TECNICHE DI RAPPRESENTAZIONE E MODELLIZZAZIONE DEI DATI

— Part 1 —

(2 CFU out of 6 total CFU)

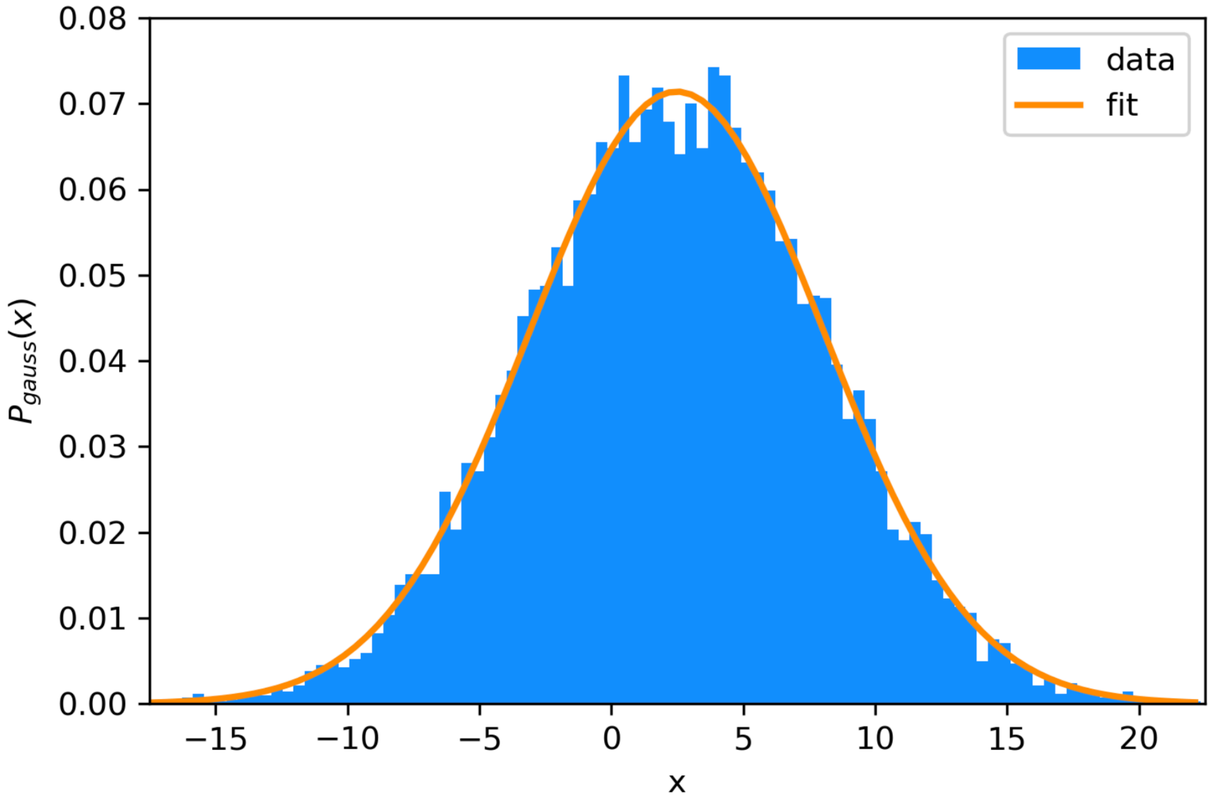
Link moodle: https://moodle2.units.it/course/view.php?id=11703

