Experiment 213: Data Analysis with Python

Aim

The aim of this experiment is to illustrate the use of PYTHON for analysing and displaying experimental data. In particular, you will learn how to fit linear functions to measured data so as to extract meaningful experimental parameters, as well as their uncertainties. The techniques discussed in this experiment may be applied to many other Advanced Lab experiments and **you are encouraged to use Python to analyse data obtained in other experiments**. In particular, this will greatly help you in quickly getting relevant uncertainties on your results.

Reference

1. Numerical recipes, Section 15.2 & 15.4;
   2nd edition free online at `http://apps.nrbook.com/c/index.html`
2. The python documentation: `http://docs.python.org/2/`
3. Numerical python: `http://www.numpy.org/`
4. Scientific python: `http://www.scipy.org/`
5. Matplotlib: `http://matplotlib.org`
6. Code academy python: `http://www.codecademy.com/tracks/python`

Introduction

PYTHON is a general-purpose interpreted language, which is high-level, has many extensions (known as "modules") and is easy to learn. Furthermore, PYTHON is free (as in free beer). Here we will use an interactive graphical user interface, called SPYDER. If you are on an Advanced Lab computer, start the program by going in the Start Menu and typing "Spyder" in the Search bar, or go into All Programs → Physics → Python Anaconda → Spyder.

If you have not used PYTHON before, you will first need to familiarize yourself with the language. Log on to a computer, start a web browser, and go to

`http://docs.python.org/2/tutorial/`

We maintain a list of common errors on the back of this pamphlet. Refer to the documentation when you get stuck.

Please feel free to ask for help from a demonstrator at any time.

There are several different ways to use Python. For the experiment following, it is convenient to start an IPython console (Menu "Interpreters" or "Consoles" → Open an IPython Console). In this console, plots may be displayed inline as in a Notebook. This may be the default, otherwise you can activate this option at any time by typing and executing (with the [Enter] key) the magic command `%matplotlib inline` in the console. Alternatively, you may prefer to display your plots in separate windows, that you can move around, and in which you can pan and zoom in your graphs. This is activated with the `%matplotlib qt` command. Note that in the latter case, figures will often pop up *behind* the SPYDER window. Check the Windows Taskbar for opened plot windows.

Python can do a lot of things, but when just started it does not know much about anything. To make something useful, you typically have to load some modules, i.e. packages of functions and routines to perform a certain task. Almost all calculations in PYTHON will involve the use of the NUMPY extension module, also known as *numerical Python.* Furthermore, plotting is easily done with the MATPLOTLIB module, which has a syntax similar to MatLab. Finally *scientific Python* (scipy) contains a large number of handy mathematical routines. Check the reference Section above for links to online documentation. Again — Python is about choice — there are different ways of loading these modules, as described below. Although both allow you to access the same routines and do the same things, it is important to know these differences as the syntax of the commands looks slightly different. Examples of code you might find on the Web to solve particular problems might be using any of these approaches, which can sometime lead to confusion.

- If you want to mimic Matlab, and will be mostly working interactively, you can load ("import") the bulk of numpy/matplotlib/scipy in one go by using the command: `from pylab import *`. This is handy as you do not need to prefix commands with the name of the modules where they are defined (see below), but not recommended for large projects (as the same command name may have different meanings in different modules). Your console might be preloaded with `pylab`. If you write commands in a script, you must however include the command `from pylab import *` at the top of your script.
- The preferred way to load numpy/matplotlib is to use the commands (in the console and/or at the top of your scripts)

  ```
  import numpy as np
  import matplotlib.pyplot as plt
  import scipy.linalg as linalg
  ```

  With this approach, numpy commands are loaded into the `np` "namespace" and needs to be prefixed by "`np.`" to be recognized. Similarly, the plotting commands are loaded into the `plt` namespace and the linear algebra commands in the `linalg` namespace. This requires a bit more work but leads to cleaner codes.

In the codes listed below, **we will assume the use of the second option above**. Make sure you have executed the commands above before you start. If you prefer to use the `pylab` approach, you will need to remove all the namespace prefixes [i.e. use `array(...)` instead of `np.array(...)` or `solve(...)` instead of `linalg.solve(...)`].

In this pamphlet, sections which require you to obtain some results are numbered. Printouts may be obtained by selecting the **Print** command from the **File** menu (you might wish to first **Copy** and **Paste** into e.g. Word so that you can group items together on one page).

Estimating means from experimental data

In the simplest type of physical measurement, we repeatedly measure a quantity which we believe to be constant. Due to the inaccuracy of the measurement process, we do not always get the same result. Instead, we get a distribution of measured values about the true value, and we try to estimate the true value by various techniques such as by taking the average of the measurements. We need to know how good this estimate is likely to be.

