [2 1 0 1]
 [2 3 4 5]]

c = [[6 5 4 3]
 [2 1 0 1]
 [2 3 4 5]
 [1 2 3 4]]
```

```
# Arrays with the same shapes
print(np.equal(a, b))              # Return array with element-wise True/False
print(np.all(a==b))                # Return true is all element is true
print(np.allclose(a, b, rtol=10)) # same as all function with relative/absolute precision
print(np.any(a==b))                # Return true if any of the element is true
```

```
[[False False False False]
 [False False  True  True]
 [ True  True  True  True]]
False
True
True
```

```
# Arrays with the possibly different shapes
print(np.array_equal(a, c)) # True if a and b have the same shape and np.equal(a, b)
print(np.array_equiv(a, b)) # True if a and b have braodcastable shapes and same elements
```

```
False
False
```

```
# Example of equivalent arrays
a = np.array([1, 2])
b = np.array([[1, 2], [1, 2]])
np.array_equiv(a, b)
```

```
True
```

2.5 Example of simple gradient descent: NumPy vs. pure Python

2.5.1 Gradient descent: What is it for?

Gradient descent is a method used to numerically find the minimum of a function $f(p_0, ..., p_n)$. Finding the minimum is necessary for most machine learning problems, including model fitting. The goal is to find the best set of parameters that describe a dataset, assuming a given function.

For example, let's say you have n pairs of measured values (x_i, y_i) and you want to predict the mathematical relationship between x and y for all points: $y_i = \text{model}(x_i)$. Usually, the mathematical function "model" depends on some unknown parameters $p_0, ..., p_N$. In this case, the function to minimize is often the error function (or cost):

$$f(p_0, ..., p_N) = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \text{model}(x_i; p_0, ..., p_N)\right)^2$$

Finding the minimum of an error (cost) function is a fundamental step in any supervised learning algorithm. These concepts will be described in more detail in other lectures.

How does gradient descent work? At each iteration (or epoch), the parameters are updated using a step value μ in the opposite direction of the gradient, evaluated at the current point:

$$(p_0, ..., p_N)^{i+1} \leftarrow (p_0, ..., p_N)^i - \mu\left(\frac{\partial f}{\partial p_0}, ... \frac{\partial f}{\partial p_N}\right)|_{(p_0,...,p_N)^i}$$

This assumes that the value of the gradient is known. There are techniques to numerically estimate the gradient for arbitrary functions. In the example below, we consider a simpler situation with an exact solution: a linear model. In other words, there are only two parameters and we assume that:

$$\text{model}(x) = p_0 + p_1 x$$

with the following loss function gradient:

$$\frac{\partial f}{\partial p_0} = -\frac{2}{n}\sum_{i=0}^{i=n}\left(y_i - p_0 - p_1 x_i\right) \tag{2.1}$$

$$\frac{\partial f}{\partial p_1} = -\frac{2}{n}\sum_{i=0}^{i=n}\left(\left(y_i - p_0 - p_1 x_i\right) \times x_i\right) \tag{2.2}$$

From a coding point of view, we will introduce an array `delta = [yi - p0 - p1*xi]` which will be used to compute both the two gradient components and the loss function. The next two sections describe the pure Python implementation and the NumPy implementation, in order to compare performance. The following content is highly inspired by a *RealPython* post on performance comparison. Before diving into the discussion, let's define our fake dataset:

```
# Fake (xi, yi) data definition
n = 1000
x = np.linspace(0, 2, n)
xfine = np.linspace(0, 2, 1000) # to draw a line
y = 3 + 2 * x + 0.1*np.random.randn(n)
```

```
# Linear model definition
def model(x, p0, p1):
    return p0 + p1*x

# Loss function definition
def loss_function(p0, p1):
    return np.mean((y - model(x, p0, p1))**2)

# Vectorize the loss function for many parameters
loss_function = np.vectorize(loss_function)
```

The above `numpy.vectorize()` function allows for making several calls to the same function much faster using vectorization (cf. this documentation page). The next function, `plot_model(p)`, simply represents the data, the model, the evolution of the loss function, and the trajectory in the (p_0, p_1) space that is followed by the gradient descent. This function is based on the matplotlib library, which will be discussed in later chapters of this lecture.

```
# Plotting function (data vs fit, loss function, gradient descent)
def plot_model(p):
    '''
    Producing three plots from the list of the 2
    parameters for all epochs: p.shape = (Nepochs, 2)
    '''
    import matplotlib.pyplot as plt
    plt.figure(figsize=(30, 7))

    # Get Best parameters (last ones), ymodel and lost functions
    p0, p1 = p[-1, 0], p[-1, 1]
    ymodel = model(x, p0, p1)
    loss = loss_function(p[:, 0], p[:, 1])

    # Plot (xi,yi) data and overlay (x, model(x, p)) points
    plt.subplot(1, 3, 1)
    plt.plot(x, y, 'o', alpha=0.3, markersize=5, label='data')
    plt.plot(xfine, ymodel, linewidth=3, color='tab:red', label='model')
    plt.xlabel('$x$'); plt.ylabel('$y$')
    plt.legend()

    # Plot the loss function v.s. epoch
    plt.subplot(1, 3, 2)
    plt.semilogy(loss, linewidth=3)
    plt.xlabel('Epochs'); plt.ylabel('Loss function')

    # Plot the gradient in the (p0, p1) space and the descent trajectory
    plt.subplot(1, 3, 3)
    P0, P1 = np.meshgrid(np.linspace(0, 4, 300), np.linspace(0, 3, 300))
    plt.imshow(np.log(loss_function(P0, P1)), extent=[0, 4, 0, 3], aspect='auto',
               origin='lower', cmap='Greys')
```

```
    plt.plot(p[:, 0], p[:, 1], linewidth=5, color='tab:red', alpha=0.8, label='Trajectory')
    plt.xlabel('$p_0$'); plt.ylabel('$p_1$')
    plt.colorbar(label='log(loss function)')
    for t in plt.legend().get_texts():
        t.set_color('white')

    return
```

```
# Tuning default matplotlib style
import matplotlib as mpl
mpl.rcParams['legend.frameon'] = False
mpl.rcParams['legend.fontsize'] = 24
mpl.rcParams['xtick.labelsize'] = 20
mpl.rcParams['ytick.labelsize'] = 20
mpl.rcParams['axes.titlesize'] = 24
mpl.rcParams['axes.labelsize'] = 24
```

2.5.2 Pure Python implementation

The pure Python implementation doesn't use any of the vectorized features of NumPy. The loop and sum over the data points are explicit, using `zip()`, `sum()`, and comprehension syntax. The function returns an array of all parameters for each epoch (for later convenience, we simply return a NumPy array - but mathematical operations are not done with NumPy in this function).

```
def python_linear_descent(x, y, mu, N_epochs):

    # Length of data
    n = len(x)

    # Initialize predictions, errors, parameters and gradients.
    ym = [0] * n
    para = [[0, 0]] * N_epochs
    grad = [0, 0]
    pm = [0, 0]

    # Looping over iterations (epochs)
    for i_epoch in range(0, N_epochs):
        delta = tuple(i - j for i, j in zip(y, ym))
        grad[0] = -2/n * sum(delta)
        grad[1] = -2/n * sum(i * j for i, j in zip(delta, x))
        pm = [i - mu * j for i, j in zip(pm, grad)]
        ym = ( model(i, pm[0], pm[1]) for i in x )

        # Save all parameters
        para[i_epoch] = pm
```

```
    # Return numpy array of the 2 paramters for all epochs
    return np.array(para)
```

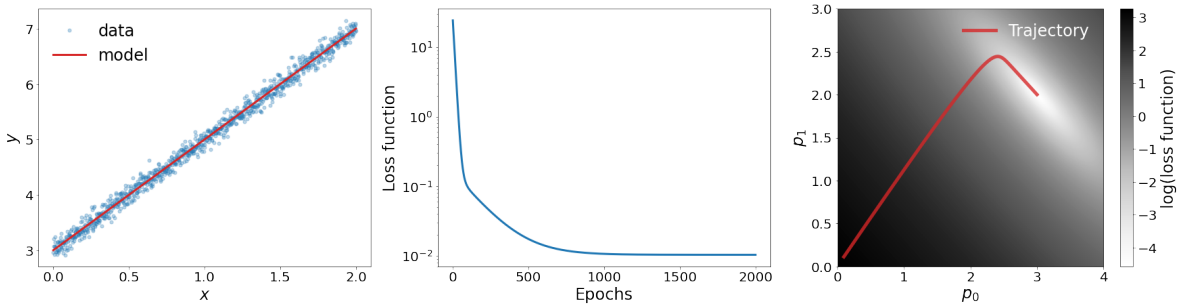
We can then try to time this function using a step of 0.01 and 2000 epochs for our 1000 data points.

```
%timeit python_linear_descent(x, y, mu=0.01, N_epochs=2000)
```

```
1.14 s ± 52.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

This takes approximately 1 second to run. What follows shows the best model prediction for the best parameters, the evolution of the loss function, as well as the descent trajectory in the parameter space.

```
parameters = python_linear_descent(x, y, mu=0.01, N_epochs=2000)
plot_model(parameters)
```



2.5.3 NumPy implementation

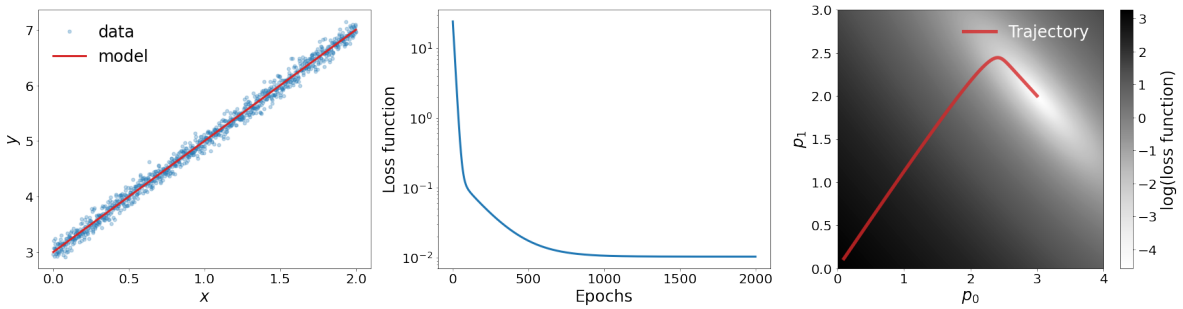
The NumPy implementation makes full use of the vectorization feature discussed several times in this chapter, as well as broadcasting. This leads to significantly clearer code and also much faster execution. We can note, in particular, the different syntax used to update the model parameters.

```
def numpy_linear_descent(x, y, mu, N_epochs):

    # To define the lost function
    n = x.shape[0]

    # Initialize predictions, errors, parameters and gradients.
    ym = np.zeros(n)
    para = np.zeros((N_epochs, 2))
    pm, grad = np.zeros(2), np.zeros(2)

    # Looping over iterations (epochs)
    for i_epoch in range(0, N_epochs):
        delta = y-ym
        grad = -2/n * np.array([np.sum(delta), np.sum(delta*x)])
        pm = pm - mu*grad
```

```
        ym = model(x, pm[0], pm[1])

        # Save all parameters
        para[i_epoch] = pm

    return para

%timeit numpy_linear_descent(x, y, mu=0.01, N_epochs=2000)
```

```
40.3 ms ± 2.26 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
parameters = numpy_linear_descent(x, y, mu=0.01, N_epochs=2000)
plot_model(parameters)
```



We check that the results are indeed the same for an execution about 25 times faster…

2.6 Example of a vectorized grid scan with NumPy

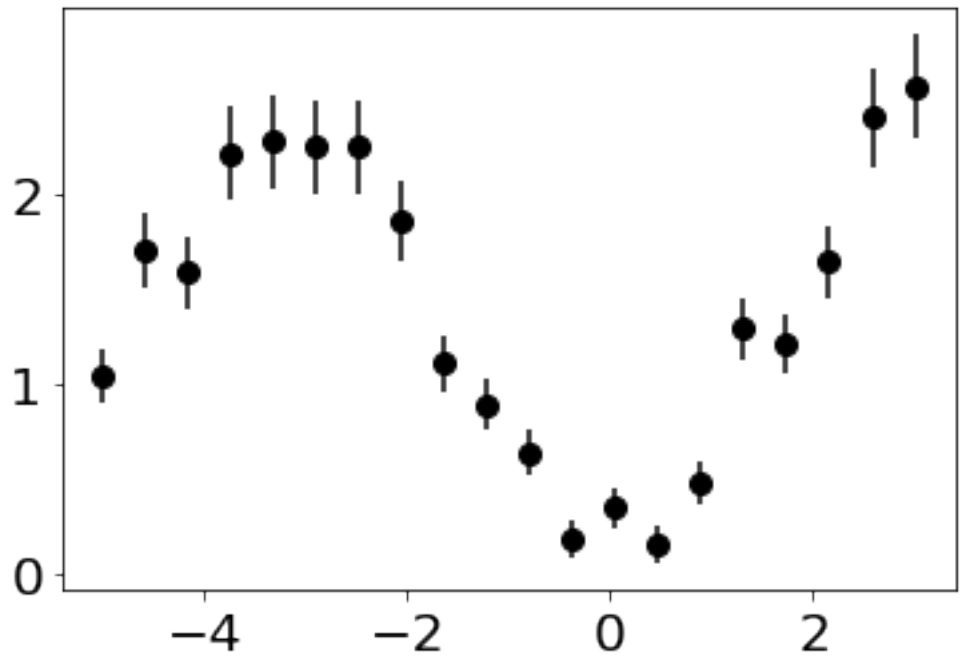
2.6.1 Context: The brute force grid scan

This approach consists of scanning the parameter space to find the optimum of a given (loss) function. This procedure is not often the best one for a real-life case, but it is interesting to know how to efficiently code this scan using the vectorization and broadcasting of NumPy, as this logic can be useful in other contexts. For this example, we will define (fake) data on which we will fit several models having a different number of parameters. The fit will be performed with a parameter grid scan search, both with pure Python and NumPy. Note that the plotting package presented in the next chapter will be used for this example.

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

```
# Fake data with noise
Npoints, Nsampling = 20, 1000
xcont = np.linspace(-5.0, 3.5, Nsampling)
x = np.linspace(-5, 3.0, Npoints)
y = 2*(np.sin(x/2)**2 + np.random.random(Npoints)*0.3)
dy = np.sqrt(0.10**2 + (0.10*y)**2)
```

```
# Data style for plotting
data_style = {'marker': 'o', 'color': 'black', 'markersize': 8,
              'linestyle': '', 'zorder': 10, 'label': 'Data'}

# Plotting the fake data
plt.errorbar(x, y, yerr=dy, **data_style);
```




```
# Linear model
def model_lin(x, p0, p1):
    return p0 + p1*x

# Trigonometric model
def model_x3(x, p0, p1, p2, p3):
    return p0 + p1*x + p2*x**2 + p3*x**3
```

2.6.2 Pure Python approach: nested loops

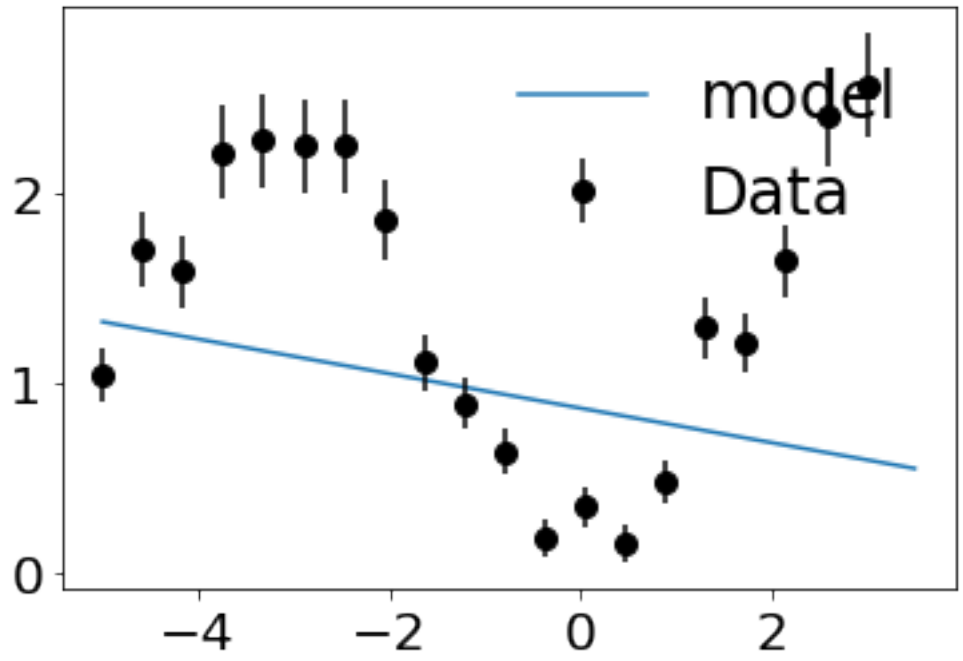
The naive way to proceed with a grid scan is to perform nested loops (as many as parameters to scan) and compute the loss function for each point in the grid. We will keep track of the minimum loss and the corresponding parameters. We will try this approach on the linear model only.

```
# Loss for linear model for nested loops
def loss_linear_loops(p0, p1):
    residus = (y - model_lin(x, p0, p1)) / dy
    return np.sum( residus**2 )
```

```
def grid_scan_linear_loops(N0=100, N1=100):

    '''
    Perform a grid search over (p0, p1) using nested loops
    for the linear model. Return the parameter for which
    the loss was found to be minimal in the grid.
    '''

    # Defining the grid
    p0s = np.linspace(-2, 2, N0)
    p1s = np.linspace(-3, 3, N1)

    # Loop over parameters
    lmin, ip0min, ip1min = 1e10, -1, -1
    for ip0, p0 in enumerate(p0s):
        for ip1, p1 in enumerate(p1s):
            l = loss_linear_loops(p0, p1)
            if l<lmin:
                lmin = l
                ip0min = ip0
                ip1min = ip1

    # Return the minimum of the loss and the associated parameters
    return p0s[ip0min], p1s[ip1min]
```

```
# Making the scan
p0, p1 = grid_scan_linear_loops()
print(f'p0={p0:.2f}, p1={p1:.2f}')

# Plotting the result
plt.plot(xcont, model_lin(xcont, p0, p1), label='model')
plt.errorbar(x, y, yerr=dy, **data_style);
plt.legend();
```

```
p0=0.87, p1=-0.09
```



2.6.3 NumPy approach: broadcasting + vectorization

The idea is to increase the dimension of the initial array so that broadcasting can be used to obtain the desired result. Let's assume we have two parameters with `N0` and `N1` values each. The final result should be a 2D array of shape `(N0, N1)` which will contain the values of the loss function for every combination of parameter values.

First, the residuals are computed for each individual data point, resulting in an array of shape `(N0, N1, Ndata)`. Then, the residuals are summed over the dataset, resulting in a 2D array of shape `(N0, N1)`.

To achieve this, we can use broadcasting and vectorization. We can change the dimensions of the arrays as follows:

```
data (Ndata) -> ( 1,  1, Ndata)
par0 (N0)    -> (N0,  1,     1)
par1 (N1)    -> ( 1, N1,     1)
residuals    -> (N0, N1, Ndata)
```

Then, we can compute the residuals using these modified arrays and perform the sum over the last axis to obtain the desired 2D array.

The code below demonstrates this approach and also provides a generalization for an arbitrary number of parameters.

```
def loss_linear_vectorized(p0s, p1s):

    '''
    Loss function generalized for arrays of parameters.
```

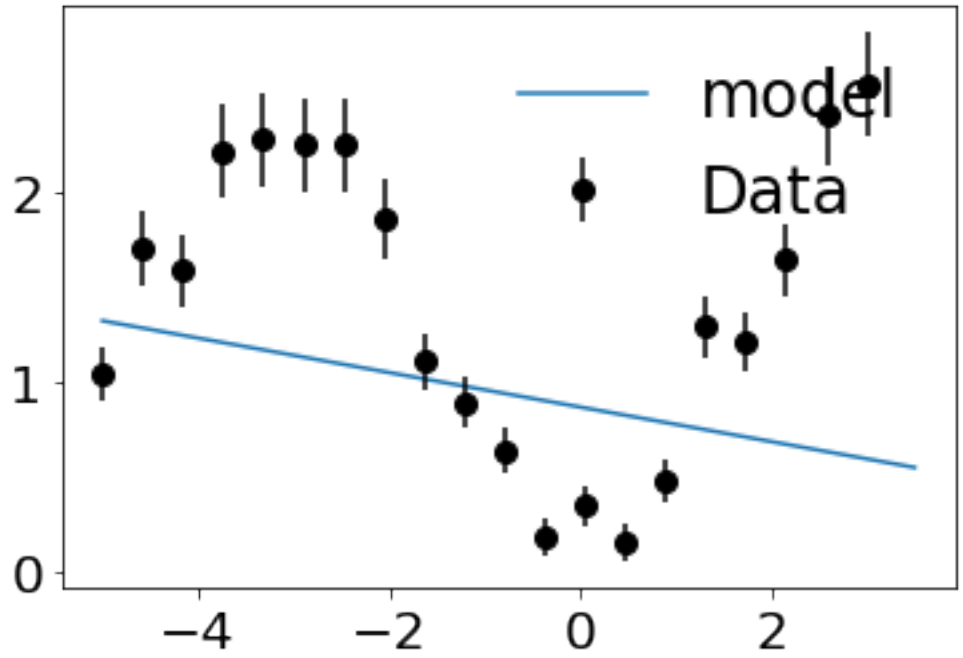
```
    Return a 2D array (Np0, Np1) being the value of the
    loss function for all parameters values.
    '''

    # Preparing shape of data for broadcasting
    xb  = x[np.newaxis, np.newaxis, :]
    yb  = y[np.newaxis, np.newaxis, :]
    dyb = dy[np.newaxis, np.newaxis, :]

    # Preparing shape of parameters for broadcasting
    p0sb = p0s[:, np.newaxis, np.newaxis]
    p1sb = p1s[np.newaxis, :, np.newaxis]

    # Compute the residus for each points and paramter values
    residus = (yb - model_lin(xb, p0sb, p1sb)) / dyb

    # Return the sum over all points for each parameters values
    return np.sum(residus**2, axis=-1)
```

```
def grid_scan_linear_vectorized(N0=100, N1=100):

    '''
    Perform a grid search over (p0, p1) using vectorization
    for the linear model. Return the parameter for which
    the loss was found to be minimal in the grid. If several
    is found, the first one found by np.where() is returned.
    '''

    # Defining the grid
    p0s = np.linspace(-2, 2, N0)
    p1s = np.linspace(-3, 3, N1)

    # Compute losses for all parameters
    ls = loss_linear_vectorized(p0s, p1s)

    # Get the optmized parameters
    lmin = np.min(ls)
    ip0min, ip1min = np.where(lmin==ls)

    # Return the minimum of the loss and the associated parameters
    return p0s[ip0min[0]], p1s[ip1min[0]]
```

```
# Making the scan
p0, p1 = grid_scan_linear_vectorized()
print(f'p0={p0:.2f}, p1={p1:.2f}')

# Plotting the result
plt.plot(xcont, model_lin(xcont, p0, p1), label='model')
```

```
plt.errorbar(x, y, yerr=dy, **data_style);
plt.legend();
```

```
p0=0.87, p1=-0.09
```



Let's try to generalize to an arbitrary model, dataset, and parameter grid. For this, the new dimension will be created on the fly, and the function `np.reshape()` will be used since it accepts a Python list to specify the dimension along each axis. As you can see below, such a function can be written, but the limiting factor will then be the available memory. This brute force approach scales very badly with the dimension of the problem (i.e. the number of parameters).

```
def loss_vectorized(xdata, ydata, dydata, model, *modelpars):

    '''
    Return the loss value for the full data set
    for a model `model` and the associated parameters.

       loss = sum_{data} ( ydata - model(xdata, *pars) )^2

    For example, if we have 20 data points with a model of 3 parameters
    having 200 steps each, the internally created arrays have the following
    dimensions:
        data = (  1, 1, 1, 20)
        par1 = (200, 1, 1,  1)
        par2 = (1, 200, 1,  1)
        par3 = (1, 1, 200,  1)

    Arguments:
```

```
    ----------

      - xdata, ydata, dydata: 1D array corresponding to the data points.
      - model: callable of type f(x, *pars).
      - *modelpars: list of 1D array correponding to the scan of each
                    parameter.
    '''

    # Parameters
    pars = [p for p in modelpars]
    Npars = len(pars)

    # Generic data broadcasting for Npars:
    # Data shape : (1, 1, 1, ..., 1, Ndata) (with as much as 1 as Npars)
    data_dim = [1 for p in pars]
    data_dim.append(xdata.shape[0]) # Append the last axis with Ndata
    xdata_b  = xdata.reshape(data_dim)
    ydata_b  = ydata.reshape(data_dim)
    dydata_b = dydata.reshape(data_dim)

    # Generic broadcasting for each parameters
    pars_b = []
    for ip in range(Npars):
        pdim = [1 if ip!=jp else pars[ip].shape[0] for jp in range(Npars)]
        pdim.append(1) # Append a last empty axis for the data
        pars_b.append(pars[ip].reshape(pdim))

    # Compute residus
    residus = (ydata_b - model(xdata_b, *pars_b)) / dydata_b

    # Return the sum over data - first axis
    return np.sum(residus**2, axis=-1)
```

```
# Trying the function
all_pars = [np.linspace(0, 1, 10) for i in range(4)]
ls = loss_vectorized(x, y, dy, model_x3, *all_pars)
print(ls.shape, ls.size)
```

