

**Basic Curve Fitting of Experimental Data with Python**

**Nabil Nasreldeen Mohamed**

Senior Student,

Computer science Department

Supervised By

**Dr / Khaled Mohamed Hussein**

**Physics** Department, Faculty of Science, South Valley University

2022 / 2023

**Contents**

Section 1 (Fitting concepts):- page 2

1. Linear fitting:- page 2

* Interpolation and extrapoation page 6

2.Curve fitting:- page 12

* Gaussian page 14
* Lorentzian page 24
* Voigt page 28

Section 2 (Python Desktop application):- page 32

1. Initial interface page 33

2.the dataset file structure page 37

3.The interface with linear fitting page 39

4.The interface with curve fitting page 41

5.The application full code:- page 43

1. main.py file page 43
2. interpolation\_functions.py file page 58
3. fitting\_functions file page 59

**Section 1 (Fitting concepts):-**

**1-Linear fitting(Linear regression)**

**Introduction:-**

Linear regression is a type of statistical analysis used to predict the relationship between two variables. It assumes a linear relationship between the independent variable and the dependent variable, and aims to find the best-fitting line that describes the relationship. The line is determined by minimizing the sum of the squared differences between the predicted values and the actual values.

The general form for the linear function:

**f(x)=mx+c**

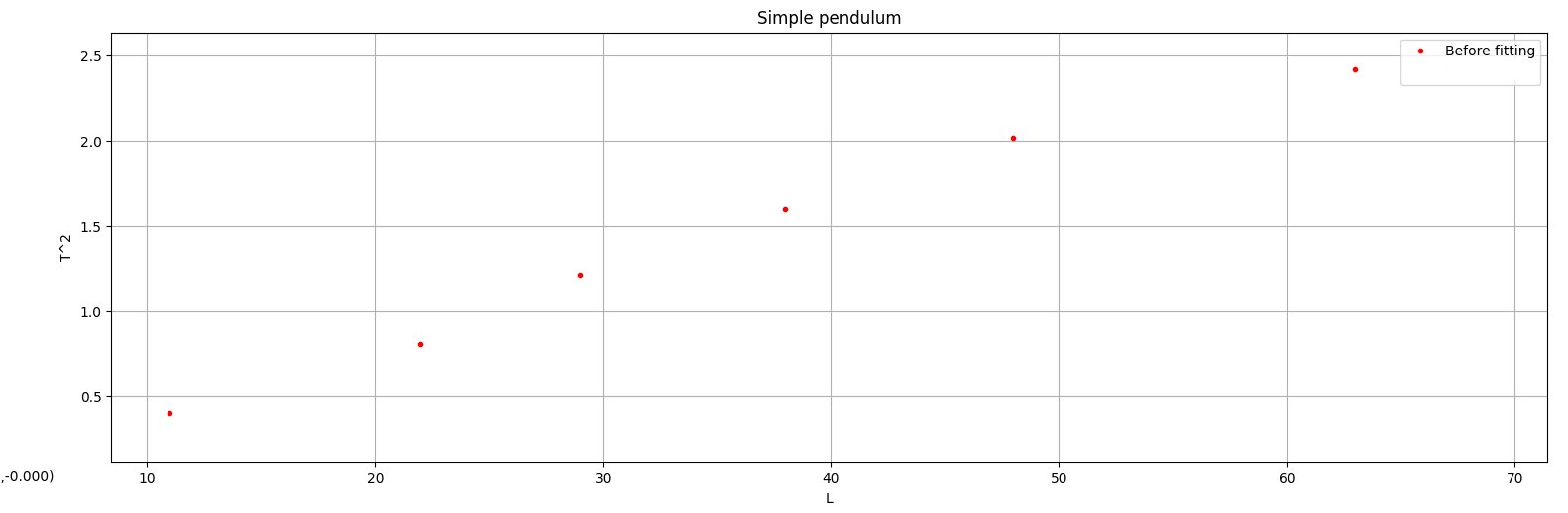
Where m corresponds to slope and c corresponds to intercept.

Consider the following points for simple pendulum experiment:

X=[11,22,29,38,48,63]

Y=[0.403,0.81,1.21,1.6,2.015,2.42]

Can be plotted as the following:

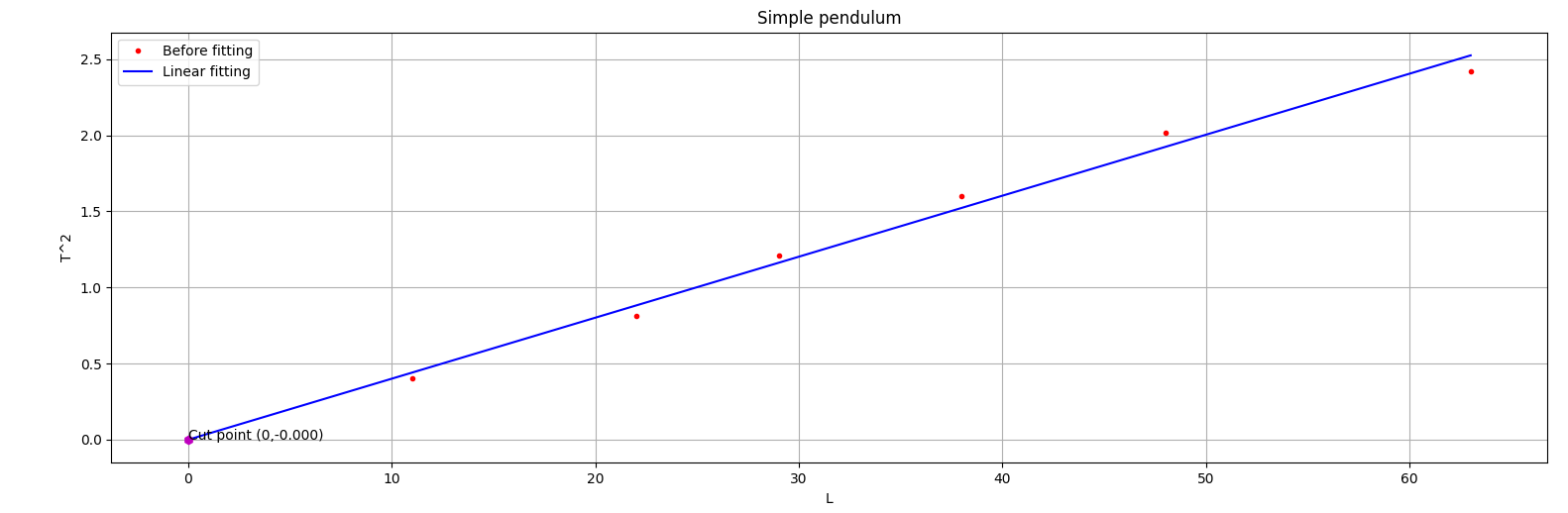


We want to find the optimal values of the slope and intercept that fit the data above.

|  |
| --- |
| *def* linear\_fit(self):  # Least squares function  self.m = (self.n\*np.sum(self.x\*self.y)-np.sum(self.x)\*np.sum(self.y))/(self.n\*np.sum(self.x\*\*2)-np.sum(self.x)\*\*2)  self.c = (1/self.n)\*(np.sum(self.y)-self.m\*np.sum(self.x))  self.x\_smooth = np.linspace(min(self.x.min(),0), self.x.max(), 300)  self.y\_fit = self.m\*self.x\_smooth+self.c |

By using the code above

That gives us the following result:



And the fitting result is :

Slope : 0.04009754 s2/cm

Intercept = 0 s2

Earth gravitational acceleration = 984.55 cm /s2

What about if we want interpolation??

Or extrapolation??

**Interpolation and extrapolation:**

1-**Interpolation** means determining a value from the existing values in a given data set. Another way of describing it is the act of inserting or interjecting an intermediate value between two other values.

In data science or mathematics, interpolation is about calculating a function's value based on the value of other data points in a given sequence. This function may be represented as f(x), and the known x values may range from x0 to xn.

2-**Extrapolation** refers to estimating an unknown value based on extending a known sequence of values or facts. To extrapolate is to infer something not explicitly stated from existing information.

to interpolate or extrapolate with linear data sets we use the first degree newton approximation formula:



Note that is a finite divided-difference approximation of the first derivative, which also represents the slope of the line of the first-degree polynomial approximation.



We can test interpolation or extrapolation as follows :

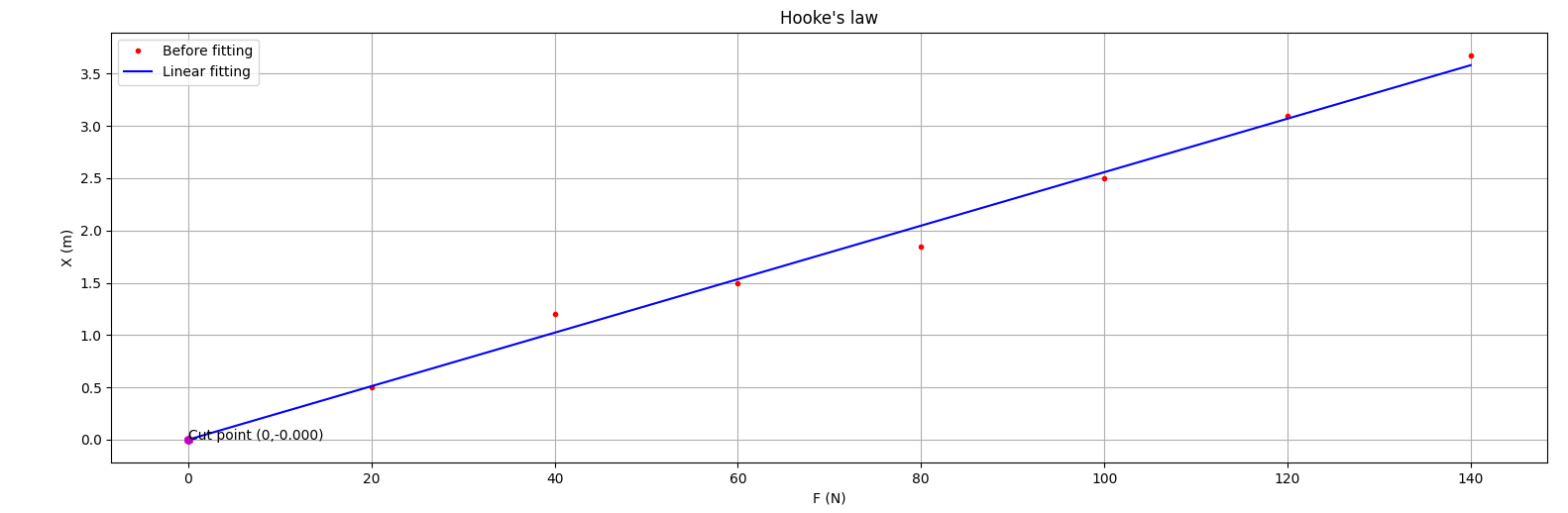
Lets take the extrapolation as an example

