Phys105-Week01

September 25, 2019

1 Introduction to Computational Physics - Week 1

1.1 Table of contents week 1

1.2 Introduction to week 1

The aim of the Introduction to Computational Physics (Phys105) module is to provide a practical introduction to programming in Python for Physics students and students on related courses. It takes a "bottom-up" approach: from day one we use examples to develop an understanding of how Python programs can be used to tackle physics problems, rather than first learning a lot of general principles and then trying to apply them. There are two reasons for this. Firstly, you will be using Python in your practical and some other modules soon, so need to be in a position to do some

things quite quickly. Secondly, most people learn more effectively by actually doing things than by just hearing how they can be done.

This first Notebook provides some of the basic information you need for the Phys105 module. It shows you how to install the tools we will be using in the course (the Python programming language and Jupyter Notebooks) and provides a few exercises that will help you start to learn how to tackle problems using them. When you have finished working through it, you should understand the basic structure of a Notebook, be able to write Markdown that allows you to present text, figures, formulae and tables attractively in Notebooks and be able to use the program `least_squares` in a Notebook to fit a straight line to a set of data points.

Please take the time to read the information provided here, don't just skip to the exercises. If you have read through the Notebook and still have difficulties with the exercises, or don't understand something, ask one of the demonstrators for help. Further information and explanation is provided in the recommended textbooks (A student's guide to Python for physical modelling, or Learning scientific programming with Python). There is also lots of information on Python and Jupyter Notebooks on the web, so you can ask Google for help!

When you have finished the exercises, ask one of the demonstrators to mark your work.

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 Phys105 practical 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 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 press *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*.

This "Phys105-Week01-Student.ipynb" file can be opened within Jupyter, which will allow you to work with the code. **It is strongly advised you do this** as opposed to just reading the PDF file!

1.4 Jupyter Notebooks

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 programming language in the code cells. We will start off by learning how to create Markdown documents, but learning Python will be the main aim of this course.

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*.

1.4.1 Week 1 exercise 1

Look through the menus in the Notebook screen. Use them to create a new cell below this one. Convert the type of the cell to Markdown, write something in the cell, then delete it. Repeat this exercise using the keyboard shortcuts described above. If you have problems doing this, ask a demonstrator for help!

1.4.2 Week 1 exercise 1 answer 1

Students should make a new cell, write something in it, and then delete it!

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. (Do this now!)

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* key and hold it down, then press the *Enter* key, then release both keys). Use *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, respectively.) You can also ~~eross 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
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 the table will be formatted for you, 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 on the web can be included as is done 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 below (double click on the cell to see it!). The first example uses the path relative to the folder/directory in which the Notebook is saved:

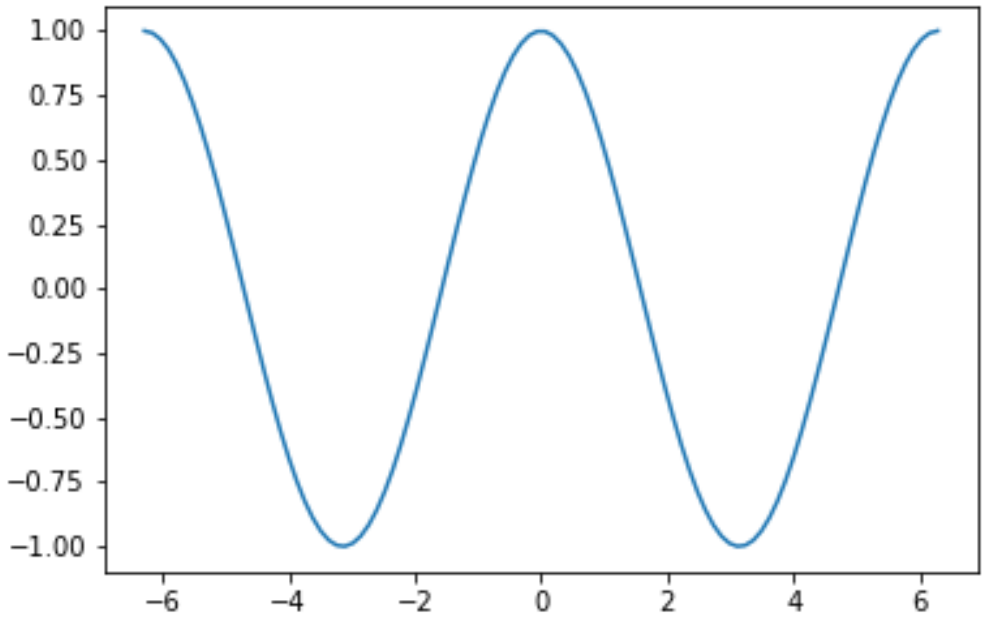


Figure 1 *Cosine as a function of angle*

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 "remote" 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.) Remove the comment marks if you want to see what this bit of Markdown produces.