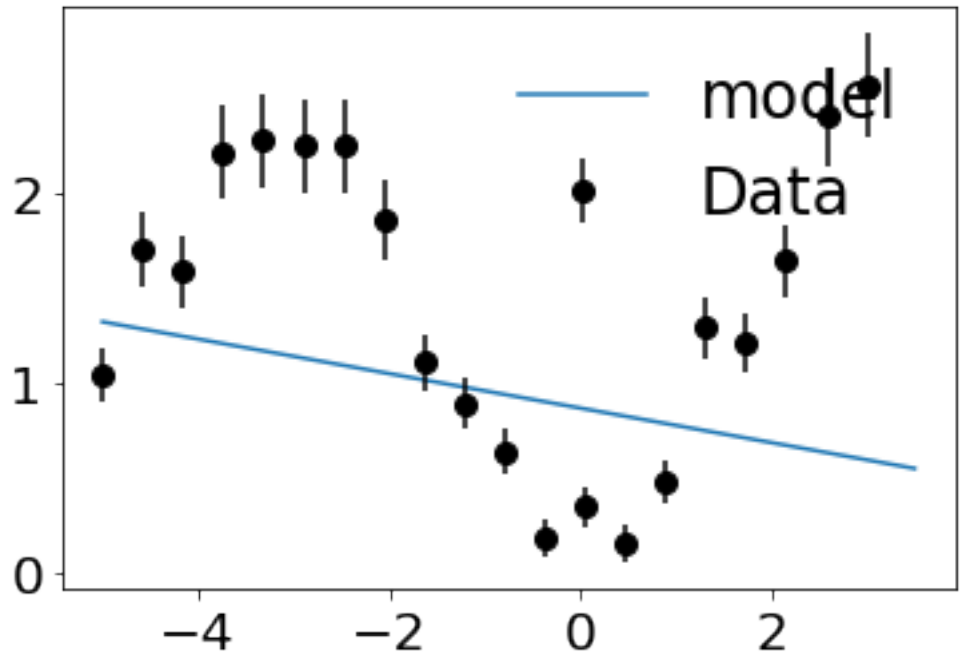
```
(10, 10, 10, 10) 10000
```

```
def grid_scan_vectorized(xdata, ydata, dydata, model, mins, maxs, steps):

    '''
    Return the parameter which minimize the loss function over the full
    dataset for a model `model` :

       loss = sum_{data} ( ydata - model(xdata, *pars) )^2
```

```
    Arguments:
    ----------
      - xdata, ydata, dydata: 1D array corresponding to the data points.
      - model: callable of type model(x, *pars).
      - mins : list of floats being the minimum for each paramters
      - maxs : list of floats being the maximum for each paramters
      - steps: list of integers being the steps for each parameters
    '''

    # Checking that the inputs for the parameters are correct
    nmins, nmaxs, nsteps = len(mins), len(maxs), len(steps)
    if not (nmins==nmaxs and nmins==nsteps and nmaxs==nsteps):
        print(f'Uncorrect numbers for min, max and/or steps: {len(mins)}, {len(maxs)},
        ↪  {len(steps)}')
        return [-1]

    # Defining the parameter grid
    npars = nmins
    pars  = [np.linspace(mins[i], maxs[i], steps[i]) for i in range(npars)]

    # Computing the loss for all parameters
    ls = loss_vectorized(xdata, ydata, dydata, model, *pars)

    # Get the optmized parameters
    lmin = np.min(ls)
    ipars_min = np.where(lmin==ls)

    # Return the minimum of the loss and the associated parameters
    return [ p[i[0]] for p, i in zip(pars, ipars_min) ]
```

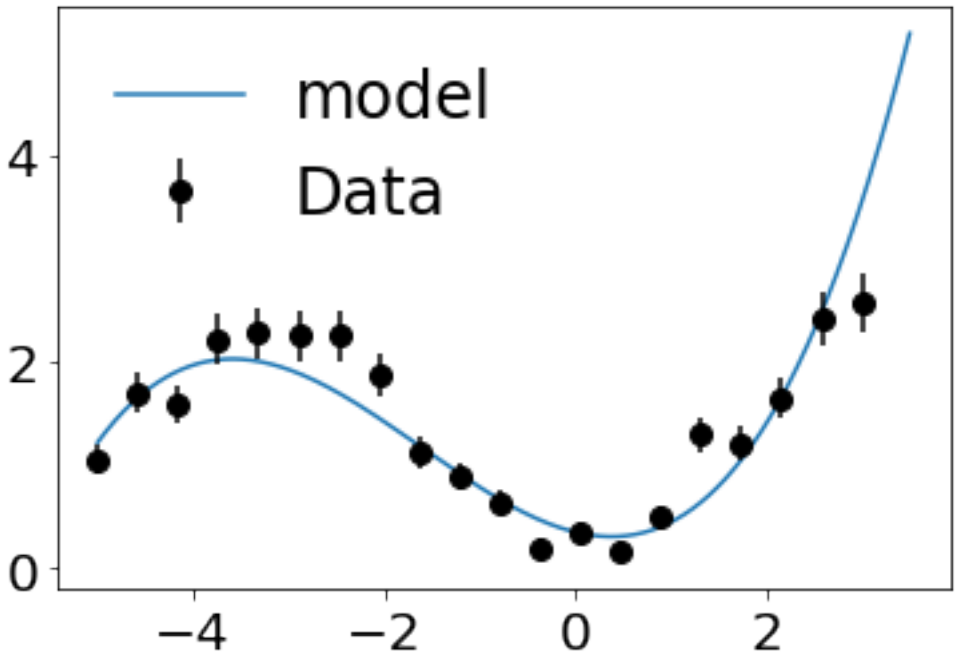
```
# Making the scan
p0, p1 = grid_scan_vectorized(x, y, dy, model_lin, [-2, -3], [2, 3], [100, 100])
print(f'p0={p0:.2f}, p1={p1:.2f}')

# Plotting the result
plt.plot(xcont, model_lin(xcont, p0, p1), label='model')
plt.errorbar(x, y, yerr=dy, **data_style);
plt.legend();
```

```
p0=0.87, p1=-0.09
```




```
# Making the scan with same function for another model.
p0, p1, p2, p3 = grid_scan_vectorized(x, y, dy, model_x3, [-1, -1, -1, -0.1], [1, 1, 1, 0.1],
↪  [50]*4)
print(f'p0={p0:.2f}, p1={p1:.2f}, p2={p2:.2f}, p3={p3:.3f}')

# Plotting the result
plt.plot(xcont, model_x3(xcont, p0, p1, p2, p3), label='model')
plt.errorbar(x, y, yerr=dy, **data_style);
plt.legend();
```

```
p0=0.35, p1=-0.22, p2=0.27, p3=0.055
```



2.6.4 Timing comparison for 2 parameters

```
%timeit grid_scan_linear_loops(N0=500, N1=500)
```

```
2.31 s ± 5.14 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%timeit grid_scan_linear_vectorized(N0=500, N1=500)
```

```
45.2 ms ± 912 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Chapter 3

Three important tools to know

> **skills to take away**
>
> - *Basic*: Plotting $y = f(x)$ and histograms with NumPy/Matplotlib, dataframes from CSV, add columns
> - *Medium*: Scatter plots, dataframe cleaning and plotting, curve fitting
> - *Expert*: Meshgrid and 3D plots for $y = f(x, y)$

3.1 A word of caution

The three important tools discussed in this section, namely Matplotlib, Pandas, and SciPy, are *only introduced*. A decently extensive presentation would deserve an entire book for each of them. The main goal of this chapter is to give the very basic and practical features of each of them, so that you can search for more detailed information when you need it.

3.2 Graphical representation of data: Matplotlib

Matplotlib is an extremely rich library for data visualization and there is no way to cover all its features in this note. The goal of this section is just to give short and practical examples to plot data. Much more details can be obtained on the webpage. Another interesting link to understand the structure of a Matplotlib plot is a post on Real Python website. The following shows how to quickly make *histograms*, *graphs*, *2D and 3D scatter plots*.

The main object of Matplotlib is `matplotlib.pyplot` imported as `plt` here (and usually). The most common functions are then called on this object, and often take NumPy arrays as arguments (possibly with more than one dimension) and a lot of `kwargs` to define the plotting style.

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

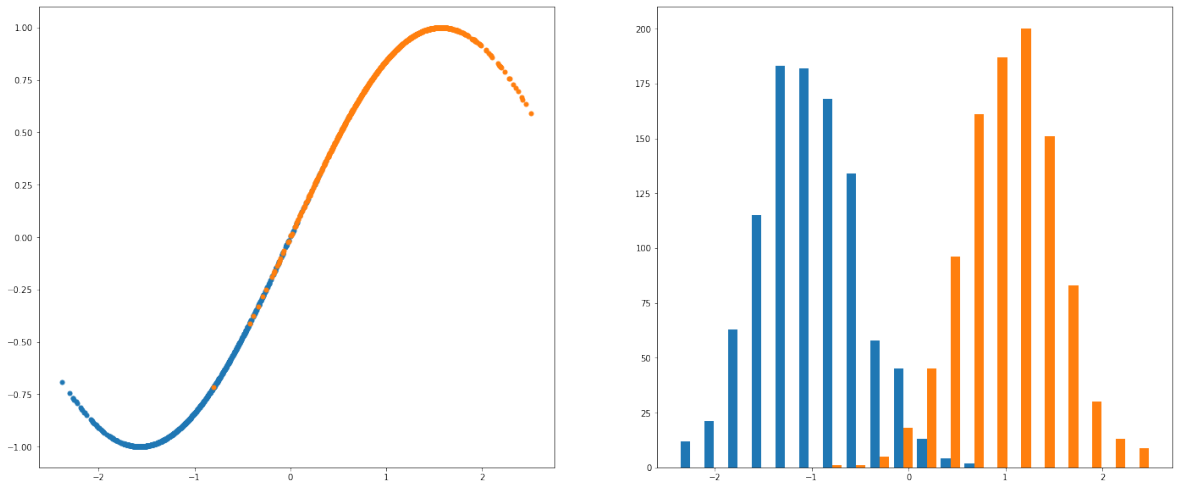
3.2.1 Example of 1D plots and histograms

To work with data, we generate two samples of 1000 values distributed according to a normal probability density function with $\mu = -1$ and $\mu = 1$ respectively, and $\sigma = 0.5$. These data are stored in a NumPy array `x` of shape `x.shape=(1000, 2)`. We then compute and store the sine of all these values into an array of the same shape called `y`.

```
x = np.random.normal(loc=[-1, 1], scale=[0.5, 0.5], size=(1000,2))
y = np.sin(x)
```

The next step is to plot this data in two ways: first, we want `y` vs. `x`; second, we want the histogram of the `x` values. We need to first create a figure, then create two *subplots* (specifying the number of rows, columns, and subplot index). Note that Matplotlib always takes the first dimension to define the numbers to plot, while higher dimensions are considered as other plots that are automatically overlaid.

```
plt.figure(figsize=(24, 10))
plt.subplot(121) # 121 means 1 line, 2 column, 1st plot
plt.plot(x, y, marker='o', markersize=5, linewidth=0.0)
plt.subplot(122) # 122 means 1 line, 2 column, 2nd plot
plt.hist(x, bins=20);
```



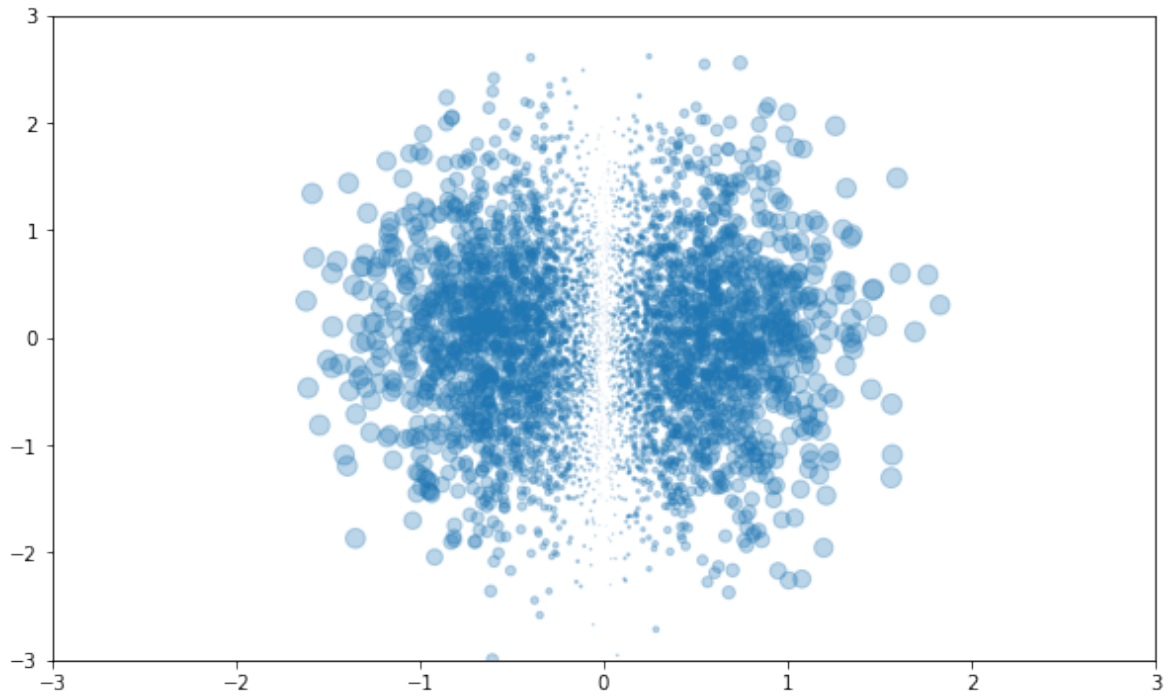
3.2.2 Example of 2D scatter plot

A scatter plot allows us to draw markers in a 2D space, with a third piece of information encoded into the marker size. To demonstrate this, we generated two sets of 5000 numbers distributed according to uncorrelated Gaussians with means $\mu_0 = \mu_1 = 0$ and standard deviations $\sigma_1 = 0.5$ and $\sigma_2 = 0.8$. These numbers are stored in a NumPy array called `points`, with a shape of `points.shape=(5000,2)`. We then interpret these two sets of numbers as (x, y) positions and load them into two `(5000, 1)` arrays called `x` and `y`:

```
points = np.random.normal(loc=[0, 0], scale=[0.5, 0.8], size=(5000,2))
x, y = points[:, 0], points[:, 1]
```

We can then plot the 5000 points in the 2D plane, and here we specify the marker size as $100 \times \sin^2(x)$ using the argument `s` of the `plt.scatter()` function (note that the arrays `x`, `y`, and `s` must have the same shape):

```
plt.figure(figsize=(10,6))
plt.scatter(x, y, s=100*(np.sin(x))**2, marker='o', alpha=0.3)
plt.xlim(-3, 3)
plt.ylim(-3, 3);
```



3.2.3 Example of 3D plots

For 3D plots, one can generate 1000 positions in space and perform a translation by a vector $\vec{r}_0$ using broadcasting.

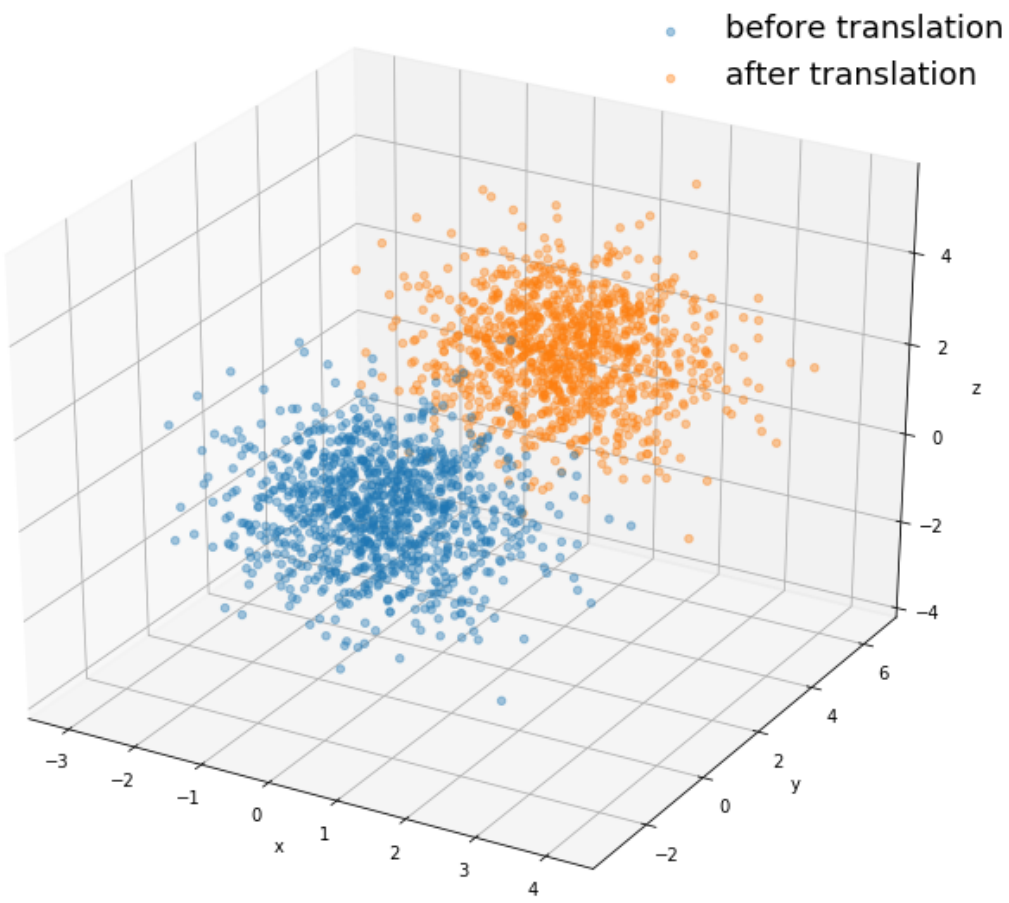
```
data = np.random.normal(size=(1000, 3))
r0 = np.array([1, 4, 2])
data_trans = data + r0
```

It is then easy to get back the spatial initial (i.e. before translation) and final (i.e. after translation) coordinates.

```
xi, yi, zi = data[:,0], data[:,1], data[:,2]
xf, yf, zf = data_trans[:,0], data_trans[:,1], data_trans[:,2]
```

An additional module must be imported in order to plot data in three dimensions, and the projection has to be specified. Once this is done, a simple call to `ax.scatter3D(x, y, z)` will create the plot. Note that we call a function of `ax` and not `plt` as before. This is because of the `ax = plt.axes(projection='3d')` command, which is necessary for 3D plotting. More details are available in the matplotlib 3D tutorial.

```
from mpl_toolkits import mplot3d
plt.figure(figsize=(12,10))
ax = plt.axes(projection='3d')
ax.scatter3D(xi, yi, zi, alpha=0.4, label='before translation')
ax.scatter3D(xf, yf, zf, alpha=0.4, label='after translation')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.legend(frameon=False, fontsize=18);
```



3.2.4 Example of 2D function $z = f(x, y)$: notion of `meshgrid`

Another typical plot we might want to create is a representation of a function of two variables (x, y) in 3D: $z = f(x, y)$. In Python, this requires the use of the `meshgrid` function, which may not be immediately obvious. Let's begin by defining a function with two variables:

```
def my_surface(x, y):
    x0 = 5*np.sin(y)
    sigma = 5+y
```

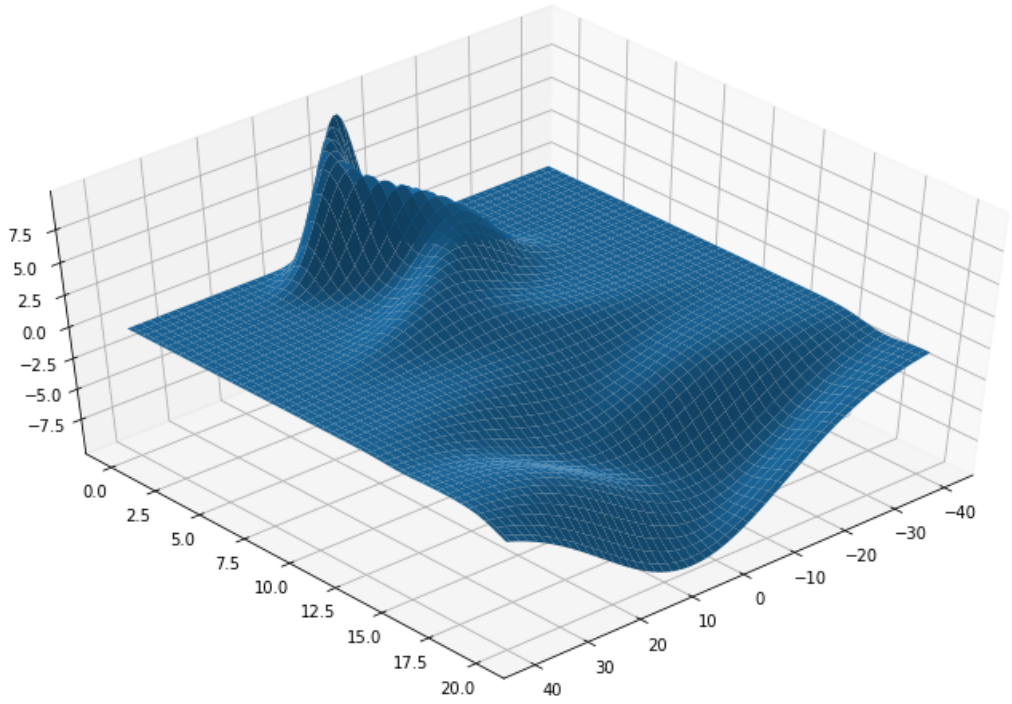
```
    amp = (10-y)
    return amp*np.exp(-(x-x0)**2/sigma**2)
```

Let's define an (x, y) interval on which we want to describe the surface:

```
x = np.linspace(-40, 40, 100)
y = np.linspace(0, 20, 200)
```

These two NumPy arrays don't have the same shape, and an explicit loop would be needed to process them, which is very time-consuming in Python. This is where the *meshgrid* notion comes in: it will provide two arrays *with the same size* and allow for vectorization.

```
# Meshgrid and function application (see after for more details)
xx, yy = np.meshgrid(x, y)
Z = my_surface(xx, yy)

# Plotting
fig = plt.figure(figsize=(13,8))
ax = fig.gca(projection='3d')
ax.plot_surface(xx, yy, Z)

# Choose the default view
ax.view_init(azim=48, elev=48)
```



Meshgrid explanation. The *meshgrid* consists of two 2D arrays made out of two 1D arrays. The main purpose is to have NumPy arrays with the same shape, which can then support vectorized operations. The logic is

relatively straightforward and can be understood with two points along each coordinate. Let's assume you want to scan the {0, 1} *x*-values and {2, 3} *y*-values, then you need to build up the four following 2D points: (0, 2), (1, 2), (0, 3), (1, 3). These four points can be encoded in the two arrays `[[0, 1], [0, 1]]` and `[[2, 2], [3, 3]]`. These two arrays have the same shape, which is similar to the result of `f(x,y)` computed on this grid, even if there are *not the same numbers of x and y values*.

```
# Create a simple 2-variables function
def f(x, y):
    return x**2+y**2

# Define x-values, y-values and create the 2D
x, y = np.arange(0, 2), np.arange(2, 4)
xx, yy = np.meshgrid(x, y)
zz = f(xx, yy)

# Printing arrays
print('Array values:')
print('xx={}'.format(xx))
print('yy={}'.format(yy))
print('zz={}'.format(zz))
```

```
Array values:
xx=[[0 1]
 [0 1]]
yy=[[2 2]
 [3 3]]
zz=[[ 4  5]
 [ 9 10]]
```

A way to explicitly create the meshgrid is to flatten the `xx` and `yy` arrays using the `ravel()` function, and then take each pair using the `zip()` syntax. This ensures that every point is formed.