With the following hooke’s law expereiment points:

X = [20,40,60,80,100,120,140]

Y = [0.5,1.2,1.5,1.85,2.5,3.1,3.675]

Which gives us after fitting :

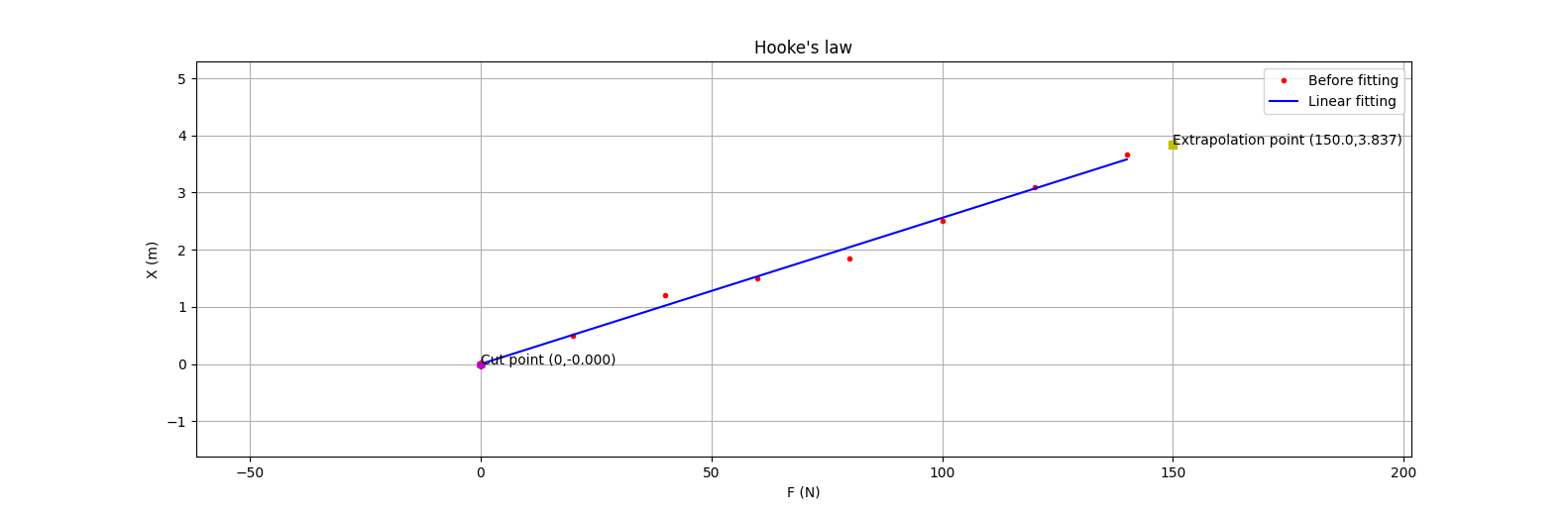


If we choose x=150 for extrapolation

By using following code without explaining for now (will be written as a full code later) :

|  |
| --- |
| *def* linear\_fit(self):  # Least squares function  self.m = (self.n\*np.sum(self.x\*self.y)-np.sum(self.x)\*np.sum(self.y))/(self.n\*np.sum(self.x\*\*2)-np.sum(self.x)\*\*2)  self.c = (1/self.n)\*(np.sum(self.y)-self.m\*np.sum(self.x))  self.x\_smooth = np.linspace(min(self.x.min(),0), self.x.max(), 300)  self.y\_fit = self.m\*self.x\_smooth+self.c  # first derivative function  *def* def1(self, i):  return (self.y\_fit[i+1]-self.y\_fit[i])/(self.x\_smooth[i+1]-self.x\_smooth[i])    # first degree newton function for interpolation or extrapolation  *def* newton1(self, X):  return self.y\_fit[0]+self.def1(0)\*(X-self.x\_smooth[0]) |

We can plot the result with extrapolation:



And the fitting result with extrapolation is :

Slope : 0.0255803 s2/gm

Intercept = 0 cm

The extrapolated point = (150,3.837)

The interpolation method will be the same as the extrapolation one except that it will belong to the fitting interval.

**2-Curve fitting**

**Introduction**:-

Let’s suppose that we are given a set of measured data points. **Curve fitting is the process of finding a mathematical function in an analytic form that best fits this set of data.** The first question that may arise is **why do we need that.** There are many cases that curve fitting can prove useful:

* quantify a general trend of the measured data
* remove noise from a function
* extract meaningful parameters from the learning curve
* summarize the relationships among two or more variables

**- In spectroscopy, data may be fitted with:-**

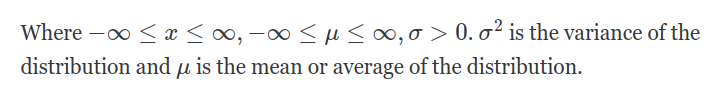
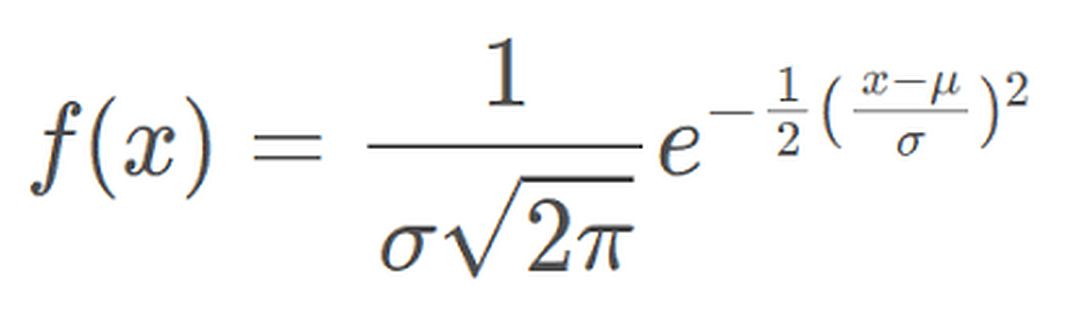
* Gaussian
* Lorentzian
* Voigt
* Other related functions

## **What is normal or Gaussian distribution?**

When we plot a dataset such as a histogram, the shape of that charted plot is what we call its distribution. The most commonly observed shape of continuous values is the bell curve, also called the Gaussian or normal distribution.

It is named after the German mathematician Carl Friedrich Gauss. Some common example datasets that follow Gaussian distribution are X-ray diffraction and photoluminescence in order to determine line widths and other properties.

In statistics, a normal distribution or Gaussian distribution is a type of continuous probability distribution for a real-valued random variable. The general form of its probability density function is :



The part:



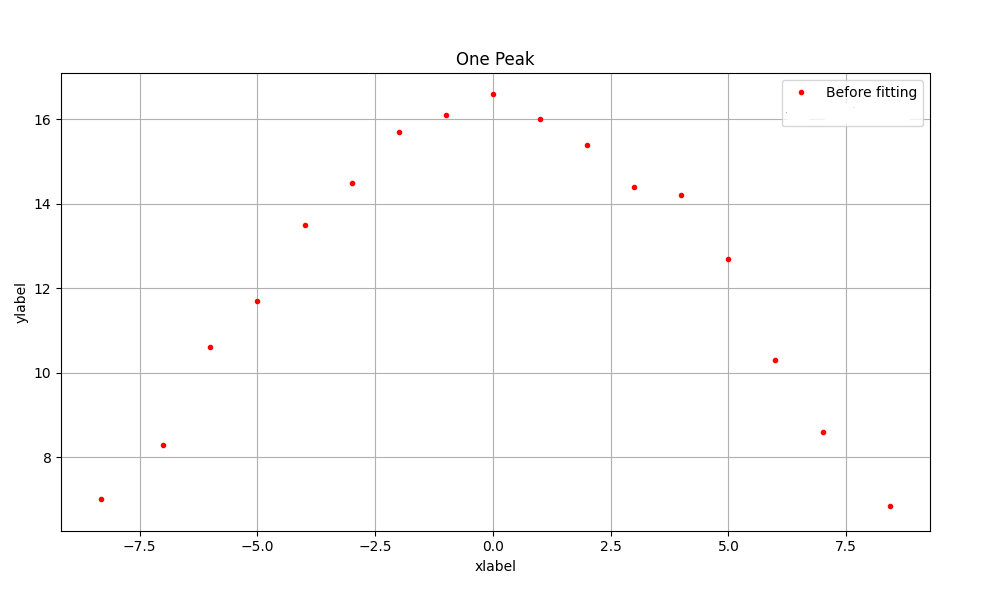
can be considered as the amplitude.

