Phys205-Week01-Student

September 26, 2019

1 Computational Physics - Week 1

1.1 Table of contents week 1

1.2 Introduction to week 1

The Computational Physics module (Phys205) builds on the knowledge you acquired in the Introduction to Computational Physics module (Phys105) you took in your first year. The first few weeks follow a format similar to that used in Phys105: there will be a lecture that introduces an aspect of Python and then a computer lab in which you will work through a Jupyter Notebook that contains some exercises. Demonstrators will be available to help with any difficulties you have, and you should get one of them to mark your work when you have finished it.

The second part of the course, which will start later this semester, consists of a set of short projects that you will tackle in groups of about three. Example projects could include determining Feigenbaum's constant by investigating the behaviour of the logistic map and writing a Monte Carlo to simulate the determination of π using raindrops. Again, demonstrators will be available to help you with any technical problems you have. The projects will be assessed through short individual reports, which should be written as Jupyter Notebooks. If you have any ideas for projects you

would like to try, discuss them with me and we will decide if they are suitable!

The final section of the course, which will run in semester 2, consists of longer projects. You will work on these in groups of six to eight. Possible topics include modelling traffic flow in the Birkenhead tunnel, and investigating the relationship between the availability of prey and the size and number of predators in a range of habitats. You will be asked to produce an initial project plan, describing how you plan to tackle the problem you have been given, and a group report at the end of the project. You will also give a presentation explaining your project to the other Phys205 groups. Again, if you have any ideas for topics you would like to tackle, please discuss them with me!

Today, we will start with a reminder on how to install the tools we will be using in the course (the Python programming language and Jupyter Notebooks) and then look at how data can be imported into a Python program and analysed using the Pandas package. (We will look at Pandas in more detail next week.) We will finish by revisiting the problem of fitting data using the `least_squares` routine from the SciPy library, which you will be using extensively in Practical Physics II (Phys206).

Today, and in all our Phys205 sessions, please read through the relevant sections of the Notebook before you try the problems. (Of course, you can miss out revision topics, like how to use Markdown, if you are familiar with them!). If you don't understand anything in the Notebook, or have difficulties with the problems, ask one of the demonstrators for help. You can also consult the recommended textbooks (A student's guide to Python for physical modelling, or Learning scientific programming with Python), both of which are available in the libary. Alternatively, ask Google - there is lots of information on Python and Jupyter Notebooks out there!

1.3 Installing Python and Jupyter software

Python and Jupyter Notebooks have aready been installed on the University PCs you will be using in the Phys205 computing sessions.

If you want to use Python and Jupyter Notebooks on your own computer, you can install the software required for Windows, Mac or Linux PCs as follows:
Go to the Anaconda web site https://www.anaconda.com/download/. Select Windows, macOS or Linux, depending on the operating system running on your computer. Download the version of Anaconda labelled Python 3.7, not the Python 2.7 version. Follow the installation instructions on the web site.

Once Anaconda is installed, open it and launch Jupyter Notebook to get started:

- On Windows, click the "Anaconda3" icon in the start menu and then "Jupyter Notebook".
- On Linux, open a terminal window , type in the command "jupyter notebook &" and presss *Enter*.
- On a Mac, click on "Anaconda-Navigator" in the LaunchPad and then "Jupyter Notebook" or open a terminal window, type in the command "jupyter notebook &" and press *Enter*.

1.4 Jupyter Notebooks revisited - cells

Jupyter Notebooks consist of cells in which nicely formatted documents can be written and cells which contain computer code. The language used to create the document cells is called Markdown, and we will use the Python in the code cells.

When you start a new Notebook, its first cell will be a code cell and any new cell you create will be a code cell by default. You can create a new cell below the current cell by clicking the "+" symbol in the Notebook menu bar or by using the *Insert* menu. (Click on *Insert*, then on *Insert Cell Above*, or *Insert Cell Below*, as appropriate.) To change a cell's type to Markdown, select the cell (by clicking in it) and then click *Cell, Cell Type* and *Markdown*. Alternatively, select the cell, press *Esc*, then *m* (for Markdown). You will have to click inside the cell (or press *Enter*) before it is selected and you are able to type in it!

A Markdown cell can be changed to a code cell using the *Cell* menu, or by typing *Esc, y*.

Cells can be deleted by selecting them and then using the *Edit* menu, or by pressing *Esc, d, d*.

1.5 Markdown cells

Markdown is a way of writing nicely formatted text, allowing the inclusion of pictures, web links, videos and other features in your Notebooks.

