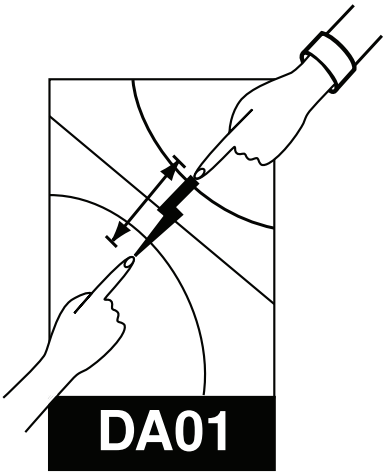
An introduction to data analysis in Python



JJL, BHF August 2019

1 Objectives

In this practical you will learn how to process data in Python using 'notebooks'. Python is a widely used open-source programming language, which emphasizes that coding should be simple, with a wealth of help available online[1]. Notebooks allow graphs and visualizations of data to be displayed and organised quite neatly, making them an ideal tool for data analysis.

An advantage of Python is the number of functions (or 'modules') that are available to add into the base programming language. If you have a programming task, chances are there is already a Python module which you can use to get the job done faster by writing less code & writing simpler, more readable code. In this practical, we will use some of the most well-known packages for data analysis, which essentially give Python the data analysis tools we need as physicists. We will be using Python as a tool and framework for data analysis (without focusing too much on the programming aspects).

2 Introduction

Physics attempts to build mathematical models of how the world around us works. Experiment is always needed to test theories: If experimental tests do not give results consistent with the mathematical model, then the model cannot be entirely correct. For example, general relativity's first success was predicting the observed precession of Mercury's perihelion, which could not be explained by Newton's theory of gravitation.

To effectively compare measured data with theory, we need to know how well we understand our measurements. No measurement has absolute precision, so we first make an estimate of the accuracy which we can achieve in our measurement (known as the *uncertainty*, or *error*). This comes from two sources: *random* and *systematic* errors. The difference between these types of error are most easily illustrated if we consider a large number of measurements of a single quantity.

Random errors are those which lead to the distribution of the measurements about the mean — in any given measurement the random error could lead to the measurement being greater or less than the actual value. The most obvious source of random error is the finite precision of our measurements: when we measure a distance with a ruler, for example, we can only measure accurately to the smallest division marked on the ruler, and estimates based on the ruler sub-divisions can vary from measurement to measurement.

Systematic errors are the errors which affect all of your measurements in the same way; this shifts the mean of the set of measurements away from the actual value. A common systematic error is poor calibration of measuring devices like scales, for example, where a scale isn't set to zero while empty.

[1]For example, a huge collection of submitted questions with answers are available at `https://stackoverflow.com/`. Getting started guides, documentation and a list of online courses is available at `https://python.org/`

These are concepts you will encounter in your labs. In each of the labs you will do, some data are acquired which always have a degree of uncertainty.

2.1 Experimental steps

This is a 3 hour lab. We work through 3 examples, the first is worked for you to demonstrate how the functions work. Then you will apply the same procedure to the remaining examples. These examples are intended to prepare you for the type of data processing that is typically required in the labs.

This lab is run completed remotely with the help of demonstrators online via Teams.

This lab will proceed in the following manner (note: there is a morning session and an afternoon session):

- 10:30-11.30am (2-2:30pm) - The demonstrator will go through the first example, with everybody, on Teams. You should be setup and ready to follow along.
- 11-11.30am (2:30-3pm) - We will split into subgroups. The first example will be discussed briefly in your subgroup.
- 11.30-12.30pm (3-4pm) - Working in subgroups, you will create a new notebook and begin the second example, applying the lessons from the first example. The demonstrator will discuss the second example with the group.
- 12.30-1.30pm (4-5pm) - Move onto the third example. Submit notebook (as PDF) to Canvas.

2.2 Record-keeping

Throughout this and all other practicals you carry out, you should keep records in your experimental logbook. It is not necessary to repeat instructions written in the script: You should however record any further details or thoughts you have about the procedure.