Figures from the web can be added in the markdown using the syntax shown below. (Again, remove the comment marks if you want to compile this Markdown.)

This section is 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 files 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. It is important in Markdown to ensure there are no spaces between the dollar signs and the characters in the formula. 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*.

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

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

Alternatively, you can use the following syntax, which also allows you to "line up" successive equations:

$$\begin{aligned}
\int_0^\pi \sin x dx &= (-\cos x)_0^\pi && (1)\\
&= -\left(\cos(\pi) - cos(0)\right) && (2)\\
&= -(-1-1) && (3)\\
&= 2. && (4)
\end{aligned}$$

The ampersand (&) tells LaTeX where to align the equations and the double backslash tells it where to break the lines. This equation also illustrates the use of the commands (and) to make brackets of the appropriate size (i.e. that expand as they are nested). (The command backslash text changes the font in the region delimited by the curly brackets.)

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}.

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.) The backslash followed by a comma in the integral produces a "slim space" between the x^2 and the dx, a full space would be obtained using a backslash with a space after it. As already mentioned, brackets which expand to the required height can be entered using () or []. Greek letters are written α, β etc. Another example combining some of these features is the formula for the Fermi function:

$$F(\epsilon) = \frac{1}{\exp\left[\frac{\epsilon-\mu}{kT}\right]+1}. \tag{5}$$

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!

1.5.5 Week 1 exercise 2

In the cell below, write a short document with a heading, a line of text, a shopping list with five numbered items, a table with two columns (one of which contains the numbers 1 to 7 and the other the corresponding days of the week), the formula for a gaussian distribution, a link to the Liverpool University web site and a picture of your choice!

1.5.6 Week 1 exercise 2 answer

The students should produce something like:

1.6 This is the heading

Here are a few words of text.

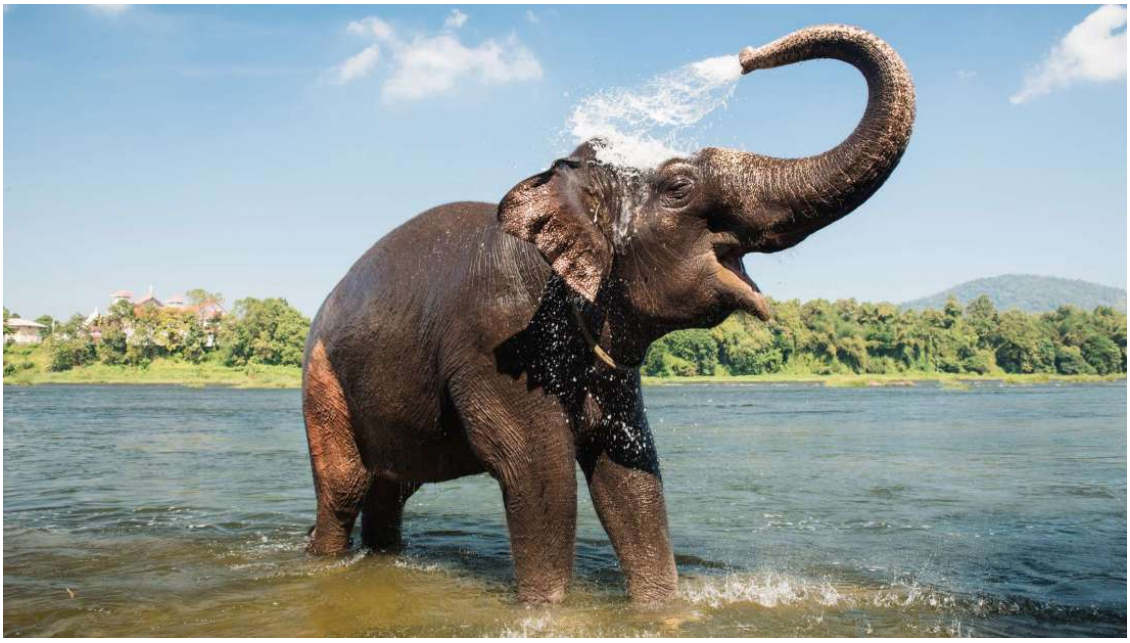
I need to buy:

1. Cheese
2. Butter
3. Bread
4. Apples

Here is a table of the days of the week:

Number	Day
1	Monday
2	Tuesday
3	Wednesday
4	Thursday
5	Friday
6	Saturday
7	Sunday

Here is a picture!