Consider the following points:

X=[-8.33, -7.0, -6.0, -5.0, -4.0, -3.0, -2.0, -1.0, 0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.44]

Y=[7.0, 8.3, 10.6, 11.7, 13.5, 14.5, 15.7, 16.1, 16.6, 16.0, 15.4, 14.4, 14.2, 12.7, 10.3, 8.6, 6.84]

Can be plotted as the following:



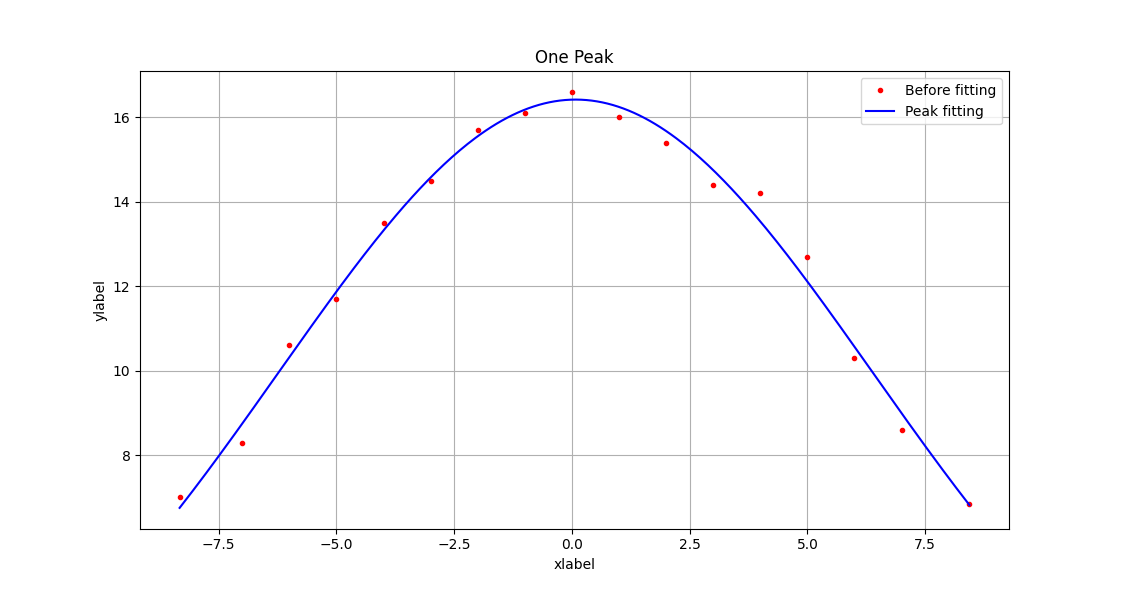
We want to find the optimal values of the amplitude ,mean and sigma that fit the data above.

|  |
| --- |
| *def* gaussian(x, amplitude, mean, sigma):  return amplitude \* np.exp(-(x - mean) \*\* 2 / (2 \*sigma\*\*2))  *def* gaussian\_fit(self):  model = Model(gaussian)  \_\_private\_shared\_fitting\_body(self,model) |

By using the code above and for now we will hide the big implementation of the

\_\_private\_shared\_fitting\_body function

That gives us the following result:



And the fitting result is :

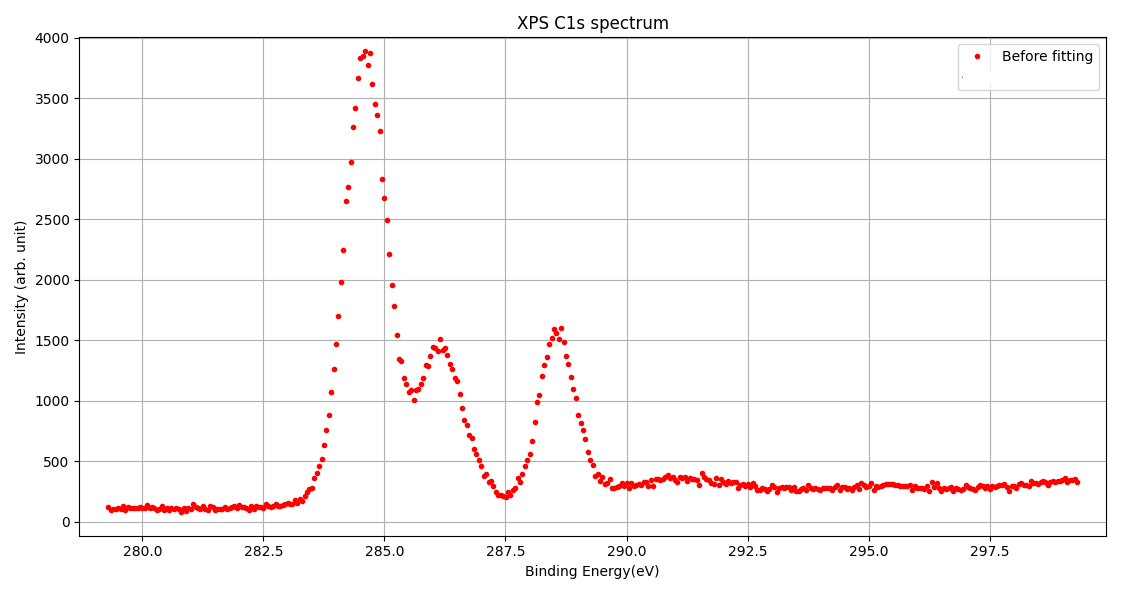
center: 0.07867061 +/- 0.06905517

fwhm: 14.8517187 +/- 0.22931927

height: 16.4218349 +/- 0.14335344

sigma: 6.30694434 +/- 0.09738293

If we have a more complex example like the image below:



That’s need a more complex model

So we will use (Composite gaussian model)

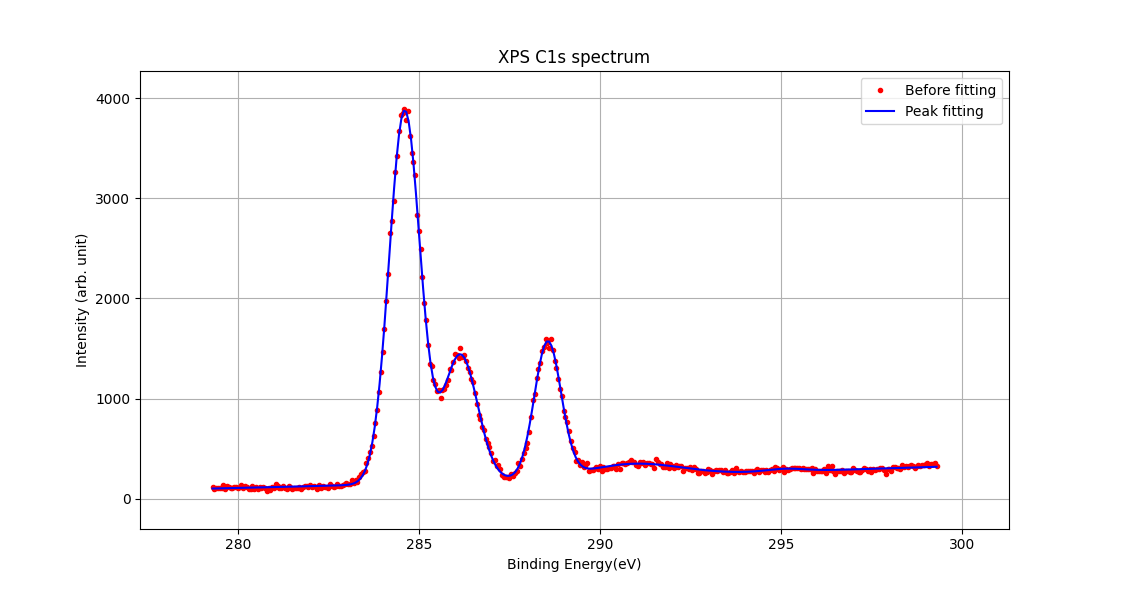
With the following initial guesses for the parameters (the highlighted initial guesses are background points):

Amplitude = [3870,1490,1590,350,316,356]

Mean = [284.56,286.11,288.57,291.33, 295.39 ,298.98]

Sigma = [1,1,1,3,10,10]

To apply gaussian model for every peak and the image below will be the result:



And the fitting result is:

peak1\_center: 284.608380 +/- 0.00132443

peak1\_fwhm: 1.00666023 +/- 0.00332189

peak1\_height: 3723.60375 +/- 8.90207580

peak1\_sigma: 0.42748925 +/- 0.00141067

peak2\_center: 286.148459 +/- 0.00406750

peak2\_fwhm: 1.11348405 +/- 0.01110725

peak2\_height: 1271.12562 +/- 9.05365056

peak2\_sigma: 0.47285315 +/- 0.00471681

peak3\_center: 288.560908 +/- 0.00305425

peak3\_fwhm: 0.86876038 +/- 0.00801579

peak3\_height: 1356.59893 +/- 10.0283242

peak3\_sigma: 0.36892857 +/- 0.00340399

**For more accuracy for the composite model we give it initial guesses at the background points too**

**That gives us the following results:-**

peak4\_center: 295.125096 +/- 0.16164702

peak4\_fwhm: 1.24749048 +/- 0.47057135

peak4\_height: 28.1504220 +/- 8.71267418

peak4\_sigma: 0.52976044 +/- 0.19983326

peak5\_center: 290.921913 +/- 0.07113044

peak5\_fwhm: 3.05715470 +/- 0.26354108

peak5\_height: 128.518442 +/- 8.80025054

peak5\_sigma: 1.29825409 +/- 0.11191560

peak6\_center: 313.929000 +/- 1.99742797

peak6\_fwhm: 48.8683479 +/- 11.8489739

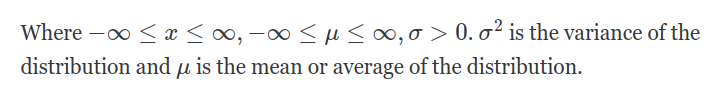
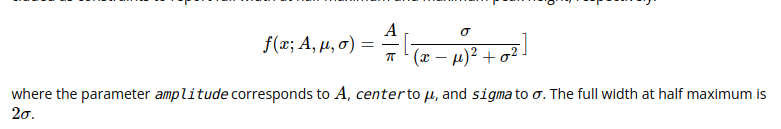
peak6\_height: 408.126497 +/- 118.156141

peak6\_sigma: 20.7524770 +/- 5.03179601

## **What is Cauchy or Lorentzian distribution?**

A Lorentzian distribution is bell shaped, but has much wider tails than does a Gaussian distribution.

The general form of lorentzian function is:



And A is the amplitude.