All results that you obtain should also be recorded in your logbook, along with their associated errors.

When recording data, you should think sensibly about how much precision is needed. For more information on basic error analysis, see auxiliary script AD02 — *estimating experimental errors*[2].

You should record the answer to any questions which appear in this script in your logbook.

3 Getting started

3.1 Choice of software

To complete this lab, you have a choice of software - a Jupyter server within physics[3] or Google Colab[4], a free service hosted by Google. Both allow you to easily create Python notebooks online.

If you use the physics server, log on to that url and sign in with your university account. Using Google Colab requires registering for a Google account. A Google account can be registered to any email address (e.g. your university address). There is more information about using the physics server on the Canvas page for this practical [5]

[2] `www-teaching.physics.ox.ac.uk/practical_course/Admin/AD02.pdf`
[3] `https://jupyterlab.physics.ox.ac.uk`
[4] `https://colab.research.google.com/`
[5] `https://canvas.ox.ac.uk/courses/275711/pages/general-information-on-data-analysis?module_item_id=2693350`

3.2 Accessing the material

The files for this lab are downloaded from Canvas. The first example: 'DA01_exercise1.ipynb', and two data files: 'hubble.csv' and 'protein_fluorescence.csv'

Go to Canvas: `https://canvas.ox.ac.uk/`, then under 'Prelims Practical pages', open the 'Data Analysis' page and download the notebook and data files.

Note: You will need to use your Single Sign On credentials to log in to Canvas.

In a web browser, open either the physics Jupyterlab server or Google Colab and select the 'Upload Notebook' option. Upload the notebook for the first example (which you downloaded from Canvas).

Then press the 'Folder' button on the left (see figure 1) and then press the button to 'Upload to Session' and upload the Hubble data (hubblev.csv).

Also log into Teams and join the lab session. Ideally, have the Jupyter window and the Teams window open as shown in figure 1.

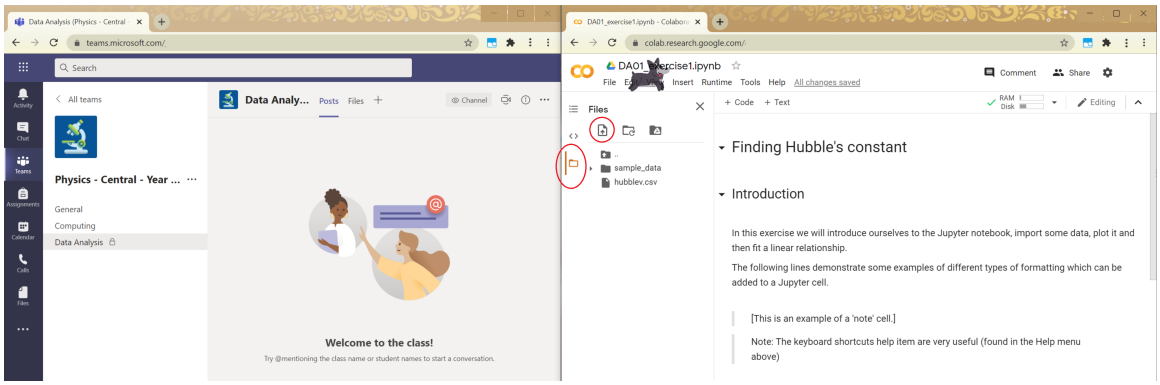


Figure 1: An example of how to work for this laboratory. In this screenshot we have the Teams website open alongside the Google Colab interface. Note: the 'Folder' and 'Upload' buttons used to upload data files have been circled.

In a Python notebook, each block of text or code is called a 'cell' which can contain comments or code. You can see specific examples of these in the first example, try double clicking the cells.

Each cell is run individually by pressing the Shift and Enter keys together. The advantage of having code in cells is we can edit and re-execute a portion without running the whole notebook (e.g. maybe we just want to change the label of a graph rather than re-run the whole analysis).