And here is a link to the Liverpool University web site.

1.7 Code cells

Markdown allows us to write attractive documents quickly and efficiently, but the power of Jupyter Notebooks is that they allow these documents to be combined with computer code. We can describe a problem and work on the programs needed to solve it in the same place. Below, we will look at an example of how this can be of use in the context of laboratory measurements, and at the same time see our first Python program. We will fit a straight line to some experimental data using a routine called `least_squares`.

A word of warning. Unless you have done some programming before, you are unlikely to understand the nuts and bolts of all of the code below; we will learn that in later weeks. For now, it is important that you understand how to get data into `least_squares`, run the program and extract the results, as you will need to do this in Practical Physics I (Phys106).

1.7.1 An imaginary experiment

Ten measurements have been made of two quantities, x and y. These are thought to be linearly related, i.e. to lie on a straight line, $y = mx + c$. The x and y values (with their measurement errors) are:

Measurement	x value	Error in x	y value	Error in y
1	1.50	0.21	14.3	2.1
2	2.31	0.11	20.2	1.7
3	2.78	0.43	30.1	3.3
4	3.91	0.10	41.5	1.1
5	3.88	0.07	42.7	0.9
6	4.76	0.08	47.1	0.8
7	5.62	0.05	52.9	0.5

Measurement	x value	Error in x	y value	Error in y
8	7.02	0.09	68.8	0.7
9	8.45	0.17	85.2	1.2
10	9.65	0.11	99.4	2.9

Table 2 *A number of data points which are expected to lie on a straight line.*

We will test whether the expected relationship holds by fitting a straight line to the data and simultaneously determine the values of m and c that best describe the data.

The program in the cells below shows how this can be done. It is broken down into sections interspersed with an explanation of what each section is doing.

1.7.2 Fitting a straight line to data

The first step is to import (load) the Python modules used by the program. Here, we need: * NumPy (Numerical Python), a library of routines for defining arrays of numbers and working with them. * Matplotlib.pyplot, a set of plotting routines for making graphs of various types. * least_squares from scipy.optimize (part of the Scientific Python library), the fitting routine we shall use.

We also use the "line magic" command

`%matplotlib inline`

to ensure that any plots we make appear in the Notebook and don't just get stored as files on the computer.

The following code cell imports the routines we will need. (Note that, in Python, comments are indicated by a "#": anything after the hash symbol is not part of the program but is a "comment" providing information on what the program is doing.) To actually do the importing, you must run the cell. As for Markdown cells, you do this by selecting it and using the menu, or selecting the cell and pressing *Ctrl + Enter*. Alternatively, *Shift + Enter* will run the cell and move to the next one or create a new one if there isn't already a cell after the current one.

[6]:
```
# <!-- Student -->
import numpy as np
import matplotlib.pyplot as plt
#
# least_squares is a fitting routine
from scipy.optimize import least_squares
#
# This "line magic" ensures plots are displayed in the Notebook
%matplotlib inline
```

Because we have used

`import numpy as np`

to load everything from the NumPy library, if we want to use a NumPy routine (e.g. `sqrt(x)`, which calculates the square root of x) we have to use the statement

`np.sqrt(x)`

not just `sqrt(x)`. The same is true for plots. Because we `imported` `matplotlib.pyplt` as `plt`, if we want to make a plot with errorbars using the relevant `matplotlib.pyplot` routine (`errorbar`), we have to write `plt.errorbar`.

We are only going to use one routine (`least_squares`) from `scipy.optimize`, so we only import that. And because we haven't imported it "as" anything, we can use its name directly.

Instead of `import numpy as np`, we could have used `import numpy` to load all the `numpy` routines, but then we would have to write `numpy.sqrt(x)` to calculate $\sqrt{x}$, not the shorter `np.sqrt(x)`. We could also have written `from numpy import *`, which would have imported all the `numpy` routines and made them directly accessible, so just using `sqrt(x)` would return $\sqrt{x}$. The problem with this is that Python offers different routines for calculating square roots, for example there is another version in the `math` module. If we don't distinguish between the two (by using `import numpy as np` and then `np.sqrt` and `import math as ma` and `ma.sqrt`), code can produce undesired results. In this case, the distinction is important as `np.sqrt` works for complex numbers and `ma.sqrt` doesn't!