To see how this cell was written using Markdown, double click on it.

To "run" or "compile" the cell, so the text is formatted nicely, select the cell (click in it) and press *Ctrl + Enter* (i.e. press the *Ctrl* and *Enter* keys at the same time, *Command + Enter* on a Mac). Alternatively, you can click on the *Run* button on the menu bar. If you press *Shift + Enter*, you will run the current cell and move to the next cell (or create a new cell below the current one if there isn't one already there).

Flipping between looking at the Markdown (double click) and the compiled cell (*Ctrl + Enter*) will allow you to see how Markdown can be used.

1.5.1 Entering text and tables

If you want to write some normal text, just type it into the cell. If you want to emphasize the text, you can *write in italics* (using asterisks before and after the section that you want to be in italics), or **make it bold** (using double asterisks). (There are *alternative ways of getting italics* and of **entering bold text**, using single and double underscores.) You can also ~~cross out text~~ (using two ~ symbols before and after the text to be "deleted").

If you want a new paragraph, press *Enter* twice, so you produce an empty line.

If you want to start a new line without having a new paragraph, leave two spaces at the end of the line, like this:
This is a new line.
So is this.

If you want a numbered list, do this: 1. This is the first entry. 2. This is the second. 3. And this the third.

Bulleted lists are also easy to produce: * This is the first bullet. * And this the second.

If you want to present information in a table, use the following syntax (double click to see the Markdown!):

Number	Angle (degrees)	Cosine of angle
0	0	1

Number	Angle (degrees)	Cosine of angle
1	30	0.866
2	45	0.707
3	60	0.5
4	90	0.0

Table 1 *The value of the cosine of several angles.*

(You don't have to line up the Markdown columns; when you run the cell that will be done automatically, but it does make reading and editing the Markdown easier!) Notice that you have to enter the caption "by hand" below the table; table and figure numbers are not entered automatically.

If you want a title or heading, use the hash symbol. (One hash is a title, two a heading, three a subheading, etc.) For example, here is a subheading…

1.5.2 Including links

Links to pages and videos on the web can be included as is done above (and below) for the introduction to Markdown. Alternatively, the link can be direct, e.g. `https://www.liverpool.ac.uk`.

Links can also be "reference style". This is where you can find the UK google home page. You can put the link at the end of the paragraph or cell. This makes the Markdown a little easier to read. There is no visible difference in the text that results when you run the cell.

Links can be made to sections or figures in the Notebook. For example, back to the section Section 1.5.1. Double clicking this cell will allow you to see how the section is labelled (using a "#" followed by the section title with the spaces replaced by hyphens, and how a link to that label is created: the label is not allowed to contain any spaces!).

1.5.3 Figures

Images stored on your computer can be added using the syntax used below (double click the cell to see it!). The first example uses the path relative to the folder/directory in which the Notebook is saved:

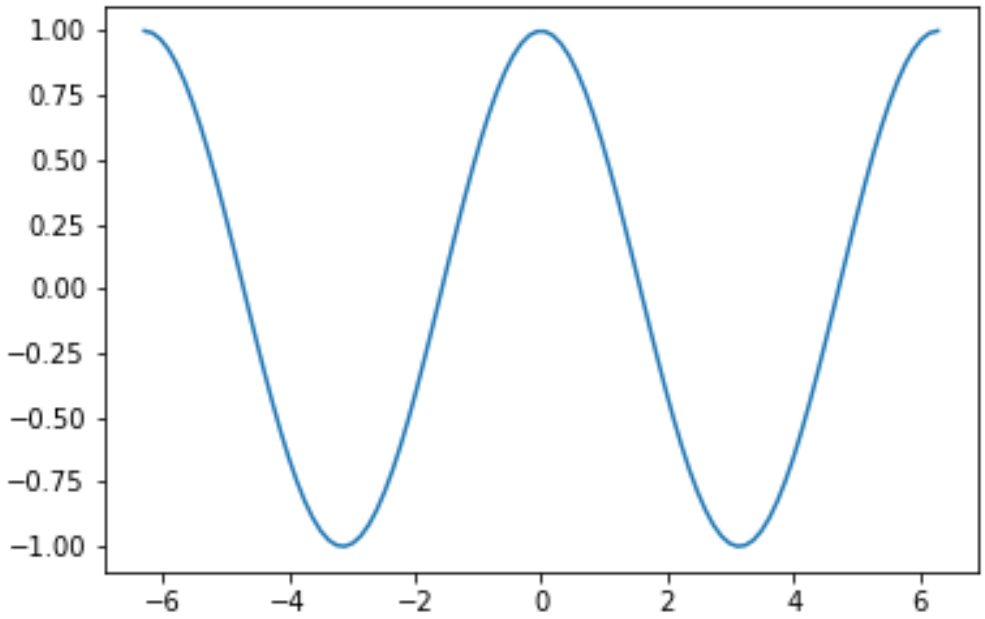


Figure 1 *Cosine as a function of angle*

The absolute path can also be used in Markdown as shown below…though you can't go back further than the Jupyter start folder/directory. These images are not automatically incorporated if you convert your markdown to pdf, so the following section is "commented out". (Anything between the "<", "!" and "–" and the "–" and ">" symbols is treated as a comment, i.e. an explanatory remark, and is not displayed when the Markdown is compiled.)

Figures from the web can be added in the markdown using the syntax shown below.

Again, this has been commented out, as there is no automatic location and conversion of web images to pdf, so the above lines would have spoiled the pdf version of this Notebook.

If you want to create a pdf file of this Notebook including the above images, you will have to download the relevant images to your computer and use a local link!

1.5.4 Entering formulae

Mathematical formulae are entered using LaTeX. How this can be done is best illustrated with a few examples. A formula can be entered "inline" like this: $E = mc^2$. (Notice how the dollar signs "bracket" the mathematical formula.)

Formulae can also be placed on their own line in the centre of the page using two dollar symbols:

$$\sin^2 x + \cos^2 x = 1.$$

Notice how functions like sin and cos (and log, exp etc.) are entered, and how superscripts (x^2) and subscripts (p_0) can be obtained. If you need more than one character in a superscript (or subscript), enclose the relevant section in curly brackets, e.g. x_{max}.

Note, it is important in Markdown to ensure there are no spaces between the "$" and the characters in inline formulae. Although spaces are allowed by the LaTeX standard, they can cause problems when you try and run LaTex on Notebooks, e.g. when using *File, Export Notebook as..., LaTeX*, or *File, Export Notebook as..., PDF*.

Fractions are obtained like this $\frac{1}{2}$, and a vast array of symbols is available, for example: $\sqrt{2}$, $2 \approx 2.5$, $\int_0^1 x^2 dx$, $\sum_{n=0}^{15} x_n$. (See here for more.) Brackets which expand to the required height can be entered using () or []. Greek letters are written α, β etc. An example combining some of these features is the formula for the Fermi function:

$$F(\epsilon) = \frac{1}{\exp\left[\frac{\epsilon-\mu}{kT}\right]+1}.$$

Finally, "inline" segments of computer code can be indicated using reverse quotes, for example: `print("This is a Python 3 print statement")`. Three reverse quotes, with a tag indicating the relevant language, can be used to produce blocks of code. We shall be interested in Python, so we will be seeing things like:

```
import numpy as np
import matplotlib.pyplot as plt
#
print("This illustrates how Markdown can be used to format a block of code")
#
for n in range(0, nMax + 1):
    print("n =",n)
```

Note that the "three reverse quotes with language tag" produces code with colours highlighting the various elements of the language; much clearer than the inline format.

And that's about all we need to know about Markdown!

Of course, the power of Jupyter Notebooks is that they allow documents and figures in Markdown cells like this one to be combined with computer code. We will now exploit this by taking a first look at the Pandas package.

1.6 Pandas

Pandas is a Python library that provides high-performance, easy-to-use data structures and data analysis tools. It can be thought of as an extremely powerful version of Excel, with a lot more features! It should already be installed on your computer as part of the Anaconda package, so in order to use it we just need to import it. We will also import `numpy` and `matplotlib.pyplot`, as we will need these, and use the `%matplotlib inline` "magic" command to ensure our plots appear in the Notebook:

[1]:
```
# <!-- Student -->
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

1.6.1 Reading data into Pandas

The most useful Pandas data structures are known as DataFrames. Data can be read into them from a wide range of formats including from comma-separated-variable (csv) files, Excel spreadsheets (xls or xlsx) and the web (html). Here, we will read in an Excel (xlsx) file. Let's first have a look at the file using Excel:

[2]:
```
#<!-- Student -->
!excel Spectrum01.xlsx
```

That looks OK. Let's read the file into a DataFrame called `df` in our Notebook:

[3]:
```
#<!-- Student -->
df = pd.read_excel("Spectrum01.xlsx")
df
```

[3]:

	Unnamed: 0	xData	xError	yData	yError
0	0	0.000000	0.048113	1.213715	1.101687
1	1	0.172414	0.048113	1.068080	1.033479
2	2	0.344828	0.048113	5.352066	2.313453
3	3	0.517241	0.048113	4.571977	2.138218
4	4	0.689655	0.048113	5.042432	2.245536
5	5	0.862069	0.048113	2.028153	1.424132
6	6	1.034483	0.048113	4.452245	2.110034
7	7	1.206897	0.048113	2.209224	1.486346
8	8	1.379310	0.048113	9.481829	3.079258
9	9	1.551724	0.048113	8.464667	2.909410
10	10	1.724138	0.048113	16.065005	4.008117
11	11	1.896552	0.048113	46.473266	6.817130
12	12	2.068966	0.048113	100.883753	10.044090
13	13	2.241379	0.048113	114.477995	10.699439
14	14	2.413793	0.048113	90.468280	9.511481
15	15	2.586207	0.048113	36.077287	6.006437
16	16	2.758621	0.048113	14.663739	3.829326
17	17	2.931034	0.048113	12.677617	3.560564
18	18	3.103448	0.048113	13.462261	3.669095
19	19	3.275862	0.048113	20.749554	4.555168
20	20	3.448276	0.048113	21.947589	4.684825
21	21	3.620690	0.048113	17.906467	4.231603
22	22	3.793103	0.048113	18.526115	4.304197
23	23	3.965517	0.048113	23.308507	4.827888
24	24	4.137931	0.048113	9.974797	3.158290
25	25	4.310345	0.048113	17.549180	4.189174
26	26	4.482759	0.048113	25.830562	5.082378
27	27	4.655172	0.048113	29.505515	5.431898
28	28	4.827586	0.048113	19.995142	4.471593
29	29	5.000000	0.048113	25.943390	5.093465

We have the data we want, but also an extra column full of row indices, and a column header

Unnamed:0 that we don't want. We will see next week how we can delete (and add) columns and rows in DataFrames. For now, we modify the way we read in the data so we don't get the unwanted material in the first place!

[4]:
```
#<!-- Student -->
df = pd.read_excel("Spectrum01.xlsx", index_col = 0)
df
```

[4]:

	xData	xError	yData	yError
0	0.000000	0.048113	1.213715	1.101687
1	0.172414	0.048113	1.068080	1.033479
2	0.344828	0.048113	5.352066	2.313453
3	0.517241	0.048113	4.571977	2.138218
4	0.689655	0.048113	5.042432	2.245536
5	0.862069	0.048113	2.028153	1.424132
6	1.034483	0.048113	4.452245	2.110034
7	1.206897	0.048113	2.209224	1.486346
8	1.379310	0.048113	9.481829	3.079258
9	1.551724	0.048113	8.464667	2.909410
10	1.724138	0.048113	16.065005	4.008117
11	1.896552	0.048113	46.473266	6.817130
12	2.068966	0.048113	100.883753	10.044090
13	2.241379	0.048113	114.477995	10.699439
14	2.413793	0.048113	90.468280	9.511481
15	2.586207	0.048113	36.077287	6.006437
16	2.758621	0.048113	14.663739	3.829326
17	2.931034	0.048113	12.677617	3.560564
18	3.103448	0.048113	13.462261	3.669095
19	3.275862	0.048113	20.749554	4.555168
20	3.448276	0.048113	21.947589	4.684825
21	3.620690	0.048113	17.906467	4.231603
22	3.793103	0.048113	18.526115	4.304197
23	3.965517	0.048113	23.308507	4.827888
24	4.137931	0.048113	9.974797	3.158290
25	4.310345	0.048113	17.549180	4.189174
26	4.482759	0.048113	25.830562	5.082378
27	4.655172	0.048113	29.505515	5.431898
28	4.827586	0.048113	19.995142	4.471593
29	5.000000	0.048113	25.943390	5.093465

That looks better!

1.6.2 Week 1 exercise 1

Use Excel to create a spreadsheet called TimeTable.xlsx with columns labelled with the days of the week and two rows, one labelled "morning" and the other "afternoon". Enter a one in each of the cells in which you have a lecture, problems class or laboratory and a zero where you are free. Read the spreadsheet into a DataFrame called dfTimeTable in the cell below this one and then display

it.

[5]:
```
#<!-- Student -->
```

1.7 Data analysis

Last year, we used Python to visualise data in various types of plots using `matplotlib.pyplot` and to fit data using the `least_squares` routine from the SciPy library. We will be doing this again in the year two laboratories (Phys206), so we will do some quick revision on plotting and fitting here.

1.7.1 Plotting with matplotlib

Let's first plot the data from Spectrum01.xslx we have read into our DataFrame `df`. We will see next week how we can plot directly from Pandas. Here, we will extract the data from the DataFrame so we can use the plotting procedures we learnt about last year. Let's look at one of the columns from the DataFrame:

[6]:
```
#<!-- Student -->
df.xData
```

[6]:
```
0     0.000000
1     0.172414
2     0.344828
3     0.517241
4     0.689655
5     0.862069
6     1.034483
7     1.206897
8     1.379310
9     1.551724
10    1.724138
11    1.896552
12    2.068966
13    2.241379
14    2.413793
15    2.586207
16    2.758621
17    2.931034
18    3.103448
19    3.275862
20    3.448276
21    3.620690
22    3.793103
23    3.965517
24    4.137931
25    4.310345
26    4.482759
27    4.655172
```