Before we get started, a word of warning: beware of copy and pasting (in general), specifically from a pdf file. Some of the characters may not 'copy' to be the character you expect (e.g. some '*' characters often copy as a very similar character). When coding, many 'bugs' are syntax errors, so ensure if you do copy and paste that you: 1. understand the material, 2. closely examine the pasted result.

4 Finding Hubble's constant

Many of you will have heard of Edwin Hubble, or at least of his namesake constant H_0. In the 1920s, Hubble discovered a linear relationship between the distance d of an object from our Solar System and its velocity relative to us v:

$$v = H_0 d \tag{1}$$

where the constant of proportionality H_0 is Hubble's constant. This relationship served as one of the first pieces of experimental evidence for the Big Bang theory.

Hubble's data[1] consist of an astronomical object's distance from Earth and its apparent velocity. In order to determine H_0, we need to find the straight line which comes as close to passing through all of the data points as possible: This is done mathematically by minimising the square of the sum of the distances from the straight line to each of the data points, a process known generally as least-squares fitting (or linear regression).

4.1 Importing data

First step, import our data. Our data consist of an array of numbers, formatted in a file which has a large number of rows with the columns separated by commas. This is a very standard formatting style referred to as 'Comma Separated Values', or 'CSV' for short.

A very popular module for working with data in Python is `Pandas`, which has a function for importing a CSV file, making things very easy for us. We load this module using the `import` command.

A central theme of `Pandas` is formatting datasets so that they can be easily manipulated. `Pandas` makes use of a data structure they call a 'data frame'. When we import our CSV file, we get a `Pandas` data frame (some of these terms are confusing, but they are just names the developers have given these things)

There are other functions specifically to provide summaries of the data. For a quick summary of our data we use the `head` function. We could also use the `describe` function, or `info()` and `columns`.

> Execute the cell (shift+enter) in the notebook containing the import command to load the data from file.
>
> To see the first few lines of the data contained in the file, execute the cell which has the `head` function.

Our data are easy to look at - it helps that we only have two columns and a handful of values.

4.2 Plotting data

To create a plot, we load another module, `matplotlib`. This module is widely used and has commands similar to another piece of analysis software: MatLab (which you will use in the Computing laboratory).

First, we need to specify which columns are the x and y axes, and which contains our uncertainties. The column names are those found when using the `head` or `describe` functions.

We extract the columns of data and store them as variables called 'x' and 'y'. Note: We access the elements of our data frame (our imported data) with square brackets.

There are a number of different methods to extract a column of data. We can ask for the column of values by either: 1. the column name, or 2. specifying the column number (we will use the column name).

We then use the `errorbar` function of the `matplotlib` library to create a plot with error bars.

> Execute the cell which contains the `errorplot` function. A plot should appear within the notebook.

4.3 Linear regression

Look at the graph: While the data have considerable variation, they do seem to relate in a roughly linear way. We can use linear regression to find an estimate of the line's slope and intercept, and the error on those two quantities.

To do our fit we need to import a statistics package, of which there are many choices, we will use one of the most common. We import the `scipy.optimize` module and use the `curve_fit` function.

The `curve_fit` function requires us to define the type of fit. In this example we will define a linear fit (in later examples, we will define different functions).

```
> def linear(x, a, b):
>   return a * x + b
```

We also specify our 'x' and 'y' data and also the uncertainties on our y-data. Here, we must specify the '`absolute_sigma=True`' option. This option changes the way the fitting function works based on the type of uncertainties we have given it. When we have uncertainties based on measurements which are the same units as our data (which will be the case for most of your lab work) we must specify this option.

We then perform the fit with the function and data we have defined. Upon executing it, the `curve_fit` function returns two things:

- The fitted set of parameters (note in our function we used 'a' and 'b' as parameters for the slope and intercept)
- The co-variance matrix. This gives information about how well the parameters are affected by each other (this is covered in the second year stats course). Here we use it to extract our uncertainties.