```
for i, j in zip(xx.ravel(), yy.ravel()):
    print('(x,y)=({},{}); f(x,y)={}'.format(i, j, f(i, j)))
```

```
(x,y)=(0,2); f(x,y)=4
(x,y)=(1,2); f(x,y)=5
(x,y)=(0,3); f(x,y)=9
(x,y)=(1,3); f(x,y)=10
```

If you want to read more about this, you can check the NumPy meshgrid documentation and this Stack Overflow post. For more advanced readers, there are two similar functions which return slightly different objects: `np.ogrid()` and `np.mgrid()`. For a nice discussion of the differences, you can check this post.

3.3 Import and Manipulate Data as NumPy Array: `pandas`

The package `pandas` is a very rich interface to read data from different formats and produce a `pandas.DataFrame` that can be based on NumPy (but containing a lot more features). There is no

way to fully describe this package here; the goal is simply to give functional and concrete examples that are easily usable. For more details, please check the pandas webpage.

3.3.1 Data importation

Many built-in functions are available to import data as a Pandas DataFrame. One particularly convenient function directly reads CSV files. It allows you to specify the columns to load, the rows to skip, and many other options.

```
import pandas as pd
df = pd.read_csv('../data/WaveData.csv')
print(df.head())
```

```
          Date/Time      Hs   Hmax      Tz      Tp  Peak Direction    SST
0  01/01/2017 00:00 -99.900 -99.90 -99.900 -99.900           -99.9 -99.90
1  01/01/2017 00:30   0.875   1.39   4.421   4.506           -99.9 -99.90
2  01/01/2017 01:00   0.763   1.15   4.520   5.513            49.0  25.65
3  01/01/2017 01:30   0.770   1.41   4.582   5.647            75.0  25.50
4  01/01/2017 02:00   0.747   1.16   4.515   5.083            91.0  25.45
```

```
# Rename columns names using df.rename() function
old_new_cols = {
    'Date/Time': 'date',
    'Hs': 'height',
    'Hmax': 'heightMax',
    'Tz': 'period',
    'Tp': 'energy',
    'Peak Direction': 'direction',
    'SST': 'temperature'
}

# The argument `inplace` means the current dataframe is overwritten with the change
df.rename(columns=old_new_cols, inplace=True)
print(df.head())
```

```
              date  height  heightMax  period  energy  direction  \
2  01/01/2017 01:00   0.763       1.15   4.520   5.513       49.0
3  01/01/2017 01:30   0.770       1.41   4.582   5.647       75.0
4  01/01/2017 02:00   0.747       1.16   4.515   5.083       91.0
5  01/01/2017 02:30   0.718       1.61   4.614   6.181       68.0
6  01/01/2017 03:00   0.707       1.34   4.568   4.705       73.0

   temperature  height_normalized  heightMax_normalized  period_normalized  \
2        25.65          -0.898215             -1.047342          -1.184338
3        25.50          -0.884973             -0.757690          -1.117565
4        25.45          -0.928484             -1.036201          -1.189723
5        25.45          -0.983346             -0.534881          -1.083102
```

```
6        25.50          -1.004155             -0.835673          -1.132643

   energy_normalized  direction_normalized  temperature_normalized
2          -1.463956             -2.044360                0.762152
3          -1.407891             -0.973294                0.694918
4          -1.643866             -0.314176                0.672506
5          -1.184467             -1.261658                0.672506
6          -1.802020             -1.055683                0.694918
```

3.3.2 Cleaning the dataset using NumPy syntax

It is possible to clean the dataframe using masking syntax. First, let's check how many default values are stored for each column (all but the date):

```
# Check which wave has -99 values for every variables
for c in ['height', 'heightMax', 'period', 'energy', 'direction', 'temperature']:
    n = np.count_nonzero(df[c]<=-99)
    print('{}: {} wave have <=-99'.format(c, n))
```

```
height: 85 wave have <=-99
heightMax: 85 wave have <=-99
period: 85 wave have <=-99
energy: 85 wave have <=-99
direction: 271 wave have <=-99
temperature: 262 wave have <=-99
```

```
# Simply take value above -99
print(df[df>-99].head())
```

```
              date  height  heightMax  period  energy  direction  temperature
0  01/01/2017 00:00     NaN        NaN     NaN     NaN        NaN          NaN
1  01/01/2017 00:30   0.875       1.39   4.421   4.506        NaN          NaN
2  01/01/2017 01:00   0.763       1.15   4.520   5.513       49.0        25.65
3  01/01/2017 01:30   0.770       1.41   4.582   5.647       75.0        25.50
4  01/01/2017 02:00   0.747       1.16   4.515   5.083       91.0        25.45
```

```
# Removing all entry (line) which has at least one default value
for c in ['height', 'heightMax', 'period', 'energy', 'direction', 'temperature']:
    df = df[df[c]>-99]

print(df.head())
```

```
              date  height  heightMax  period  energy  direction  temperature
2  01/01/2017 01:00   0.763       1.15   4.520   5.513       49.0        25.65
3  01/01/2017 01:30   0.770       1.41   4.582   5.647       75.0        25.50
4  01/01/2017 02:00   0.747       1.16   4.515   5.083       91.0        25.45
5  01/01/2017 02:30   0.718       1.61   4.614   6.181       68.0        25.45
6  01/01/2017 03:00   0.707       1.34   4.568   4.705       73.0        25.50
```

We can now check how many default values we get:

```
for c in ['height', 'heightMax', 'period', 'energy', 'direction', 'temperature']:
    n = np.count_nonzero(df[c]<=-99)
    print('{}: {} wave have <=-99'.format(c, n))
```

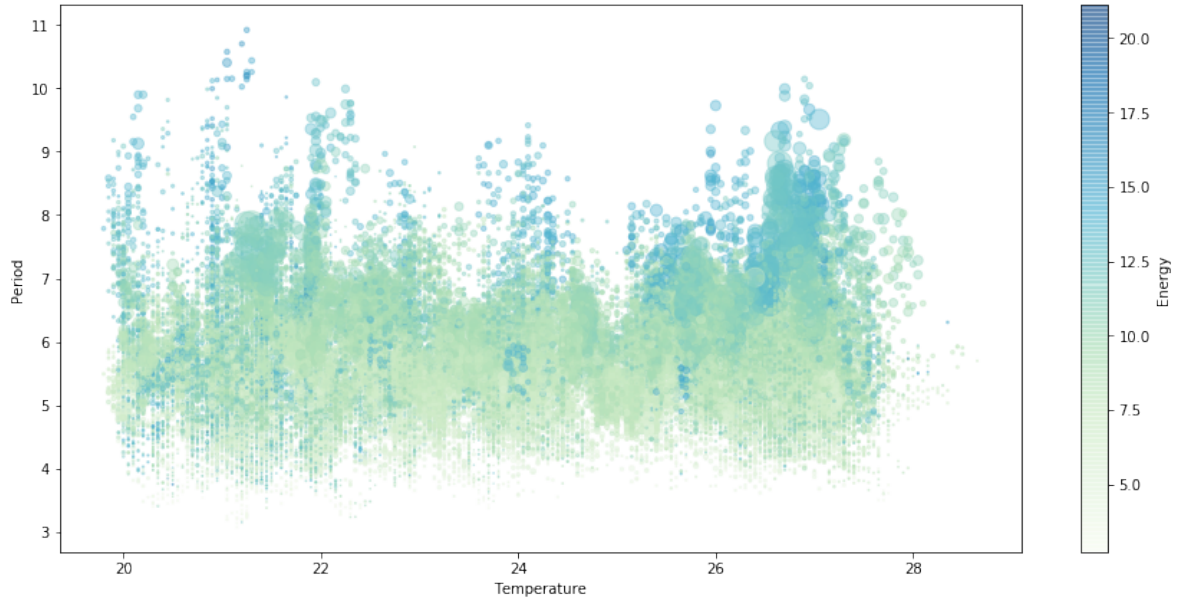
```
height: 0 wave have <=-99
heightMax: 0 wave have <=-99
period: 0 wave have <=-99
energy: 0 wave have <=-99
direction: 0 wave have <=-99
temperature: 0 wave have <=-99
```

3.3.3 Extracting NumPy arrays and plotting

It is possible to perform computations using pandas columns directly, but it can be useful to extract NumPy arrays in case of more complex broadcasting or indexing. This can be done using the `df[col].values` command.

```
# Get numpy array for further manipulations
T = df['temperature'].values
P = df['period'].values
H = df['heightMax'].values
E = df['energy'].values
```

```
# Plot temperature vs period vs max_height vs energy
plt.figure(figsize=(15, 7))
plt.scatter(T, P , s=H**3, c=E, cmap='GnBu', alpha=0.4)
plt.colorbar(label='Energy')
plt.xlabel('Temperature')
plt.ylabel('Period');
```



3.3.4 Add information in a DataFrame

One of the nice features of pandas is the ability to easily store the result of a computation as a new column. For instance, it's a common practice in machine learning to *normalize* the input variables, i.e., transform them to have a mean of 0 and a variance of 1.0. The following example shows how to add new columns that are normalized:

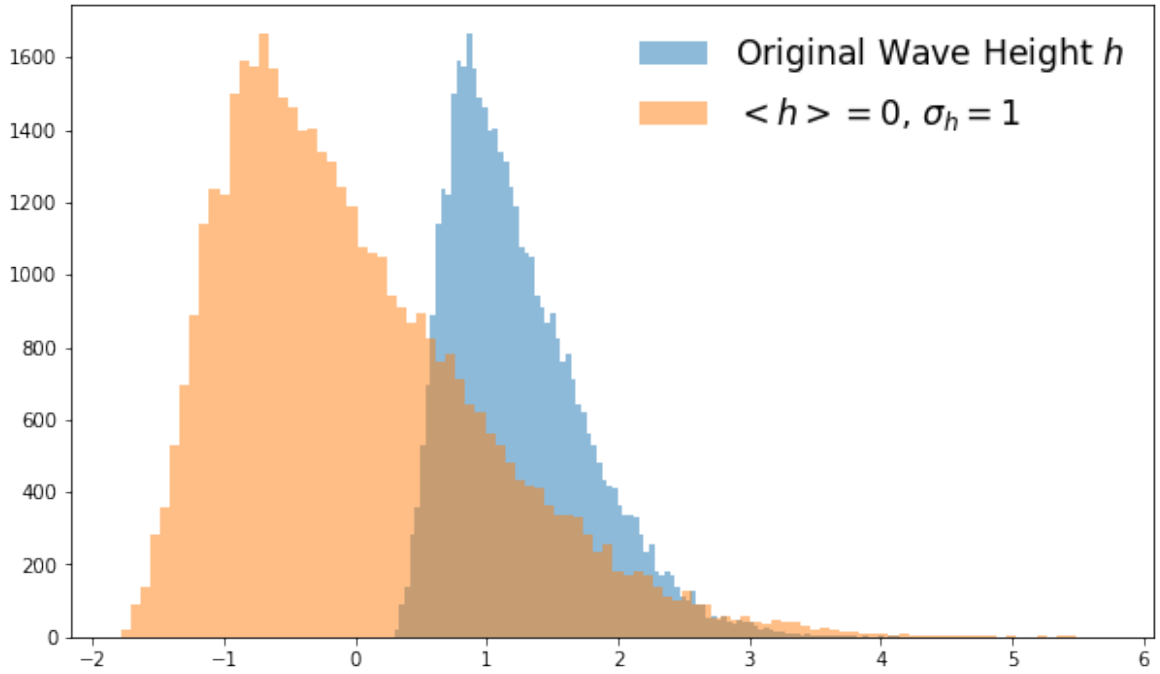
```
def add_normalized_variable_to_df(col):

    # Get a numpy arrays
    v = df[col].values

    # Replace NaN by 0.0
    v[np.isnan(v)] = 0

    # Compute quantities
    v_mean = np.mean(v)
    v_rms = np.sqrt(np.mean((v-v_mean)**2))

    # Add them into the pandas dataframe
    df[col+'_normalized'] = (v-v_mean)/v_rms

    return


for c in ['height', 'heightMax', 'period', 'energy', 'direction', 'temperature']:
    add_normalized_variable_to_df(c)

# Get only normalized column
```

```
normalized_cols = [c for c in df.columns.tolist() if '_normalized' in c]
print(df[normalized_cols].head())
```

```
   height_normalized  heightMax_normalized  period_normalized  \
2          -0.898215             -1.047342          -1.184338
3          -0.884973             -0.757690          -1.117565
4          -0.928484             -1.036201          -1.189723
5          -0.983346             -0.534881          -1.083102
6          -1.004155             -0.835673          -1.132643

   energy_normalized  direction_normalized  temperature_normalized
2          -1.463956             -2.044360                0.762152
3          -1.407891             -0.973294                0.694918
4          -1.643866             -0.314176                0.672506
5          -1.184467             -1.261658                0.672506
6          -1.802020             -1.055683                0.694918
```

One can simply plot the content of a Pandas DataFrame using the name of the column (a more direct alternative than extracting a NumPy array). For instance, one can compare the evolution of the wave height after the transformation:

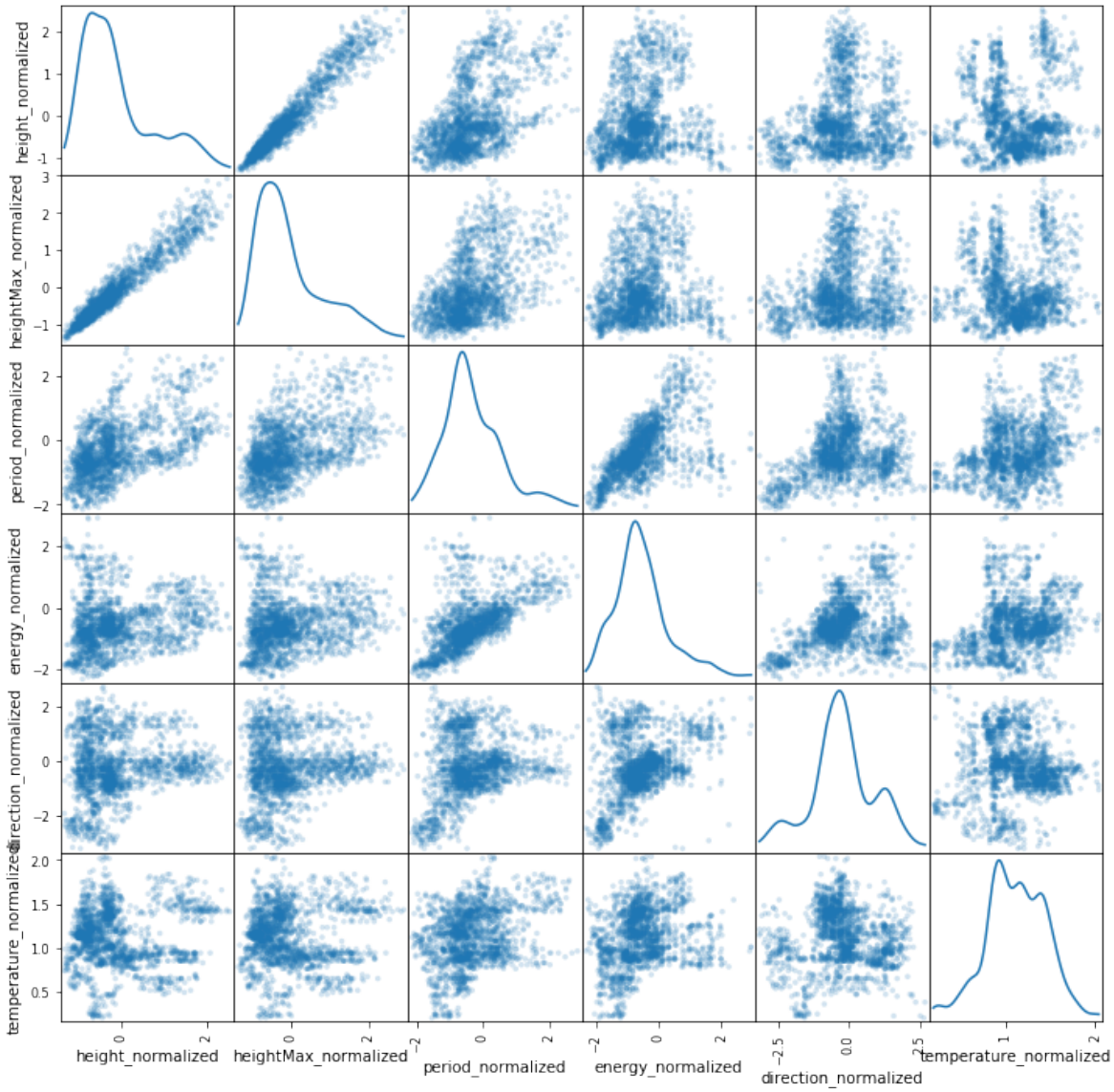
```
plt.figure(figsize=(10, 6))
plt.hist(df['height'], bins=100, alpha=0.5, label='Original Wave Height $h$')
plt.hist(df['height_normalized'], bins=100, alpha=0.5, label='$<h> = 0$, $\sigma_{h}=1$')
plt.legend(frameon=False, fontsize='xx-large');
```



3.3.5 Data visualization with Pandas

There are also many plotting functions already included in the Pandas library. To show only one example (all functions are described in the Pandas Visualization Tutorial), here is the *scatter matrix* between variables (defined as a subset of the ones stored in the dataframe) obtained in a single line of code:

```
from pandas.plotting import scatter_matrix
scatter_matrix(df[normalized_cols][:2000], figsize=(12, 12), alpha=0.2, s=50,
↪  diagonal='kde');
```



3.4 Mathematics, physics, and engineering: SciPy

The SciPy project is a Python-based ecosystem of open-source software for mathematics, science, and engineering. In particular, the following core packages are part of it: NumPy, Matplotlib, Pandas, the SciPy

library, and SymPy (symbolic calculations with mathematical expressions *à la* Mathematica).

3.4.1 General overview

Obviously, there is no way to extensively present the SciPy library in this short introduction, but one can quickly summarize a few features and illustrate one with a concrete and useful example: fitting data points with a function. Among the main features, the SciPy library contains:

- Integration (`scipy.integrate`): integrals, differential equations, etc.
- Optimization (`scipy.optimize`): minimization, fits, etc.
- Interpolation (`scipy.interpolate`): smoothing methods, etc.
- Fourier Transforms (`scipy.fftpack`): spectral analysis, etc.
- Signal Processing (`scipy.signal`): transfer functions, filtering, etc.
- Linear Algebra (`scipy.linalg`): matrix operations, diagonalization, determinant, etc.
- Statistics (`scipy.stats`): random numbers, probability density function, cumulative distribution, etc.

3.4.2 Curve Fitting Example

```
from scipy import optimize
from scipy import stats
```

Let's now show how to perform a fit of data with error bars using one particular function of `scipy.optimize`. First, we need to generate some data. We will choose 20 measurements with some noise of ~30% and a combined uncertainty of an absolute 0.1 uncertainty and 10% relative uncertainty.

```
Npoints, Nsampling = 20, 1000
xcont = np.linspace(-5.0, 3.5, Nsampling)
x = np.linspace(-5, 3.0, Npoints)
y = 2*(np.sin(x/2)**2 + np.random.random(Npoints)*0.3)
dy = np.sqrt(0.10**2 + (0.10*y)**2)
```

Then we need to define functions with which we want to fit our data, for example a degree 1 polynomial. The syntax has to be `func(x, *pars)`:

```
def pol1(x, p0, p1):
    return p0 + x*p1
```

The following lines actually perform the fit and return both the optimal parameters and the covariances for the degree 1 polynomial:

```
p, cov = optimize.curve_fit(pol1, x, y, sigma=dy)
```

One can then generalize the procedure by plotting the result of the fit for polynomials of several degrees, after having plotted the data. This is a good way to compare different models for the same data. First, we define an arbitrary degree polynomial `poly_func()` and we *vectorize* it using `np.vectorize` so that it can accept NumPy arrays.

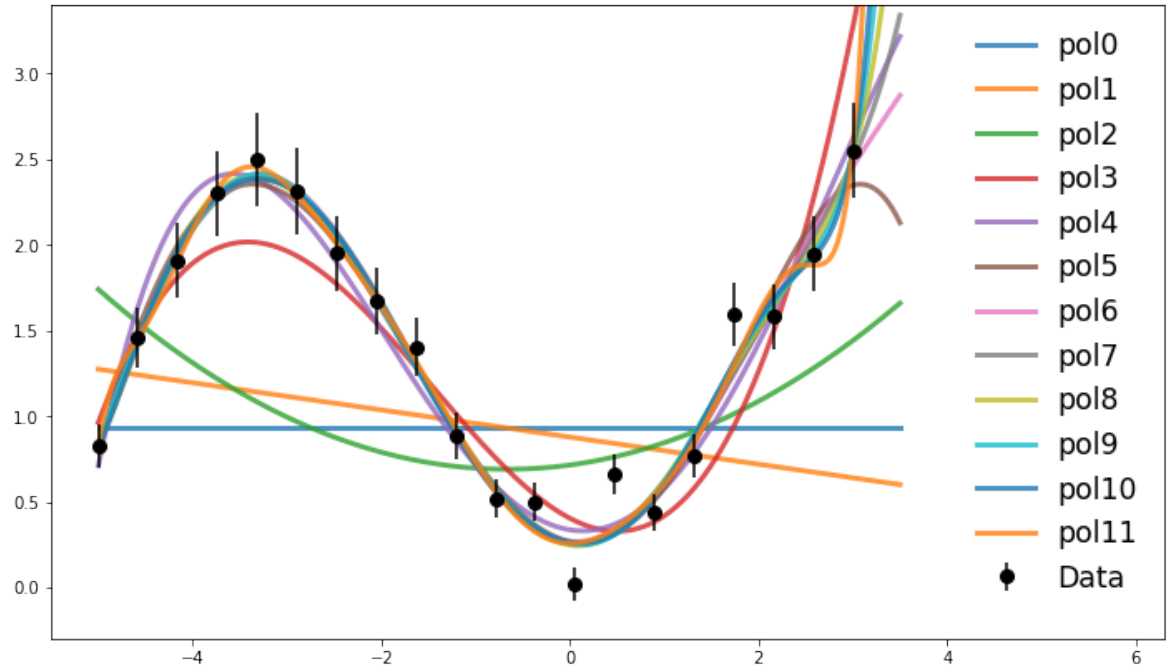
```
def pol_func(x, *coeff):
    '''Arbitrary degree polynom: f(x) = a0 + a1*x + a2*x^2 + ... aN*x^N'''
    a = np.array([coeff[i]*x**i for i in range(len(coeff))])
    return np.sum(a)

pol_func = np.vectorize(pol_func)
```

In the previous call to `optimize.curve_fit()`, we didn't use an additional argument. For this example, we need to specify at least the starting point of the parameters `p0` because the number of parameters will be assessed using `len(p0)` (it's not known a priori since it is dynamically allocated). Other options can be specified, such as the minimum and maximum allowed values of parameters. Here is a wrap-up function performing the fit for an arbitrary polynomial degree:

```
def fit_polynom(degree):
    nPars = degree+1
    p0, pmin, pmax = [1.0]*nPars, [-10]*nPars, [10]*nPars
    fit_options = {'p0': p0, 'bounds': (pmin, pmax), 'check_finite': True}
    par, cov = optimize.curve_fit(pol_func, x, y, sigma=dy, **fit_options)
    return par, cov

degree_max = 12
```

The following code tries every polynomial function up to a degree `degree_max`, performs the fit, and overlays the result for each function together with the experimental data on the same figure.