If we used the one peak example that we used

With gaussian distribution above but with lorentzian model from lmfit library instead

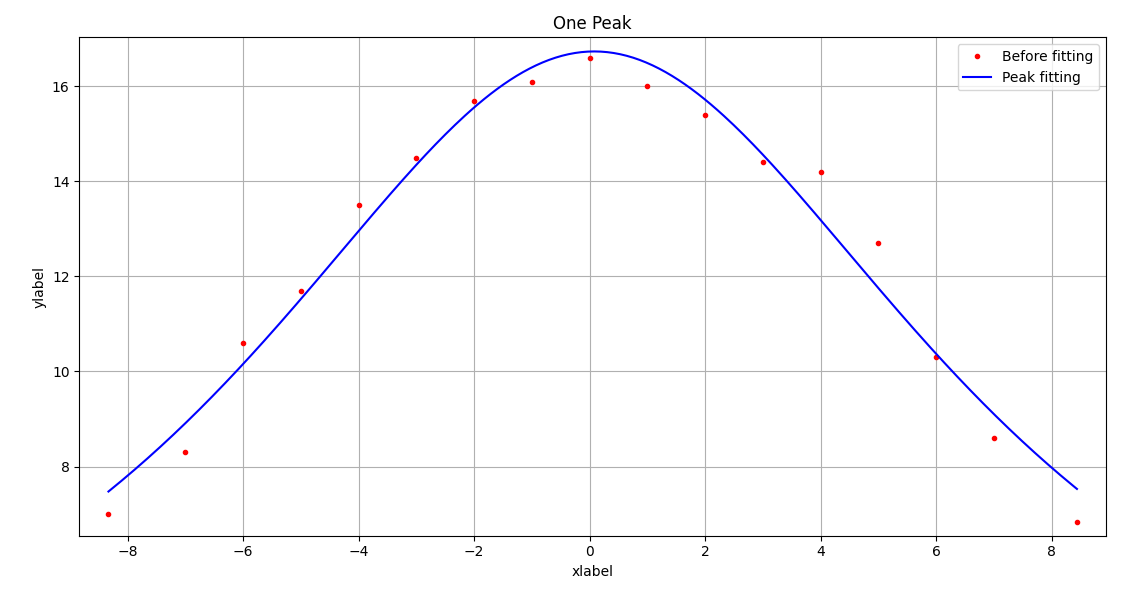
To not reinvent the wheel:

|  |
| --- |
| *def* lorentzian\_fit(self):  model = lorentzianModel()  \_\_private\_shared\_fitting\_body(self,model) |

By using the code above and for now we will hide the big implementation of the

\_\_private\_shared\_fitting\_body function

That gives us the following result:



And the fitting result is :

center: 0.08267810 +/- 0.11651662

fwhm: 15.1123113 +/- 0.49164474

height: 16.7372339 +/- 0.25243151

sigma: 7.55615564 +/- 0.24582237

We can see that the result is slightly changed from the result of gaussian function because of

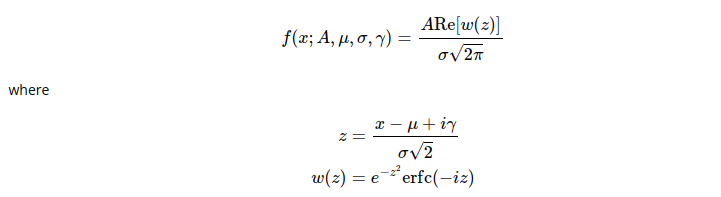
Lorentzian function has much wider tails than does a Gaussian distribution.

...We can use composite model with lorentzian model like what we did with gaussian model.

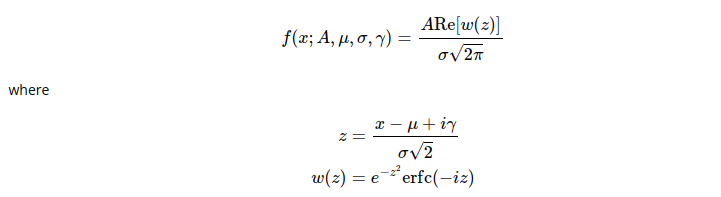
## **What is Voigt profile or Voigt distribution?**

The Voigt profile (named after Woldemar Voigt) is a probability distribution given by a convolution of a Lorentzian distribution and a Gaussian distribution. It is often used in analyzing data from spectroscopy or diffraction.

The general form of lorentzian function is:



Where



and erfc() is the complementary error function. As above, amplitude corresponds to A, center to μ, and sigma to σ. The parameter gamma corresponds to ɣ. If gamma is kept at the default value (constrained to sigma), the full width at half maximum is approximately 3.6013σ .

If we used the one peak example that we used

With gaussian distribution above but with voigt model from lmfit library instead

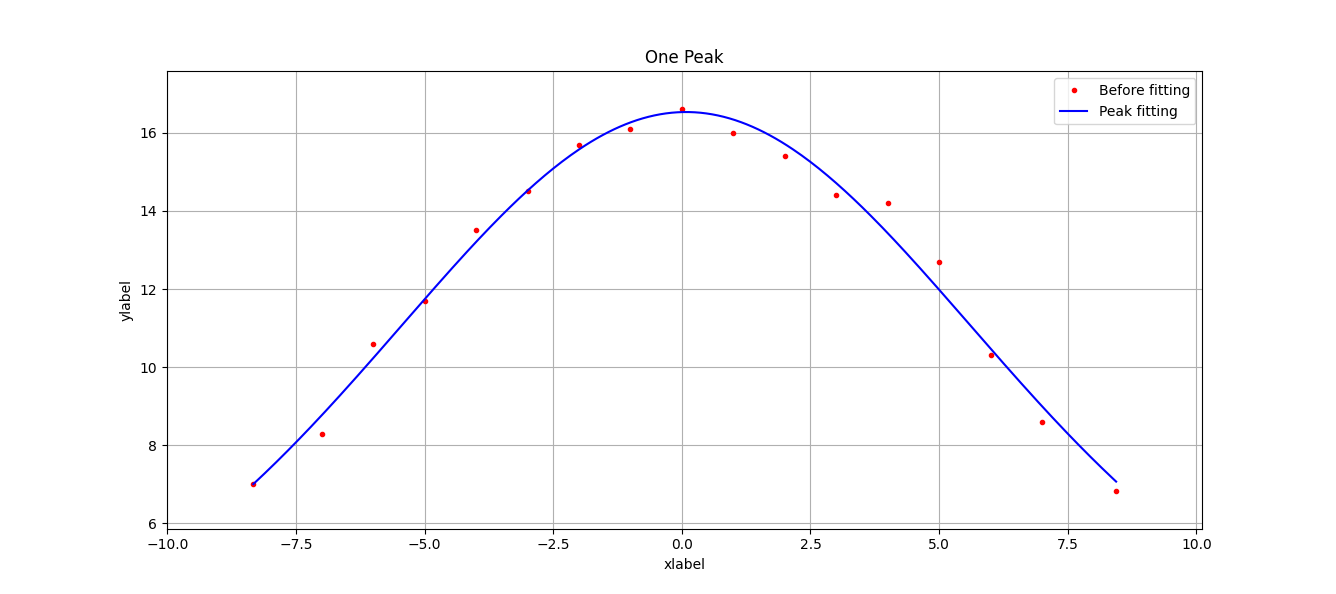
To not reinvent the wheel:

|  |
| --- |
| *def* voigt\_fit(self):  model = voigtModel()  \_\_private\_shared\_fitting\_body(self,model) |

By using the code above and for now we will hide the big implementation of the

\_\_private\_shared\_fitting\_body function

That gives us the following result:



And the fitting result is :

center: 0.08138378 +/- 0.07887461

fwhm: 14.8821118 +/- 0.28397769

**gamma**: 4.13243843 +/- 0.07885442

height: 16.5324930 +/- 0.16565426

sigma: 4.13243843 +/- 0.07885442

We can see that the result is slightly changed from the result of gaussian and lorentzian functions and we have a new parameter in the result which is **gamma.**

...We can use composite model with voigt model like what we did with gaussian model.

**Section 2 (Python Desktop application):-**

First of all you can visit the following link to download the whole project files with the full structure of the project with the installation instructions :- [**https://github.com/Nabil-Nasr/Fitting-GUI**](https://github.com/Nabil-Nasr/Fitting-GUI)

**Small introduction:-**

The following tutorial is for Python desktop

Application **(built from scratch)** which is used to solve all the problems in **section 1** and all python codes that used before are parts from this application.

**1-The initial interface:**

(1)

(13)

(12)

(11)

(10)

(7)

(9)