Execute the cells to process the linear with using the `scipy` package and extract the slope and intercept from the set of parameters.

Usually when carrying out a least squares fit, we need to know how good the fit is, i.e., how well the fitted model matches the points. The simplest way to extract this information is by looking at the errors on the fitted parameters. To find the errors on the parameters we have to take the square root of the diagonal elements of the co-variance matrix. The co-variance matrix is a concept you will encounter in your second year, here, we process it in this way as an example to show how to compute the uncertainties on the fit parameters. In many programs, e.g. Excel, these numbers will just be displayed along with a fit. Here, we have introduced how they are computed (and you will likely go into more detail in later lectures).

Execute the cells which process the errors on the fit parameters.

4.4 Systematic error

Hubble's original paper did not include estimates of the measurement error on his values. Hubble had derived the distances from what he thought were bright super-giant stars, but it turned out they were actually giant clouds of ionised hydrogen, known as HII regions, which were much further away. This is a good example of a systematic error — an error in the design of, or fundamental assumptions behind an experiment. Although they can be very hard to spot, it is good practice to think about possible sources of systematic error.

As astronomical methods developed through the twentieth century, systematic errors were found in the data, and better distance references found. The accepted value for H_0 is now much smaller than Hubble's original value of 400–500 km s^{-1} Mpc^{-1}. There have been a large number of papers calculating a value for Hubble's constant since then. If you're interested in the different methods used and values obtained, try looking at [2].

▶ Interpret the fit results. Is the intercept significantly different from zero?

4.5 Organising and saving plots

It is important to include graphs in your logbook. We can save the plots made by `matplotlib` to files using the `savefig` function. It is a good idea to save your plots as separate files as they are often useful after the experiment, for example, when writing a report. Important plots should be printed and stuck firmly into your logbook.

Execute the cell which plots the fit and saves it as a Portable Network Graphics (PNG) image file. Ensure to label the axes appropriately. The file is saved to the same directory as the workbook file. One useful option to note: we make the plot larger with the `figsize` and `dpi` options. Tip: If we make it over-sized now, we can easily shrink it where-ever we use it later.

▶ Make sure that your graph axes are labeled. Save and print this graph, and fasten the page into your logbook. Discuss your work with a demonstrator.

5 Exponential and logarithmic relationships

In biological physics, compounds of interest are often tracked within a cell or organism using fluorescent proteins. These proteins emit photons when they are illuminated by light of a particular wavelength: by detecting these photons, the compound can be located within the cell. However, each time they emit a new photon there is a fixed chance that the protein will decay in such a way that it can't fluoresce any more. Thus emission from a fixed number of proteins will decay exponentially over the period of illumination. In order to accurately determine the total number of fluorescent proteins present, the decay constant must be determined and the decay must be accounted for in data processing.

For this exercise we are going to start a fresh notebook, just as you would begin a new laboratory experiment. The commands will all be very similar to those used in the previous exercise.

On the main Jupyter browser window (as in figure 1), from the 'File' menu, select 'New' notebook.

A new window will open with a fresh notebook. Convert a cell to markdown and title the notebook appropriately. Save the workbook with an appropriate name (e.g., 'DA01') - and don't forget to save periodically (although the notebook should 'auto-save' occasionally, best to save just in case...).

Some typical data for the emission of a fluorescent protein can be found in the data file named `protein_fluorescence.csv`. Import and plot this data, in the same process as in the previous exercise. The code begins like this:

```
> import pandas as pd
> df = pd.read_csv("protein_fluorescence.csv")
> df.head()
```

Define your x and y variables and add the plotting code to produce a plot of the data.

▶ From the plot, do you think that an exponential decay might be an appropriate description of this data?

Two good methods to test whether an exponential relationship is an appropriate description are: 1. take logs of one of your sets of data and to look for a linear fit to this data (which you may be familiar from A-level), or, 2. fitting an exponential function to the data. These two methods will be discussed in the following sections.