```
# Figure for the result
fig = plt.figure(figsize=(12,7))

# Fitting & plotting
for d in np.arange(0, degree_max):
    par, cov = fit_polynom(d)
    plt.plot(xcont, pol_func(xcont, *par), label='pol{}'.format(d),
             linewidth=3, alpha=0.8)

# Plotting data
style = {'marker': 'o', 'color': 'black', 'markersize': 8,
         'linestyle': '', 'zorder': 10, 'label': 'Data'}
plt.errorbar(x, y, yerr=dy, **style)

# Plot cosmetics
plt.xlim(-5.5, 6.3)
plt.ylim(-0.3, 3.4)
plt.legend(frameon=False, fontsize='xx-large');
```



It is possible to quantify how well a given model explains the observations by computing what we call the *goodness of fit*. In a frequentist approach, this can be assessed by the fraction of pseudo-data coming from - in principle - repeating the exact same experiment, with a worse agreement for a given model. The agreement can be quantified using $\chi^2 = \sum_{i=1}^{n} \frac{(y_i - f(x_i))^2}{\sigma_i^2}$ and its probability density function (PDF) directly gives access to the fraction of "worst pseudo-data" (by integrating the PDF from χ^2 to ∞). More precisely, one can use the cumulative distribution function (CDF) of χ^2 computed with n degrees of freedom, for instance `Npoints`, *i.e.* `len(x)`. More details can be found, for example, in the statistics review of the Particle Data Group. The following two functions allow computing the goodness of fit:

```
def get_chi2_nDOF(y, dy, yfit):
    r = (y-yfit)/dy
    return  np.sum(r**2), len(y)

def get_pvalue(chi2, nDOF):
    return 1-stats.chi2.cdf(chi2, df=nDOF)
```

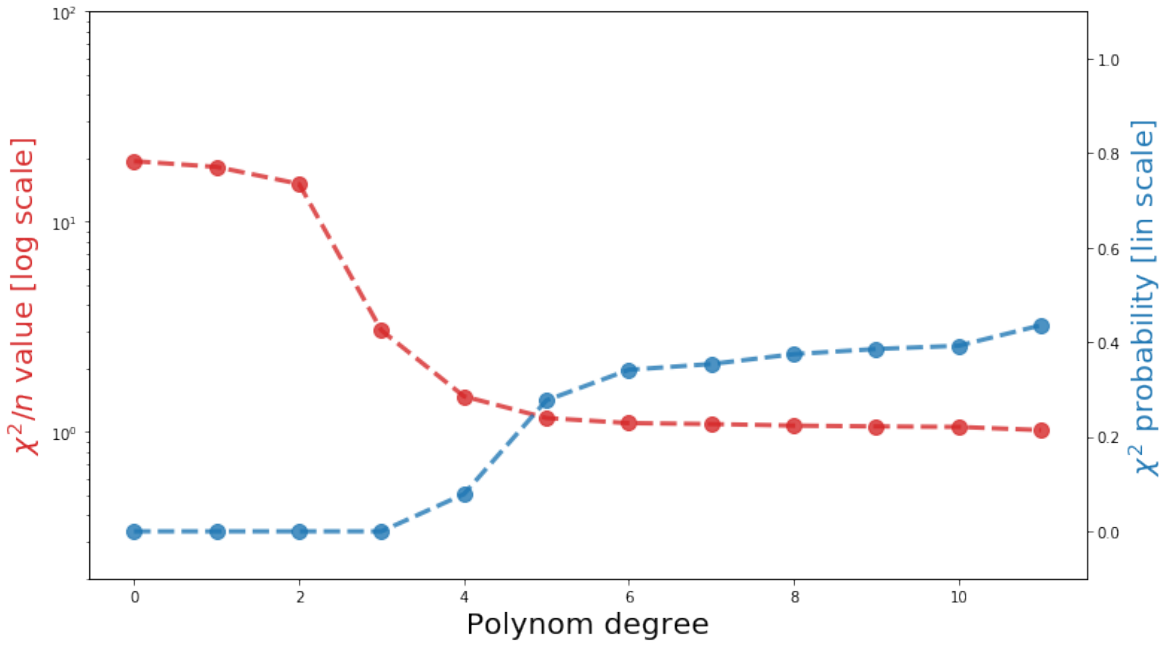
We can now perform all of these fits and extract the goodness of fit (χ^2 and p-value) for each model.

```
# Fitting and getting p-value
degree, chiSquare, pvalue = [], [], []
for d in np.arange(degree_max):
    par, cov = fit_polynom(d)
    c2, n = get_chi2_nDOF(y, dy, pol_func(x, *par))
    degree.append(d), chiSquare.append(c2), pvalue.append(get_pvalue(c2, n))
```

The following piece of code plots both the χ^2 and the p-value versus the degree of the polynomial using two different y-axes. This provides another way to use Matplotlib by defining explicit objects such as `ax` and `fig` and calling methods on those (referred to as the *stateless approach*), instead of using functions on `plt`

(referred to as the *stateful approach*). For more details on these different approaches, see this RealPython post.

```
# Plotting the result with 2 different axis
fig, ax1 = plt.subplots(figsize=(12,7))
ax1.set_xlabel('Polynom degree', fontsize=20)
style = {'marker': 'o', 'markersize': 10, 'alpha': 0.8,
         'linestyle': '--', 'linewidth': 3}

# Plot chi2/n
ax1.semilogy(degree, np.array(chiSquare)/Npoints, color='tab:red', **style)
ax1.set_ylim(0.2, 100)
ax1.set_ylabel('$\chi^2/n$ value [log scale]', color='tab:red', fontsize=20)

# Plot p-values
ax2 = ax1.twinx()
ax2.plot(degree, pvalue, color='tab:blue', **style)
ax2.set_ylim(-0.1, 1.1)
ax2.set_ylabel('$\chi^2$ probability [lin scale]', color='tab:blue', fontsize=20);
```



Chapter 4

High-dimensional data manipulation

> **Skills to take away**
>
> - *Basic*: Computation along each axis, distance computation, plotting of n-dim arrays
> - *Medium*: Find closest elements along a direction, select pairs based on their distance
> - *Expert*: Pairing objects along a given dimension

4.1 Introduction

The present chapter makes use of the concepts previously introduced to perform computations that one would do with high-dimensional data. Typically, if a given dataset consists of several 3D positions for each observation, one has to deal with many numbers. It is possible that grouping these vectors by pairs is relevant to understand the problem. Or maybe other operations within these various 3D vectors are useful. Since we want to use the full power of NumPy, all these computations cannot be done with an explicit loop over observations and/or over vectors.

This chapter considers a few of these typical use cases and their implementation using NumPy, using a simple toy dataset made by hand. Most likely, you will never face such a situation for a machine learning algorithm, but it is good to go through these examples to show some of the limitations of not being able to loop over observations.

Let's first perform the usual imports:

```
import itertools
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Then, one can set up the default Matplotlib style for all following plots. More details on available options can be found in how to customize Matplotlib.

```
import matplotlib as mpl
mpl.rcParams['legend.frameon'] = False
mpl.rcParams['legend.fontsize'] = 'xx-large'
mpl.rcParams['xtick.labelsize'] = 16
mpl.rcParams['ytick.labelsize'] = 16
mpl.rcParams['axes.titlesize'] = 18
mpl.rcParams['axes.labelsize'] = 18
mpl.rcParams['lines.linewidth'] = 2.5
mpl.rcParams['figure.figsize'] = (10, 7)
```

4.2 Data model and goals

We consider 1 million observations, each defined by ten 3D vectors $(r_0, ..., r_9)$ where $r_i = (x, y, z)$ (arrow for vector will be omitted from now on). These pseudo-data can represent positions in space or RGB colors for an image. This is just an example to play with and apply NumPy concepts for both simple computations (element-by-element functions, statistics calculations) and more complex computations exploiting the multi-dimensional structure of the data. For example, one might want to compute the distance between all pairs (r_i, r_j), which has to be done without a loop.

Using the `np.random` module, it is possible to generate n-dimensional arrays easily. In our case, we want to generate an array containing our observations with 3 dimensions (or *axes* in NumPy language), and the size along each of these axes will have the following value and meaning:

- `axis=0`: over 1 million events
- `axis=1`: over 10 vectors
- `axis=2`: over 3 coordinates

```
r = np.random.random_sample((1000000, 10, 3))
```

It is possible to print the first two observations as follows:

```
print(r[0:2])
```

```
[[[0.01789452 0.66209367 0.06888675]
  [0.39922964 0.87771381 0.01192809]
  [0.88259998 0.88655328 0.30233561]
  [0.53178797 0.69359593 0.05459176]
  [0.69979407 0.82627363 0.63028438]
  [0.19121361 0.80913874 0.2813931 ]
  [0.22248198 0.78734303 0.39821198]
  [0.41256443 0.81143611 0.71874392]
  [0.10597153 0.61954029 0.15438807]
  [0.0586176  0.5567407  0.19918271]]

 [[0.36893905 0.29645313 0.64392255]
```

```
  [0.6084518  0.71718717 0.18079204]
  [0.79397165 0.75437849 0.81858946]
  [0.28424271 0.32153672 0.29690342]
  [0.62484107 0.23856055 0.4057989 ]
  [0.72375291 0.77319732 0.68777312]
  [0.98876848 0.32340734 0.49148584]
  [0.55165453 0.02655616 0.65149546]
  [0.42893549 0.25157921 0.26770818]
  [0.60597909 0.79425594 0.24029925]]]
```

4.3 Mean over the different axes

4.3.1 Mean over observations (axis=0)

This mean will average all observations, i.e., over the first dimension, returning an array of dimension `(10, 3)` corresponding to the average $r_i = (x_i, y_i, z_i)$ over the observations.

```
m0 = np.mean(r, axis=0)
print(m0.shape)
```

```
(10, 3)
```

Note the computation time of 30 ms for 30 averages over a million numbers.

```
%timeit np.mean(r, axis=0)
```

```
33.8 ms ± 1.28 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

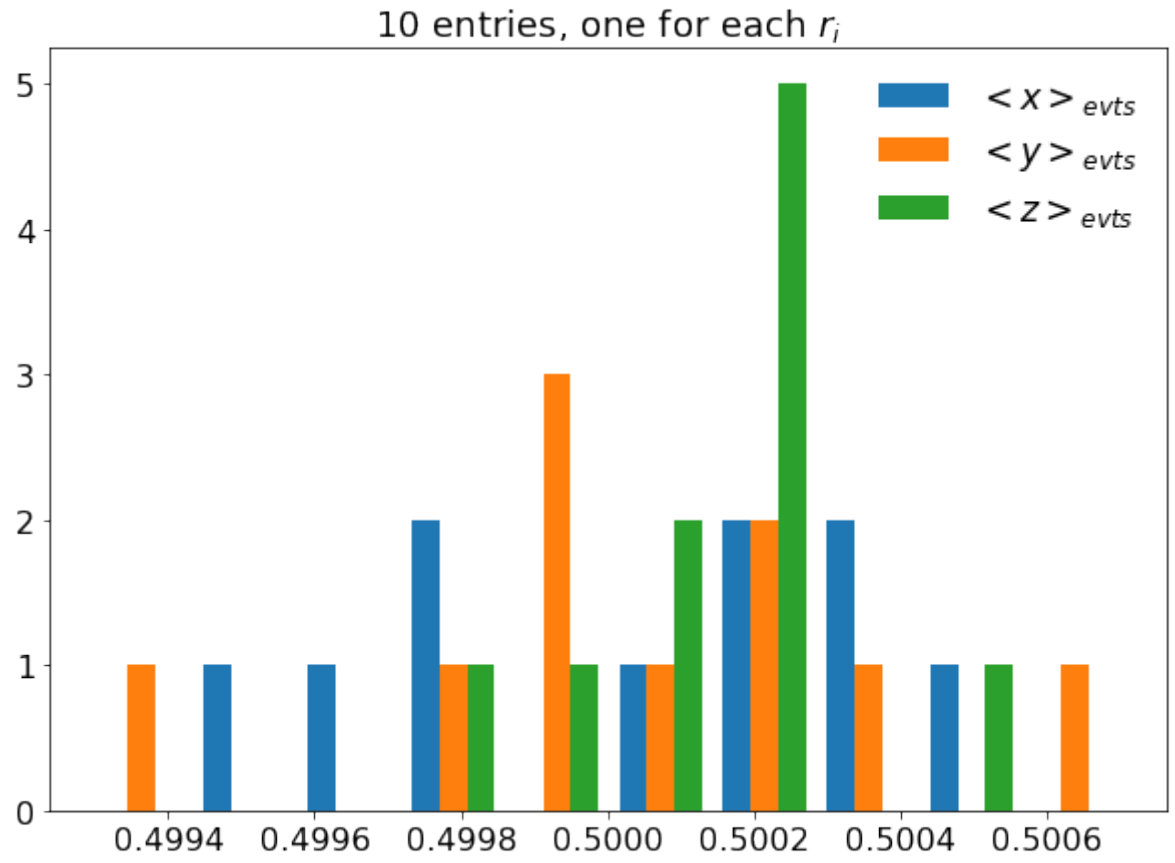
While it takes 10 times longer for a single mean over a million numbers with an explicit loop, the gain of vectorization is a factor of 300.

```
def explicit_loop(array):
    res=0
    for a in array:
        res += a/len(array)

%timeit explicit_loop(np.random.random_sample(size=1000000))
```

```
328 ms ± 7.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

The distributions of `m0` obtained with `plt.hist()` result in three separate histograms (one for each x, y, z), each having 10 entries (one per r_i).

```
plt.hist(m0, label=['$<x>_{evts}$', '$<y>_{evts}$', '$<z>_{evts}$'])
plt.title('10 entries, one for each $r_i$')
plt.legend();
```



4.3.2 Mean over the 10 vectors (axis=1)

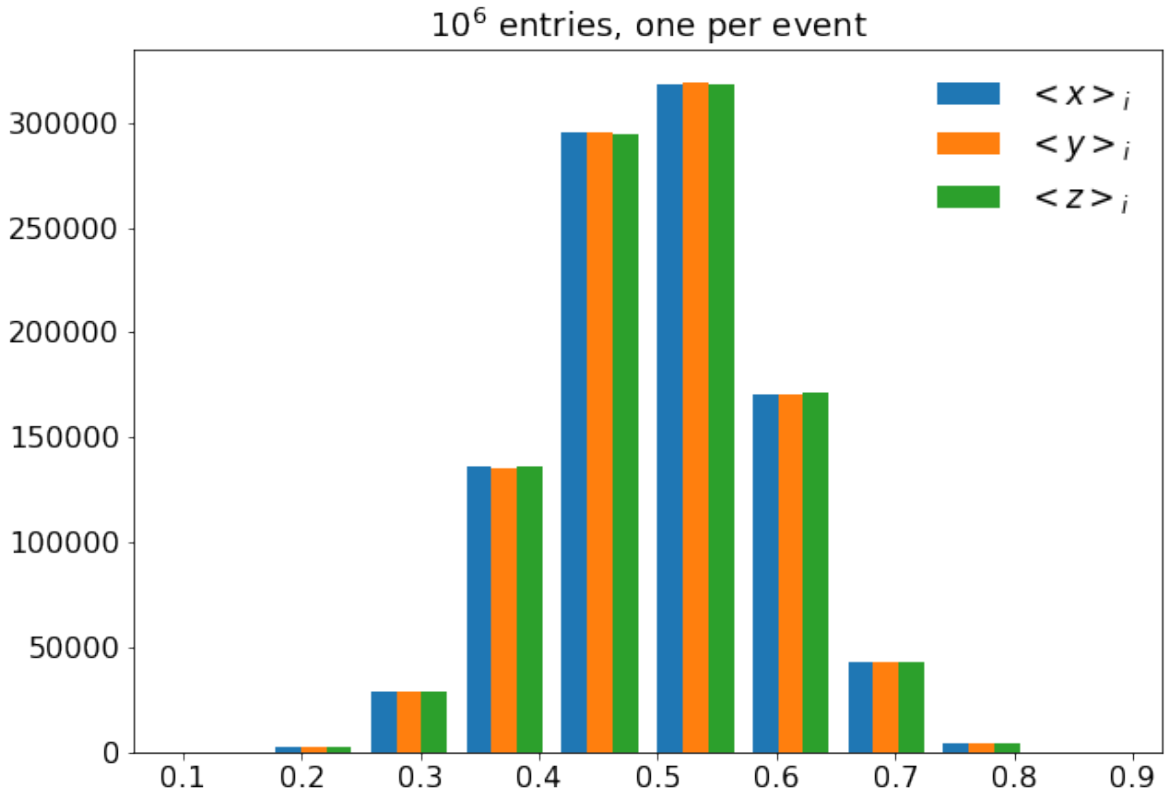
This one will compute the average over the 10 vectors for each observation, resulting in a `(1000000, 3)` shaped array as shown below. This represents the 3D barycenter of each observation.

```
m1 = np.mean(r, axis=1)
print(m1.shape)
```

```
(1000000, 3)
```

One can plot the obtained array `m1` using `plt.hist()`, which results in 3 histograms, each with a million entries:

```
plt.hist(m1, label=['$<x>_{i}$', '$<y>_{i}$', '$<z>_{i}$'])
plt.title('$10^6$ entries, one per event')
plt.legend();
```



4.3.3 Mean over the coordinates (`axis=2`)

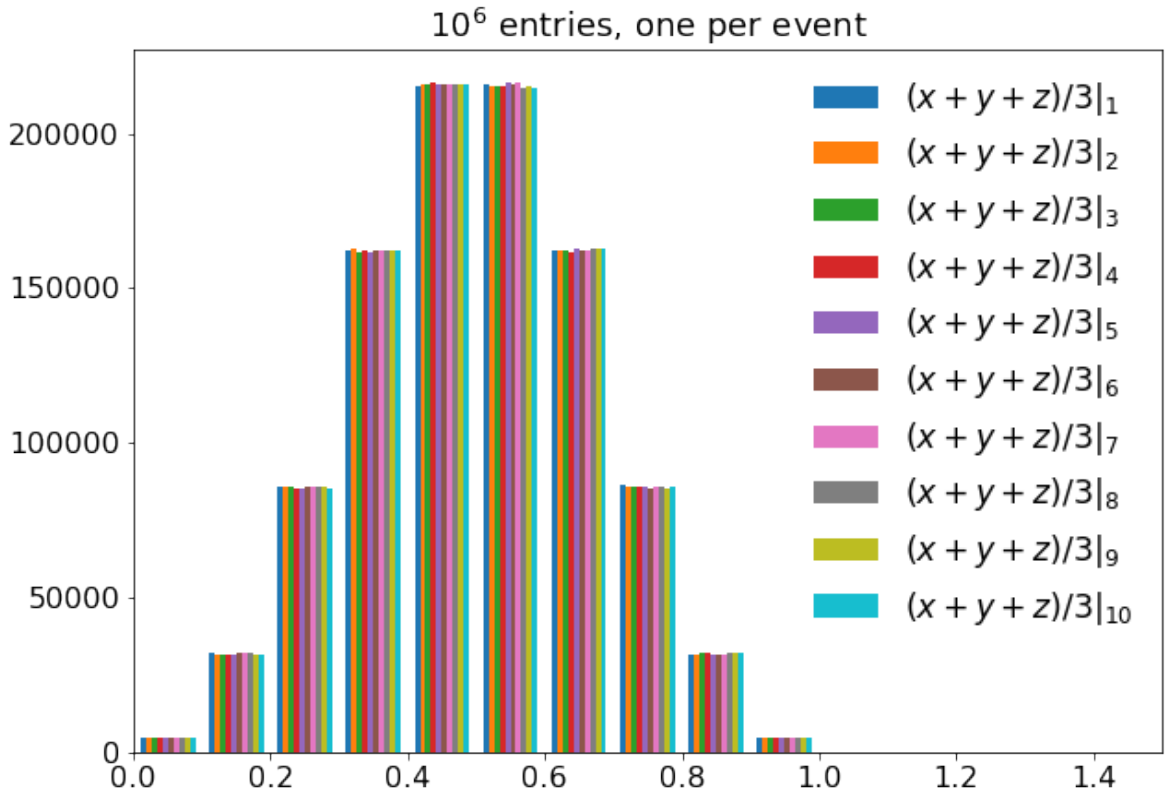
This directly computes the average over the three coordinates $(x + y + z)/3$ for each vector of each event, resulting in 10 values per event.

```
m2 = np.mean(r, axis=2)
print(m2.shape)
```

```
(1000000, 10)
```

The `plt.hist()` of the resulting array `m2` corresponds to 10 histograms, each with a million entries.

```
names = ['$(x+y+z)/3|_{'+'{}'.format(i)+'}$' for i in range(1, 11)]
plt.hist(m2, label=names)
plt.title('$10^6$ entries, one per event')
plt.xlim(0, 1.5)
plt.legend();
```



4.4 Distance computation

Computing specific distances within a given event is relevant for many applications (distances here can be seen as any type of metric). For example, these computations are crucial in learning algorithms based on the nearest neighbor approach. In collider physics, it is always useful to compute the angle between two objects (tracks, deposits, particles, etc.) in order to compute invariant masses or isolation in a given cone, etc.

4.4.1 Distance to a reference r_0

We can start simple by defining a new origin `r0`

```
r0 = np.array([1, 2, 1])
```

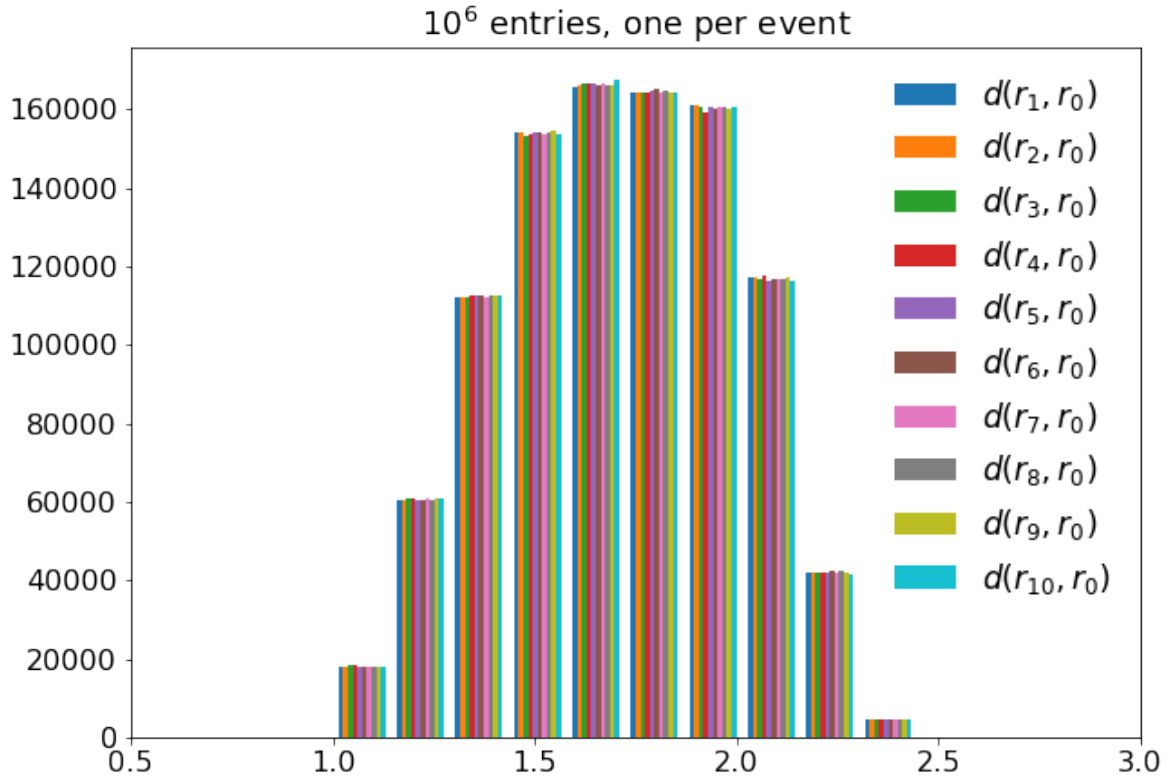
and compute the distance to this new origin for all points, using `**2` to square all numbers, performing the sum over the coordinates (`axis=2`), and taking the square root of everything with `**0.5`:

```
d = np.sum((r-r0)**2, axis=2)**0.5
print(d.shape)
```

```
(1000000, 10)
```

As expected, the result is 10 numbers for each of the events, which can be easily plotted.

```
names = ['$d(r_{'+'{}'.format(i)+'},r_0)$' for i in range(1, 11)]
plt.hist(d, label=names)
plt.title('$10^6$ entries, one per event')
plt.xlim(0.5, 3)
plt.legend();
```



4.4.2 Distance between r_i and $< r >_i$ for each event

Another calculation is to compute the average position for each event and see how distant each vector is from this position. To perform such a calculation, we will use NumPy array broadcasting. Let's first compute the average position for every event:

```
r_mean = np.mean(r, axis=1)
```

Now, let's broadcast this array of shape `(1e6, 3)` with the full dataset, i.e. an array of shape `(1e6, 10, 3)`, by computing the distance for each point.

```
try:
    d_to_mean = np.sum((r-r_mean)**2, axis=2)**0.5
except ValueError:
    print('Impossible for {} and {}'.format(r.shape, r_mean.shape))
```

```
Impossible for (1000000, 10, 3) and (1000000, 3)
```

There is one missing dimension, describing the 10 positions, which has to be created so that the array can be copied 10 times along this dimension.

