Lab5_Notebook

September 26, 2020

1 Lab 5: Reading data and fitting data

In this session you will learn about how to read data from a data file and the various ways you can fit lines or curves to experimental data. You will use techniques like this in the exprimental labs from now on to plot, and fit lines/curves to, your experimental data.

1.1 Reading data from text files

Often in physics we will make a measurement and our data will be stored in a file. Usually, our data will be stored in a standard text file (ASCII). Imagine we have a text file called `data_1column.txt` which contains the following

0.17
0.23
0.51
0.57
0.72
0.83

We want read these numbers and do some maths with them. First let us open the data file (it needs to be in the same directory as your code).

1.1.1 Opening and closing files

[ ]:
```
datafile = open('data_1column.txt','r')  # Open the data file for input
```

The r means the file is for reading. We could read this line by line by using commands like this:

[ ]:
```
y=datafile.readline()
print(y)
print(type(y))

datafile.close()
```

So, we can see that (as usual with python) the read command has assumed that the format is text - i.e. a string. We will usually want to read a number, so we should use:

It is essential that once you have finished reading the data file, you should "close" it again. This is important so dont forget. This is because the processor remembers where you are in the file (like a cursor in a data file). So, you cannot read a data file twice without either (a) closing it and opening it again, or (b) rewinding it. Its safest to close it when you've finished, with:

`filename.close()`

or you can rewind it with:

`datafile.seek(0)`

where `filename` is the name we have given the data file in our code.

1.1.2 Extracting data from file

[ ]:
```
datafile = open('data_1column.txt','r')   # Open the data file for input
y=float(datafile.readline())
print(y)

datafile.close()   # closing the file having read it
```

It is usually impractical to read one line at time. Here we have six lines in our data file, so the sensible thing is to create an array with six elements in it, and read in the six rows into these six elements. There are many ways to do this but this is the simplest:

[ ]:
```
import numpy as np

x = np.zeros(6)  # This creates an array of six elements, called x, and sets␣
 ↪the values initially to zero

datafile = open('data_1column.txt','r')  # Open the data file for input

i=0
for line in datafile:                  # this loops over every line in the file
    x[i]=float(line.strip())           # the "line.strip()" function strips out␣
 ↪the contents of a line
    i=i+1
print(x)

datafile.close()   # closing the file having read it
```

So now we have an array, x, that we can use in mathematical functions, we can make plots, figures etc.

1.1.3 Tables of data

Often our data will be in two or more columns - i.e. a data table. Something like this:

0.17 | 2.9
0.23 | 4.3
0.51 | 3.9
0.57 | 5.3
0.72 | 4.8
0.83 | 6.9

If we used the same code, then `line.strip( )` will create a single string with two numbers in it. We need to split this row up into its separate columns. To do this we use the `split( )` command:

```
[ ]: import numpy as np
     x = np.zeros(6)  # This creates an array of six elements, and sets the six␣
      ↪elements initially to zero
     y = np.zeros(6)
     datafile = open('data_2column.txt','r')  # Open the data file for input
     i=0
     for line in datafile:
          row=line.strip()            # Strips out one line and stores it in a string␣
      ↪"row"
          column=row.split()          # splits up the string into a list (called␣
      ↪column) of items found in that line
          x[i]=float(column[0])       # float(column[0]) is the first item converted␣
      ↪to a float
          y[i]=float(column[1])
          i=i+1
     datafile.close()   # closing the file having read it
```

```
[ ]: print(x,y)
```

Here `row` is a string that contains the text in that line. The `split( )` function splits up the string into anything separated by a space or comma and stores the result in a list - with one entry for each thing it found in the line. So, here, the two items on the line will have the variable names column[0] and column[1]. Now both x and y values are stored in arrays, to do with as you wish !

Sometimes, especially with big data files, you may not know the number of lines in the file, which means setting up the arrays is hard. Here is a trick to find out the number of lines in a data file.

```
[ ]: datafile = open('data_2column.txt','r')
     numlines = sum(1 for line in datafile)
     datafile.seek(0)
     print (numlines)


     datafile.close()   # closing the file having read it
```

You could now use `numlines` to set up your arrays etc. The `seek(0)` command basically rewinds the data file back to the beginning, otherwise the “cursor” would be stuck at the end of the file and it would read nothing.