5.1 Fitting an exponential decay

In this example, the count data are fitted with an exponential that depends on time. We will use the `curve_fit()` function in a similar manner as we did previously but replacing the linear fit with an exponential.

In non-linear fitting, initial guesses for the constants are usually needed, this example shows how to include these.

The code below is similar to that we used in the previous exercise.

Inspect the form of the function. The parameters `a`, `b` and `c` are those we seek to 'fit'. In the code snippet below, we specify estimates of these parameters for the `curve_fit()` function, into: `p0=(A, B, C)`).

From the plot of the raw data estimate initial parameters for the exponential function and substitute those in place of 'A, B, C'.

If you get an error message then ask for help from a demonstrator.

```
> from scipy.optimize import curve_fit
> import numpy as np

> def exponential(x, a, b, c):
>   return a*np.exp(-b*x)+c

> parameters, covariant_matrix = curve_fit(exponential, x, y, p0=(A, B, C))
> a_fitted = parameters[0] #slope parameter is parameter #0
> b_fitted = parameters[1] #exponential parameter is parameter #1
> c_fitted = parameters[2] #intercept parameter is parameter #2
```

Use the 'print' function to display the values of the fitted parameters (defined above as 'a_fitted', 'b_fitted', 'c_fitted') i.e. `print(a_fitted)`.

Assess the quality of the fit by plotting the fit over the points. Use functions as we did in the previous exercise, but with one additional feature: here, we will specify the range we want on the x and y axes using the `xlim` and `ylim` methods; see below:

```
> import matplotlib.pyplot as plt
> plt.plot(x, y, "o", label="original data")
> plt.plot(x, a_fitted*np.exp(-1*b_fitted*x)+ c_fitted, "r", label="fitted line")
> plt.xlabel("time/s")
> plt.ylabel("counts")
> plt.xlim(left = 0)
> plt.ylim(bottom = 400, top = 1300)
> plt.legend()
> plt.show()
```

▶ Write down the equation of the fitted decay (in your log book), with the values (and errors) derived from your fit.

What is your best estimate of the time it would take for the fluorescence signal to drop to 50% of its original value?

Do these data support the hypothesis that there is a constant background signal due to scattered light from the illumination?

5.2 Linearising an exponential decay

Consider a general exponential relationship:

$$y = Ae^{\alpha x} + y_0 \tag{2}$$

If we take the natural log of both sides of this equation, we obtain:

$$\ln(y - y_0) = \ln A + \alpha x \tag{3}$$

Note how the constant offset y_0 has been moved to the left-hand side, equivalent to subtracting the value of any background or offset from all data values.

Where we suspect there is a relationship of the form of equation 2, we can test it by plotting the corrected y-values on a logarithmic axis. For a simple exponential decay or increase, this procedure should give a straight line plot. Note the offset y_0 can cause complications. Here, we will subtract our previous fitted result (the 'c_fitted' parameter) from our data, which gives us the quantity '$y - y_0$'.

We then define a new column of data, which has the y_0 parameter subtracted (hence this is referred to as 'baseline' subtracted data). This offset we fitted in the previous section. We then take the log of that 'baseline corrected' data (note we imported the `numpy` module for this).

We then take a look at our data frame (with the `head` function) and note the 'corrected log' values have been added as a new column of data.

Follow the code below to calculate the log of the count values in the manner described above:

```
> df["counts corrected"] = df["counts"].values - c_fitted
> df["log counts corrected"] = np.log(df["counts corrected"])
> df.head()
```

Again, define x and y variables and plot the corrected log counts.

Now produce a linear fit to the log of the corrected count data and plot it together with the data. Now the data are in a linear form, the code will look very similar to the fit we made for the Hubble data.

Save this plot as an image to print out into your log book (as before with the `savefig` function).

▶ Are the parameters extracted from the linear fit the same as the parameters calculated for an exponential fit? Are they consistent with each other?