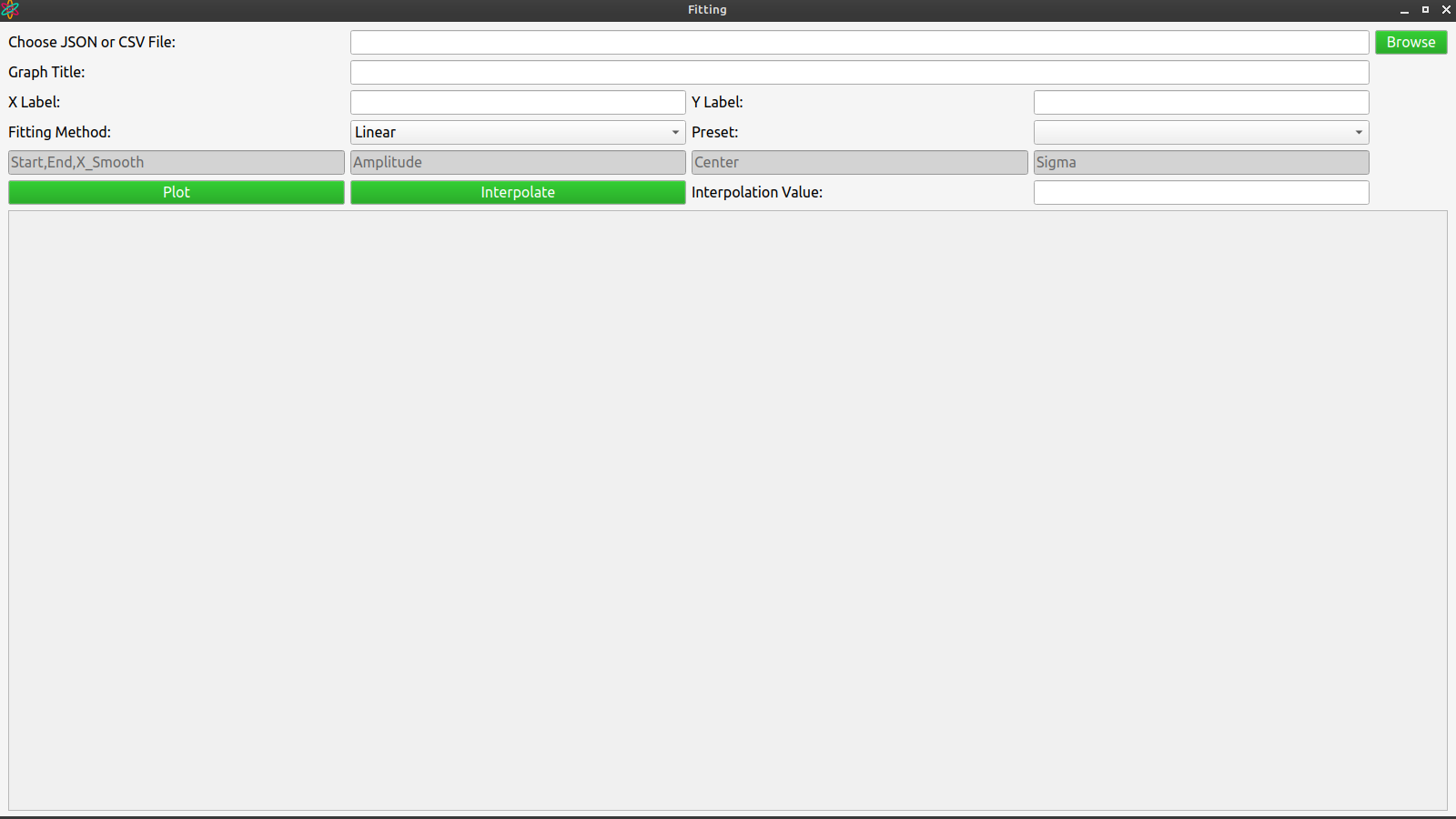
(8)

(6)

(5)

(4)

(3)



(2)

Notes:-

A) The fields (7),(8),(9) and (10) are enabled only when using curve fitting methods.

B) The field (13) is enabled only when using linear method.

**(1)**The first step to start the application by choosing the dataset file(CSV :a generated file by XPS device for example and JSON: a generated file by modern programming languages).

**(2)**Graph title (experiment title).

**(3)**X-axis label.

**(4)**Y-axis label.

**(5)**Choosing fitting method depending on the dataset (the available methods are :[linear,gaussian,lorentzian,voigt]).

**(6)**Add Preset with the experiment result (the available presets are:[Hooke’s law,Simple Pendulum] for adding the measuring units and

the experiment requirements ).

**(7)** limit x interval with start and end and you can choose the number of fitting points (x\_smooth) for more curve smoothness with the big datasets (start ,end and x\_smooth all separated by comma).

**(8)**Add the initial guesses for the amplitude parameter separated by comma.

**(9)**Add the initial guesses for the center(mean) parameter separated by comma.

**(10)**Add the initial guesses for sigma parameter separated by comma.

**(11)**Plot the points when all the fields are filled except the interpolation value field.

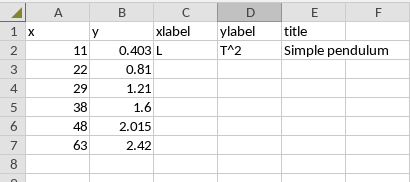
**(12)**Plot the points with interpolation when all the fields are filled with the interpolation value field.

**(13)**Add the interpolation or extrapolation value before pressing button (12) (when pressing enter in this field interpolate button will be pressed too).

**2-the dataset file structure(partially optional):**

A)CSV file:-

By opening the file with excel and choosing the comma delimiter then the structure of the file should be like this:

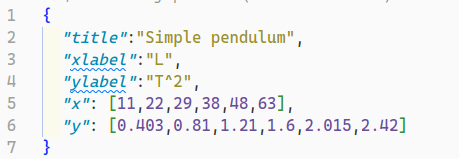


Where A1=x and B1=y are required for the program to read the file

And the columns C,D and E every one of them is optional (but will be good for program to do autofill the fields of the experiment)

B) JSON file:-

By opening the file with any text editor then the structure of the file should be like this if we compared with previous CSV file constraints:



Where x and y keys are required for the program to read the file

And the keys title,xlabel and ylabel every one of them is optional (but will be good for program to do autofill the fields of the experiment).

**3-The interface with linear fitting:-**

(1)

(11)

(10)

(9)

(8)

(7)

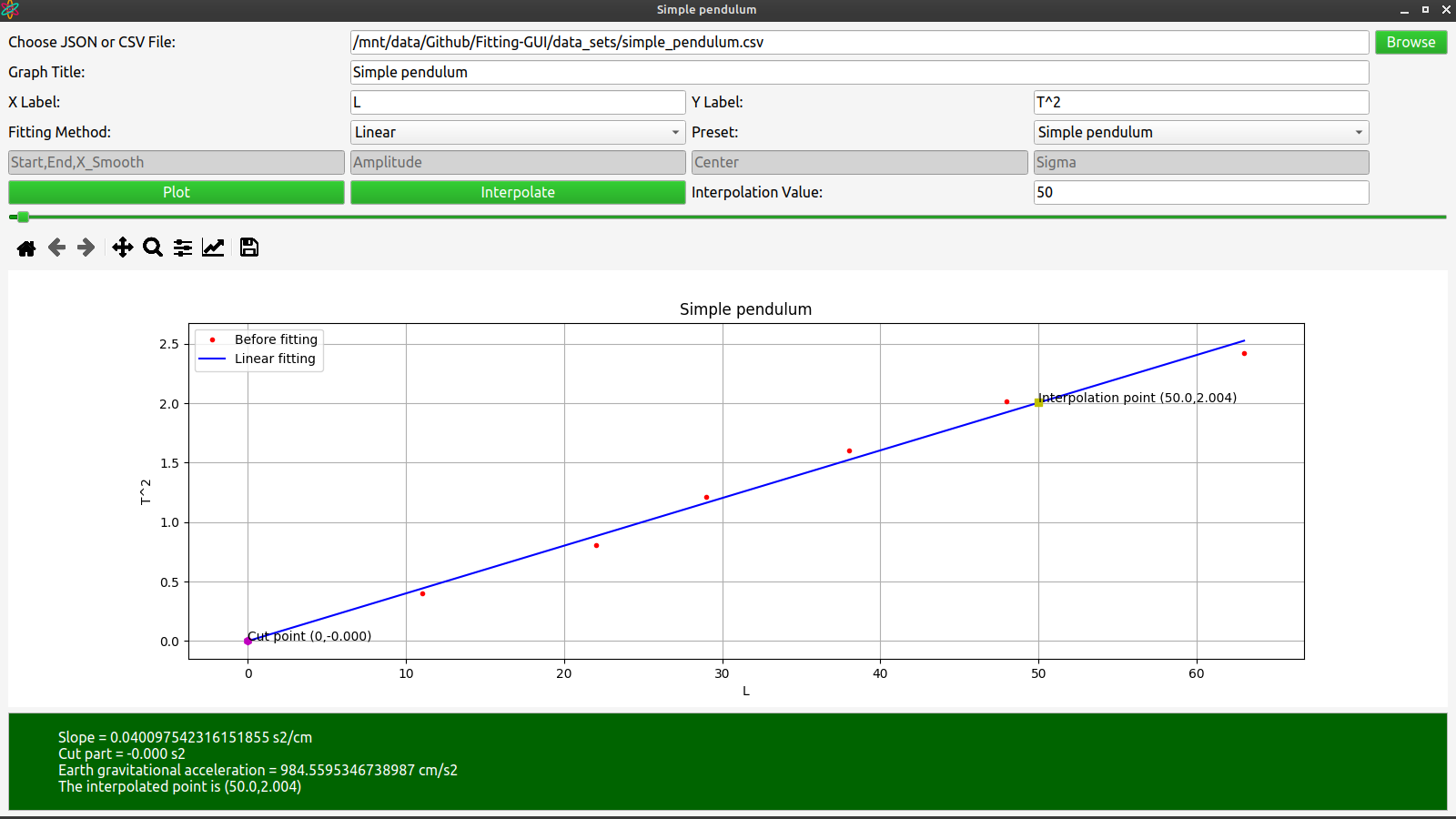
(6)

(5)

(4)

(2)

(3)



**(1)**Autofilled dataset file path.

**(2)**Zoom-in and zoom-out for the figure.

**(3)** Save png image for the figure.

**(4)** Edit the figure like changing the line or the points color and shape and more.

**(5)** configure subplots.

**(6)** zoom for any custom section in the figure.

**(7)** move the figure to any direction with mouse left click or zoom in or out for one axis like x or y with mouse right click.

