Simple Graphing and Data Analysis in Python

Will Luckin

February 26, 2016

Abstract

This document serves as a guide for any kind of `Python` based plot that a self-respecting Physicist may have to whip up in the course of duty. Published at `physics.luckin.co.uk`.

Contents

1 Introduction

This document aims to serve as a guide for simple data analysis, for Physics, in `Python, Numpy and Matplotlib`.

2 Quick start: Plotting a simple graph

2.1 Importing the data from a file

Firstly, one must open a blank .py file in a suitable editor (GNU Emacs, Sublime Text 3, etc).

At the head of the `Python` file, one must import the libraries we are to use. This allows Python to be used to plot graphs and analyse data.

```
import numpy as np
from matplotlib import pyplot as plt
```

Then, the data to be plotted must be inputted into Python. This can either be in the form of a raw Python list or numpy array, or by importing an external data file. [1]

```
# simple python flat lists
x_data = [1, 2, 3, 4, 5]
y_data = [1, 2, 4, 5, 6]

# numpy arrays
x_data = np.array([1, 2, 3, 4, 5])
y_data = np.array([1, 2, 4, 5, 6])
```

Importing external data is slightly more complicated. There are a few use cases to cover:

```
# simple, tab-seperated file with no headers
data = np.loadtxt('file.txt')
# simple, comma-separated file with no headers
data = np.loadtxt('file.csv', delimiter=',')
# tab-seperated file with headers occupying one row
data = np.loadtxt('file.csv', skiprows=1)
```

Any kind of `ValueError` is most likely due to needing the `skiprows` argument to skip column headers. Data imported in this way, or manufactured as a single Numpy array in Python, will be a 2D array; different columns can be accessed via Numpy's slicing.

```
# first column
x_data = data[:, 0]
# second column
y_data = data[:, 1]
```

2.2 Generating the data from a functional form

If you instead do not have data, but wish to plot a graph of a function of x, this is also simple to achieve. However, it must be noted that Numpy is not a symbolic library in that it doesn't interpret mathematical notation in the same way as `Mathematica` or `Sympy`, (another Python library). One must generate an array of the numeric data to be plotted.

Consider the example of plotting a simple sine curve over the range of $-\pi$ to π. First, one generates the x data using `linspace`.

[1] Creating a valid external data file is covered in the appendices.

```
x_data = np.linspace(-np.pi, np.pi, 100)
```

where the 100 refers to the number of values in the array. Higher values lead to smoother curves. Then, one transforms the x data to get the desired y data. Numpy arrays can be used in nearly all Numpy functions instead of a single value, leading to an incredibly easy transformation.

```
y_data = np.sin(x_data)
```

2.3 Rendering the simple plot

This graph can be simply plotted using `pyplot`, the easy interface to `Matplotlib's` full plotting capabilities. In this case, to plot a simple line graph, this requires one line.

```
plt.plot(x_data, y_data)
```

To plot a scatter graph instead, replace the word `plot` with `scatter`.

Finally, to actually view this graph, `pyplot` needs to be told to actually open a window and render it to the screen. In addition, at this point a `png` file of the graph can be saved to the current working directory.

```
plt.savefig('graph1.png')
plt.show()
```

This would end up with a plot similar to the following.

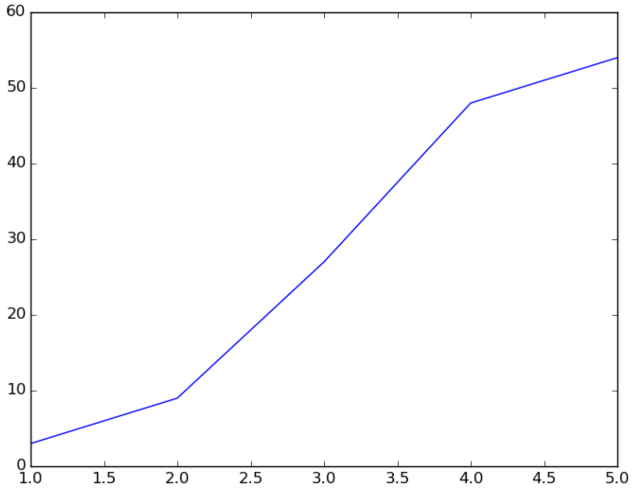

Figure 1: A sample, simple plot

2.4 Tidying things up

Real graphs have axis labels, titles, etc. These are very simple to add with `pyplot`. Any mathematical notation must be surrounded with dollar signs.

```
plt.title("Example of a graph title")
plt.xlabel("An x-axis label")
plt.ylabel("A y-axis label using $LaTeX$ syntax")
```