```
28    4.827586
29    5.000000
Name: xData, dtype: float64
```

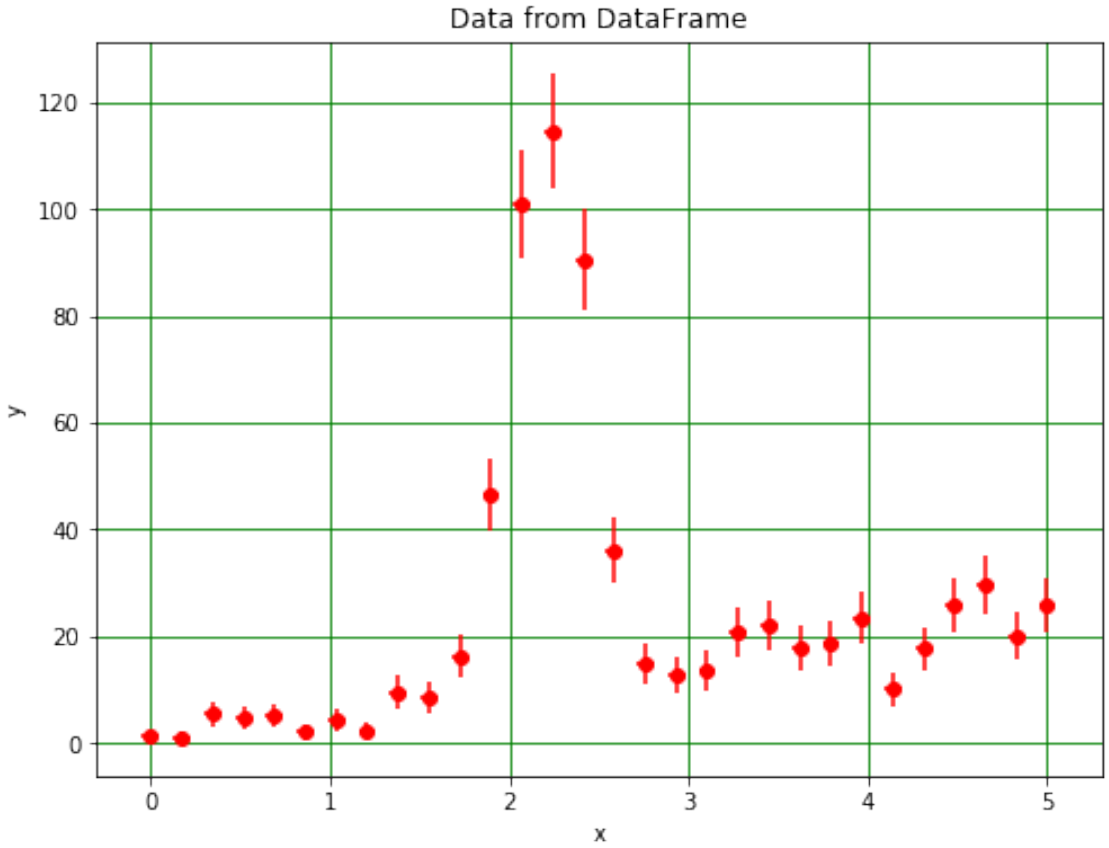
We see that we get the data we want, but also the indices which we don't. If we turn the column into a numpy array, we get the format we need for making a plot:

[7]:
```
#<!-- Student -->
xData = np.array(df.xData)
xError = np.array(df.xError)
yData = np.array(df.yData)
yError = np.array(df.yError)
print(yData)
```