The next step, in the code cell below, is to enter the data for the fit and print them out for checking. We do this by defining `numpy` arrays for the x and y data and their errors, and setting these equal to the relevant numbers from the table of data above. (Note that Python counts array elements from 0, not 1!) Remember, the lines that are preceded by a "#" are *comment* lines used to explain what the code is doing; they don't influence the computations carried out when the cell is run!

1.7.3 Week 1 exercise 3

Run the code cell below and check the data that it prints out against the numbers in Table 2. Look for the mistakes and fix them by editing the Python in the cell. Re-run and re-check the cell until all the data are correctly entered.

[7]:
```
# <!-- Student -->
# nPoints is the number of data points in the fit
nPoints = 10
#
# Define NumPy arrays, initially filled with zeros, to store the x and y data␣
 ↪values
xData = np.zeros(nPoints)
yData = np.zeros(nPoints)
#
# Define arrays to store the errors in x and y
xError = np.zeros(nPoints)
yError = np.zeros(nPoints)
#
# Enter the x data points.
xData[0] = 1.50
xData[1] = 2.31
xData[2] = 2.78
xData[3] = 3.58
xData[4] = 4.08
```

```
xData[5] = 4.76
xData[6] = 5.62
xData[7] = 7.02
xData[8] = 8.45
xData[9] = 9.65
#
# Enter the y data points.
yData[0] = 14.3
yData[1] = 20.2
yData[2] = 30.1
yData[3] = 36.5
yData[4] = 42.7
yData[5] = 47.1
yData[6] = 52.9
yData[7] = 68.8
yData[8] = 85.2
yData[9] = 99.4
#
# Enter the errors in the x values
xError[0] = 0.21
xError[1] = 0.11
xError[2] = 0.43
xError[3] = 0.13
xError[4] = 0.17
xError[5] = 0.18
xError[6] = 0.15
xError[7] = 0.19
xError[8] = 0.17
xError[9] = 0.11
#
# Enter the errors in the y values
yError[0] = 2.1
yError[1] = 1.7
yError[2] = 3.3
yError[3] = 1.1
yError[4] = 0.9
yError[5] = 1.1
yError[6] = 1.5
yError[7] = 0.9
yError[8] = 1.2
yError[9] = 2.9
#
# The code below prints out the data we have entered. We will explain how it␣
 ↪works
# later in the course
print(" ")
print("Check the data points:")
```

```
print("Point\t x\t\t\ty\t")
for n in range(0, nPoints):
    print("{:d}\t{:5.2f}\t+-{:5.2f}\t\t\t{:5.2f}\t+-{:5.2f}".\
          format(n, xData[n], xError[n], yData[n], yError[n]))
```

```
Check the data points:
Point    x                    y
0        1.50   +- 0.21       14.30   +- 2.10
1        2.31   +- 0.11       20.20   +- 1.70
2        2.78   +- 0.43       30.10   +- 3.30
3        3.58   +- 0.13       36.50   +- 1.10
4        4.08   +- 0.17       42.70   +- 0.90
5        4.76   +- 0.18       47.10   +- 1.10
6        5.62   +- 0.15       52.90   +- 1.50
7        7.02   +- 0.19       68.80   +- 0.90
8        8.45   +- 0.17       85.20   +- 1.20
9        9.65   +- 0.11       99.40   +- 2.90
```

1.7.4 Week 1 exercise 3 answer

Check that the students have corrected xData[3..4], yData[3], xError[3..9], yError[5..7].

The routine we use for fitting is optimize.least_squares from `scipy.optimize`. This requires us to define the function that we want to fit to the data (called `fitLine` here) and a function (`fitChi`), which describes how differences between the data and the fit are treated. (This function is referred to as a *cost* or *merit function* in the SciPy documentation.) The function `fitChi` depends on the derivative of `fitline`, `fitLineDiff`. You should be able to work out why this is the case when you have done some error analysis in Maths for Physics I (Phys107) or from the information on errors given in the Phys106 handbook!

Note that three quotes before and after a section of text are another way of writing comments in Python - the text between the quotes does not have any effect when the code is run, it serves only to explain what the code does.