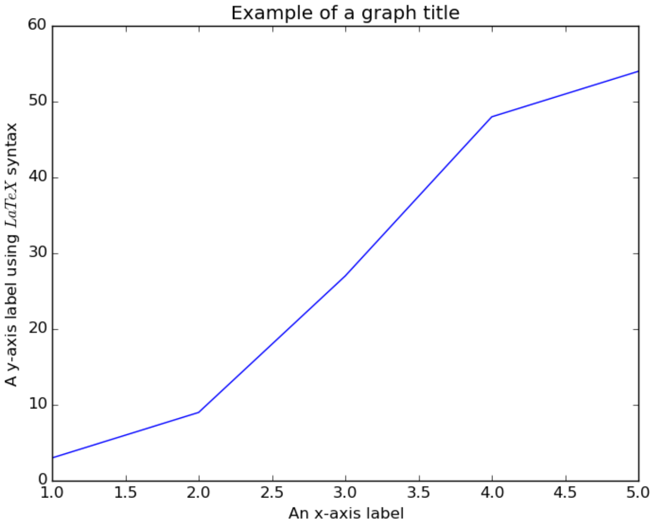

Figure 2: Simple plot with axis titles

2.5 Error bars

Error bars can be added to `Matplotlib` plots, although it is somewhat non-trivial due to a few quirks in the library. The key function here is `errorbars`. This takes the place of a `plt.plot` or a `plt.scatter`.

```
plt.errorbars(x_data, y_data, x_err, y_err)
```

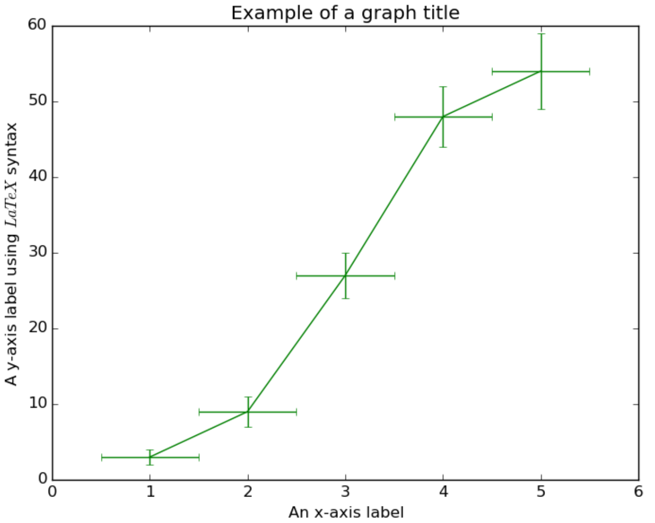

Figure 3: Simple errorbars

However, this code will automatically plot a line connecting the data. As this is often not what we want, we can pass in a `format string`, or fmt. This string allows the parameters of the plotted line to be set, and a parameter of `'o'` invokes a simple scatter plot.

```
plt.errorbars(x_data, y_data, x_err, y_err, fmt='o')
```

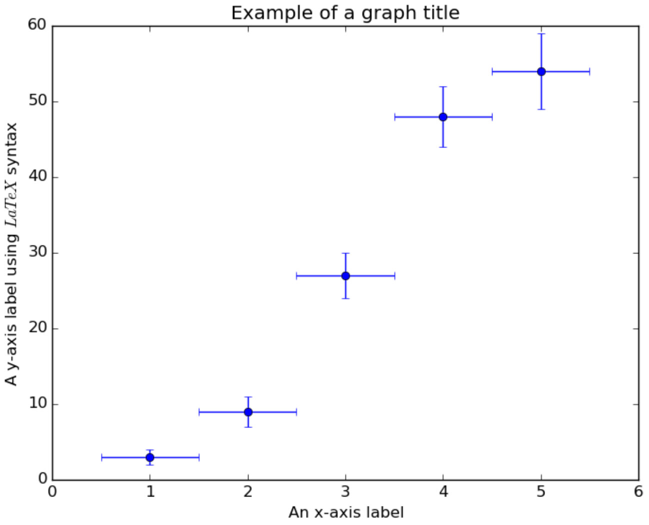

Figure 4: Errorbars with a scatter plot

3 Fitting a curve to the data

There are numerous ways to fit a curve to data in the Scipy stack, but the method explained here is universal, works in every case where you know the functional form of the data and is quick and simple to implement. It revolves around a function from the `scipy.optimize` submodule, called `curve_fit`. This uses non-linear least-squares fit, to optimise [2] a given function to an array of data. Firstly, therefore, one must prepare this function. You must know the functional form you wish to fit, but the general principles are as follows;

1. Your function must accept the x variable as its first parameter
2. Your function must accept a non-zero amount of additional, positional parameters. It cannot accept named parameters.