```
[  1.21371531   1.06807951   5.35206586   4.5719766    5.04243153
   2.02815262   4.45224528   2.20922434   9.48182923   8.46466692
  16.06500484  46.4732664  100.88375314 114.47799496  90.46827961
  36.07728691  14.66373865  12.67761662  13.46226134  20.74955384
  21.94758904  17.90646675  18.52611518  23.30850696   9.97479697
  17.54918024  25.830562    29.50551547  19.99514177  25.94339013]
```

Now we can create a figure, define the axes we need and make a plot of the data.

[8]:
```
#<!-- Student -->
#
plt.figure(figsize = (8, 6))
plt.errorbar(xData, yData, xerr = xError, yerr = yError, color = 'r', marker =␣
↪'o', linestyle = '')
plt.title('Data from DataFrame')
plt.xlabel('x')
plt.ylabel('y')
plt.grid(color = 'g')
plt.savefig('Spect01.png')
plt.show()
```



1.7.2 Fitting with least_squares

The next step is to fit this data. From the plot, we can see that it looks like a peak on top of some background. The data could perhaps be the measurements of the photon energy spectrum produced when a radiaoctive material decays. We will assume the peak has a gaussian shape and that the background can be described by a second order polynomial. The formula for a gaussian function is:

$$G(x) = \frac{N}{\sqrt{2\pi}\sigma} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right].$$

Here, N describes the number of events in the gauussian peak, μ is the central (mean) value of the peak and σ its standard deviation.

The second order (background) polynomial can be written:

$$P(x) = a + bx + cx^2.$$

We want to find the values of the parameters N, μ, σ, a, b and c which best describe the data. That is, we want to choose these values so that the function $F(x) = G(x) + P(x)$ is as close to all

of the measured points as possible. We could do this by finding the parameters which minimise the distance between the measured points $(x_i \pm \varepsilon_{x,i}, y_i \pm \varepsilon_{y,i})$ and the value of the function at those points $(x_i, F(x_i))$. That is, choose N, μ, σ, a, b and c so that $\sum |y_i - F(x_i)|$ is as small as possible.

1.7.3 Week 1 exercise 2

Explain why we have to minimise $\sum |y_i - F(x_i)|$ rather than $\sum y_i - F(x_i)$.

1.7.4 Week 1 exercise 3

In which of the Python functions below: 1. Is $\frac{\partial F}{\partial x}$ calculated? 2. Is $\chi = \frac{y - F(x)}{e}$ calculated?

[9]:
```
# <!-- Student -->
from scipy.optimize import least_squares
#
def fitFunc(p, x):
    '''
    Gaussian plus polynomial (order 2)
    '''
    f = p[0]/(np.sqrt(2*np.pi)*p[2])*np.exp(-(x - p[1])**2/(2*p[2]**2)) + p[3]␣
 ↪+ p[4]*x + p[5]*x**2
    return f
#
def fitFuncDiff(p, x):
    '''
    Differential of fit function
    '''
    df = p[0]/(np.sqrt(2*np.pi)*p[2])*(x - p[1])/p[2]**2*np.exp(-(x - p[1])**2/
 ↪(2*p[2]**2)) + p[4] + 2*p[5]*x
    return df
#
def fitChi(p, x, y, xerr, yerr):
    '''
    Calculation of chi for fit function and data
    '''
    chi = (y - fitFunc(p, x))/(np.sqrt(yerr**2 + fitFuncDiff(p, x)**2*xerr**2))
    return chi
```

We provide initial values of the `nParams` parameters in `p` then run the fit. The better our initial guesses are, the more likely the fit is to find the correct values. We can use the plot we have made to help us pick sensible values. E.g. the mean of the gaussion `mu` can be seen to be about 2.5. We can also provide upper and lower bounds for the values of `p` usung the arrays `lBounds` and `uBounds`. The values used here ($\pm\infty$) clearly have no effect! We do not have to give bounds; if they are not needed, just omit the argument `bounds` from the call to `least_squares`.

[10]:
```
# <!-- Student -->
#
# Set initial values of fit parameters
```