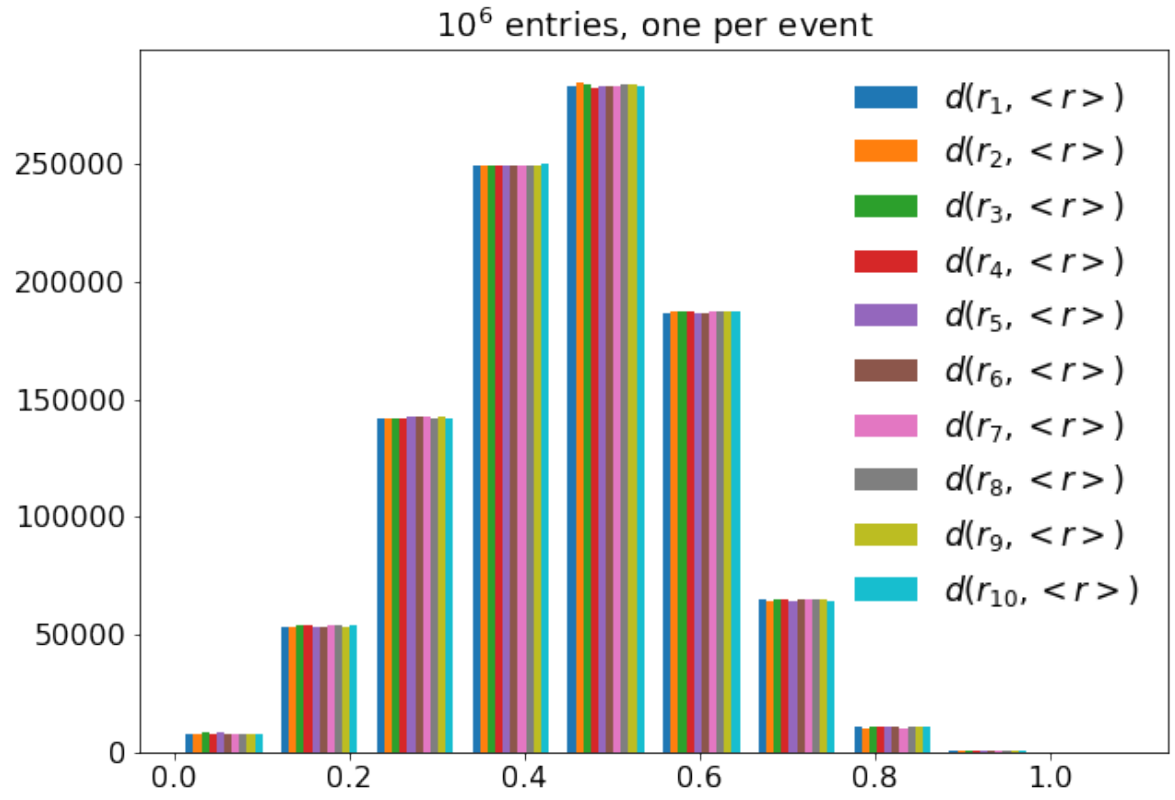
```
r_mean_3d = r_mean[:, np.newaxis, :]
```

We can now retry the operation:

```
try:
    d_to_mean = np.sum((r-r_mean_3d)**2, axis=2)**0.5
    print('Possible for {} and {}'.format(r.shape, r_mean_3d.shape))
except ValueError:
    print('Impossible for {} and {}'.format(r.shape, r_mean_3d.shape))
```

```
Possible for (1000000, 10, 3) and (1000000, 1, 3)
```

```
names = ['$d(r_{'+'{}'.format(i)+'},<r>)$' for i in range(1, 11)]
plt.hist(d_to_mean, label=names)
plt.title('$10^6$ entries, one per event')
plt.legend();
```



4.5 pairing 3d vectors for each observation, without a loop

Being able to pair objects is obviously important for many types of calculations. This allows us to probe correlations at the first order, identify sub-systems, etc. In the traditional way, pairing would involve a *for* loop in which the combinatorics can be done for each event. However, when working with NumPy, we can perform the combinatorics in a vectorized way and return a new NumPy array containing all the pairs. Once this is done, we can perform many types of computations on this new array.

4.5.1 Finding all possible (r_i, r_j) pairs for all events

One solution to perform this task without a *for* loop was found on stackoverflow. The idea is to simply work with indices to build the pairs (since it doesn't really matter what the nature of the objects is), and use NumPy's *fancy* indexing. Let's proceed step by step with a smaller array to understand the procedure (namely 2 observations of 5 positions):

```
a = r[0:2,0:5]
print(a)
```

```
[[[0.01789452 0.66209367 0.06888675]
  [0.39922964 0.87771381 0.01192809]
  [0.88259998 0.88655328 0.30233561]
  [0.53178797 0.69359593 0.05459176]
  [0.69979407 0.82627363 0.63028438]]

 [[0.36893905 0.29645313 0.64392255]
  [0.6084518  0.71718717 0.18079204]
  [0.79397165 0.75437849 0.81858946]
  [0.28424271 0.32153672 0.29690342]
  [0.62484107 0.23856055 0.4057989 ]]]
```

Since we want to work with the indices of the 5 vectors, we create a NumPy array of integers going from 0 to 4 (`a.shape[1]` is the number of elements along the second dimension, i.e., 5):

```
array_indices = np.arange(a.shape[1])
print(array_indices)
```

```
[0 1 2 3 4]
```

Then, we use the package `itertools` to deal with combinatorics. This will return an *iterator* that can be turned into a NumPy array using `np.fromiter()`. However, this function requires specifying the data type `dt`, which is done using a structured array syntax here (i.e., `[(varName1, type1), (varName2, type2)]`). For more details on data types, check this documentation page.

```
dt = np.dtype([('index1', np.intp), ('index2', np.intp)])
print(dt)
```

```
[('index1', '<i8'), ('index2', '<i8')]
```

```
array_indice_comb = np.fromiter(itertools.combinations(array_indices, 2), dt)
print(array_indice_comb)
```

```
[(0, 1) (0, 2) (0, 3) (0, 4) (1, 2) (1, 3) (1, 4) (2, 3) (2, 4) (3, 4)]
```

The next step is to format these numbers into an indices array with the proper dimensions, so that when we do `a[indices]`, we get all the pairs. For instance, we need to have all 10 pairs, each with two elements corresponding to a shape `indices.shape=(10,2)`. We can achieve this in two steps:

- `array_indice_comb.view(np.intp)` returns the exact same data as `array_indice_comb`, but as a 1D array of positive integers.
- We reshape the resulting array with `reshape(-1, 2)`, where -1 means "compute the size of the first dimension to have 2 objects (we want pairs!) in the second dimension.

```
indices = array_indice_comb.view(np.intp).reshape(-1, 2)
print(indices)
```

```
[[0 1]
 [0 2]
 [0 3]
 [0 4]
 [1 2]
 [1 3]
 [1 4]
 [2 3]
 [2 4]
 [3 4]]
```

The final step is to exploit fancy indexing along `axis=1`, i.e., the 5 spatial positions. In practice, for each observation `iobs`, we want to have `a[iobs, indices]`. There are two ways to do this: - (a) `a[:, indices]` - (b) using the NumPy function `np.take(a, indices, axis)` which makes the code more independent from the structure of `a`.

```
a_pairs = np.take(a, indices, axis=1)
print(a_pairs.shape)
```

```
(2, 10, 2, 3)
```

```
a_pairs = a[:,indices]
print(a_pairs.shape)
```

```
(2, 10, 2, 3)
```

We now have 2 events, each consisting of 10 pairs. Each pair consists of 2 objects (still a pair!), each with 3 coordinates (spatial positions). We can print all 10 pairs for the first observation:

```
print(a_pairs[0])
```

```
[[[0.01789452 0.66209367 0.06888675]
  [0.39922964 0.87771381 0.01192809]]

 [[0.01789452 0.66209367 0.06888675]
  [0.88259998 0.88655328 0.30233561]]

 [[0.01789452 0.66209367 0.06888675]
  [0.53178797 0.69359593 0.05459176]]

 [[0.01789452 0.66209367 0.06888675]
  [0.69979407 0.82627363 0.63028438]]

 [[0.39922964 0.87771381 0.01192809]
  [0.88259998 0.88655328 0.30233561]]

 [[0.39922964 0.87771381 0.01192809]
  [0.53178797 0.69359593 0.05459176]]

 [[0.39922964 0.87771381 0.01192809]
  [0.69979407 0.82627363 0.63028438]]

 [[0.88259998 0.88655328 0.30233561]
  [0.53178797 0.69359593 0.05459176]]

 [[0.88259998 0.88655328 0.30233561]
  [0.69979407 0.82627363 0.63028438]]

 [[0.53178797 0.69359593 0.05459176]
  [0.69979407 0.82627363 0.63028438]]]
```

Once understood, we can wrap up this code into a function where we generalize the number of objects we want to group `n` and the axis along which we want to group `axis`:

```
def combs_nd(a, n, axis=0):
    i = np.arange(a.shape[axis])
    dt = np.dtype([('', np.intp)]*n)
    i = np.fromiter(itertools.combinations(i, n), dt)
    i = i.view(np.intp).reshape(-1, n)
    return np.take(a, i, axis=axis)
```

As a sanity check, we can recompute `a_pair` and compare it with the previous results.

```
a_pairs = combs_nd(a=r[0:2,0:5], n=2, axis=1)
print(a_pairs[0])
```

```
[[[0.01789452 0.66209367 0.06888675]
  [0.39922964 0.87771381 0.01192809]]

 [[0.01789452 0.66209367 0.06888675]
  [0.88259998 0.88655328 0.30233561]]

 [[0.01789452 0.66209367 0.06888675]
  [0.53178797 0.69359593 0.05459176]]

 [[0.01789452 0.66209367 0.06888675]
  [0.69979407 0.82627363 0.63028438]]

 [[0.39922964 0.87771381 0.01192809]
  [0.88259998 0.88655328 0.30233561]]

 [[0.39922964 0.87771381 0.01192809]
  [0.53178797 0.69359593 0.05459176]]

 [[0.39922964 0.87771381 0.01192809]
  [0.69979407 0.82627363 0.63028438]]

 [[0.88259998 0.88655328 0.30233561]
  [0.53178797 0.69359593 0.05459176]]

 [[0.88259998 0.88655328 0.30233561]
  [0.69979407 0.82627363 0.63028438]]

 [[0.53178797 0.69359593 0.05459176]
  [0.69979407 0.82627363 0.63028438]]]
```

It can be interesting to see that this operation takes less than a second for a million observations of 10 vectors, meaning 45 pairs.

```
%timeit combs_nd(a=r, n=2, axis=1)
```

```
614 ms ± 12.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

4.5.2 Computing (minimum) distances on these pairs

Once we have these pairs, we can, for example, compute all the distances and find which pair has the closest objects. Starting with the pairs:

```
pairs = combs_nd(a=r, n=2, axis=1)
```

We can then define the vectorial difference between the two positions of a pair and compute the Euclidean distance.

```
dp = pairs[:, :, 0, :]-pairs[:, :, 1, :]
distances = (np.sum(dp**2, axis=2))**0.5
```

And get the minimum distance for each event:

```
smallest_distance = np.min(distances, axis=1)
print(smallest_distance.shape)
```

```
(1000000,)
```

All of these instructions can be put into a function that can be timed:

```
def compute_dr_min(a):
    pairs = combs_nd(a, 2, axis=1)
    i1 = tuple([None, None, 0, None])
    i2 = tuple([None, None, 1, None])
    return np.min(np.sum((pairs[i1]-pairs[i2])**2, axis=2)**0.5, axis=1)
```

```
%timeit compute_dr_min(r)
```

```
712 ms ± 196 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Note that performing all operations in the fewest possible number of lines can significantly speed up the process. Let's define another function where the difference between the pair elements is computed separately.

```
def compute_dr_min_more_steps(a):
    pairs = combs_nd(a, 2, axis=1)
    dp = pairs[:, :, 0, :]-pairs[:, :, 1, :]
    return np.min(np.sum(dp**2, axis=2)**0.5, axis=1)
```

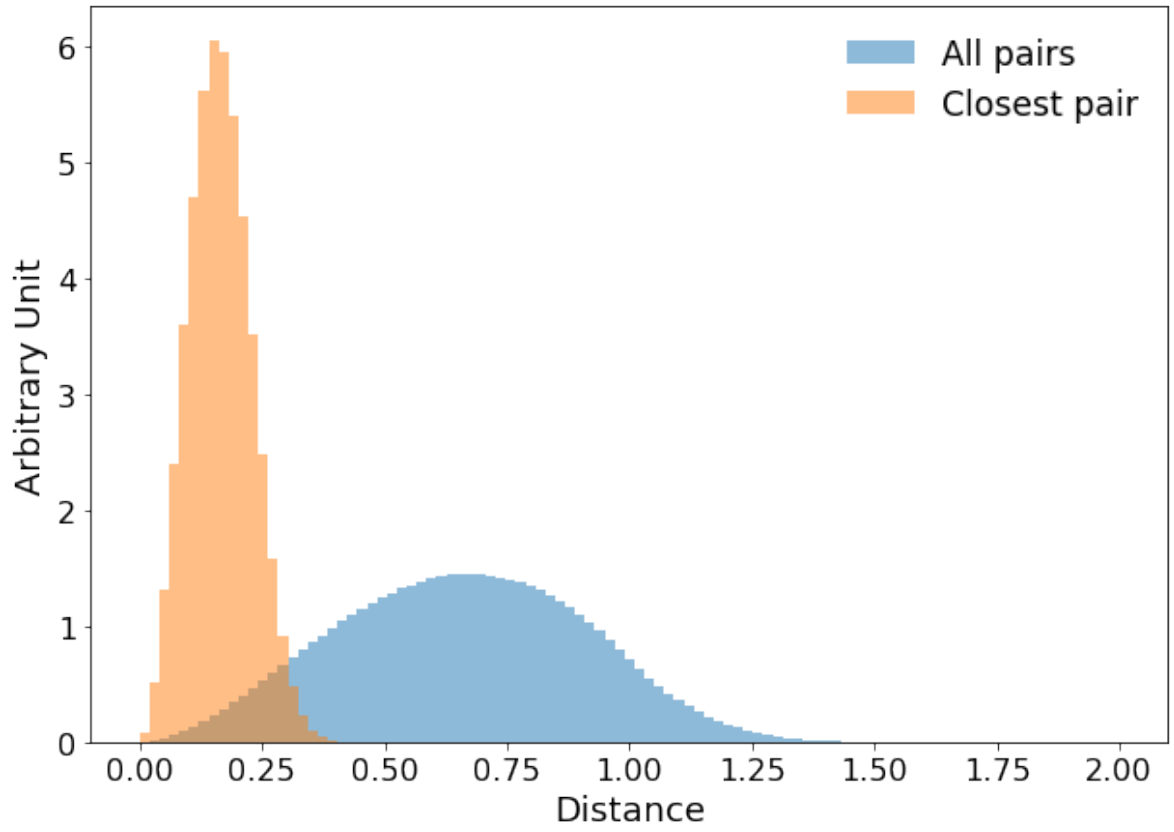
And let's compare the performance on 0.2 million observations:

```
%timeit compute_dr_min(a=r[:200000])
%timeit compute_dr_min_more_steps(a=r[:200000])
```

```
123 ms ± 3.55 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
561 ms ± 84.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Let's now plot the distributions of all distances for all the pairs (using the `flatten()` function, which returns a 1D array), and only the pair with the smallest distances.

```
plot_style = {
    'bins': np.linspace(0, 2, 100),
    'alpha': 0.5,
    'density': True,
}

plt.hist(distances.flatten(), label='All pairs' ,**plot_style)
plt.hist(smallest_distance, label='Closest pair', **plot_style)
plt.xlabel('Distance')
plt.ylabel('Arbitrary Unit')
plt.legend();
```



4.6 Selecting a subset of r_i based on (x, y, z) values, without a loop

The next step in our exploration of "loop-less calculations" is to be able to perform the same kind of computation described above, but only on a subset of positions selected according to a given criteria. For example, we might want to keep particles only if they have a positive charge. Many obvious applications can be found in other physics fields and/or machine learning. Let's start by accessing the three arrays of coordinates in order to select points based on some simple criteria.

```
x, y, z = r[:, :, 0], r[:, :, 1], r[:, :, 2]
```

4.6.1 Counting the number of points among the 10 where $x_i > y_i$ in each event

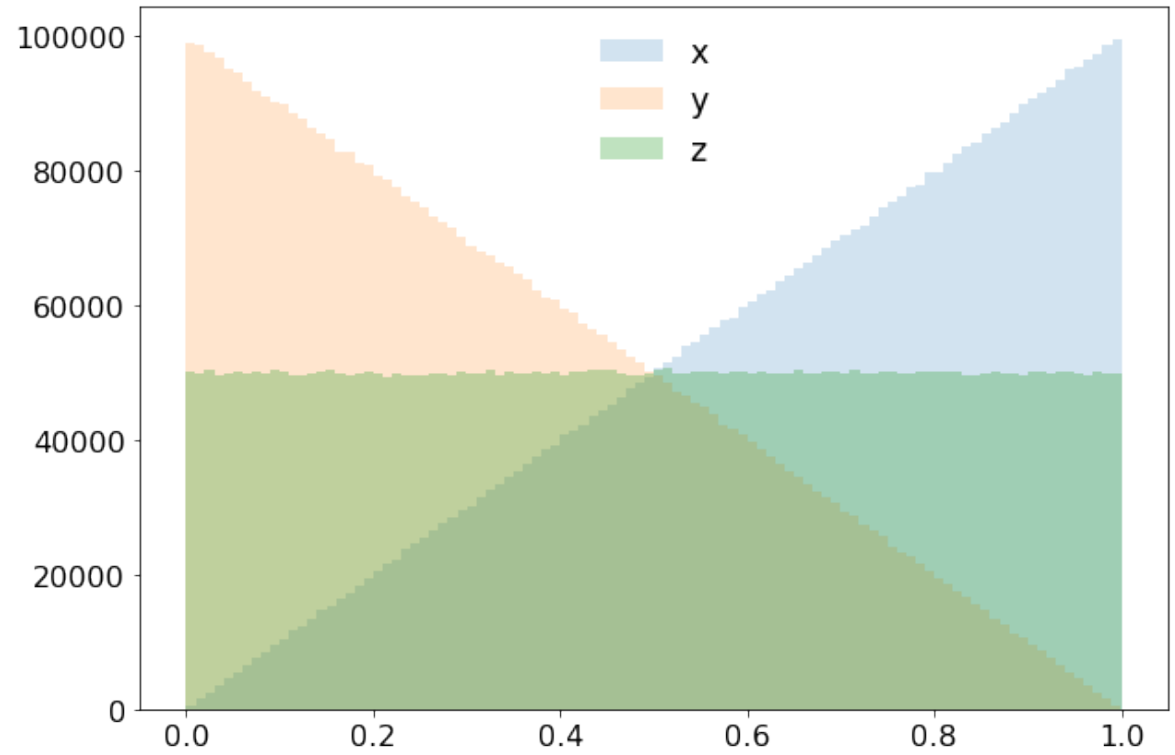
We will use the numpy masking feature described in the first chapter, defining an index of booleans based on the `x` and `y` arrays:

```
idx = x > y
print(idx.shape)
```

```
(1000000, 10)
```

We can quickly check the distribution for the selected coordinates: x and y are anti-correlated - as expected - while z is flat - as expected.

```
plt.hist(x[idx], bins=100, alpha=0.2, label='x')
plt.hist(y[idx], bins=100, alpha=0.2, label='y')
plt.hist(z[idx], bins=100, alpha=0.3, label='z')
plt.legend();
```



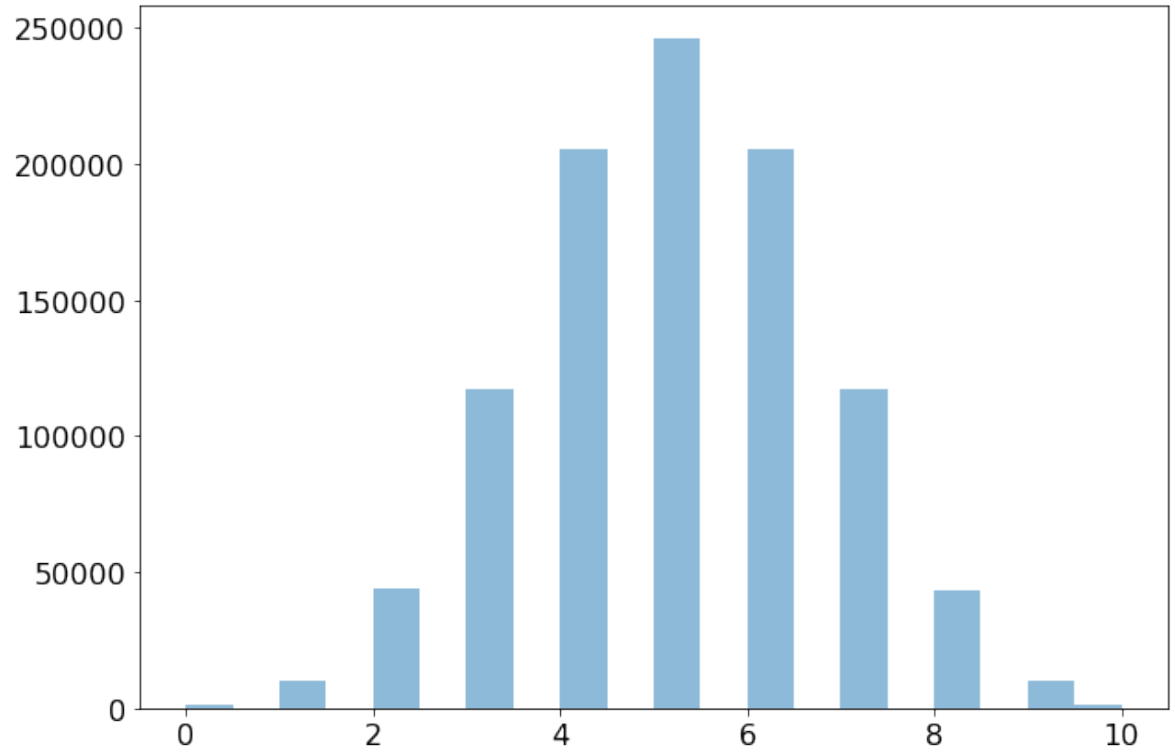
If we want to better understand how this selection affects our data, one might want to count the number of points per event that satisfy this selection. This can be done by using `np.count_nonzero()` on the boolean array along the axis representing the 10 vectors (`axis=1`):

```
c = np.count_nonzero(idx, axis=1)
print(c.shape)
```

```
(1000000,)
```

We can then plot the distribution of this number for all the events.

```
plt.hist(c, bins=20, alpha=0.5);
```



4.6.2 Plotting z for the two types of populations ($x > y$ and $x < y$)

This is obviously useful to inspect the different populations - something we want to do very often. For the plotting purpose, let's consider only the first 500 observations that we dump into `sx`, `sy`, `sz` (`s` for small):

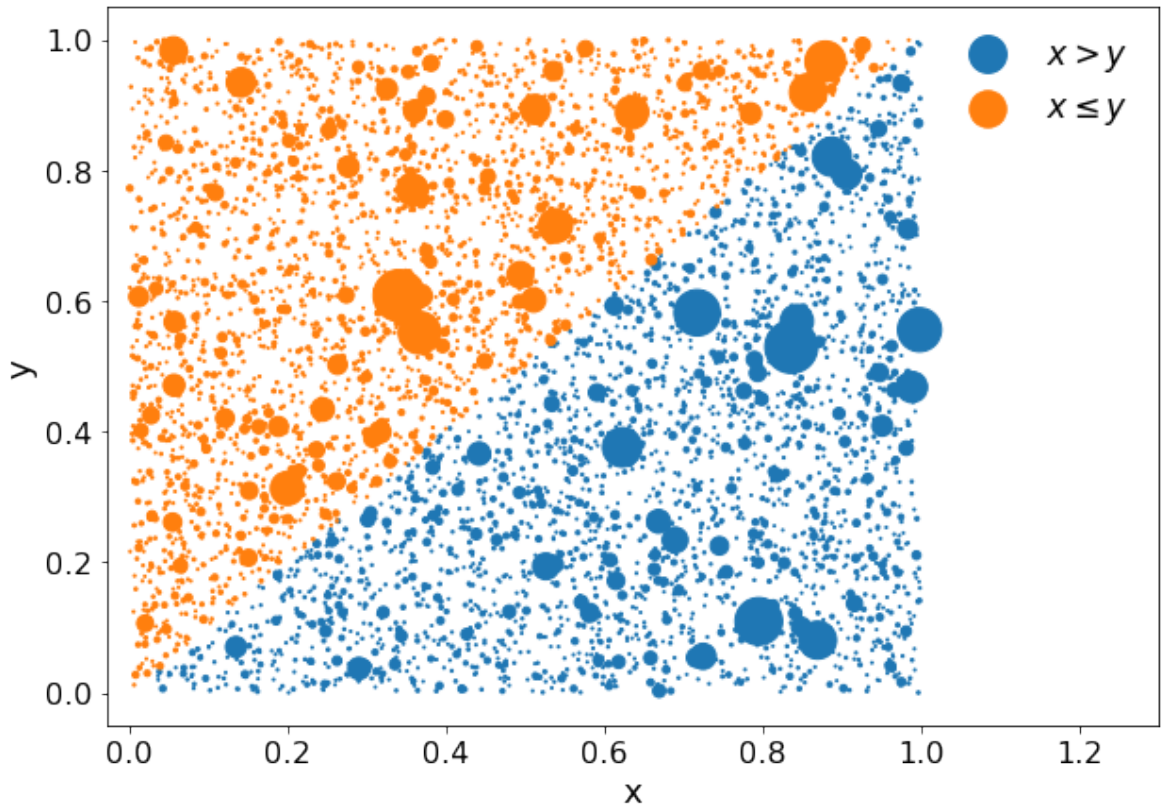
```
sx, sy, sz = x[0:500, ...], y[0:500, ...], z[0:500, ...]
```

We define the mask computed on these small arrays as `smask`.