The definition of `fitLine` tells us that it accepts two arguments, `p` and `x` (just as though it were a mathematical function fitLine(p, x)). In this case, `p` is a two-component vector, or `array`. The first component, `p[0]`, is the intercept of the straight line, the second component, `p[1]`, is its gradient. That is, `p[0]`$= c$ and `p[1]`$= m$ in the expression $y = mx + c$. Notice again that Python counts array elements from 0, not 1!

[8]:
```
# <!-- Student -->
def fitLine(p, x):
    '''
    Straight line
    '''
    f = p[0] + p[1]*x
    return f
#
```

```
def fitLineDiff(p, x):
    '''
    Differential of straight line
    '''
    df = p[1]
    return df
#
def fitChi(p, x, y, xerr, yerr):
    '''
    Cost function
    '''
    e = (y - fitLine(p, x))/(np.sqrt(yerr**2 + fitLineDiff(p, x)**2*xerr**2))
    return e
```

Here, we set our first guesses for the values of the intercept and the gradient, run the fit, and transfer the information on whether the fit has worked to the variable `fitOK`.

[9]:
```
# <!-- Student -->
#
# Set initial values of fit parameters, run fit
pInit = [1.0, 1.0]
out = least_squares(fitChi, pInit, args=(xData, yData, xError, yError))
#
fitOK = out.success
```

Now we check if the fit has worked, and if it has, transfer the fitted parameters into the array `pFinal`. This contains the fitted values of the intercept (`pFinal[0]`) and the gradient (`pFinal[1]`). We also calculate and print out the values of the statistics χ^2 and χ^2/NDF which describe how well the fitted straight line 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 on the fitted intercept (`covar[0, 0]`) and gradient (`covar[1, 1]`). Finally, we plot the data with error bars (using the routine `plt.errorbar` from `matplotlib.pyplot`) and draw the fitted line on the plot (using `plt.plot`).