```
nParams = 6
pInit =  [50.0, 2.0,  0.2,  1.0,  1.0,  1.0]
lBounds = [-np.inf, -np.inf, -np.inf, -np.inf, -np.inf, -np.inf]
uBounds = [np.inf,  np.inf,  np.inf,  np.inf,  np.inf,  np.inf]
out = least_squares(fitChi, pInit, args = (xData, yData, xError, yError),␣
 ↪bounds = (lBounds, uBounds))
#
```

The values returned by `least_squares` are all contained in the object `out`. The first one we look at we label `fitOK`. This is a boolean variable that tells us if the fit has run. We also transfer the fitted parameters into the array `pFinal`. This contains the fitted values of the parameters N, μ, σ, a, b and c in the same order that we used to `p`.

We also calculate and print out the values of the statistics χ^2 and χ^2/NDF which describe how well the fitted function matches the data. We calculate the errors on the fitted parameters by inverting the squared Jacobian matrix to get the covariance matrix `covar`. The diagonal components of `covar` are the squared errors of the fitted parameters. E.g. the error on N is `covar[0, 0]`, the error on μ is `covar[1, 1]`, the error on σ is `covar[2, 2]` etc.

Finally, we plot the data again and add a line describing the fitted function.

[11]:
```
# <!-- Student -->
#
fitOK = out.success
#
# Test if fit failed
if not fitOK:
    print(" ")
    print("Fit failed")
else:
    #
    # Fit worked, so get output
    pFinal = out.x
    N = pFinal[0]
    mu = pFinal[1]
    sigma = pFinal[2]
    a = pFinal[3]
    b = pFinal[4]
    c = pFinal[5]
    #
    #   Calculate chis**2 per point, summed chi**2 and chi**2/NDF
    chisqArr = fitChi(pFinal, xData, yData, xError, yError)**2
    chisq = np.sum(chisqArr)
    nPoints = len(xData)
    NDF = nPoints - nParams
    redchisq = chisq/NDF
#
    np.set_printoptions(precision = 3)
```

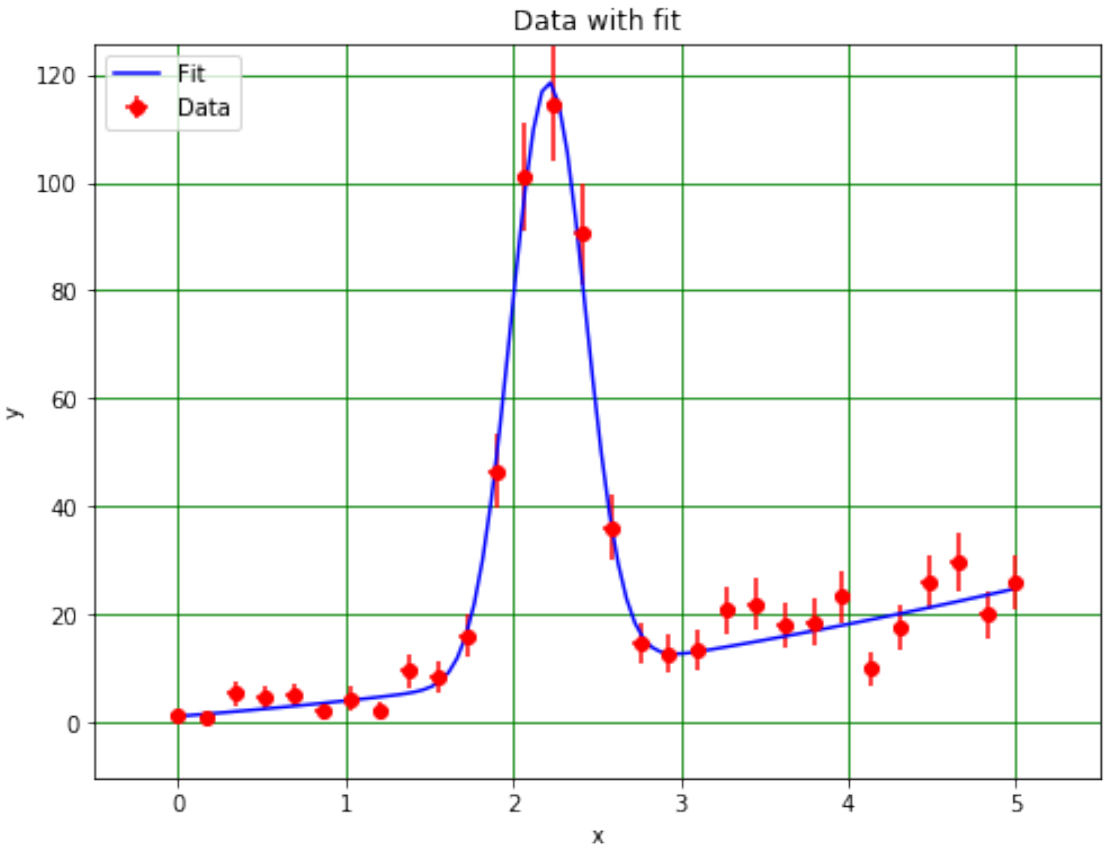
```
    print(" ")
    print("Fit quality:")
    print("chisq per point = \n",chisqArr)
    print("chisq = {:5.2f}, chisq/NDF = {:5.2f}.".format(chisq, redchisq))
    #
    # Compute covariance
    jMat = out.jac
    jMat2 = np.dot(jMat.T, jMat)
    detJmat2 = np.linalg.det(jMat2)
    #
    # Check whether cavariance matrix can be calculated
    if detJmat2 < 1E-32:
        #
        # It can't, print out values of parameters without errors
        print("Value of determinat detJmat2",detJmat2)
        print("Matrix singular, error calculation failed.")
        print(" ")
        print("Parameters returned by fit:")
        print("N = {:5.2f}".format(N))
        print("mu = {:5.2f}".format(mu))
        print("sigma = {:5.2f}".format(sigma))
        print("a = {:5.2f}".format(a))
        print("b = {:5.2f}".format(b))
        print("c = {:5.2f}".format(c))
        print(" ")
        errN = 0.0
        errmu = 0.0
        errsigma = 0.0
        erra = 0.0
        errb = 0.0
        errc = 0.0
    else:
        #
        # Covariance matrix is OK, calculate errors
        covar = np.linalg.inv(jMat2)
        errN = np.sqrt(covar[0, 0])
        errmu = np.sqrt(covar[1, 1])
        errsigma = np.sqrt(covar[2, 2])
        erra = np.sqrt(covar[3, 3])
        errb = np.sqrt(covar[4, 4])
        errc = np.sqrt(covar[5, 5])
        #
        print(" ")
        print("Parameters returned by fit:")
        print("N = {:5.2f} +- {:5.2f}".format(N, errN))
        print("mu = {:5.2f} +- {:5.2f}".format(mu, errmu))
        print("sigma = {:5.2f} +- {:5.2f}".format(sigma, errsigma))
```