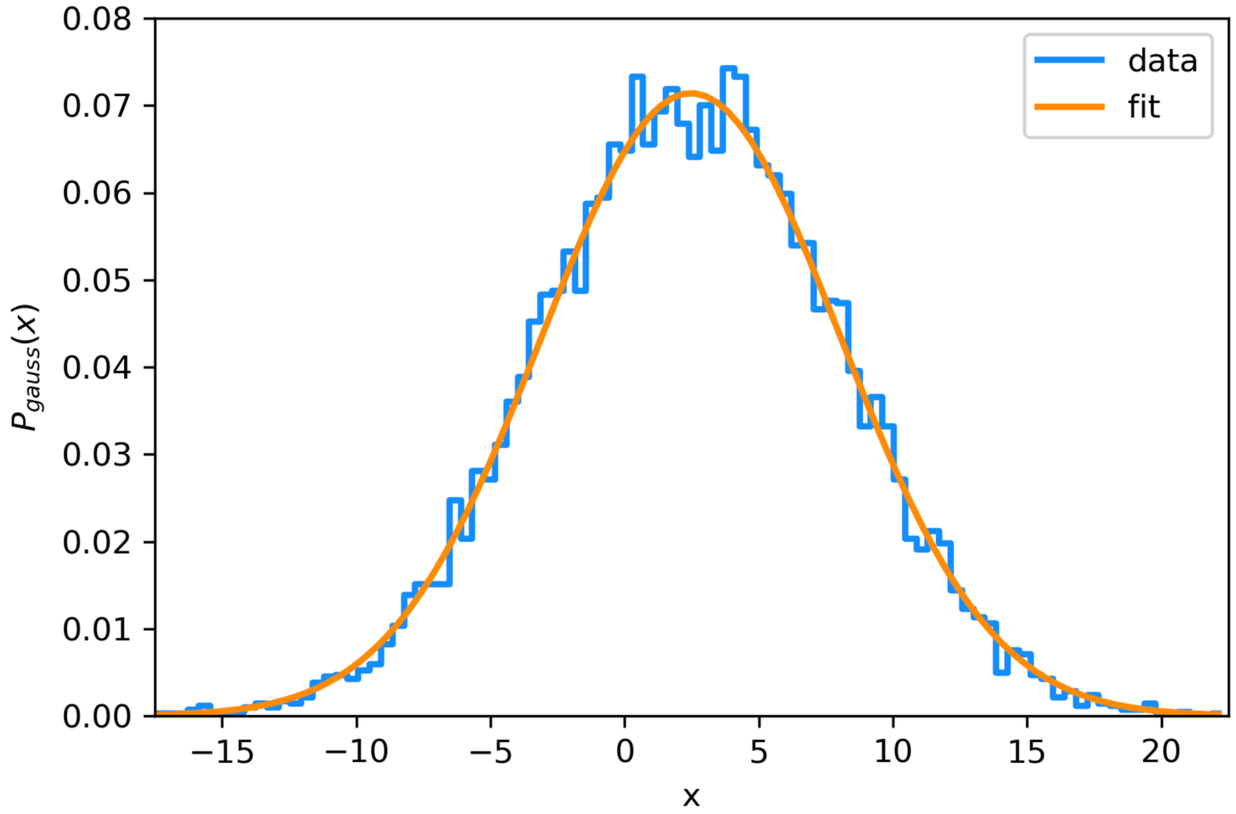
Teams code: 0ftoqj8

Python: plotting examples

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize as sopt

#We generate random data, drawn from a gaussian centered at 2.5, standard deviation of 5.5
original_mu = 2.5
original_sigma = 5.5
my_data = np.random.normal(original_mu, original_sigma, 10000)

#We create an histogram of these data
#We do not use weights, but we ask for the probability density function to be returned. The integral of this
curve will be 1.
#We use 100 bins, we could have defined bins ourselves
gauss_hist, bin_edges = np.histogram(my_data, bins = 100, density = True)