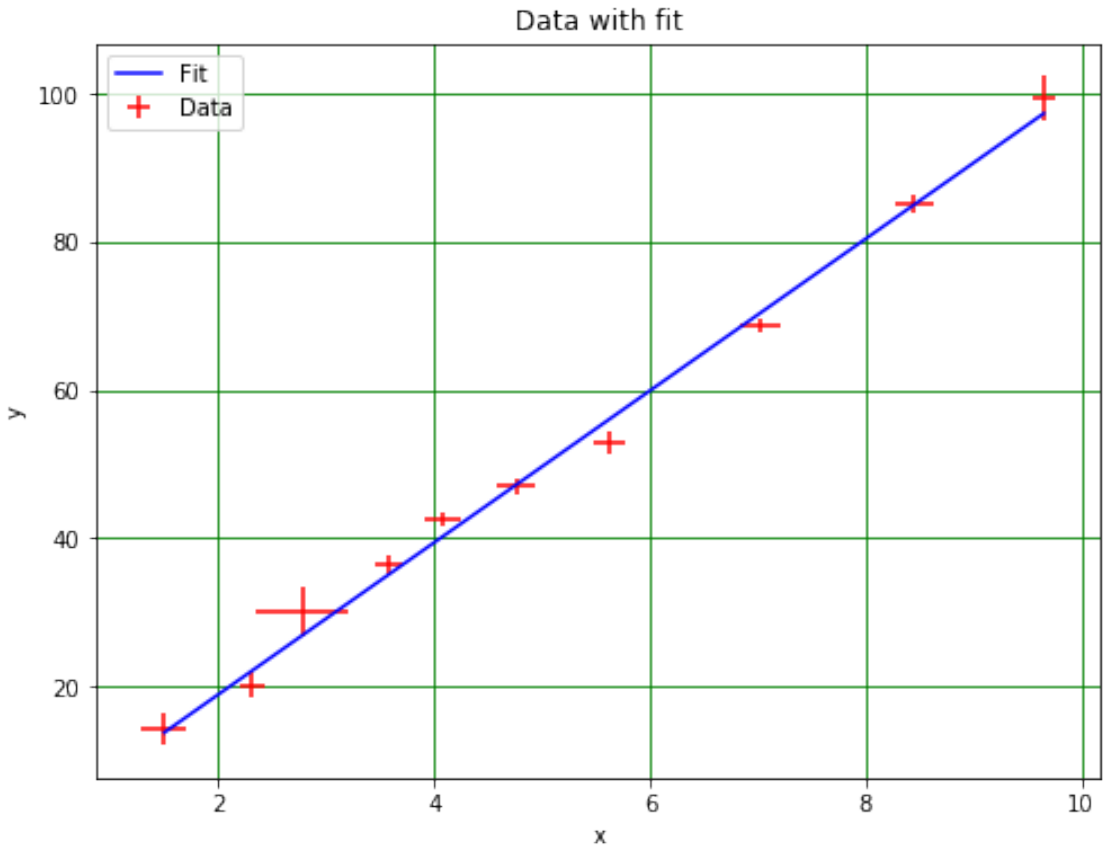
[10]:
```
# <!-- Student -->
#
# Test if fit failed
if not fitOK:
    print(" ")
    print("Fit failed")
else:
    #
    # get output
    pFinal = out.x
    cVal = pFinal[0]
    mVal = pFinal[1]
    #
    #   Calculate chis**2 per point, summed chi**2 and chi**2/NDF
```

```
    chisqArr = fitChi(pFinal, xData, yData, xError, yError)**2
    chisq = np.sum(chisqArr)
    NDF = nPoints - 2
    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)
    #
    if detJmat2 < 1E-32:
        print("Value of determinat detJmat2",detJmat2)
        print("Matrix singular, error calculation failed.")
        print(" ")
        print("Parameters returned by fit:")
        print("Intercept = {:5.2f}".format(cVal))
        print("Gradient = {:5.2f}".format(mVal))
        print(" ")
        cErr = 0.0
        mErr = 0.0
    else:
        covar = np.linalg.inv(jMat2)
        cErr = np.sqrt(covar[0, 0])
        mErr = np.sqrt(covar[1, 1])
        #
        print(" ")
        print("Parameters returned by fit:")
        print("Intercept = {:5.2f} +- {:5.2f}".format(cVal, cErr))
        print("Gradient = {:5.2f} +- {:5.2f}".format(mVal, mErr))
        print(" ")
    #
    # Calculate fitted function values
    fitData = fitLine(pFinal, xData)
    #
    # 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, fmt='r', \
                 linestyle = '', label = "Data")
```

```
    plt.plot(xData, fitData, color = 'b', linestyle = '-', label = "Fit")
    #plt.xlim(1.0, 7.0)
    #plt.ylim(0.0, 3.0)
    plt.grid(color = 'g')
    plt.legend(loc = 2)
    plt.show()
```

```
Fit quality:
chisq per point =
 [0.025 0.891 0.33  0.626 1.556 0.003 2.123 0.535 0.008 0.453]
chisq =  6.55, chisq/NDF =  0.82.

Parameters returned by fit:
Intercept = -1.54 +-  1.74
Gradient = 10.24 +-  0.32
```