We usually assume that the distribution of measurements about the true value is a normal (Gaussian) distribution. Instead of actually making a set of measurements, we can let Python generate a set of random data with a normal distribution by using the random number generator functions in Python. The `normal(mean, std, length)` function from the `np.random` sub-module returns `length` random samples from a normal distribution that has a mean of `mean` and a standard deviation of `std`. As an example, type the following line:

```
y = np.random.normal(25, 3, 20)
```

You can now inspect the array you have just created by typing

```
print(y)
```

You should find that the vector `y` is a 20-element array. For a normal distribution, 68% of samples are expected to fall within one standard deviation of the mean. Record the number of elements of the vector `y` which lie between 22 and 28. What do you expect this number to be if the elements are normally distributed around 25 with a standard deviation of 3?

If you now type:

```
plt.plot(y, '-o')
```

the components of the vector will be plotted as interconnected dots (that is the meaning of the `-o` format specifier) as a function of the index.

Note that subsequent `plt.plot` commands will generate overlapping curves by default, which is not necessarily what you want. Use the command `plt.clf()` to *clear* the current figure if needed. Alternatively, you can close the plotting windows with the mouse. You can also use the `plt.close('all')` command to close all opened figure windows in one go. Such a command may be useful at the top of your scripts.

(1) Plot out a graph of a 200 element vector `z` (as a function of the index) whose elements are normally distributed with mean 15 and standard deviation 2.

Let us now examine what happens as we take the average of the measurements. After making n measurements, $\{z_1, z_2, \ldots, z_n\}$, the mean m of these measurements is

$$m = \frac{1}{n}\left(z_1 + z_2 + \ldots + z_n\right)$$

It is interesting to see how m changes as we make more and more measurements. Given the vector `z` defined in (1) above, the following line calculates the mean of the first 1, first 2, first 3, ... measurements and places the result in the array `zm`

```
zm = np.cumsum(z)/(np.arange(z.size)+1)
```

(2) Enter the above and make sure you understand what it is doing. Explain this in your report. If you have not met the cumulative sum function `np.cumsum` before, try it out on a simple array such as `np.array([1,2,3,4])` and on a matrix array such as `np.array([[1,16],[3,9],[5,4],[7,1]])`. `np.cumsum(a)` will use the "flattened" array (row-by-row), `np.cumsum(a,1)` will sum along the horizontal dimension (across rows) only.

   Note the use of the `np.arange(n)` function above, which generates an array of subsequent integers starting from zero (from `0` to `n-1`). Note that all PYTHON indexing are zero-based. The first element of the vector `z` is `z[0]` while the last one is `z[z.size-1]`.

(3) Plot out a graph of the successive means of the vector `z` which you generated above and notice how the mean approaches the true value as the number of points becomes large. Note, however, that there is always some residual fluctuation which means that we can only *estimate* the true value from measurements — there will always be some residual uncertainty.

We now repeat the above procedure for further ensembles of measurements, all of which come from the same "population" with a mean of 15 and a standard deviation of 2. This will give an idea of the amount of fluctuation we can expect after taking averages. According to theory, the standard error in the mean of n measurements is $\sigma/\sqrt{n}$ where σ is the standard deviation of the population.

(4) We will now write a script file that generates (within a `for` loop) twenty random vectors `z` of 200 points with mean $\mu = 15$ and standard deviation $\sigma = 2$, and calculates and plots the successive averages for each of these just as in the above. Superimposed underneath the graphs, in different colours, we will create shaded areas delineating $\mu \pm \sigma/\sqrt{n}$ and $\mu \pm 2\sigma/\sqrt{n}$.

   Type the following code as a script file (File $\rightarrow$ New file) and run it *in the current interpreter*. Make sure you start from a cleared figure. (You do not need to type in the comments that appear after the # signs — they are there so that you understand the purpose of each line):

```
mean = 15
sigma = 2

# The for loop runs over the indented code.
# White spaces are important in python!

for i in range(20):                             # Sets up a for loop to execute 20x
    z = np.random.normal(mean, sigma, 200)      # Generates vector z
    zm = np.cumsum(z)/(np.arange(z.size)+1)     # Finds successive averages
    plt.plot(zm)                                # Plots the successive averages

n = np.arange(200)+1                  # Sets up an index vector

su = mean + 2*sigma/np.sqrt(n)        # Calculates mu+2sigma/root(n) for each value of n
sl = mean - 2*sigma/np.sqrt(n)        # Calculates mu-2sigma/root(n) for each value of n
plt.fill_between(n-1, sl, su,         # Shades the area from sl to su on the graph
        facecolor='b', alpha=0.3)     # Note how you can break incomplete statements
                                      # over multiple lines

su = mean + sigma/np.sqrt(n)          # Do the same for mu+-1sigma/root(n)
sl = mean - sigma/np.sqrt(n)
plt.fill_between(n-1, sl, su,
        facecolor='b', alpha=0.3)     # alpha denotes the transparency level
```

Print a copy of the graph you obtain (or copy-paste into your report), and explain its form.