…

#Fit curve to data using scipy.optimize
#We need to define a function to use for the fit. In our case, we fit a gaussian
#I place the function here for example, but this should really go at the top of the script
#This function returns the probability density of a Gaussian curve, i.e. consistent with our 'observed'
values (our histogram)
def gauss_for_fit(xvals, mu, sigma):

    p_of_x = (1./np.sqrt(2*np.pi*sigma**2))*np.exp(-(((xvals-mu)**2)/(2*sigma**2)))

    return p_of_x

#We perform the fit: scipy.optimize is only one of the many methods to do this!
fit_params, fit_covariance_matrix = sopt.curve_fit(gauss_for_fit, bin_centers, gauss_hist)
bestfit_mu, bestfit_sigma = fit_params

…
```

Python: plotting examples





Python: input/output

Input from the keyboard

```
>>>
>>> a = input('Insert value: ')
Insert value: 3
>>> print(a)
3
>>>
```

Python: input/output

How to read from a file

```
#Inizio dal file contenente le proprieta' delle parent particles
filename = ‘my_file.dat'
# load data from the file my_file.dat, whose header is as follows:
# M(M_Sun) rho(gr/cm3) T(K) …
mass = []
density = []
temperature = []
…

with open(filename, 'r') as ppf:
    header = ppf.readline() # skip the first line of the file (header)
    for line in ppf:
        line = line.strip()
        columns = line.split()

        mass.append(float(columns[0]))
        density.append(float(columns[1]))
        temperature.append(float(columns[2]))
        …
#Transform lists into arrays
mass_array = np.array(mass)
density_array = np.array(density)
…
```

Python: input/output

How to read from a file

with numpy.loadtxt

```
206 import numpy as np
207
208 data_filename = 'path/filename'
209 data = np.loadtxt(data_filename, delimiter=' ', usecols=(0, 1, 8, 12), unpack=True)
210 1st_column = data[0]
211 2nd_column = data[1]
212 8th_column = data[2]
213 12th_column = data[3]
```

and/or other keywords, e.g. comments='#', skiprows=1

Python: input/output

How to write a file

```
# here I open my file
datafile = "/path/filename.txt"
datafile_id = open(datafile, 'w+')

array1 = …
array2 = …
array3 = x_s[ids]
array4 = …
array5 = …
…
array13 = …
…

data = np.array([array1, array2, array3, array4, array5, …, array13, …])
data = data.T
# here I transpose my data, so to have it in columns

np.savetxt(datafile_id, data, fmt=['%d','%e','%e','%e','%e', …,'%e', ...], header='ID
                    d_sun (pc)    x_star(pc)    y_star(pc)   z_star(pc)   …   [M/H]    …')

# here the ascii file is populated

datafile_id.close()
# close the file
```

Python: exercise

https://github.com/MilenaValentini/TRMD_2024/blob/main/file1_Group_Pos_Data_100Mpc-N256-DM.txt

1. Plottare prima colonna VS ciascuna delle altre tre, e poi seconda VS terza e quarta, terza VS seconda e quarta, quarta VS seconda e terza.
2. Considerare la struttura più massiccia. Calcolare le distanze di ciascuna delle altre strutture dalla più massiccia, e plottare nel piano y=massa VS x=distanza.
3. Graficare l'istogramma che mostra la distribuzione delle masse degli aloni.
4. Ripetere 1., organizzando i plot in modo che contengano due panel (ad esempio tramite https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.subplot.html), così da vedere le proiezioni delle posizioni (ad esempio top panel mostra coordinata y VS x, mentre bottom panel mostra z VS x; o anche left panel che mostra y VS x e right panel che mostra y VS z). Gestire gli assi che i due panel di volta in volta condividono.
5. Ripetere 3., prevedendo una scala logaritmica per le masse delle strutture, e provando poi ad usare bin logaritmici (non equispaziati linearmente).

Python: Exceptions

Exceptions are runtime errors occurring during the execution of a program.

Exceptions are objects that an instruction returns when its execution does not run successfully.
Either they are somehow handled by the developer, or they halt the script execution.

Python: Exception handling

Exceptions are runtime errors occurring during the execution of a program.

Exceptions are objects that an instruction returns when its execution does not run successfully.
Either they are somehow handled by the developer, or they halt the script execution.

However, exceptions in Python can also point to something else (e.g. see the case of *StopIteration*)

It’s useful to catch exceptions as soon as they arise and decide how to continue with following program instructions.

The blocks **try / except** allow you to handle exceptions.

Python: Exception handling

The blocks **try / except** allow you to handle exceptions.

try allows you to test a block of code for errors.

except lets you handle the error.

else lets you execute code when there is no error.

The **finally** block lets you execute code, regardless of the result of the try and except blocks.

Python: Exception handling

try allows you to test a block of code for errors