**(8)**back or forward to a view that changed by the buttons (6) and (7).

**(9)**reset to original view.

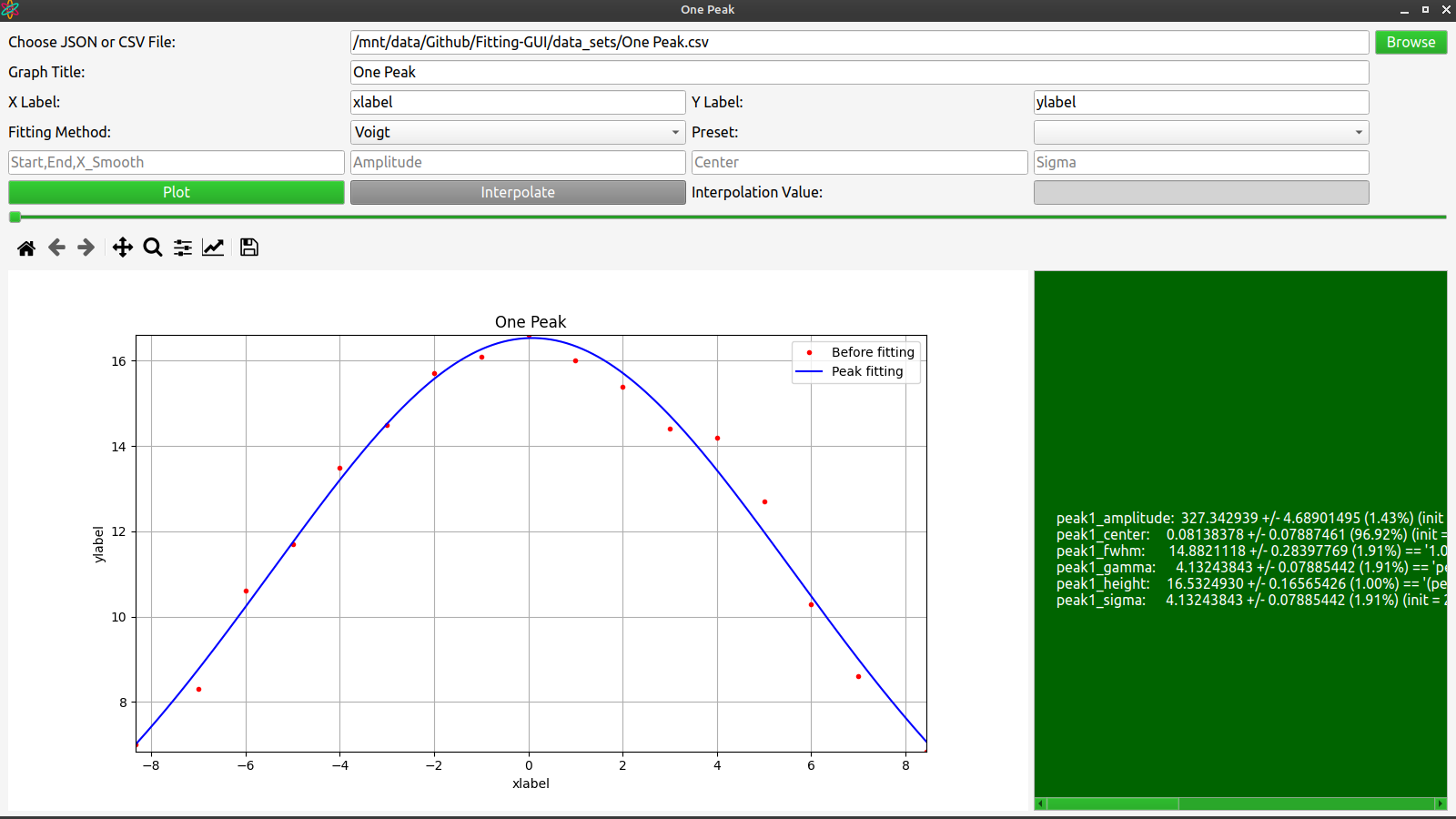
**(10)**The linear experiment result.

**(11)**Autofilled preset because it matches the title in the dataset file.

**4-The interface**

**with curve fitting:-**

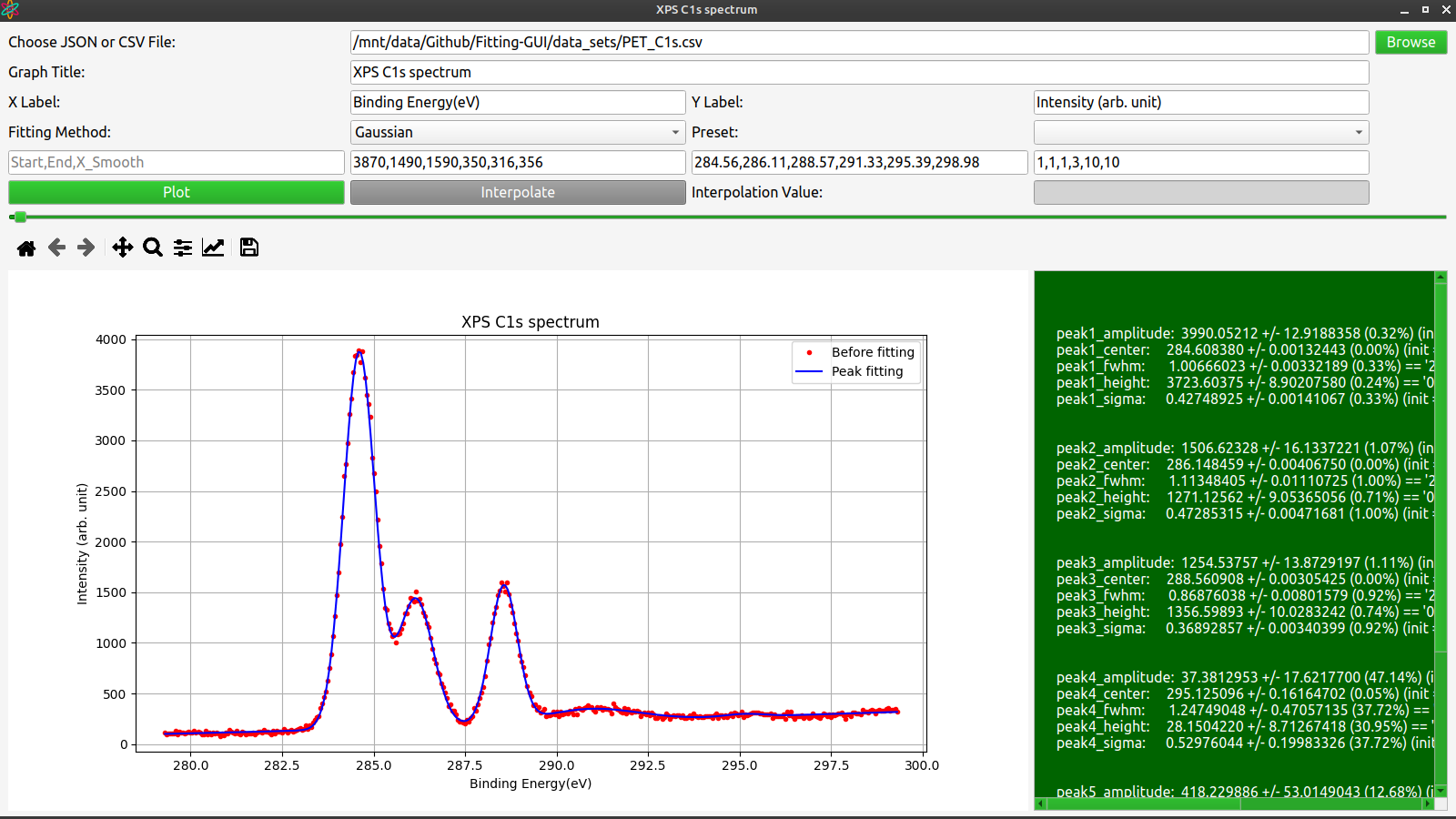
(1)



Note:-

One peak doesn’t need initial guesses in the fields Amplitude,Center and Sigma.

**(1)**Peak fitting result.



**5-The application full code**:-

The main structure of the project is the following files:-

(main.py , fitting\_functions.py and interpolation\_functions.py)

Where main.py is the execution file of the project.

code

**1)main.py**

Importing the used libraries in the file:

|  |
| --- |
| #!/usr/bin/python3    import sys  import matplotlib.pyplot as plt  from matplotlib.backends.backend\_qt5agg import FigureCanvasQTAgg as FigureCanvas  from matplotlib.backends.backend\_qt5agg import NavigationToolbar2QT as NavigationToolbar  from PyQt5.QtWidgets import QApplication, QMainWindow, QLabel, QGridLayout, QWidget, QPushButton, QFileDialog, QLineEdit, QMessageBox,QComboBox,QSlider,QScrollArea  from PyQt5.QtGui import QIcon, QPalette, QColor,QCursor  from PyQt5.QtCore import Qt  from fitting\_functions import \*  import interpolation\_functions  import os.path as path  import pandas as pd |

Constructing the GUI:

|  |
| --- |
| *class* MainWindow(*QMainWindow*):  *def* \_\_init\_\_(self):  *super*().\_\_init\_\_()    self.app\_dir=path.dirname(\_\_file\_\_)    MainWindow.linear\_fit=linear\_fit  MainWindow.gaussian\_fit=gaussian\_fit  MainWindow.lorentzian\_fit=lorentzian\_fit  MainWindow.voigt\_fit=voigt\_fit  MainWindow.def1=interpolation\_functions.def1  MainWindow.newton1=interpolation\_functions.newton1  MainWindow.newton2=interpolation\_functions.newton2    # Set window properties  self.setWindowTitle("Fitting")  self.setStyleSheet("""  \*{  font-size:16px;  }  QPushButton:disabled{  background-color:grey;  }  QLineEdit:disabled{  background-color:lightgrey  }  QMainWindow{  background:rgb(245,245,245);  }  QPushButton::hover{  background:green  }    """)    self.setWindowIcon(QIcon(path.join(self.app\_dir,'assets','icon.png')))  self.setGeometry(100, 100, 800, 720)    # Create central widget and grid layout  central\_widget = QWidget(self)  self.setCentralWidget(central\_widget)  self.grid\_layout = QGridLayout(central\_widget)    # Create labels and line edits for file input  file\_label = QLabel("Choose JSON or CSV File:")  self.file\_edit = QLineEdit()  self.file\_edit.setReadOnly(True)  file\_button = QPushButton("Browse")  file\_button.clicked.connect(self.browse\_file)  file\_button.setCursor(QCursor(Qt.PointingHandCursor))    # Create labels and line edits for plot input  title\_label = QLabel("Graph Title:")  self.title\_edit = QLineEdit()  xlabel\_label = QLabel("X Label:")  self.xlabel\_edit = QLineEdit()  ylabel\_label = QLabel("Y Label:")  self.ylabel\_edit = QLineEdit()    fitting\_method\_label=QLabel("Fitting Method:")  self.fitting\_method\_edit=QComboBox()  self.fitting\_method\_edit.addItems(["Linear","Gaussian","Lorentzian","Voigt"])  self.fitting\_method\_edit.currentIndexChanged.connect(self.disable\_enable\_interpolation)  experiment\_label=QLabel("Preset:")  self.experiment\_edit=QComboBox()  self.experiment\_edit.addItems(["","Simple pendulum","Hooke's law"])    # will be used if we used custom fitting functions instead of the models  self.start\_end\_xsmooth\_edit = QLineEdit()  self.start\_end\_xsmooth\_edit.setPlaceholderText('Start,End,X\_Smooth')  self.amplitude\_edit = QLineEdit()  self.amplitude\_edit.setPlaceholderText('Amplitude')  self.center\_edit = QLineEdit()  self.center\_edit.setPlaceholderText('Center')  self.sigma\_edit = QLineEdit()  self.sigma\_edit.setPlaceholderText('Sigma')    # Create buttons for plot and interpolation  plot\_button = QPushButton("Plot")  plot\_button.clicked.connect(self.plot)  plot\_button.setCursor(QCursor(Qt.PointingHandCursor))  self.interp\_button = QPushButton("Interpolate")  self.interp\_button.clicked.connect(self.interpolate)  self.interp\_button.setCursor(QCursor(Qt.PointingHandCursor))  # Create label for interpolation input  self.interp\_label = QLabel("Interpolation Value:")  self.interp\_edit = QLineEdit()  # When pressing enter button  self.interp\_edit.returnPressed.connect(self.interp\_button.click)    # Create slider for graph zoom  self.slider=QSlider(Qt.Horizontal)  self.slider.setRange(0,1000)  self.slider.setValue(0)  self.slider.valueChanged.connect(self.graph\_draw\_zoom)    # Create figure canvas for plot output  self.canvas = FigureCanvas(plt.Figure())    # Create label for results  self.result\_label = QLabel()  self.scroll\_area = QScrollArea(self)  self.scroll\_area.setWidgetResizable(True)  self.scroll\_area.setWidget(self.result\_label)    # Add widgets to grid layout  self.grid\_layout.addWidget(file\_label, 0, 0)  self.grid\_layout.addWidget(self.file\_edit, 0, 1, 1, 3)  self.grid\_layout.addWidget(file\_button, 0, 4)    self.grid\_layout.addWidget(title\_label, 1, 0)  self.grid\_layout.addWidget(self.title\_edit, 1, 1, 1, 3)    self.grid\_layout.addWidget(xlabel\_label, 2, 0)  self.grid\_layout.addWidget(self.xlabel\_edit, 2, 1)  self.grid\_layout.addWidget(ylabel\_label, 2, 2)  self.grid\_layout.addWidget(self.ylabel\_edit, 2, 3)    self.grid\_layout.addWidget(fitting\_method\_label, 3, 0)  self.grid\_layout.addWidget(self.fitting\_method\_edit, 3, 1)  self.grid\_layout.addWidget(experiment\_label, 3, 2)  self.grid\_layout.addWidget(self.experiment\_edit, 3, 3)    # will be used if we used custom fitting functions instead of the models  self.grid\_layout.addWidget(self.start\_end\_xsmooth\_edit,4,0)  self.grid\_layout.addWidget(self.amplitude\_edit,4,1)  self.grid\_layout.addWidget(self.center\_edit,4,2)  self.grid\_layout.addWidget(self.sigma\_edit,4,3)  self.start\_end\_xsmooth\_edit.setEnabled(False)  self.amplitude\_edit.setEnabled(False)  self.center\_edit.setEnabled(False)  self.sigma\_edit.setEnabled(False)    self.grid\_layout.addWidget(plot\_button, 5, 0)  self.grid\_layout.addWidget(self.interp\_button, 5, 1)  self.grid\_layout.addWidget(self.interp\_label, 5, 2)  self.grid\_layout.addWidget(self.interp\_edit, 5, 3)    self.grid\_layout.addWidget(self.scroll\_area, 9, 0, 1, 5) |

The method of reading the dataset file and autofilling fields **:**

|  |
| --- |
| *def* browse\_file(self):  # Open file dialog and get selected file path  file\_path,\_= QFileDialog.getOpenFileName(self, "Open JSON or CSV File", path.join(self.app\_dir,'data\_sets'), "JSON or CSV Files (\*.json \*.csv)")    if file\_path:  file\_ext = path.splitext(file\_path)[1]  # Read file and set default input values  if(file\_ext==".csv"):  file= pd.read\_csv(file\_path)  elif(file\_ext==".json"):  file= pd.read\_json(file\_path)    try:  title=file['title'][0]  self.title\_edit.setText(title)  self.experiment\_edit.setCurrentIndex(0)  for i in range(self.experiment\_edit.count()):  if(self.experiment\_edit.itemText(i)==title):  self.experiment\_edit.setCurrentIndex(i)  except *KeyError*:  self.title\_edit.setText("")  self.experiment\_edit.setCurrentIndex(0)    try:  xlabel=file['xlabel'][0]  self.xlabel\_edit.setText(xlabel)  except *KeyError*:  self.xlabel\_edit.setText("")    try:  ylabel=file['ylabel'][0]  self.ylabel\_edit.setText(ylabel)  except *KeyError*:  self.ylabel\_edit.setText("")    try:  self.x = file['x']  self.y = file['y']  indexes = *list*(range(len(self.x)))  indexes.sort(key=self.x.\_\_getitem\_\_)  self.x=np.array(*list*(map(self.x.\_\_getitem\_\_,indexes)))  self.y=np.array(*list*(map(self.y.\_\_getitem\_\_,indexes)))  # those value values to restore x and y when errors happens  self.x\_temp=self.x  self.y\_temp=self.y  except *KeyError* as error:  self.title\_edit.setText("")  self.experiment\_edit.setCurrentIndex(0)  self.xlabel\_edit.setText("")  self.ylabel\_edit.setText("")  QMessageBox.warning(self, "Data error", *f*"Please provide a valid {error} column name in the chosen file .")  return    self.n = len(self.x)  self.file\_edit.setText(file\_path) |

The shared method between the regular plotting and the plotting with interpolation:

|  |
| --- |
| *def* shared\_plot(self):  # Get input values  filename = self.file\_edit.text()  xlabel = self.xlabel\_edit.text()  ylabel = self.ylabel\_edit.text()  title = self.title\_edit.text()  self.setWindowTitle(title)    # Check if all input values are provided  if not all([filename, xlabel, ylabel, title]):  QMessageBox.warning(self, "Error", "Please provide all input values.")  return  experiment=self.select\_fitting\_method()  self.result = *f*"{self.add\_experiment\_result()}"  self.x=self.x\_temp  self.y=self.y\_temp    # Clear previous plot and draw new plot on canvas  self.canvas.figure.clear()  self.ax = self.canvas.figure.add\_subplot(111)  self.ax.plot(experiment.x, experiment.y,'ro',markersize=3,label="Before fitting")  self.ax.plot(experiment.x\_smooth, experiment.y\_fit,'b',label=experiment.after\_fitting\_label)    # Plot cut part point  if(self.fitting\_method\_edit.currentText()=="Linear"):  self.ax.plot(0, experiment.c, "h m")  self.ax.text(0, experiment.c, *f*"Cut point (0,{experiment.c*:.3f*})")  self.result\_label.setText(self.result)  self.result\_label.setStyleSheet("background:darkgreen;color:white;padding:5px 50px;font-size:16px;")  else:  self.result\_label.setText(self.peak\_result)  self.result\_label.setStyleSheet("background:darkgreen;color:white;padding:5px;font-size:16px;")    self.min\_x=min(\*experiment.x,\*experiment.x\_smooth)  self.max\_x=max(\*experiment.x,\*experiment.x\_smooth)  self.min\_y=min(\*experiment.y,\*experiment.y\_fit)  self.max\_y=max(\*experiment.y,\*experiment.y\_fit)    self.ax.legend(loc="best")  self.ax.set\_xlabel(xlabel)  self.ax.set\_ylabel(ylabel)  self.ax.set\_title(title)  self.ax.grid() |

The method for plotting without interpolation:

|  |
| --- |
| *def* plot(self):  self.shared\_plot()  self.graph\_draw\_zoom() |

The method for plotting with interpolation:

|  |
| --- |
| # plot with interpolation  *def* interpolate(self):  interp\_value = self.interp\_edit.text()  try:  interp\_value = *float*(interp\_value)  except *ValueError*:  QMessageBox.warning(self, "Error", "Please provide a valid interpolation value.")  return  self.shared\_plot()    experiment=self  y\_interp= experiment.newton1(interp\_value)  # Plot interpolated/extrapolated point  self.ax.plot(interp\_value, y\_interp, "s y")    # Determine if interpolated/extrapolated point is within plot range  interp\_point=*f*"({interp\_value},{y\_interp*:.3f*})"  if min(experiment.x) <=interp\_value <= max(experiment.x):  interpolation\_text=*f*"The interpolated point is {interp\_point}"  self.ax.text(interp\_value, y\_interp, *f*"Interpolation point {interp\_point}")  else:  interpolation\_text=*f*"The extrapolated point is {interp\_point}"  self.ax.text(interp\_value, y\_interp, *f*"Extrapolation point {interp\_point}")  self.result = *f*"{self.result}\n{interpolation\_text}"  self.result\_label.setText(self.result)    self.graph\_draw\_zoom() |

The method for disabling or enabling the parameters fields and the interpolation field depending on the selected fitting method:

|  |
| --- |
| *def* disable\_enable\_interpolation(self):  if(self.fitting\_method\_edit.currentText()=="Linear"):  self.interp\_button.setEnabled(True)  self.interp\_edit.setEnabled(True)  self.start\_end\_xsmooth\_edit.setEnabled(False)  self.amplitude\_edit.setEnabled(False)  self.center\_edit.setEnabled(False)  self.sigma\_edit.setEnabled(False)  else:  self.interp\_button.setEnabled(False)  self.interp\_edit.setEnabled(False)  self.start\_end\_xsmooth\_edit.setEnabled(True)  self.amplitude\_edit.setEnabled(True)  self.center\_edit.setEnabled(True)  self.sigma\_edit.setEnabled(True) |