Finding the best straight line through a set of points

In many applications, we have a set of n data points (x_i, y_i) for which a linear relationship $y = ax + b$ is expected to hold between the variables. Due to measurement inaccuracies, the data points do not lie exactly on the line and we need to find the best values of a and b given the data. The classical least-squares approach is to assume that the x_i are known exactly. With this assumption, the aim is to find a straight line

$$\widehat{y}_i = ax_i + b$$

that approximates to the y_i such that the value of the sum of the squared deviations

$$\varepsilon(a,b) = (\widehat{y}_1 - y_1)^2 + (\widehat{y}_2 - y_2)^2 + \ldots + (\widehat{y}_n - y_n)^2 \tag{1}$$

is minimized over all a and b (hence the name *least-squares*). $\varepsilon(a,b)$ and its generalizations is the misfit function. The solution to this problem can be found by differentiating $\varepsilon(a,b)$ with respect to each of its unknowns a and b and looking for zeros of the corresponding expressions (minima are points of zero derivatives). It is not difficult to check that a and b must be solutions of the simultaneous linear equations (for a proof, see Reference *Numerical Recipes*, Section 15.2)

$$\begin{pmatrix} \sum_{i=1}^n x_i^2 & \sum_{i=1}^n x_i \\ \sum_{i=1}^n x_i & n \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^n x_i y_i \\ \sum_{i=1}^n y_i \end{pmatrix} \tag{2}$$

(5) Write a function `c = lregress(x, y)` which is to be called with the column vectors `x` and `y` containing the x_i and y_i respectively. It should return the coefficients a and b of the best straight line in the vector `c=[a,b]`. You will find it convenient to use the `np.sum` command to construct the matrix on the left and the vector on the right of the equation. You will also find the `linalg.solve(A, B)` function, which solves $Ax = B$, useful.

Note that functions in PYTHON are defined as follows (again the indentation shows the extent of the function definition)

```
def myfunction(myarg1, myarg2):
    localvar = 2.0*myarg1          # Mind the indentation
    result = localvar + myarg2     # Of course this can be any code
    return result
```

(6) To test your code, download this data file:

http://www.psmsl.org/data/obtaining/rlr.monthly.data/150.rlrdata

save it, and import it by dragging the file into Spyder's variable explorer (found in one of the tab above the console). More details on the import procedure can be found in the following document

http://advlabsvr.phy.auckland.ac.nz/pamphlets/e301.pdf

Alternatively, you can load the data using numpy `np.loadtext('filename', delimiter=';')` command. The data represents mean sea level height (in mm; measured from about 7 m below sea level so that all values are positive) from the Auckland Harbour over the years. Fit a straight line to the data using the function `lregress` you just wrote. You can extract the individual columns of data using array indexing (`data[:,0]` extract all rows of the first column e.g). In your report, include a graph of the data superimposed with the best fitting line, together with the value of the fitting coefficients (slope a and y-intercept b) as well as all your code.

Note: Some data are spurious because of errors in recordings. The levels for those are set at -99999. Make sure you ignore/delete those meaningless outliers. For example, define a mask for positive `y` values, `mask = y>0`. Then index the `x` and `y` values with that mask, i.e. `x = x[mask]`, etc.

Linear least-squares model fitting

Fitting a straight line to a set of points as in the above section can be written in a more general matrix form as follows. The linear estimates $\widehat{y}_i$ of the y_i can be written as the system of equations

$$\begin{pmatrix} \widehat{y}_1 \\ \widehat{y}_2 \\ \vdots \\ \widehat{y}_n \end{pmatrix} = \begin{pmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} \tag{3}$$

or in matrix form as

$$\widehat{y} = Ac \tag{4}$$

where

$$\widehat{y} = \begin{pmatrix} \widehat{y}_1 \\ \widehat{y}_2 \\ \vdots \\ \widehat{y}_n \end{pmatrix}, \qquad A = \begin{pmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{pmatrix}, \qquad \text{and } c = \begin{pmatrix} a \\ b \end{pmatrix}$$

For a given set of coefficients c, the misfit (error) function (1) can be written as

$$\varepsilon\left(c\right) = \left\|\widehat{y} - y\right\|^2 = \left\|Ac - y\right\|^2 \tag{5}$$

We want to find c so that this is minimized. It can be shown that this happens when c is the solution of the system of equations

$$A^t A\, c = A^t\, y \tag{6}$$

These are called the *normal equations* of the least-squares problem: they are actually the same as equation (2) (check it; it is easy), just written in terms of the matrix A and y. A^t is the transpose of A (rows are swapped with columns). If $A^t A$ is non-singular, the unique solution of the normal equation is

$$c = \left(A^t A\right)^{-1} A^t\, y \tag{7}$$

In Python, the matrix A can be built with the `np.column_stack` and `np.ones` functions. Check the online numpy documentation for how to use these functions. The transpose of a matrix `A` can be obtained with the syntax `A.T` and matrix products, e.g. AB, calculated with `np.dot(A, B)`. You can then use `linalg.solve` as above to solve equation (6) for c.