```
>>> 3 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

Python: Exception handling

try allows you to test a block of code for errors

except lets you handle the error

```
>>> 3 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

```
>>> try:
...     3 / 0
... except ZeroDivisionError:
...     print("I cannot perform this operation")
...
I cannot perform this operation
>>>
```

Python: Exception handling

try allows you to test a block of code for errors

except lets you handle the error

```
>>> 3 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

```
>>> try:
...     3 / 0
... except ZeroDivisionError as error_text:
...     print("I cannot perform this operation: ", error_text)
...
I cannot perform this operation:  division by zero
>>>
```

Python: Exception handling

A single **try - except** block can deal with multiple operations and different errors

```
>>> 4 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
>>> int("hello")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'hello'
>>>
>>> 4 + "hello"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>>
```

Python: Exception handling

A single **try - except** block can deal with multiple operations and different errors

```
>>> try:
...     4 / 0
...     int("hello")
...     4 + "hello"
... except ZeroDivisionError:
...     print("I cannot perform this operation")
... except ValueError:
...     print("I cannot cast a string into an int")
... except TypeError:
...     print("I cannot sum a string to an int")
...
I cannot perform this operation
>>>
```

Python: Exception handling

A single **try - except** block can deal with multiple operations and different errors

```
>>> try:
...     4 / 0
...     int("hello")
...     4 + "hello"
... except ZeroDivisionError:
...     print("I cannot perform this operation")
... except ValueError:
...     print("I cannot cast a string into an int")
... except TypeError:
...     print("I cannot sum a string to an int")
...
I cannot perform this operation
>>>
```

```
>>> try:
...     int("hello")
...     4 / 0
...     4 + "hello"
... except ZeroDivisionError:
...     print("I cannot perform this operation")
... except ValueError:
...     print("I cannot cast a string into an int")
... except TypeError:
...     print("I cannot sum a string to an int")
...
I cannot cast a string into an int
>>>
```

Python: Exception handling

Exceptions can be grouped in different (sub-)types.

The **except** block catches all the exceptions of a given type.

Main type is **Exception**. All the others stem from it.

```
>>> try:
...     6 / 0
... except Exception as error_text:
...     print("I cannot perform this operation: ", error_text)
...
I cannot perform this operation:  division by zero
>>>
```

ZeroDivisionError is a sub-type of **Exception**.

```
>>> try:
...     2 + "bye"
... except Exception as error_text:
...     print("I cannot perform division: ", error_text)
...
I cannot perform division:  unsupported operand type(s) for +: 'int' and 'str'
>>>
```

Python: Exception handling

Exceptions can be grouped in different (sub-)types.

The **except** block catches all the exceptions. Main type is **Exception**. All the other types stem from it.

Exceptions should be rather captured from the most to the least specific.

```
>>> try:
...     6 / 0
... except ZeroDivisionError as error_text:
...     print("I cannot perform this operation: ", error_text)
... except Exception as error_text:
...     print("Issues due to: ", error_text)
...
I cannot perform this operation:  division by zero
>>>
>>> try:
...     2 + "bye"
... except ZeroDivisionError as error_text:
...     print("I cannot perform this operation: ", error_text)
... except Exception as error_text:
...     print("Issues due to: ", error_text)
...
Issues due to:  unsupported operand type(s) for +: 'int' and 'str'
>>>
```

Python: Exception handling

Exceptions can be grouped in different (sub-)types.