1.2 Fitting a polynomial to data (e.g. linear fit)

This is one of the most common tasks in physics: we have measured some (x, y) data and we wish to fit a line or curve to it. There are many least-squares fitting/optimisation functions in Numpy and Scipy. One simple function in Numpy is `polyfit`, which fits a polynomial of any order (1= linear, 2 = quadratic etc). This works as follows.

Let's use a data file called `test_fitting.txt` (see VLE page to download this file). We will read the data from the file and store these data into two arrays, called x and y.

```
[ ]: datafile = open('test_fitting.txt','r')
     numlines = sum(1 for line in datafile)
     datafile.seek(0)
     print (numlines)
```

```
[ ]: import numpy as np
     x = np.zeros(numlines)
     y = np.zeros(numlines)
     datafile = open('test_fitting.txt','r')
     i=0
     for line in datafile:
          row=line.strip()
          column=row.split()
          x[i]=float(column[0])
          y[i]=float(column[1])
          i=i+1

     datafile.close()   # closing the file having read it
```

```
[ ]: print(x, y)
```

Fitting a straight line can then be done simply by using the `polyfit` function, with parameter '1' to indicate the order of the polynomial fit.

```
[ ]: para=np.polyfit(x,y,1)
     print(para)
```

The output for para is a list. Let's plot x against y and see if we can see what these parameters may refer to.

```
[ ]: import matplotlib.pyplot as plt
     plt.rcParams['figure.dpi'] = 120   # Controls the size and resolution of the␣
      ↪figures
     plt.plot(x,y, 'o')
     plt.show()

     print(para)
```

The numbers refer to the gradient and the y-intercept of the linear fit.

If you want errors on the parameters (and usually you do), then use the following command :

```
[ ]: para,covar=np.polyfit(x,y,1,cov=True)
     errors=np.sqrt(np.diag(covar))
     print(para)
     print(errors)
```

The second list printed containes the errors on the two parameters. Since we have used the `cov=True` option, the function also calculates something called a "covariance matrix" which contains information on the errors. The errors on the fitted parameters are the square roots of the diagonal elements of this matrix. This explains what the second line in the code is doing. You can use this for any fitting function that returns a covariance matrix. Since the function returns lists, you can refer to specific elements of the list e.g. `print(para[0])` will print the gradient, and `print(errors[1])` will print the error on the intercept, etc.

Errors can be included on the y data points, refer to the `polyfit()` help page for more details on this.

1.3 Fitting a specific function to data (non-polynomial)

This again is a very common task in physics. Essentially, we wish to fit our own function (that we define) to a set of data, rather than a polynomial. The best function to use is the `curve_fit( )` function in the `scipy.optimize` module. As an example, lets fit a simple linear function (again) to the same data as above - so we should get the same result.

```
[ ]: from scipy.optimize import curve_fit
     import numpy as np
```

```
[ ]: def func(x,m,c):        # here we are defining the function that we will fit to␣
      ↪the data
         return m*x + c      # in this case a simple linear fit with two parameters,␣
      ↪m and c.

     para,covar=curve_fit(func,x,y)    # This is scipy's curve_fit function.
     errors=np.sqrt(np.diag(covar))
     print(para)
     print(errors)
```

That's it - again, `para` will return the best-fit values of m and c. To fit a more complicated function, just replace the linear equation above with a more complex one, and you can have as many fit parameters as you like (here there are just two - m and c). Important: if your own function contains trigonometric or maths functions, you must use the Numpy versions of these (e.g. `np.exp( ), np.cos( ),` etc etc).

Again, as with `polyfit( )` you can include errors on the y-data points. The method of doing this is slightly different than for `polyfit( )`, so check out the Scipy reference guide to curve_fit here.

Finally, you will find that complicated functions will usually require an inital guess for the parameters. If you get a message like

"OptimizeWarning: Covariance of the parameters could not be estimated"

then you will need provide some guesses. For `curve_fit( )` this is done by using `p0=( )` where the brackets contains a list of the guesses of the paramaters. In our above example (even though we don't need to) we could specify `p0=(10, 0)` as one of our options. The guesses only need to very approximate.

[ ]:

```
```