```
smask = sx>sy
```

And we can plot the result in the 2D plane (x, y) with the z coordinate as the marker size, for instance $1/(z+10^{-3})$. The two populations are defined using both `smask` and `~smask` to ensure that the union of the two is the original dataset.

```
plt.scatter(sx[smask], sy[smask], s=(sz[smask]+1e-3)**-1, label='$x>y$')
plt.scatter(sx[~smask], sy[~smask], s=(sz[~smask]+1e-3)**-1, label='$x\leq y$')
plt.xlabel('x')
plt.ylabel('y')
plt.xlim(-0.03, 1.3)
plt.legend();
```



4.6.3 Computation of $x_i + y_i + z_i$ sum over a subset of the 10 positions

Once we are able to isolate a subset of points, we might want to compute new numbers based only on those. This is what is proposed here with the sum of the three coordinates. Let's first compute the sum, called `ht`, over all 10 points:

```
ht1 = np.sum(x+y+z, axis=1)
print(ht1.shape)
```

```
(1000000,)
```

Apply now a selection, which multiplies the value by 0 (i.e., `False`) if the condition is not satisfied.

```
selection = x>y
ht2 = np.sum((x+y+z)*selection, axis=1)
```

Of course, this only works for computations that are not affected by a zero. If we want to compute the product of coordinates, this approach will obviously not work.

```
prod = np.product((x+y+z)*selection, axis=1)
eff = np.count_nonzero(prod>0)/len(prod)
print('Efficiency of prod>0: {:.5f}'.format(eff))
```

```
Efficiency of prod>0: 0.00102
```

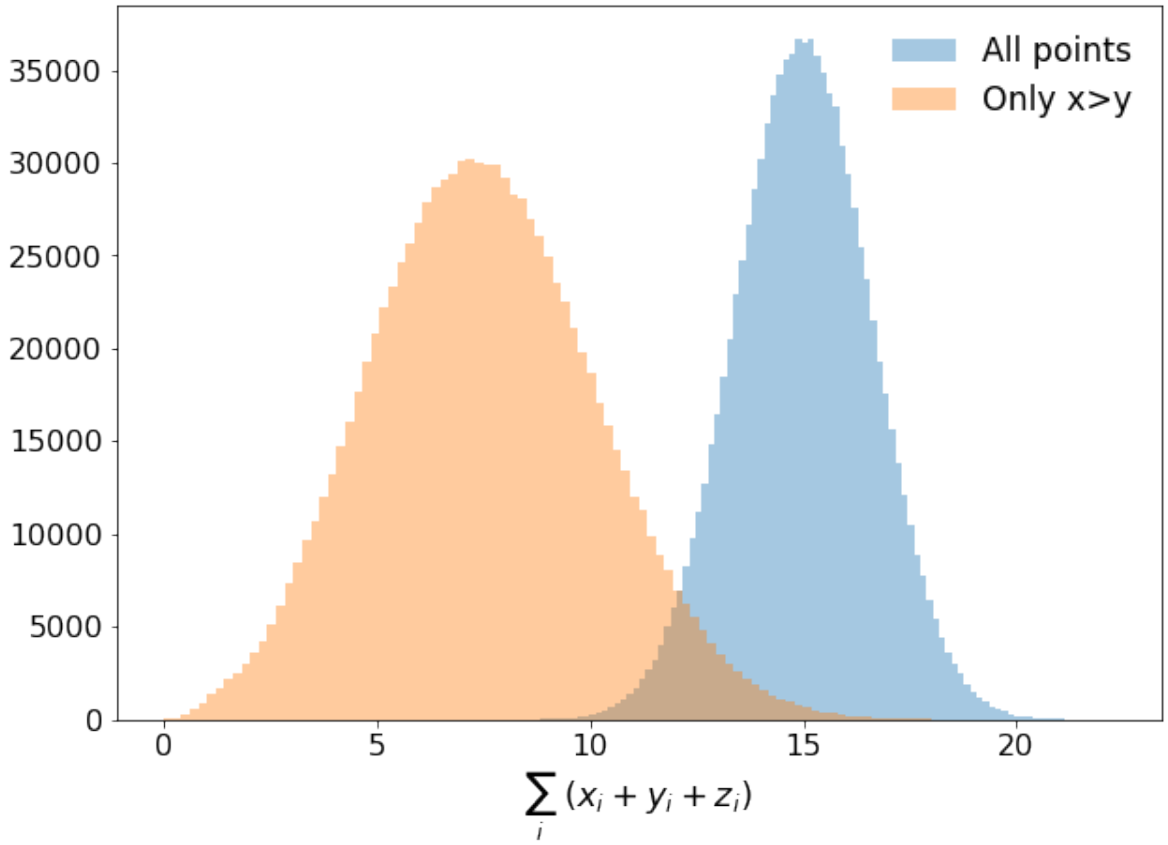
In a more general manner, we should use *masked arrays*, which completely remove the masked elements from any computations.

```
mx = np.ma.array(x, mask=selection)
my = np.ma.array(y, mask=selection)
mz = np.ma.array(z, mask=selection)
prod = np.product((mx+my+mz), axis=1)
eff = np.count_nonzero(prod>0)/len(prod)
print('Efficiency of prod>0: {:.5f}'.format(eff))
```

```
Efficiency of prod>0: 1.00000
```

Finally, one can plot the result, removing the observation with `ht2==0` (the case where all 10 points have $x \leq y$).

```
plt.hist(ht1, bins=100, alpha=0.4, label='All points')
plt.hist(ht2[ht2>0], bins=100, alpha=0.4, label='Only x>y')
plt.xlabel('$\sum_{i} \; (x_i+y_i+z_i)$')
plt.legend();
```



4.6.4 Pairing with a subset of r_i verifying $x_i > y_i$ only

Another computation would be to redo the pairing on the subset of selected positions. In order to do so, we follow the same logic, except that we will directly replace removed values with `np.nan` in order to easily identify them after the pairing. It's *very important to copy the original data using the `copy` module*, otherwise the original data will be modified in the following piece of code.

```
import copy
selection = x>y
selected_r = copy.copy(r)
selected_r[selection] = np.nan
print(selected_r[0])
```

```
[[0.01789452 0.66209367 0.06888675]
 [0.39922964 0.87771381 0.01192809]
 [0.88259998 0.88655328 0.30233561]
 [0.53178797 0.69359593 0.05459176]
 [0.69979407 0.82627363 0.63028438]
 [0.19121361 0.80913874 0.2813931 ]
 [0.22248198 0.78734303 0.39821198]
 [0.41256443 0.81143611 0.71874392]
 [0.10597153 0.61954029 0.15438807]
 [0.0586176  0.5567407  0.19918271]]
```

One can now call the pairing function on the filtered dataset.

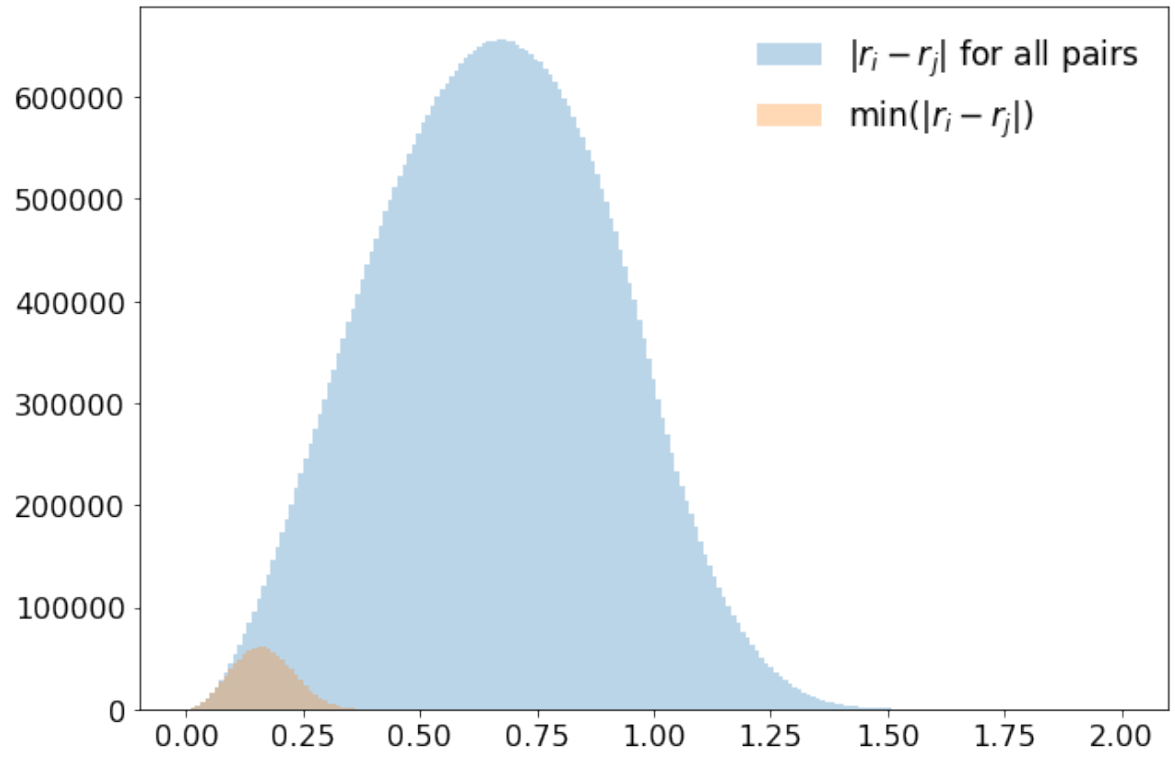
```
selected_pairs = combs_nd(selected_r, n=2, axis=1)
```

And compute the distances, but replace back the `np.nan` with a default value that will not be seen on a plot.

```
p1, p2 = pairs[:, :, 0, :], pairs[:, :, 1, :]
dp = np.sum((p1-p2)**2, axis=2)**0.5
dp[np.isnan(dp)] = 999
```

And plotting the distributions of both all distances and minimum distances for pairs made out of points verifying $x > y$:

```
plot_style = {'bins': np.linspace(0, 2, 200), 'alpha':0.3}
plt.hist(dp.flatten(), label='$|r_i-r_j|$ for all pairs', **plot_style)
plt.hist(np.min(dp, axis=1), label='min$(|r_i-r_j|)$', **plot_style)
plt.legend();
```



4.7 Some comments

Manipulating numpy arrays is quite powerful and fast for both computation and plotting, provided that we use numpy optimizations, namely vectorization, indexing, and broadcasting. However, there are some limitations to consider, as we saw above. In some cases, we need to use "patchwork approaches" to achieve our desired

results without using loops, as we did in the last two sections. What works for one computation may not work for another (for example, replacing rejected values with 0 works for addition but not for multiplication). Additionally, when pairing values, we had to replace all rejected values with `np.nan` in order to filter them later on. This kind of practice can make the code less readable as complexity increases, in my opinion. Perhaps there are smarter ways to accomplish these tasks.

Chapter 5

Introduction to image processing

> **Skills to take away**
>
> - *Basic*: Load/plot an image (data type), color/grayscale, add/subtract two images
> - *Medium*: Zone modification, filters (kernels/blocks/windows), apply them (given functions)
> - *Expert*: Filter application function (notion/use of NumPy strides, 2D convolutions)

5.1 Motivations

Image processing plays an important role in data science and science in general. The typical digit recognition problem is one (classic) example of image processing. The notion of *convolutional neural networks* (CNN) is also a key point in image processing based on machine learning algorithms. Another more recent example is the *generative adversarial neural networks* (GAN), which are able to generate images of a given nature, after being properly trained. You can check out https://thispersondoesnotexist.com, which shows generated images of people who don't actually exist.

The very first step is to understand how an image is encoded in NumPy and how to manipulate it, even without talking about sophisticated algorithms. This is the goal of this notebook, which is split into three different sections:

1. **Basic Investigations**: Load/plot/write an image, get image histogram, grayscale, cropping, ...
2. **Numerical Operations**: Addition, subtraction, masking some pixels based on a given condition, ...
3. **Applying Basic Filters**: Image split into blocks versus windows, blurring, sharpening, edge detection, ...

Note that there are a few Python packages dedicated to image processing, such as Pillow or scikit-image. The `scipy` package also has a module called `ndimage` dedicated to image processing (see this online lecture). I chose not to use these tools here in order to not increase the number of libraries (which is very easy to do in Python), so *only NumPy and matplotlib will be used in this chapter*. However, if you are interested in doing intensive image processing, I would recommend looking at Pillow. Another tool, more oriented toward machine learning and computer vision, is OpenCV - good to keep in mind depending on your applications.

5.2 Basic investigations

An image is a NumPy array of 8-bit integers with a shape `(Nx, Ny, 3)`, where `Nx` and `Ny` represent the number of pixels in the x and y directions, respectively. The array contains three color channels (red, green, and blue), with each channel represented by an integer in the range [0, 255]. In this range, lower pixel values correspond to darker areas, while values closer to 255 represent brighter pixels. To load an image in a common format (such as png or jpg) as a NumPy array, you can use the `plt.imread()` function:

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

# Load a test image
im = plt.imread('../data/image_test.jpg')
```

Indeed, we can investigate the NumPy array we loaded:

```
# Print caracteristics
print('Object    : {}'.format(type(im)))
print('Shape     : {}'.format(im.shape))
print('Data type : {}'.format(im.dtype))

# Print the values of four first pixels along x (for y=0):
print('pixels(x<5, y=0): \n{}'.format(im[0:4, 0, :]))
```

```
Object    : <class 'numpy.ndarray'>
Shape     : (3008, 4008, 3)
Data type : uint8
pixels(x<5, y=0):
[[18 18  8]
 [21 21  9]
 [31 31 19]
 [29 29 17]]
```

5.2.1 Plotting

The obvious first thing we want to do with an image is to see it! This can be achieved using the `plt.imshow()` function. Below, we write a function that uses `plt.imshow()`, with an arbitrary figure size while keeping the figure ratio:

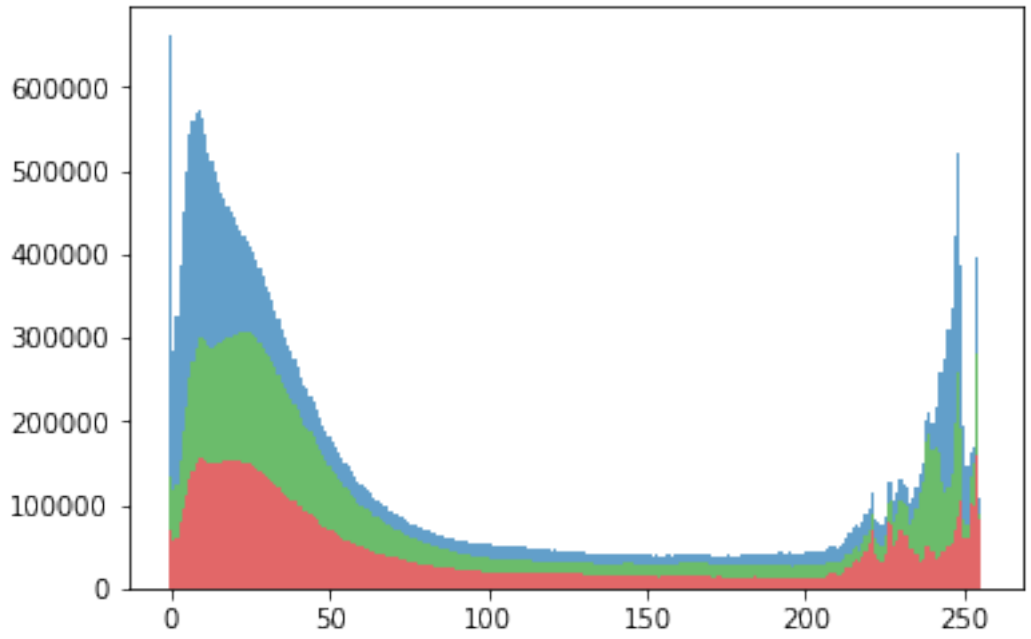
```
def plot_image(im, h=5, **kwargs):
    lx, ly = im.shape[:2]
    w = (ly/lx)*h
    plt.figure(figsize=(w,h))
    plt.imshow(im, interpolation=None, **kwargs)
    return
```

```
plot_image(im, h=7)
```

5.2.2 Histograms

The histogram of an image is often shown on a camera or in post-processing picture software. This allows us to appreciate how all pixels are distributed in terms of intensity. This can be obtained with a rather straightforward function:

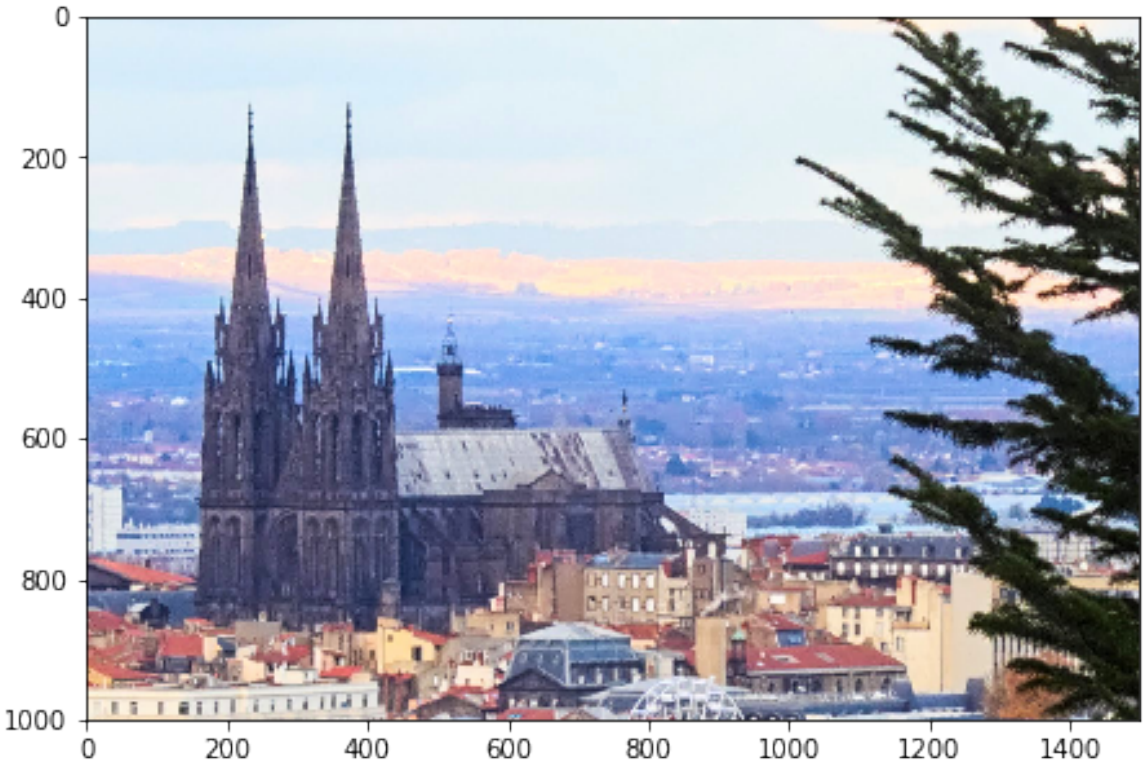
```
def plot_histogram(image):

    # Get the 2D array for each of the three colors, and flat it.
    pixels = [p.ravel() for p in np.array_split(image, 3, axis=2)]

    # Produce the histogram for each color and stack them
    style_hist = {'bins': np.arange(-0.5, 256.5, 1.0), 'alpha': 0.7, 'stacked':True}
    plt.hist(pixels, color=['tab:red', 'tab:green', 'tab:blue'], **style_hist)
    return

plot_histogram(im)
```



5.2.3 Color and grayscale

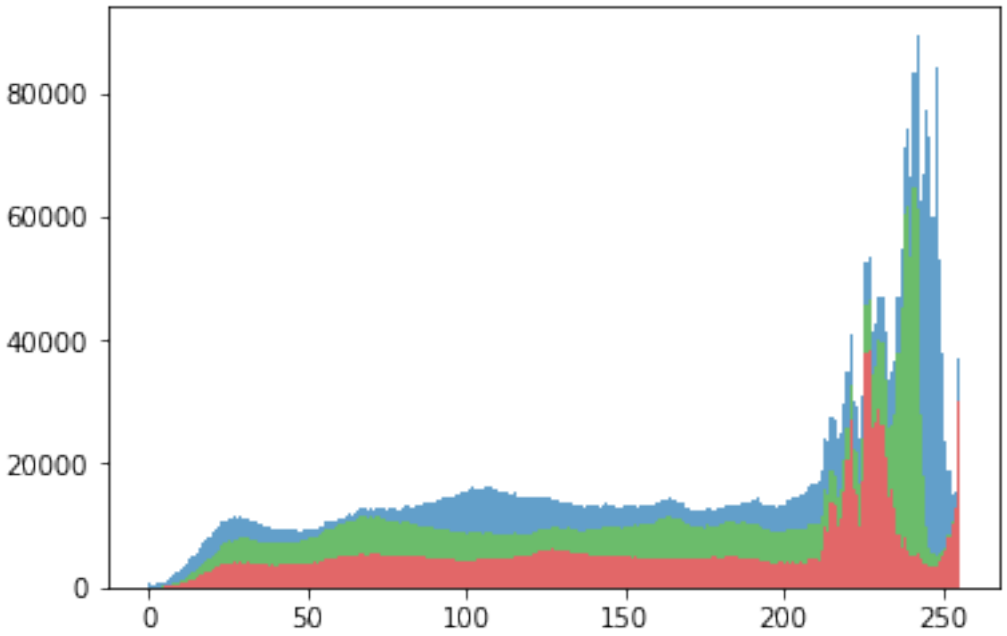
It is also possible to plot each channel (color) separately. First, we can crop the picture to focus on the interesting part of it: the Clermont-Ferrand cathedral. Since the image is a NumPy array, cropping is just taking a sub-array using NumPy indexing:

```
image = im[500:1500, 1500:3000]
plot_image(image)
```



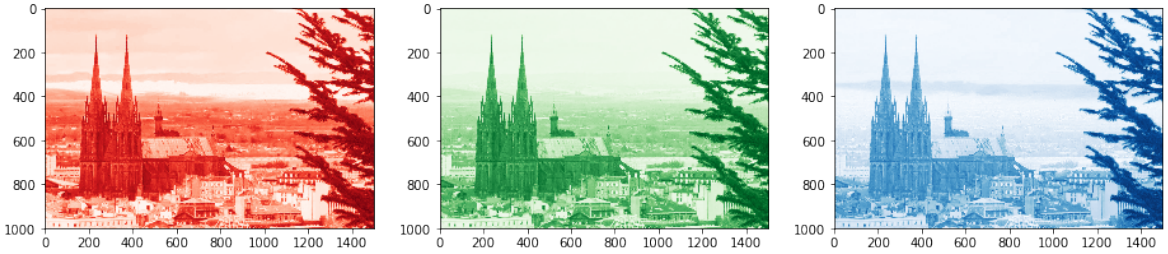
For instance, we can plot the histogram of this new image and check that the dark parts corresponding to surrounding trees (low values) are significantly decreased.

```
plot_histogram(image)
```



In order to investigate how the colors are spatially distributed, one can plot each channel separately using the appropriate color map. This is what the next function does:

```
def plot_RGB(image):
    '''Plot each color channel'''

    # Get each color channel
    R, G, B = image[...,0], image[...,1], image[...,2]

    # Figure shape preserving the image ratio
    lx, ly = image.shape[:2]
    w = (ly/lx)*3
    fig = plt.figure(figsize=(w*3.5,5))

    # One subplot per channel (Nrow, Ncol, Nplot)
    for i, (pixel, color) in enumerate(zip([R, G, B], ['Reds_r', 'Greens_r', 'Blues_r'])):
        plt.subplot(1, 3, i+1)
        plt.imshow(pixel, interpolation=None, cmap=color)
    return

plot_RGB(image)
```



Another usual operation is to switch from colored to grayscale picture. This can be done in several ways (check for *e.g.* the corresponding Wikipedia article), but one which is relatively simple to implement is based on luminance preservation:

```
def get_gray_scale(image):

    # Get RGB individual values
    R, G, B = image[...,0], image[...,1], image[...,2]

    # Get gray scale from RGB colors: PIX = 0.299 R + 0.587 G + 0.114 B
    pixels = np.array(0.299*R + 0.587*G + 0.114*B, dtype=np.uint8)

    # Replace each channel by this gray scale
    im_gs = np.stack([pixels, pixels, pixels], axis=2)

    # Retrun the gray image
    return im_gs
```