1.7.5 Results

The data look to be reasonably well described by a straight line (provided you have found and fixed the mistakes in the input!). The value of the statistic χ^2/NDF backs this up. (Values in the range $0.25 < \chi^2$/NDF \$ < 4\$ represent good agreement, the closer to one the better.) The gradient of the straight line and its intercept can be read off from the output of `least_squares`.

1.7.6 Week 1 exercise 4

Fit a straight line to the data in Table 3 (which is the same as Table 2 with two new points added). Do this by copying the relevant cells above and pasting them below this cell (use the *Edit* menu). Make the changes you need to add the new data points.

Measurement	x value	Error in x	y value	Error in y
1	1.50	0.21	14.3	2.1
2	2.31	0.11	20.2	1.7
3	2.78	0.43	30.1	3.3
4	3.91	0.10	41.5	1.1
5	3.88	0.07	42.7	0.9
6	4.76	0.08	47.1	0.8
7	5.62	0.05	52.9	0.5
8	7.02	0.09	68.8	0.7
9	8.45	0.17	85.2	1.2
10	9.65	0.11	99.4	2.9
11	10.25	0.21	110.5	3.1
12	11.85	0.41	122.1	3.3

Table 3 *Extended range of data points*

1.7.7 Week 1 exercise 4 answer

[11]:
```
# <!-- Demo -->
#
# nPoints is the number of data points in the fit
nPoints = 12
#
# Define arrays to store the x and y data values
xData, yData = np.zeros(nPoints), np.zeros(nPoints)
#
# Define arrays to store the errors in x and y
xError, yError = np.zeros(nPoints), np.zeros(nPoints)
#
# Enter the x data points
xData[0] = 1.50
xData[1] = 2.31
xData[2] = 2.78
xData[3] = 3.58
```

```
xData[4] = 4.08
xData[5] = 4.76
xData[6] = 5.62
xData[7] = 7.02
xData[8] = 8.45
xData[9] = 9.65
xData[10] = 10.25
xData[11] = 11.85
#
# Enter the y data points
yData[0] = 14.3
yData[1] = 20.2
yData[2] = 30.1
yData[3] = 36.5
yData[4] = 42.7
yData[5] = 47.1
yData[6] = 52.9
yData[7] = 68.8
yData[8] = 85.2
yData[9] = 99.4
yData[10] = 110.5
yData[11] = 122.1
#
# Enter the errors in the x values
xError[0] = 0.21
xError[1] = 0.11
xError[2] = 0.43
xError[3] = 0.13
xError[4] = 0.17
xError[5] = 0.18
xError[6] = 0.15
xError[7] = 0.19
xError[8] = 0.17
xError[9] = 0.11
xError[10] = 0.21
xError[11] = 0.41
#
# Enter the errors in the x and y values
yError[0] = 2.1
yError[1] = 1.7
yError[2] = 3.3
yError[3] = 1.1
yError[4] = 0.9
yError[5] = 1.1
yError[6] = 1.5
yError[7] = 0.9
yError[8] = 1.2
```

```
yError[9] = 2.9
yError[10] = 3.1
yError[11] = 3.3
#
print(" ")
print("Check the data points:")
print("Point\t x\t\t\ty\t")
for n in range(0, nPoints):
    print("{:d}\t{:5.2f}\t+-{:5.2f}\t\t\t{:5.2f}\t+-{:5.2f}".\
          format(n, xData[n], xError[n], yData[n], yError[n]))
```

```
Check the data points:
Point    x                      y
0        1.50   +- 0.21         14.30   +- 2.10
1        2.31   +- 0.11         20.20   +- 1.70
2        2.78   +- 0.43         30.10   +- 3.30
3        3.58   +- 0.13         36.50   +- 1.10
4        4.08   +- 0.17         42.70   +- 0.90
5        4.76   +- 0.18         47.10   +- 1.10
6        5.62   +- 0.15         52.90   +- 1.50
7        7.02   +- 0.19         68.80   +- 0.90
8        8.45   +- 0.17         85.20   +- 1.20
9        9.65   +- 0.11         99.40   +- 2.90
10      10.25   +- 0.21         110.50  +- 3.10
11      11.85   +- 0.41         122.10  +- 3.30
```