Before we begin, one must import the fitting function from Scipy:

```
from scipy.optimize import curve_fit
```

Considering the form of a straight line fit as:

$$y = mx + c \tag{1}$$

One can translate into `Python` simply:

[2] Bastardised American spellings be damned

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

Then, assuming one has arrays of x- and y-data to fit this function to, the remainder of this follows simply. curve_fit returns an array of optimised fit parameters, typically dubbed popt, and the covariance matrix of this fit, typically dubbed pcov.

```
popt, pcov = curve_fit(func, x_data, y_data)
```

The errors in this fit are wrapped up in the matrix. For an array of errors, one per each parameter of the fit, one can take the diagonal elements of the square-rooted matrix.

```
errors = np.diag(np.sqrt(pcov))
```

To plot this fit onto the graph, one can use the simple plotting tools as covered before, using linspace to generate the range of *x_data*. This *x_range* is an array of values interpolated between the minimum and maximum values in the *x* data. This allows for smooth plotting of the regression line in the case where the fit is non-linear.

```
popt, pcov = curve_fit(func, x_data, y_data)
x_range = np.linspace(np.amin(x_data), np.amax(x_data), 100)
plt.plot(x_range, func(x_range, *popt))
```

This would result in the following straight-line fit to the sample data.

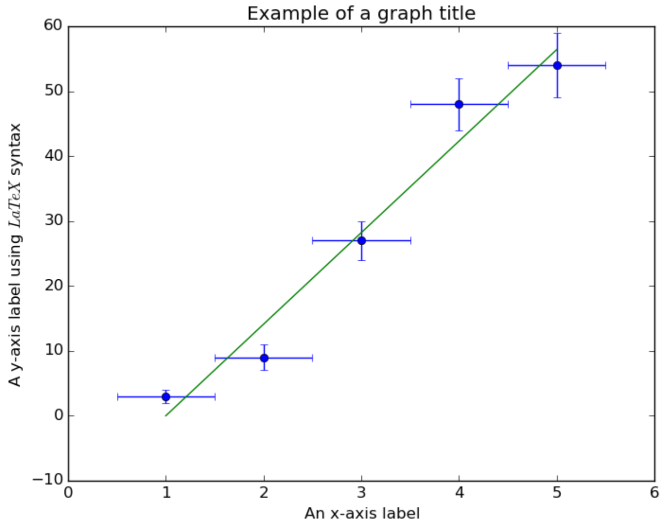

Figure 5: Straight line fit

To fit **any other functional fit** to this data, simply replace the definition of func. That's all that's required!

4 More complex plots (unfinished)

Using the pyplot interface to Matplotlib is a simple way to create plots, but it makes it difficult to access the more advanced features that the library offers.

Instead of using the imperitive commands from pyplot and manipulating the output sequentially, Matplotlib itself uses the object-oriented abstraction to make it easy to understand and create complex plots. Before embarking on this, one must understand the terminology used by Matplotlib.

Figure The figure object is the collection of axes, titles, etc, that make up your plot. A figure can have multiple axes, but each axis belongs to one figure. This is the uppermost level of the abstraction. From the Matplotlib website: "The figure keeps track of all the child Axes, a smattering of special artists (titles, figure legends, etc), and the canvas. (Dont worry too much about the canvas, it is crucial as it is the object that actually does the drawing to get you your plot, but as the user it is more-or-less invisible to you). A figure can have any number of Axes, but to be useful should have at least one."

Axes An axes object represents one set of co-ordinate axes. This object belongs to a figure. Here, one could set axis labels, the background of a specific set of axes, etc.

Line A line object represents a plotted set of data points on an axis. On this object, one can set the colour of the lines, legend labels, etc.

Let's begin with a blank Python file, and construct a plot using these new abstractions, instead of letting pyplot do all the heavy work for us! In this way, we can get a better understanding of the library, and therefore understand how to create better and more complicated plots.

Once again, we start by importing all the libraries we shall use:

```
import numpy as np
from matplotlib import pyplot as plt
```

5 Appendices

5.1 Creating a valid external data file

This is an incredibly simple process - numpy's `loadtxt` function can understand most flat files. The simplest way to get your data into Python, assuming you don't wish to manually create multi-dimensional arrays in pure Python is to use spreadsheet software, such as LibreOffice Calc or Microsoft Excel, and export the data as a `Comma Seperated Value, or CSV` file. This can then easily be loaded into Python, remembering to inform it that you have values seperated by commas - else it will interpret the entire file as a single, completely invalid, Python statement!

```
data = np.loadtxt('mydata.csv', delimiter=',')
```

You may also need to include the `skiprows` named argument, if you have included column headers in your data. Else, you will get a `ValueError` as Python attempts to cast English words to floats with disasterous results.