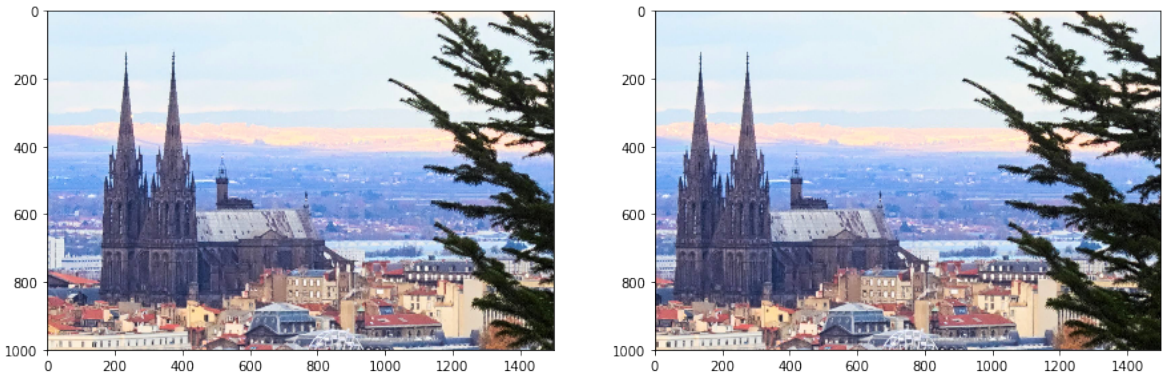
```
# Get the gray scale image
gray_image = get_gray_scale(image)
```

```
# Plot the result
plot_image(gray_image, h=7)
```

5.3 Numerical operations on images

Since images are NumPy arrays, we can perform numerical operations very easily. Not all of them have a proper meaning, though, but it is interesting to explore the possibilities. First, we define two images which are the Clermont-Ferrand cathedral slightly shifted:

```
# Create the two images
image1 = im[500:1500, 1500:3000]
image2 = im[500:1500, 1600:3100]

# Plot the two images side-by-side
plt.figure(figsize=(15, 5))
plt.subplot(1, 2, 1)
plt.imshow(image1)
plt.subplot(1, 2, 2)
plt.imshow(image2);
```



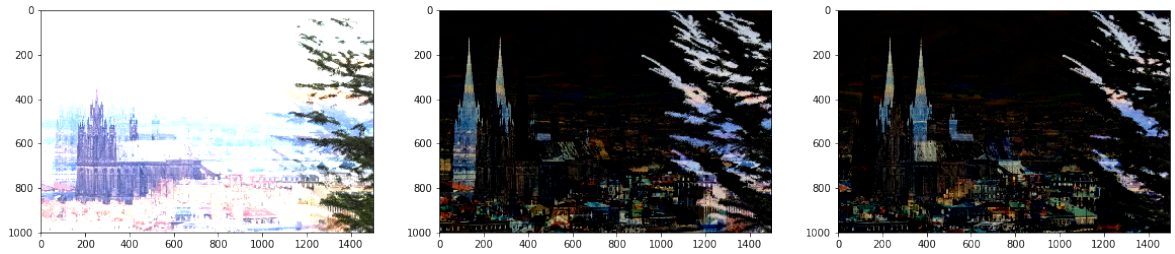
5.3.1 Addition & subtraction

What if we add or subtract these pictures? One has to define what happens if the sum (or the difference) is out of the permitted range $[0, 255]$. Let's take the following convention: if the pixel is below 0, we set it to 0, and if it is above 255, we set it to 255. This can be done by taking the image with float (allowing above values) and operating the truncation at the end. The two following functions implement this "image addition/subtraction":

```
def add_pictures(im1, im2):
    s = im1.astype(float) + im2.astype(float)
    s[s>255] = 255
    s[s<0] = 0
    return s.astype(np.uint8)

def subtract_pictures(im1, im2):
    s = im1.astype(float) - im2.astype(float)
    s[s>255] = 255
    s[s<0] = 0
    return s.astype(np.uint8)
```

Let's try to plot the added and subtracted images:

```
# Perform the operations
s12 = add_pictures(image1, image2)
d12 = subtract_pictures(image1, image2)
d21 = subtract_pictures(image2, image1)

# Plot the results side-by-side
plt.figure(figsize=(20, 5))
plt.subplot(1, 3, 1)
plt.imshow(s12)
plt.subplot(1, 3, 2)
plt.imshow(d12)
plt.subplot(1, 3, 3)
plt.imshow(d21);
```



When we sum the two pictures, we get something very bright (as expected) and we see the echo of the cathedral. After the subtraction (middle), we still see the echo but we get something very dark. The last plot shows the other difference, which looks quite cool, especially at the bottom of the cathedral!

5.3.2 Modifying certain pixels

Another useful operation we can easily do with NumPy is to mask pixels satisfying a given condition. Let's say we want to mask all pixels whose red level is higher than the sum of their blue level and green level:

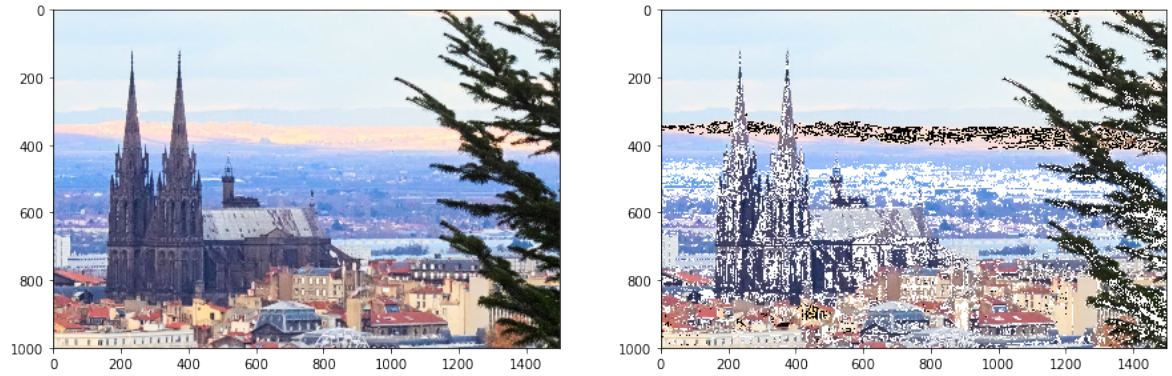
```
# Get the copy of colors (to be modifed latter)
r, g, b = image[...,0].copy(), image[..., 1].copy(), image[..., 2].copy()

# Get the mask
th = add_pictures(g, b)
to_black = r>=th

# Perform the mask
r[to_black] = 0
g[to_black] = 0
b[to_black] = 0
```

One can also decide to set the too dark regions to white, for example when `r+g+b<=60`, without modifying the previous pixels:

```
# Get the new indices to set to white
to_white = r+b+g<=60

# White only pixel that were didn't touch by the previous mask
to_white = to_white * ~to_black

# Perform the whitening
r[to_white] = 255
g[to_white] = 255
b[to_white] = 255
```

```
# Combine the three colors together
image_masked = np.stack([r, g, b], axis=2)

# Plot the results side-by-side
```

```
plt.figure(figsize=(15, 5))
plt.subplot(1, 2, 1)
plt.imshow(image1)
plt.subplot(1, 2, 2)
plt.imshow(image_masked);
```



5.3.3 Modifying regions

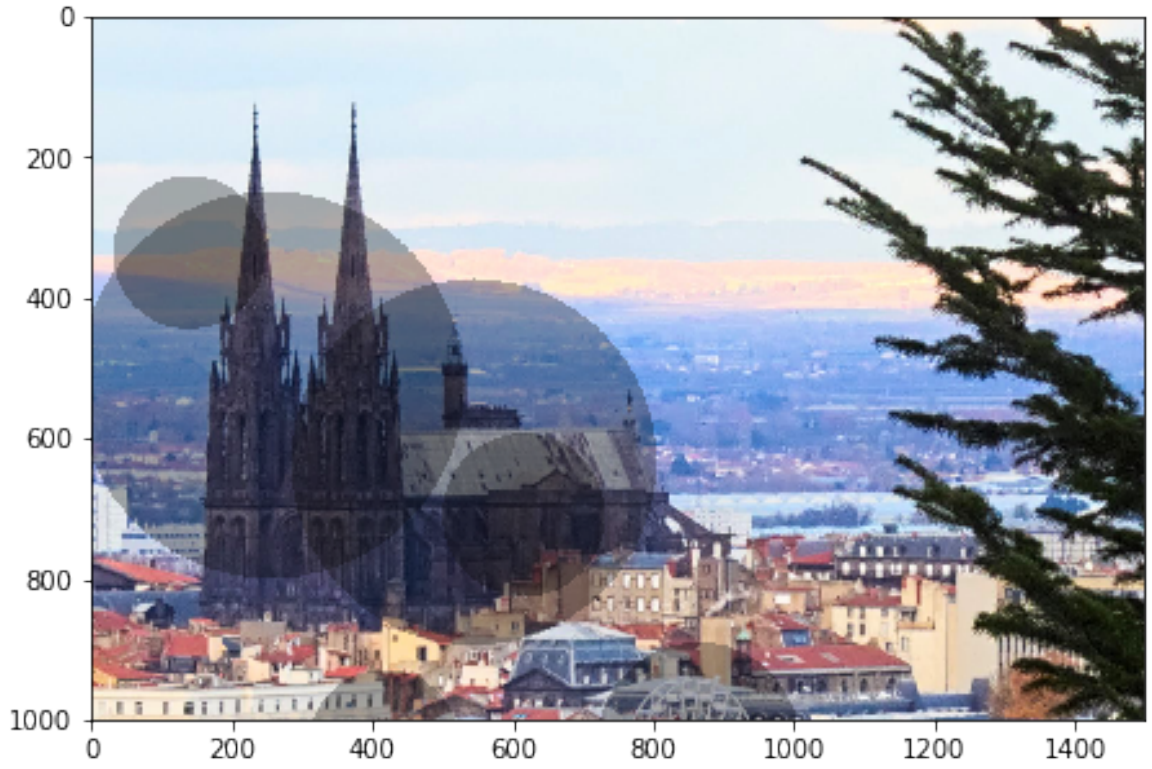
We can use fancy indexing of NumPy to define a shape and modify only the pixels that are within this shape. For this example, let's consider a circle with a given center position $r_0 = (x_0, y_0)$ and radius R. We will decrease the luminosity by scaling down all the pixels together. To start, let's write a function that returns a boolean 2D array which tells us whether a given (x, y) pixel is in the circle or not. This will, again, involve using a meshgrid:

```
def idx_in_circle(im, x0, y0, R):
    lx, ly = im.shape[:2]
    X, Y = np.meshgrid(np.arange(0, ly), np.arange(0, lx))
    radius = ((X-x0)**2 + (Y-y0)**2)**0.5
    return radius<=R
```

We will now use this function to add 10 random shadowed circles. We will generate the center and radius of the circles using the `np.random.randint()` function. Note the use of packing/unpacking of arguments to call the `idx_in_circle()` function in a more concise and clear way, together with the `zip()` syntax:

```
# Copy original image and modify it
result = image.copy()

# Get 10 random circles (in position and radius)
x0 = np.random.randint(low=100, high=1000, size=10)
y0 = np.random.randint(low=100, high=1500, size=10)
R0 = np.random.randint(low=50 , high=300 , size=10)

# For each of them, modify the picture
for circle in zip(x0, y0, R0):
    idx = idx_in_circle(image, *circle)
    result[idx] = result[idx]*0.7
```

```
# Plot the result
plt.figure(figsize=(10, 5))
plt.imshow(result);
```



5.4 Image filters with NumPy

5.4.1 Kernels, image blocks vs. windows

In image processing, a filter is a small 2D array $n \times n$ (also called a *kernel*) that is used to modify the value of each pixel using a *convolution* between a portion of the image and the kernel. These portions can either use every pixel only once by splitting the image into $(n \times n)$ blocks, or they can use every pixel several times by sliding $(n \times n)$ overlapping windows. The mathematical operation behind the name convolution is a simple sum over all elements from the window, weighted by the elements of the kernel.

To better understand the concept of kernels, blocks, and windows, let's now take an example of a `12x12` image and build both the blocks and sliding windows:

```
# Image definition
image = np.arange(12*12).reshape(12, 12)
print('image = \n{}'.format(image))

# Build-up 3x3 independant blocks
```

```
step3 = range(0, 12, 3)
blocks_3x3 = np.array([image[i:i+3, j:j+3] for i in step3 for j in step3])
blocks_3x3 = blocks_3x3.reshape(4, 4, 3, 3) # Organize the 16 blocks into a 4x4 grid
print('\nBlock[1, 1] = \n{}'.format(blocks_3x3[1, 1]))

# Built-up 3x3 windows for pixel far from the border (to avoid technical issues)
windows_3x3 = np.array( [image[i-1:i+2, j-1:j+2] for i in range(1, 11) for j in range(1, 11)]
↪  )
windows_3x3 = windows_3x3.reshape(10, 10, 3, 3) # Organize the 100 blocks into a 10x10 grid
print('\nWindow[3, 2] = \n{}'.format(windows_3x3[3, 2]))
```

```
image =
[[  0   1   2   3   4   5   6   7   8   9  10  11]
 [ 12  13  14  15  16  17  18  19  20  21  22  23]
 [ 24  25  26  27  28  29  30  31  32  33  34  35]
 [ 36  37  38  39  40  41  42  43  44  45  46  47]
 [ 48  49  50  51  52  53  54  55  56  57  58  59]
 [ 60  61  62  63  64  65  66  67  68  69  70  71]
 [ 72  73  74  75  76  77  78  79  80  81  82  83]
 [ 84  85  86  87  88  89  90  91  92  93  94  95]
 [ 96  97  98  99 100 101 102 103 104 105 106 107]
 [108 109 110 111 112 113 114 115 116 117 118 119]
 [120 121 122 123 124 125 126 127 128 129 130 131]
 [132 133 134 135 136 137 138 139 140 141 142 143]]

Block[1, 1] =
[[39 40 41]
 [51 52 53]
 [63 64 65]]

Window[3, 2] =
[[38 39 40]
 [50 51 52]
 [62 63 64]]
```

For instance, the number 39 can only be on an edge of a block (used once), while it can be anywhere for the sliding windows (used several times). Let's now define a 3x3 kernel and apply it to `blocks_3x3[1,1]`.

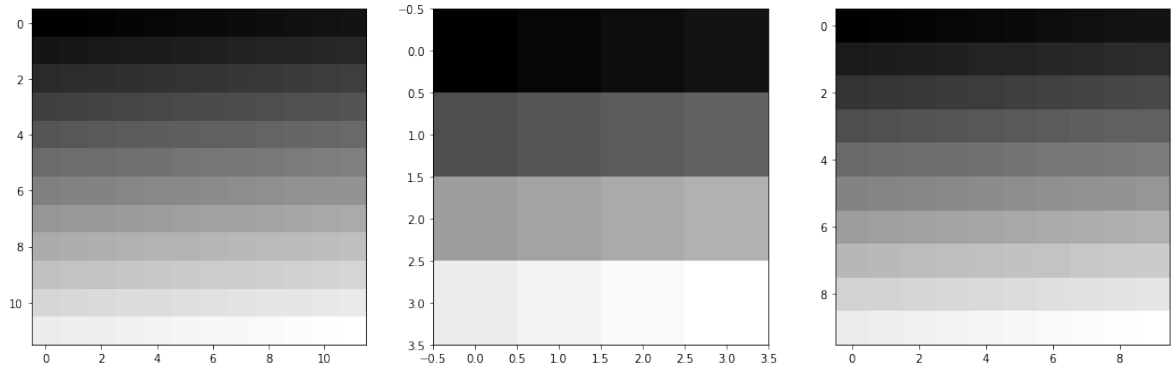
```
# Definition
kernel = np.arange(9).reshape(3, 3)/20
print('kernel = \n{}'.format(kernel))

# Convolution with the block[1, 1]
this_block = blocks_3x3[1, 1]
new_pixel = np.sum(kernel * this_block)
print('\nProduct of elements = \n{}'.format(kernel * this_block))
print('\nNew pixel = {:.1f} (vs an old pixel of {})'.format(new_pixel, this_block[1, 1]))
```

```
kernel =
```

```
[[0.   0.05 0.1 ]
 [0.15 0.2  0.25]
 [0.3  0.35 0.4 ]]

Product of elements =
[[ 0.    2.    4.1 ]
 [ 7.65 10.4  13.25]
 [18.9  22.4  26.  ]]

New pixel = 104.7 (vs an old pixel of 52)
```

Let's now apply the kernel defined above to both blocks and sliding windows. We can also represent the image before filtering, after block-based filtering, and window-based filtering.

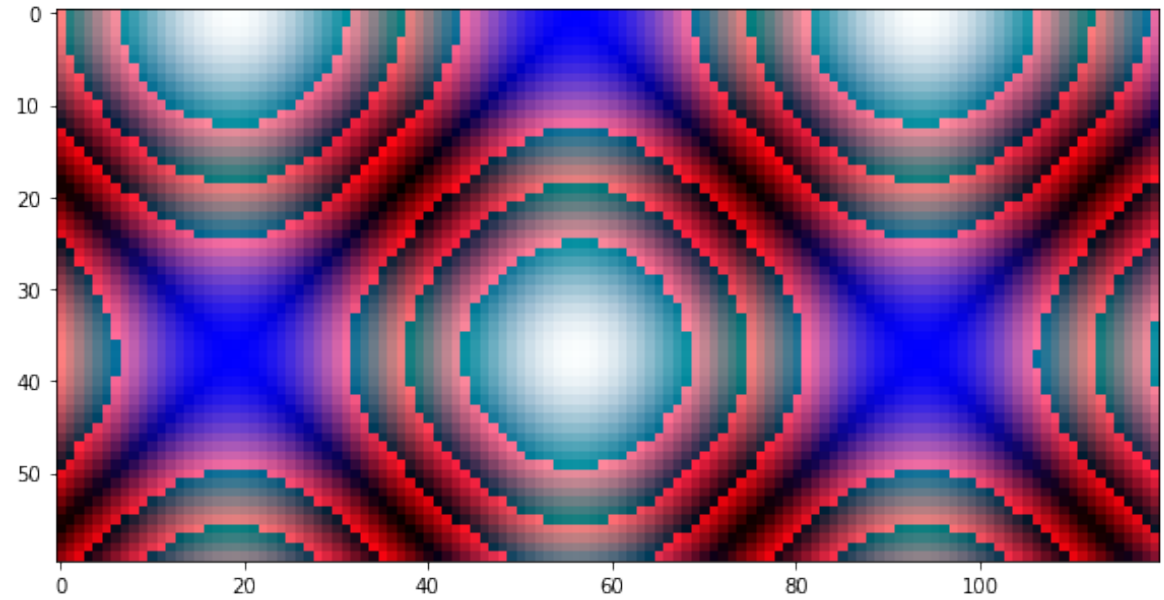
```
# Apply block-based filter
block_filtered = np.sum(blocks_3x3*kernel[np.newaxis, np.newaxis], axis=(2, 3))
block_filtered[block_filtered>255]=255
block_filtered[block_filtered<0]=0

# Apply window-based filter
window_filtered = np.sum(windows_3x3*kernel, axis=(2, 3))
window_filtered[window_filtered>255]=255
window_filtered[window_filtered<=0]=0

# Plot the results side-by-side
plt.figure(figsize=(18, 7))
plt.subplot(1, 3, 1)
plt.imshow(image, cmap='gray')
plt.subplot(1, 3, 2)
plt.imshow(block_filtered, cmap='gray')
plt.subplot(1, 3, 3)
plt.imshow(window_filtered, cmap='gray');
```



Important comment: The block view is not too heavy on memory, but the window view can explode quite rapidly. Indeed, for a kernel of $n \times n$, the window view is n^2 larger than the original array. If you manipulate millions of images, this can be problematic. For this reason, we will use a built-in `scipy.signal` function to perform the "sliding windows application" called `convolve2D()`. However, in the next section, we will study how to efficiently perform the "block approach" using a deeper NumPy feature: strides.

Generalization to RGB image. Before moving forward, we need to consider the three colors of an image to apply a filter, which has some implications in terms of broadcasting structure. First, let's define a dummy RGB image using a meshgrid (careful, x and y are reversed with respect to imshow) and three functions for each color:

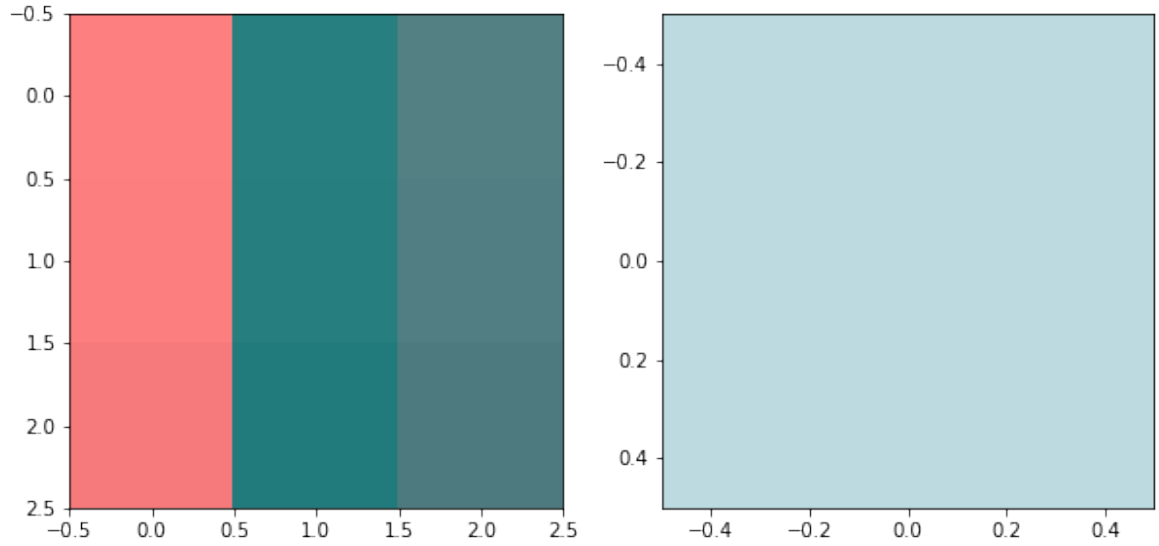
```
# Create a shaped image
def get_dummy_image(nx=600, ny=1200):
    X, Y = np.meshgrid(np.linspace(0, 10, ny), np.linspace(0, 5, nx))
    f = lambda n: np.abs(np.sin(X)**n+np.cos(Y)**n)
    im = np.stack([2*f(1), 0.5*f(3), 0.5*f(2)], axis=2)*255
    return im.astype(np.uint8)

im = get_dummy_image(nx=60, ny=120)
plot_image(im)
```



In order to achieve proper broadcasting, we need to extend the kernel with a third dimension for the colors, which can be done via the syntax `kernel[:,:,np.newaxis]`. This way, the kernel will be automatically duplicated for each color, and its application can be properly vectorized.

```
# Kernel application with the proper broadcasting over colors only for the first block
kernel = np.arange(9).reshape(3, 3)/20
new_pixel = np.sum(im[0:3, 0:3, :]*kernel[:,:,np.newaxis], axis=(0,1), dtype=np.uint8)
print(new_pixel)

# Plotting the 9 considered pixels and the result
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(im[0:3, 0:3, :])
plt.subplot(1, 2, 2)
plt.imshow(new_pixel.reshape(1, 1, 3));
```

```
[188 218 223]
```



5.4.2 Image blocks: Intuitive but inefficient approach

The most intuitive approach is to apply the filter to each block, involving an explicit *loop* over all the blocks of the image. Let's follow this approach for now, defining a function that applies the kernel to the block indexed by (i, j). Note that we didn't handle the boundaries properly, i.e. if the size of the image is not exactly n times the size of the kernel.

```
def apply_filter_one_block(im, kn, i, j):
    dx, dy = kn.shape
    start_i, end_i = i*dx, (i+1)*dx
    start_j, end_j = j*dy, (j+1)*dy
    indices = (slice(start_i, end_i), slice(start_j, end_j), slice(None, None))
    pixel = np.sum(im[indices].astype(float)*kn[:, :, np.newaxis], axis=(0, 1))
    pixel[pixel<0]=0
    pixel[pixel>255]=255
    return pixel.astype(np.uint8)

# Testing with a dummy image and averaging filter (1 every where)
im = get_dummy_image(nx=12, ny=12)
kernel = np.ones(shape=(3, 3))/9.
print('new pixel = {}'.format(apply_filter_one_block(im, kernel, 1, 1)))
```

```
new pixel = [98 57 79]
```

Let's now try to apply the strategy to a real image with dimensions 1200 $\times$ 1200 using two kernel sizes: 3 $\times$ 3 and 6 $\times$ 6:

```
# Testing with a real image
image_full = plt.imread('../data/image_test.jpg')
im = image_full[500:1700, 1500:2700]

# 3x3 kernel with all ones
kernel = np.ones(shape=(3, 3))/(3*3)
im3x3 = np.array([[apply_filter_one_block(im, kernel, i, j) for j in range(0, 400)] for i in
↪  range(0, 400)])

# 6x6 kernel with all ones
kernel = np.ones(shape=(6, 6))/(6*6)
im6x6 = np.array([[apply_filter_one_block(im, kernel, i, j) for j in range(0, 200)] for i in
↪  range(0, 200)])

# 12x12 kernel with all ones
kernel = np.ones(shape=(12, 12))/(12*12)
im12x12 = np.array([[apply_filter_one_block(im, kernel, i, j) for j in range(0, 100)] for i
↪  in range(0, 100)])

# Plotting the result
fig = plt.figure(figsize=(40, 10))
for i, this_im in enumerate([im, im3x3, im6x6, im12x12]):
    plt.subplot(1, 4, i+1)
    plt.imshow(this_im)
```

This is also possible to time the loop operation with the `%timeit` magic command. Let's do it in the full picture.

```
im_test = image_full[:3006, :4008]
kernel = np.ones(shape=(3, 3))/(3*3)
%timeit np.array([[apply_filter_one_block(im_test, kernel, i, j) for j in range(0, 1336)] for
↪  i in range(0, 1002)])
```

```
19.6 s ± 1.27 s per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

It takes approximately 20 seconds to process a 12 megapixel image. This shows that this approach is too slow for systematic treatment, especially when considering a larger number of images to process. This was expected since we have already mentioned that explicit loops in Python are slow. The goal of the next section is to use the power of NumPy to remove the loop and speed up this computation.

5.4.3 Image blocks: fast NumPy-based approach

The idea is to first convert the image array from the dimensions `(Nx, Ny, 3)` to `(Nx_new, Ny_new, Nx_kernel, Ny_kernel, 3)`, where `(Nx_new, Ny_new)` is the dimension of the "image of blocks". Once this is done, one can simply multiply and sum over axes 2 and 3.

The other approach is quite advanced but also quite powerful: `numpy.lib.stride_tricks.as_strided()`. Strides are basically a tuple of bytes of memory to jump from one element to another in each dimension. In other words, it's the byte separation between consecutive items for each dimension. This byte manipulation *doesn't duplicate the data but rather views them in a different way*, which is much more efficient from a memory point of view. This is the approach behind broadcasting and is, by far, *the fastest approach*. Let's go step by step to understand the strides concept.