6 Polynomial fitting

Here we analyse data from cold atom experiments using laser cooling, trapping and magnetic trapping to produce a Bose-Einstein condensate: a few hundred thousand bosons all in the same wave-function. Pictures of the falling atoms were taken with a digital camera at a number of times after the atoms had been released from the trap and the position of the atoms on the camera recorded against the time of flight (see Table 1). As you already know, if gravity is the only force acting on an object released from rest it will obey the equation of motion:

$$s = \frac{1}{2}gt^2 \tag{4}$$

where s is displacement, g is the acceleration due to gravity and t is the time. When this motion is imaged by a digital camera the apparent displacement in pixels should obey the equation:

$$p = \frac{1}{2D}gt^2 \tag{5}$$

where D is the distance corresponding to a single pixel. As p, g and t are generally known to a very high precision, by fitting a parabola to such a set of data we can calibrate the system and convert later measurements from pixels into real distances.

Here we present two options for entering data, both are useful to know for lab circumstances:

1. We enter data into a spreadsheet, save the spreadsheet as a CSV file and import (as we have done in the previous two examples), or,

time of flight (ms)	observed displacement of atoms (pixels)
3.00	4.2
6.00	22.1
9.00	51.6
12.00	91.8
15.00	141.6
18.00	200.6

Table 1: The time of flight and z-position of a Bose-Einstein condensate falling from rest under gravity.

2. We type the data directly into the notebook. In this example there are only a few data-points so manually entering them isn't difficult, but with more data this method would become time consuming and error-prone!

- The first method requires a CSV file to be created. One simple way to do this is to type the data into the first two columns of a blank spreadsheet (Excel or other), putting suitable column labels in the first row, and then export the spreadsheet to CSV format File › Save As › Save as type › CSV (Comma delimited) (*.csv), choosing an appropriate file name.
- The second method we type the data directly into the notebook, specifying columns names as shown below:

```
> x = [3, 6, 9, 12, 15, 18]
> y = [4.2, 22.1, 51.6, 91.8, 141.6, 200.6]
> df = pd.DataFrame({"time of flight (ms)":x, "displacement (pixels)":y})
```

Create a new section (of the same notebook), appropriately named (e.g. 'Exercise 3').

Import the data from Table 1 using one of the methods described above. Quickly check the data using the `df.head()` function.

Define x and y variables, and plot the data.

Note the time of flight is in milliseconds. We should work in SI units. Convert the data to more appropriate unit. As in the previous exercise with the `log` operation, we perform an operation on the whole column:

```
> df["time of flight (s)"] = df["time of flight (ms)"]*1e-3
```

6.1 Fitting a polynomial to the data

The `curve_fit()` function is incredibly customisable and will permit fitting of almost any equation. Here it should be used to fit a general second order polynomial, $at^2 + bt + c$.

We shall leave this exercise up to you to finish (with help from demonstrators). One hint when defining the polynomial: a square of a number in Python is typed as `x**2`.

Record your plot (save it as an image) and findings in your logbook.

▶ Is the polynomial you get consistent with what you thought about the intercept of the curve being zero? What about the other terms? i.e. Are these parameters consistent with equation 5?

Should we set the intercept of the curve to be zero?

From the parameters you obtained, calculate and record your estimate of the value of D. Take $g = 9.81\,\mathrm{m\,s^{-2}}$.

7 Summary

Now hand-write a brief summary of the experiment in your logbook. State the aims of the exercise, the method used, and what you have learnt, in a few sentences.

'Print' your notebook to a PDF file. Submit this in Canvas and discuss with a demonstrator (on Teams).

Don't forget to save a copy of your notebook before you leave!

Acknowledgements

The data for section 5 was provided by Prof C. Foot in the Department of Physics, University of Oxford.

Bibliography

[1] E. Hubble, A relation between distance and radial velocity among extra-galactic nebulae, *Proc. Nat. Acad. Sci. United States of America*, 15, 3, 168–173, 1929.

[2] S. Roser and G. A. Tammann, The ups and downs of the Hubble constant, *Rev. Mod. Astron.*, 19, 2008.