```
        print("a = {:5.2f} +- {:5.2f}".format(a, erra))
        print("b = {:5.2f} +- {:5.2f}".format(b, errb))
        print("c = {:5.2f} +- {:5.2f}".format(c, errc))
        print(" ")
    #
    # Calculate fitted function values (using lots of points so we get a nice␣
↪smooth curve!)
    nPlot = 100
    xPlot = np.linspace(np.min(xData), np.max(xData), nPlot)
    fitPlot = fitFunc(pFinal, xPlot)
    #
    # Plot data
    fig = plt.figure(figsize = (8, 6))
    plt.title('Data with fit')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.errorbar(xData, yData, xerr = xError, yerr = yError, color = 'r',␣
↪marker = 'o', linestyle = '', label = "Data")
    plt.plot(xPlot, fitPlot, color = 'b', marker = '', linestyle = '-', label =␣
↪"Fit")
    edge = 0.1
    xLow = np.min(xData) - edge*np.abs(np.max(xData) - np.min(xData))
    xHigh = np.max(xData) + edge*np.abs(np.max(xData) - np.min(xData))
    plt.xlim(xLow, xHigh)
    yLow = np.min(yData) - edge*np.abs(np.max(yData) - np.min(yData))
    yHigh = np.max(yData) + edge*np.abs(np.max(yData) - np.min(yData))
    plt.ylim(yLow, yHigh)
    plt.grid(color = 'g')
    plt.legend(loc = 2)
    plt.show()
```

```
Fit quality:
chisq per point =
 [3.107e-03 2.418e-01 2.049e+00 9.183e-01 8.000e-01 1.156e+00 2.251e-02
 2.847e+00 1.712e+00 1.372e-01 1.164e-02 1.216e-02 3.339e-02 7.673e-02
 2.610e-01 1.253e-03 1.370e-01 3.822e-04 8.455e-03 2.176e+00 2.210e+00
 2.121e-01 1.314e-01 1.205e+00 8.209e+00 3.855e-01 7.995e-01 1.693e+00
 6.467e-01 4.979e-02]
chisq = 28.15, chisq/NDF =  1.17.

Parameters returned by fit:
N = 61.11 +-  5.68
mu =  2.21 +-  0.03
sigma =  0.22 +-  0.02
a =  1.15 +-  0.77
b =  2.40 +-  1.32
```

```
c =  0.47 +-  0.30
```



1.7.5 Week 1 exercise 4

Read in the spreadsheet Spectrum02.xlsx. Plot the data. Try to fit it using a gaussian to describe the peak and a straight line to represent the background.

That's the end of Week 1 for Phys205. You will have a chance to practice using the plotting and fitting routines we have revised here in Practical Physics II (Phys 206)!