```
# Small image as a copy of a sub-image from out favorite test
im_small = image_full[1000:1006, 1000:1006].copy()

# Print few information, including the number of bytes 6x6x3x1 = 108
print('Shape     : {}'.format(im_small.shape))
print('Dtype     : {}'.format(im_small.dtype))
print('Item size : {}'.format(im_small.itemsize))
print('Nbytes    : {}'.format(im_small.nbytes))
```

```
Shape     : (6, 6, 3)
Dtype     : uint8
Item size : 1
Nbytes    : 108
```

Reminder: 1 byte (or octet) is 8 bits. One bit is a single number being 0 or 1. The 8-bit image we are looking at in this lecture is then 1 byte per color and per pixel. The above number then makes perfect sense.

Let's try to compute the byte jump by hand first. In the third axis (color) axis, the jump between two consecutive items is just the item size so 1 byte. For the y-axis you jump 3 by 3 element to jump over all colors and reach the next position. This lead to a byte jump of 3 bytes. Finally, to jump over the next x-axis value, one needs to loop over all y-values and 3 colors, which makes $6 \times 3 = 18$. So the `im.strides` command should give `(18, 3, 1)`.

```
print(im_small.strides)
```

```
(18, 3, 1)
```

Now, we can manipulate these jumps to organize the numbers differently. Let's do it for only one color and a 4×4 array, to have a smaller array and follow the numbers individually.

```
# 4x4 array and getting strides
a = im_small[:4, :4, 0].copy()
jumps = a.strides
print('a = \n{}'.format(a))
print('strides = {}'.format(jumps))
```

```
a =
[[124 120 114 111]
 [123 119 115 112]
 [122 119 115 113]
 [120 118 116 115]]
strides = (4, 1)
```

Let's say that we want to create an array of 2x2 blocks, with each block containing 2x2 elements. This would result in a new shape of `(2, 2, 2, 2)`. The first block would look like:

```
[124, 120]
[123, 119]
```

In terms of strides, this means that the memory jump between two elements along the second-to-last axis is equivalent to a jump of 4 bytes. This corresponds to the jump between two elements along the first axis of the original array. In order to have the second block like:

```
[114, 111]
[115, 112]
```

We need to jump along the second axis by 2 bytes, which is the size of the jump to go from 124 to 114. Finally, for a jump along the first axis, we need to go from 124 to 122, which accounts for 8 bytes (corresponding to the first two lines of a).

```python
# Re-agencemement using the strides
new_shape = (2, 2, 2, 2)
new_jumps = (8, 2, 4, 1)

# Create the new array
from numpy.lib.stride_tricks import as_strided
a_new = as_strided(a, strides=new_jumps, shape=new_shape)

# Print the result
print('a_new = \n{}'.format(a_new))
print('strides = {}'.format(new_jumps))
```

```
a_new =
[[[[124 120]
   [123 119]]

  [[114 111]
   [115 112]]]


 [[[122 119]
   [120 118]]

  [[115 113]
   [116 115]]]]
strides = (8, 2, 4, 1)
```

Once we did this by hand, we can try to automate it given the size of the blocks we want to make and the size of the image. Let's take the example of a `(100, 100)` image with a `(3, 3)` block.

```
# 100x100 image
im100x100 = image_full[:100, :100, 0].copy()

# Convering each shape into numpy array for element-wise operation
im_shape = np.array(im100x100.shape) # image dimensions
bl_shape = np.array((3, 3))      # block dimensions

# Shape of the "image of blocks", as floor division
im_blocks_shape = im_shape // bl_shape
print(im_blocks_shape)

# Full dimension is a concatenation of corresponding shapes
new_shape = tuple(im_blocks_shape) + tuple(bl_shape)
print(new_shape)
```

```
[33 33]
(33, 33, 3, 3)
```

Now, it is a matter of automatically obtaining the new strides based on the image and block sizes. The jumps corresponding to the first two dimensions are simply the original picture jumps multiplied by the block shape. Specifically, we want the following:

- Along the *y*-axis (2nd axis): take the 1st element, then the `1+Ny_block`'s one, etc … so the memory jump is `old_jump_y * Ny_block`.
- Along the *x*-axis (1st axis): take the 1st element, then the `1+Nx_block`'s one, so the memory jump is also `old_jump_x * Nx_block`.

As for the last two axes, i.e., the navigation inside a block, this is just the original memory jumps.

```
# Get old strides as numpy array
old_strides = np.array(im100x100.strides)
print(old_strides)

# Form new strides
new_strides = tuple(old_strides*bl_shape) + tuple(old_strides)
print(new_strides)
```

```
[100   1]
(300, 3, 100, 1)
```

```
# Form the blocks and check this is correct for the first few
im100x100_blocked = as_strided(im100x100, strides=new_strides, shape=new_shape,
↪  writeable=False)
```

```
# Print the original image
print('Original image: \n{}'.format(im100x100[:6, :6]))

# Print the first few blocks
print('\nFirst 3x3 blocks: \n',format(im100x100_blocked[:2, :2]))
```

```
Original image:
[[18 18 22 26 23 21]
 [21 23 19 21 29 30]
 [31 31 17 13 25 24]
 [29 24 15 17 28 31]
 [25 12 12 19 20 23]
 [13  2 12 27 28 38]]

First 3x3 blocks:
 [[[[18 18 22]
    [21 23 19]
    [31 31 17]]

   [[26 23 21]
    [21 29 30]
    [13 25 24]]]


 [[[29 24 15]
    [25 12 12]
    [13  2 12]]

   [[17 28 31]
    [19 20 23]
    [27 28 38]]]]
```

We are now ready to build a function that applies a kernel per block (accounting for the additional axis for colors - not taken into account above), where the last step is just operating the sum:

```
def apply_filter_strides(image, kernel):
    from numpy.lib.stride_tricks import as_strided

    # Get the new shape
    m_shape, image_shape = np.array(kernel.shape), np.array(image.shape)
    new_shape = tuple(image_shape[:2] // m_shape) + tuple(m_shape) + (image_shape[-1],)

    # Get the new strides
    new_strides = tuple(image.strides[:2] * m_shape) + image.strides

    # Get the new blocked image (Nx_new, Ny_new, Nx_mask, Ny_mask, 3)
    blocked_image = as_strided(image, shape=new_shape, strides=new_strides, writeable=False)
```

```
    # Apply the mask with the proper broadcasting
    kernel_reshaped = kernel[np.newaxis, np.newaxis, :, :, np.newaxis]
    result = np.sum(blocked_image*kernel_reshaped, axis=(2, 3))

    # Cleaning
    result[result<0] = 0
    result[result>255] = 255

    # Return
    return result.astype(np.uint8)
```

It is worth mentioning that all this code is already written in some of the tools mentioned at the beginning of this chapter. However, the goal here is to learn how the tools are made (and possibly make new ones!). To finalize the process, we can first compare if we get the same result as the explicit loop function, and then compare the timing for those two functions:

```
# Check compatibility with the previous function
im = image_full[500:1700, 1500:2700]
kernel = np.ones(shape=(12, 12))/(12*12)
im12x12_loop = np.array([[apply_filter_one_block(im, kernel, i, j) for j in range(0, 100)]
↪  for i in range(0, 100)])
im12x12_strides = apply_filter_strides(im, kernel)

# Check that all pixels are the same in both images
print('Are the two results equal? {}'.format(np.all(im12x12_strides==im12x12_loop)))
```

```
Are the two results equal? True
```

```
# Exectution time
kernel = np.ones(shape=(3, 3))/(3*3)
%timeit apply_filter_strides(image_full, kernel)
```

```
866 ms ± 6.46 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

The fully vectorized approach based on stride manipulations is more than 20 times faster than the explicit loop.

5.5 Few typical filters

5.5.1 Few utility functions

In order to perform image processing in a systematic way, we need to build a few helper functions which are written below. They are all based on the above developments, except the sliding windows application using the `scipy.signal.convolve2d()` function. The full documentation of this function, together with some examples, can be found here and will not be discussed in this lecture.

Note: There is a slight difference in the definition of the `convolve2d()` function and the weighted sum used in the blocks approach. The `convolve2d()` function performs the weighted sum using the transposed kernel, which explains why the considered kernel in the below function is `kernel.T`.

Clean image (value and data type):

```
def clean_image(image):
    image[image < 0] = 0
    image[image > 255] = 255
    return image.astype(np.uint8)
```

Normalise the kernel values:

```
def normalize_filter(kernel):
    if np.sum(kernel) != 0:
        return kernel/np.sum(kernel)
    else:
        return kernel
```

Apply filter with sliding windows (based on `scipy.signal` 2D convolution function)

```
def apply_filter_windows(image, kernel):
    from scipy.signal import convolve2d

    # Performing 2D convolution on every color
    args = {'mode': 'same', 'boundary': 'fill', 'fillvalue': 0}
    r, g, b = image[..., 0], image[..., 1], image[..., 2]
    filtered_colors = [convolve2d(ch, kernel.T, **args) for ch in [r, g, b]]

    # Stacking all colors together
    filtered_image = np.stack(filtered_colors, axis=2)

    # Result
    return clean_image(filtered_image)
```

Applying filters with blocks (based on stride-based manipulation)

```
def apply_filter_blocks(image, kernel):
    from numpy.lib.stride_tricks import as_strided

    # Get the new shape
    m_shape, image_shape = np.array(kernel.shape), np.array(image.shape)
    new_shape = tuple(image_shape[:2] // m_shape) + tuple(m_shape) + (image_shape[-1],)

    # Get the new strides
    new_strides = tuple(image.strides[:2] * m_shape) + image.strides

    # Get the new blocked image (Nx_new, Ny_new, Nx_mask, Ny_mask, 3)
```

```
    blocked_image = as_strided(image, shape=new_shape, strides=new_strides, writeable=False)

    # Filtered image
    kernel_reshaped = kernel[np.newaxis, np.newaxis, :, :, np.newaxis]
    filtered_image = np.sum(blocked_image*kernel_reshaped, axis=(2, 3))

    # Result
    return clean_image(filtered_image)
```

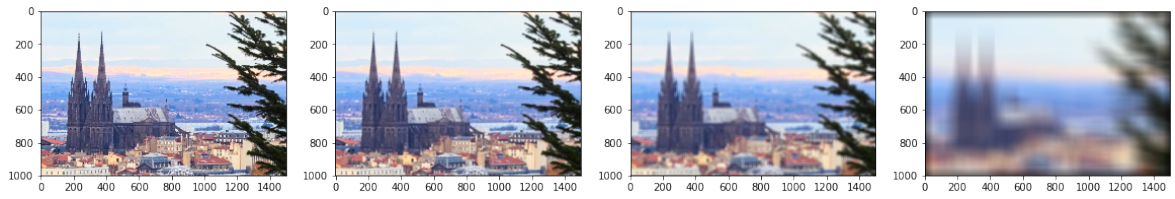
Test image definition:

```
image = image_full[500:1500, 1500:3000]
```

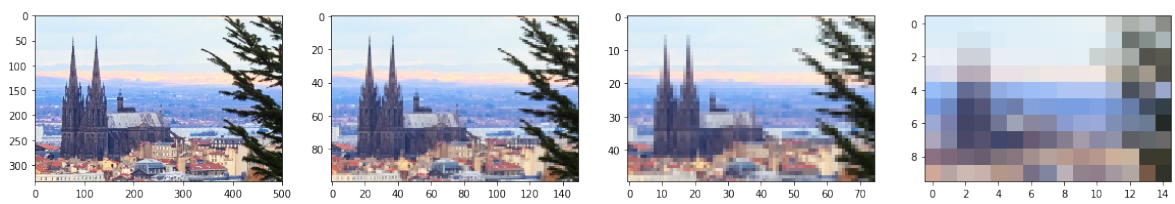
5.5.2 Blurry filter

The blurry filter just perform an average of all surrouding pixels. The corresponding kernel is a constant value in at position, with a sum of 1.0. Below, the blurry filter is applied for both sliding window and bloc approaches. *Caution:* the sliding window for a (100 $\times$ 100) kernel takes several minutes.

```
# Windows application
plt.figure(figsize=(20, 5))
for i, n in enumerate([3, 10, 20, 100]):
    kernel = normalize_filter(np.ones(shape=(n, n)))
    result = apply_filter_windows(image, kernel)
    plt.subplot(1, 4, i+1)
    plt.imshow(result)
```

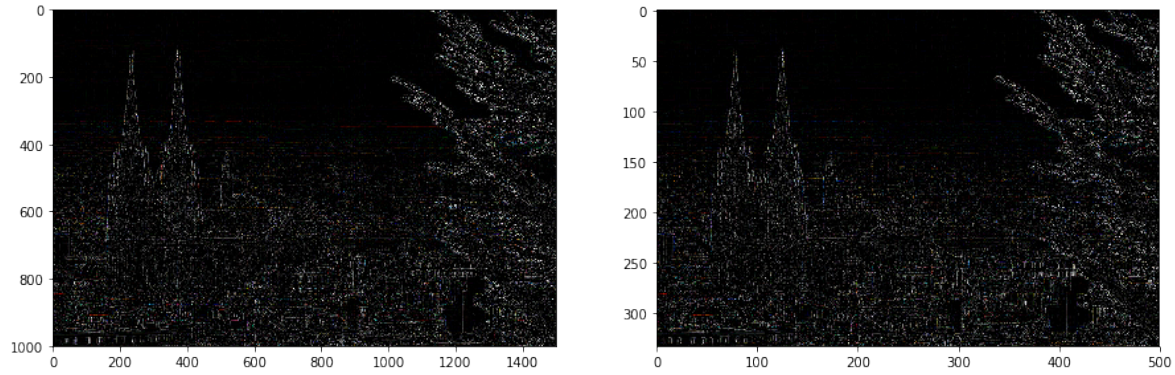


```
# Block application
plt.figure(figsize=(20, 5))
for i, n in enumerate([3, 10, 20, 100]):
    kernel = normalize_filter(np.ones(shape=(n, n)))
    result = apply_filter_blocks(image, kernel)
    plt.subplot(1, 4, i+1)
    plt.imshow(result)
```

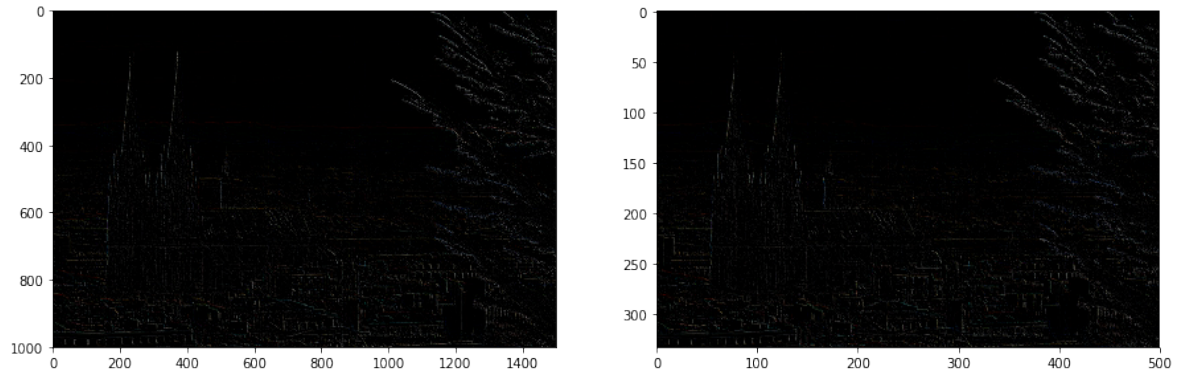


5.5.3 Edge detection

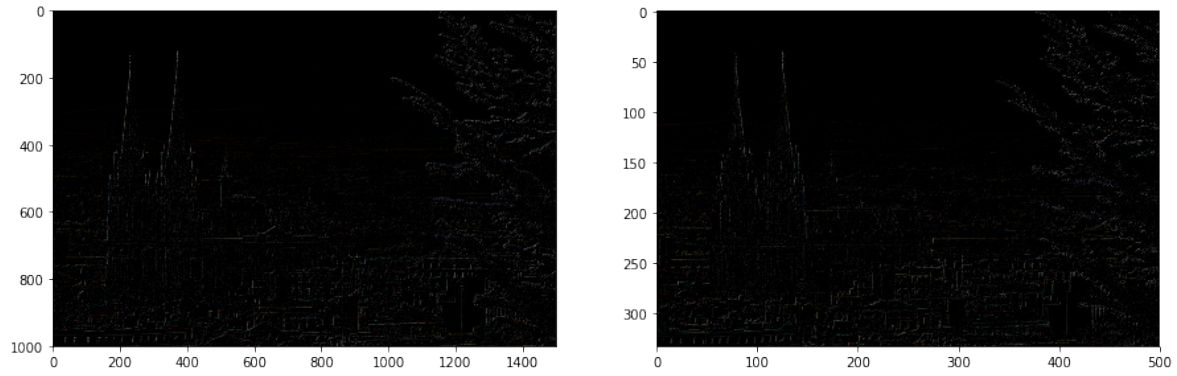
```
filter_border = np.array([
    [-1, -1, -1],
    [-1,  8, -1],
    [-1, -1, -1]
])

filter_border = normalize_filter(filter_border)
border_windows = apply_filter_windows(image, filter_border)
border_blocks = apply_filter_blocks(image, filter_border)

# Plotting the result
plt.figure(figsize=(15, 5))
plt.subplot(1, 2, 1)
plt.imshow(border_windows)
plt.subplot(1, 2, 2)
plt.imshow(border_blocks);
```




```
# Border which are direction-dependent: top-left
filter_topleft = np.array([
    [ 0,  0, -1],
    [ 0,  1,  0],
    [ 0,  0,  0]
])

filter_topright = normalize_filter(filter_topleft)
border_windows = apply_filter_windows(image, filter_topleft)
border_blocks = apply_filter_blocks(image, filter_topleft)

# Plotting the result
plt.figure(figsize=(15, 5))
plt.subplot(1, 2, 1)
plt.imshow(border_windows)
plt.subplot(1, 2, 2)
plt.imshow(border_blocks);
```

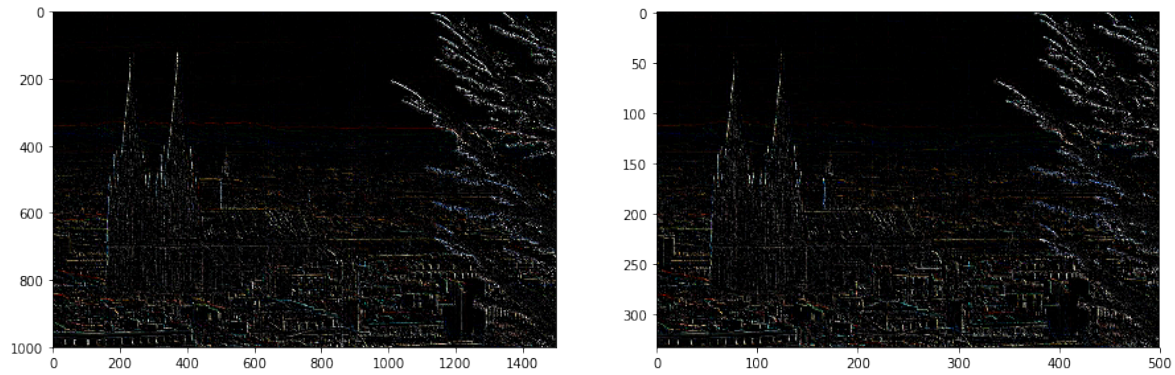



```
# Border which are direction-dependent: top-right
filter_topright = np.array([
    [ -1,  0,  0],
    [  0,  1,  0],
    [  0,  0,  0]
])

filter_topleft = normalize_filter(filter_topright)
border_windows = apply_filter_windows(image, filter_topright)
border_blocks = apply_filter_blocks(image, filter_topright) # Still a difference between
↪  block/window-approaches

# Plotting the result
plt.figure(figsize=(15, 5))
plt.subplot(1, 2, 1)
plt.imshow(border_windows)
plt.subplot(1, 2, 2)
plt.imshow(border_blocks);
```



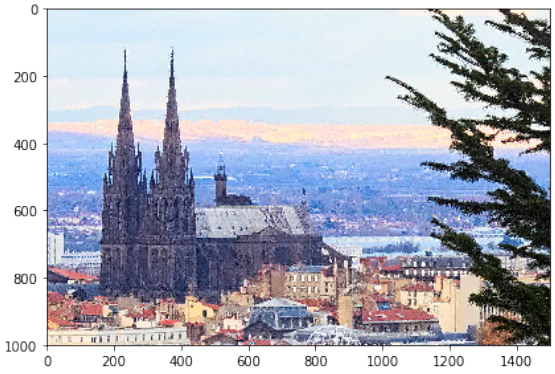

```
# Playing with intensity of the border
filter_topleft = np.array([
    [  0,  0, -3],
    [  0,  3,  0],
    [  0,  0,  0]
])
```

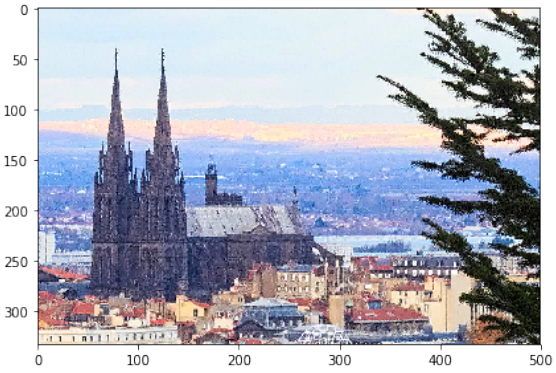
```
filter_topleft = normalize_filter(filter_topleft)
border_windows = apply_filter_windows(image, filter_topleft)
border_blocks = apply_filter_blocks(image, filter_topleft)

# Plotting the result
plt.figure(figsize=(15, 5))
plt.subplot(1, 2, 1)
plt.imshow(border_windows)
plt.subplot(1, 2, 2)
plt.imshow(border_blocks);
```



5.5.4 Sharpen filter

```
filter_sharpen = np.array([
    [  0, -1,  0],
    [ -1,  5, -1],
    [  0, -1,  0]
])

filter_sharpen = normalize_filter(filter_sharpen)
sharpen_windows = apply_filter_windows(image, filter_sharpen)
sharpen_blocks = apply_filter_blocks(image, filter_sharpen)

# Plotting the result
plt.figure(figsize=(15, 5))
plt.subplot(1, 2, 1)
plt.imshow(sharpen_windows)
plt.subplot(1, 2, 2)
plt.imshow(sharpen_blocks);
```


0
200
400
600
800
1000
0
200
400
600
800
1000
1200
1400


0
50
100
150
200
250
300
0
100
200
300
400
500