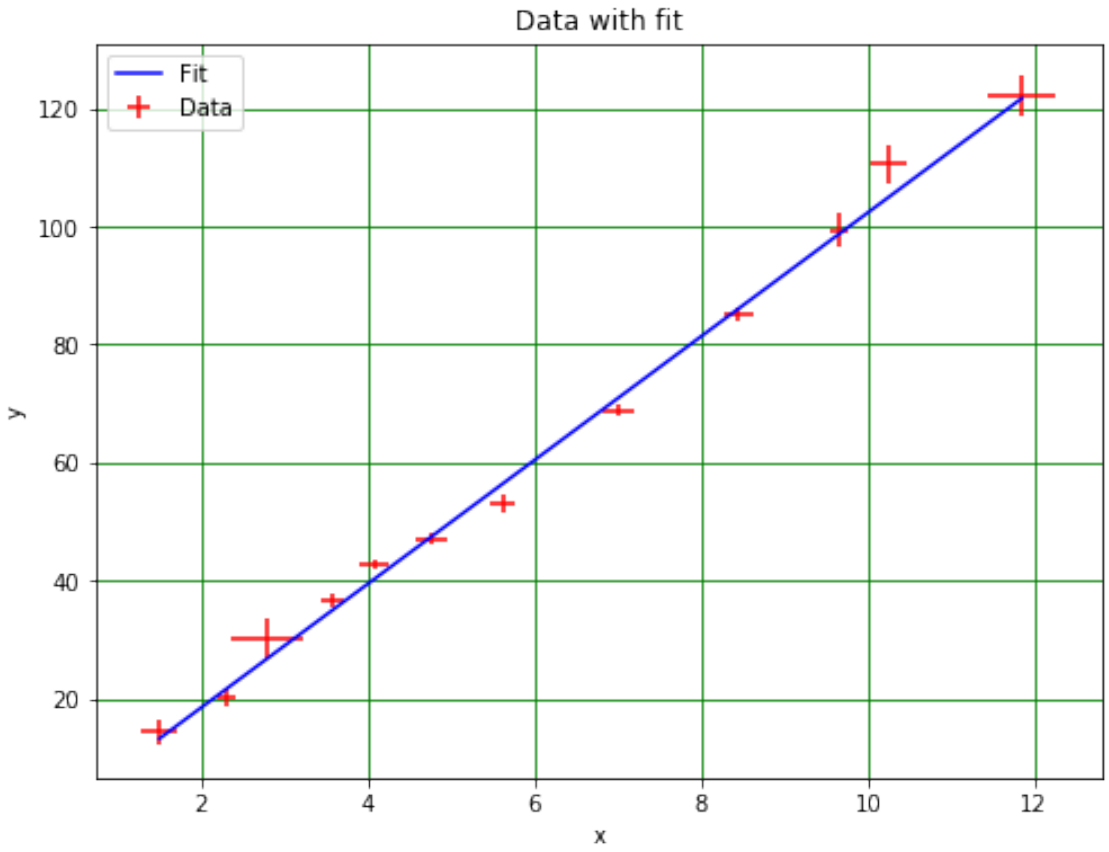
[12]:
```
# <!-- Demo -->
#
def fitLine(p, x):
    '''
    Straight line
    '''
    f = p[0] + p[1]*x
    return f
#
def fitLineDiff(p, x):
    '''
    Differential of straight line
    '''
    df = p[1]
    return df
#
def fitChi(p, x, y, xerr, yerr):
    '''
    Chi function for straight line fit
    '''
```

```
    e = (y - fitLine(p, x))/(np.sqrt(yerr**2 + fitLineDiff(p, x)**2*xerr**2))
    return e
#
# Set initial values of fit parameters, run fit
pInit = [1.0, 1.0]
out = least_squares(fitChi, pInit, args=(xData, yData, xError, yError))
#
fitOK = out.success
#
# Test if fit failed
if not fitOK:
    print(" ")
    print("Fit failed")
else:
    #
    # get output
    pFinal = out.x
    cVal = pFinal[0]
    mVal = pFinal[1]
    #
    #   Calculate chis**2 per point, summed chi**2 and chi**2/NDF
    chisqArr = fitChi(pFinal, xData, yData, xError, yError)**2
    chisq = np.sum(chisqArr)
    NDF = nPoints - 2
    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))
    #
    # Calculate fitted function values
    fitData = fitLine(pFinal, xData)
    #
    # Compute covariance
    jMat = out.jac
    jMat2 = np.dot(jMat.T, jMat)
    detJmat2 = np.linalg.det(jMat2)
    #
    if detJmat2 < 1E-32:
        print("Value of determinat detJmat2",detJmat2)
        print("Matrix singular, error calculation failed.")
        print(" ")
        print("Parameters returned by fit:")
        print("Intercept = {:5.2f}".format(cVal))
        print("Gradient = {:5.2f}".format(mVal))
```

```
        print(" ")
        cErr = 0.0
        mErr = 0.0
    else:
        covar = np.linalg.inv(jMat2)
        cErr = np.sqrt(covar[0, 0])
        mErr = np.sqrt(covar[1, 1])
        #
        print(" ")
        print("Parameters returned by fit:")
        print("Intercept = {:5.2f} +- {:5.2f}".format(cVal, cErr))
        print("Gradient = {:5.2f} +- {:5.2f}".format(mVal, mErr))
        print(" ")
    #
    # 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, fmt='r', \
                 linestyle = '', label = "Data")
    plt.plot(xData, fitData, color = 'b', linestyle = '-', label = "Fit")
    plt.grid(color = 'g')
    plt.legend(loc = 2)
    plt.savefig("LineFitPlot.png")
    plt.show()
```

```
Fit quality:
chisq per point =
 [0.128 0.533 0.387 0.724 1.507 0.016 2.579 1.072 0.156 0.062 2.159 0.006]
chisq =  9.33, chisq/NDF =  0.93.

Parameters returned by fit:
Intercept = -2.51 +-  1.63
Gradient = 10.48 +-  0.28
```



1.7.8 Week 1 exercise 5

Does the straight line still describe the data? Have the values of the gradient or the intercept changed significantly?

1.7.9 Week 1 exercise 5 answer

Straight line should still describe data. Gradient and intercept should be compatible with previous values.

1.8 Week 1 marks

Exercise	Mark	Comments
1	0	
2	2	
3	2	
4	4	
5	2	
Total	**10**	

That's the end of Week 1 for Phys105. You will have a chance to practice using the line-fitting routine that we have worked with here in Practical Physics I (Phys 106).