The method for changing the place of experiment result label depending on the fitting method:

|  |
| --- |
| *def* select\_fitting\_method(self):  current\_method = self.fitting\_method\_edit.currentText()    if current\_method == "Linear":  self.linear\_fit()  self.after\_fitting\_label="Linear fitting"  self.grid\_layout.removeWidget(self.scroll\_area)  self.grid\_layout.addWidget(self.canvas, 8, 0, 1, 5)  self.grid\_layout.addWidget(self.scroll\_area, 9, 0, 1, 5)  else:  self.grid\_layout.removeWidget(self.scroll\_area)  self.grid\_layout.addWidget(self.canvas, 8, 0, 2, 3)  self.grid\_layout.addWidget(self.scroll\_area, 8, 3, 2, 2)  if current\_method == "Gaussian":  self.gaussian\_fit()  elif current\_method == "Lorentzian":  self.lorentzian\_fit()  elif current\_method == "Voigt":  self.voigt\_fit()  self.after\_fitting\_label="Peak fitting"    return self |

The method for adding the linear experiment result depending on The selected preset:

|  |
| --- |
| *def* add\_experiment\_result(self):  current\_experiment=self.experiment\_edit.currentText()  result=""  if(self.fitting\_method\_edit.currentText()=="Linear"):  if(current\_experiment=="Simple pendulum"):  g = 4\*(np.pi\*\*2)/self.m  result=*f*"Slope = {self.m} s2/cm\nCut part = {self.c*:.3f*} s2\nEarth gravitational acceleration = {g} cm/s2"  elif(current\_experiment=="Hooke's law"):  result=*f*"Slope = {self.m} s2/gm\nCut part = {self.c*:.3f*} cm"  else:  result=*f*"Slope = {self.m}\nCut part = {self.c*:.3f*}"  return result |

The method for drawing the graph when changing the value of the slider:

|  |
| --- |
| *def* graph\_draw\_zoom(self):  lim\_percentage=self.slider.value()/100  self.ax.set\_xlim(self.min\_x-(self.max\_x-self.min\_x)\*lim\_percentage, self.max\_x+(self.max\_x-self.min\_x)\*lim\_percentage)  self.ax.set\_ylim(self.min\_y-(self.max\_y-self.min\_y)\*lim\_percentage, self.max\_y+(self.max\_y-self.min\_y)\*lim\_percentage)    self.grid\_layout.addWidget(self.slider, 6, 0, 1, 5)  self.toolbar = NavigationToolbar(self.canvas, self)  self.toolbar.locLabel.setStyleSheet("color:initial")  self.grid\_layout.addWidget(self.toolbar, 7, 0, 1, 5)  self.canvas.draw() |

Some Global styles and the execution of the application:

|  |
| --- |
| if \_\_name\_\_ == "\_\_main\_\_":  app = QApplication(sys.argv)    # Set application style to light mode  palette = QPalette()  palette.setColor(QPalette.Window, QColor(240, 240, 240))  palette.setColor(QPalette.WindowText, Qt.black)  palette.setColor(QPalette.Base, QColor(255, 255, 255))  palette.setColor(QPalette.AlternateBase, QColor(240, 240, 240))  palette.setColor(QPalette.ToolTipBase, Qt.black)  palette.setColor(QPalette.ToolTipText, Qt.white)  palette.setColor(QPalette.Text, Qt.black)  palette.setColor(QPalette.Button, QColor(0,150,0))  palette.setColor(QPalette.ButtonText, QColor("white"))  palette.setColor(QPalette.BrightText, Qt.black)  palette.setColor(QPalette.Link, QColor(0, 0, 255))  palette.setColor(QPalette.Highlight, QColor("green"))  palette.setColor(QPalette.HighlightedText, Qt.white)  app.setPalette(palette)    window = MainWindow()  combo\_palette = QPalette()  combo\_palette.setColor(QPalette.Button, QColor(240,240,240))  combo\_palette.setColor(QPalette.ButtonText, QColor("black"))  combo\_palette.setColor(QPalette.Highlight, QColor(0,255,0,100))  window.fitting\_method\_edit.setPalette(combo\_palette)  window.experiment\_edit.setPalette(combo\_palette)    window.show()  sys.exit(app.exec\_()) |

**B)interpolation\_functions.py**

|  |
| --- |
| # first derivative function  *def* def1(self, i):  return (self.y\_fit[i+1]-self.y\_fit[i])/(self.x\_smooth[i+1]-self.x\_smooth[i])    # first degree newton function  *def* newton1(self, X):  return self.y\_fit[0]+self.def1(0)\*(X-self.x\_smooth[0])    # second degree newton function(temporary)  *def* newton2(self, X):  return self.newton1(X)+((self.def1(1)-self.def1(0))/(self.x\_smooth[2]-self.x\_smooth[0]))\*(X-self.x\_smooth[0])\*(X-self.x\_smooth[1]) |

**C)fitting\_functions.py**

Importing the used libraries:

|  |
| --- |
| import numpy as np  from lmfit.models import Model,VoigtModel,LorentzianModel,GaussianModel  from scipy.optimize import curve\_fit  from scipy.special import wofz  import re  import copy |

Linear fitting function:

|  |
| --- |
| *def* linear\_fit(self):  # Least squares function  self.m = (self.n\*np.sum(self.x\*self.y)-np.sum(self.x)\*np.sum(self.y))/(self.n\*np.sum(self.x\*\*2)-np.sum(self.x)\*\*2)  self.c = (1/self.n)\*(np.sum(self.y)-self.m\*np.sum(self.x))  self.x\_smooth = np.linspace(min(self.x.min(),0), self.x.max(), 300)  self.y\_fit = self.m\*self.x\_smooth+self.c |

The shared function between the curve fitting function that can create one model and composite model:

|  |
| --- |
| *def* \_\_private\_shared\_fitting\_body(self,model:Model):  # params = model.make\_params(amplitude=self.amplitude, mean=self.mean, sigma=self.sigma,gamma=self.gamma)    # ======= creating interval start end x\_smooth started ================================  start\_end\_xsmooth\_arr=[self.x[0],self.x[-1],300]  if(self.start\_end\_xsmooth\_edit.text()!=''):  start\_end\_xsmooth=self.start\_end\_xsmooth\_edit.text().split(',')  n=len(start\_end\_xsmooth)  for i in range(n):  if(start\_end\_xsmooth[i]!=''):  start\_end\_xsmooth\_arr[i]=*float*(start\_end\_xsmooth[i])  # ======= creating interval start end x\_smooth ended ==================================    # ======= creating amplitude center sigma started ================================  params\_length=0  params\_list=[[],[],[]]  if(self.amplitude\_edit.text()!='' and self.center\_edit.text()!='' and self.sigma\_edit.text()!= ''):  amplitude=self.amplitude\_edit.text().split(',')  center=self.center\_edit.text().split(',')  sigma=self.sigma\_edit.text().split(',')  params\_length=len(amplitude)  for i in range(params\_length):  params\_list[0].append(*float*(amplitude[i]))  params\_list[1].append(*float*(center[i]))  params\_list[2].append(*float*(sigma[i]))    # ======= creating amplitude center sigma ended ==================================  model\_temp = copy.deepcopy(model)  model.prefix="peak1\_"  for i in range(params\_length):  amplitude=params\_list[0][i]  center=params\_list[1][i]  sigma=params\_list[2][i]  if(i==0):  model.set\_param\_hint('amplitude',value=amplitude,min=0)  model.set\_param\_hint('center',value=center,min=center\*0.95,max=center\*1.05)  model.set\_param\_hint('sigma',value=sigma,min=0)  continue  new\_model = copy.deepcopy(model\_temp)  new\_model.prefix=*f*"peak{i+1}\_"    new\_model.set\_param\_hint('amplitude',value=amplitude,min=0)  new\_model.set\_param\_hint('center',value=center,min=center\*0.95,max=center\*1.05)  new\_model.set\_param\_hint('sigma',value=sigma,min=0)  model+=new\_model    self.x\_smooth=np.linspace(start\_end\_xsmooth\_arr[0],start\_end\_xsmooth\_arr[1],*int*(start\_end\_xsmooth\_arr[2]))    params=model.make\_params()  if(params\_length==0):  params=model.guess(self.y,x=self.x)  result = model.fit(self.y,params,x=self.x)  self.y\_fit = result.eval(x=self.x\_smooth)  self.peak\_result=re.sub(*r*" (?=peak\d+\_amplitude)","\n\n ",result.fit\_report(show\_correl=False,sort\_pars=True).split('[[Variables]]')[-1]) |

The non shared function for adding the models of the curve fitting:

|  |
| --- |
| *def* gaussian(x, amplitude, mean, sigma):  return amplitude \* np.exp(-(x - mean) \*\* 2 / (2 \* sigma \*\* 2))    *def* gaussian\_fit(self):  # model = Model(gaussian)  # GaussianModel used function as the function above  model= GaussianModel()  \_\_private\_shared\_fitting\_body(self,model)    # def lorentzian(x, amplitude, mean, sigma):  # return (amplitude \* sigma\*\*2) / ((x - mean)\*\*2 + sigma\*\*2)    *def* lorentzian\_fit(self):  # Fit Lorentzian distribution  # model = Model(lorentzian)  # LorentzianModel used function as the function above  model=LorentzianModel()  \_\_private\_shared\_fitting\_body(self,model)    *def* voigt(x, amplitude, mean, sigma, gamma):  z = ((x - mean) + 1*j*\*gamma) / (sigma \* np.sqrt(2))  return amplitude \* np.real(wofz(z))    *def* voigt\_fit(self):  # Fit Voigt distribution  # VoigtModel used function as the function above  model = VoigtModel()  \_\_private\_shared\_fitting\_body(self,model) |

The variables and functions that can be imported when using from fitting\_functions import \*

|  |
| --- |
| \_\_all\_\_=["linear\_fit","gaussian\_fit","lorentzian\_fit","voigt\_fit","np"] |