The **except** block catches all the exceptions. Main type is **Exception**. All the other types stem from it.

```
>>> try:
...       int("hello")
...       4 / 0
...       4 + "hello"
... except (ZeroDivisionError, ValueError, TypeError) as error_text:
...     print("Issue: ", error_text)
...
Issue:  invalid literal for int() with base 10: 'hello'
>>>
>>> try:
...       4 + "hello"
... except (ZeroDivisionError, ValueError, TypeError) as error_text:
...     print("Issue: ", error_text)
...
Issue:  unsupported operand type(s) for +: 'int' and 'str'
>>>
>>> try:
...       4 / 0
... except (ZeroDivisionError, ValueError, TypeError) as error_text:
...     print("Issue: ", error_text)
...
Issue:  division by zero
```

A single **except** instruction can handle different types of exceptions: they have to be listed within a tuple.

Python: Exception handling

Besides **except**, the **try** block can be followed by **finally**.

Finally contains all the instructions which will be executed independently of the presence of exceptions.

```
>>> a = 12
>>> b = 3
>>> try:
...     a / b
... except ZeroDivisionError as error_text:
...     print("Problem with division by 0: ", error_text)
... finally:
...     print("Done!")
...
4.0
Done!
```

Python: Exception handling

Besides **except**, the **try** block can be followed by **finally**.

Finally contains all the instructions which will be executed independently of the presence of exceptions.

Finally: extremely useful to perform instructions even if errors/exceptions occur.

```
>>> a = 12
>>> b = 0
>>> try:
...     a / b
... except ZeroDivisionError as error_text:
...     print("Problem with division by 0: ", error_text)
... finally:
...     print("Done!")
...
Problem with division by 0:  division by zero
Done!
>>>
```

Python: Exception handling

Last block that **try** can execute before **finally** is **else**.

```
>>> for number in [6, 4, 0, 3]:
...     try:
...             print("Try division by {}".format(number))
...             12 / number
...             print("OK!")
...     except ZeroDivisionError as error_text:
...             print("Issue with {}: ".format(number), error_text)
...     finally:
...             print("Continue...")
...
Try division by 6
2.0
OK!
Continue...
Try division by 4
3.0
OK!
Continue...
Try division by 0
Issue with 0:  division by zero
Continue...
Try division by 3
4.0
OK!
Continue...
>>>
```

Python: Exception handling

Last block that **try** can execute before **finally** is **else**.

Else contains instructions which are computed only if exceptions did not occur.

```
>>> for number in [6, 0, 3]:
...     try:
...             print("Try division by {}".format(number))
...             12 / number
...             print("OK!")
...     except ZeroDivisionError as error_text:
...             print("Issue with {}: ".format(number), error_text)
...     else:
...             print("No issue")
...     finally:
...             print("Continue...")
...
Try division by 6
2.0
OK!
No issue
Continue...
Try division by 0
Issue with 0:  division by zero
Continue...
Try division by 3
4.0
OK!
No issue
Continue...
>>>
```

Python: plotting exercise

Exercise 1.

a) — Create two arrays (for instance with numpy.arange).
The first one has 60 even numbers.
The second one has 60 odd numbers.
Plot them (e.g. scatter; x-array is array_even, y-array is array_odd) and use a label for them.

b) — On the same plot, overplot (with a different colour) a selection of the aforementioned arrays, where you only pick every other array entry (i.e., one entry yes, the following one no, and so on).

c) — In addition, use a function to compute the sum of the x-array in a) and its log_10, and square the result (i.e., **2), and do the same for y-array.
Plot the new vectors, always choosing different colours and label.
Use log scale if needed.

d) — Write all the plotted vectors in a file.

Python: exercise

https://github.com/MilenaValentini/TRMD_2024/blob/main/file2_Groups_AGN-wWU_500Mpc_Data.txt