(7) Check that for the straight line fitting problem, c as given by equation (6) is the same as the solution of equation (2) by writing a function `mregress(x, y)` based on equation (6). To do this add a call to `mregress(x, y)` to the test code which you used in procedure (5) above. List your code in the report. Comment on the result.

The advantage of the matrix formalism is that it is readily generalized to models other than the straight line. If the model is one in which the data is a linear combination of a set of known functions — corresponding to the *columns* of the A matrix, the above method of solution can be used to find the coefficients of the linear combination. As an example, consider the problem of estimating the amounts of two radioactive substances with known decay rates (α_1, α_2) in a sample. The measured number of disintegrations y per second at time t is expected to be given by

$$y = S_1 \exp(-\alpha_1 t) + S_2 \exp(-\alpha_2 t)$$

We want to estimate S_1 and S_2 from measurements of the decay rate y_i at times t_i. We solve this problem by formulating it exactly as above. The y_i are approximated using the model by

$$\widehat{y}_i = S_1 \exp(-\alpha_1 t_i) + S_2 \exp(-\alpha_2 t_i)$$

The misfit is computed as

$$\varepsilon(S_1, S_2) = (\widehat{y}_1 - y_1)^2 + (\widehat{y}_2 - y_2)^2 + \ldots + (\widehat{y}_n - y_n)^2 \tag{8}$$

and this is minimized over all S_1 and S_2. It is again convenient to write the model parameters as a single column vector $c = (S1, S2)^{\mathrm{T}}$.

In the above expressions, note again **the important difference between y_i and $\widehat{y}_i$**: the y_i's are the actual data (typically experimentally *measured* and noisy), while the $\widehat{y}_i$'s are *calculated* and represent the values predicted by the fit equation (the model) for the corresponding t_i's.

(8) Show that this problem can be written using the same matrix formalism (3) except that the matrix `A` is changed — clearly explain how and gives its form in your report. The result matrix `c` is now `[S1,S2]` rather than `[a,b]`. Write two functions as follows:

`emodel(c, t, a1, a2)`	which returns model data points in a column vector given the times in `t` and the model parameters `c=[S1,S2]`, `a1` $= \alpha_1$ and `a2` $= \alpha_2$.
`eregress(t, y, a1, a2)`	which finds the least-squares optimal value of `c` given the data points (t_i, y_i) in vectors `t` and `y`.

(9) Test your code with the following script file

```
a1 = 0.1
a2 = 0.7
c = [1000,8000]

t = np.arange(0, 20, 0.1)
ymod = emodel(c, t, a1, a2)

y = ymod + np.sqrt(ymod)*np.random.normal(0, 1, t.size)  # Adding Poissonian noise

cfit = eregress(t, y, a1, a2)
yhat = emodel(cfit, t, a1, a2)

plt.semilogy(t, y, 'go', t, yhat, 'b')
print(cfit)
```

This should plot out the actual data together with your fitted values on a graph with a logarithmic y axis. Notice how `emodel` is used *twice* in this file: once to compute the initial data to which some simulated noise is added; second to compute the $\widehat{y}_i$ to display the line of best fit. Report your fitted parameters and comment. Are these what you expect?

Weighted least squares fitting to a straight line

In the previous sections, all of the data points were given equal weighting, i.e. all of the points were considered equally important when doing the fitting. In many cases we have a set of data where the standard errors in the points varies from point to point; in these situations we need to 'weight' the fitting so that the points which are known most accurately have a greater influence on the values of the fitted coefficients.

The misfit function defined in equation (1) must be changed to take this weighting into account. For a weighted least-squares problem, the misfit is called the chi-squared value and is defined as:

$$\chi^2(c) = \sum_{i=1}^{n} \frac{(\widehat{y}_i - y_i)^2}{\sigma_i^2} \tag{9}$$

where σ_i is the standard error in data point i, i.e. y_i is known to within $\pm\sigma_i$. Again we assume that the x_i are known perfectly. Note: Fitting a line through points with uncertainties in *both* the x_i and the y_i is much harder to tackle and will not be addressed here (see *Numerical Recipes* section 15.3 if you are interested). In practice, if you encounter this case, you would arrange the fit such that the values with the largest uncertainties are chosen to be along the y-coordinate (and swap the x_i and the y_i otherwise).

If we redo the algebra for the unweighted fit to a straight line, it is easy to show that a and b are now solutions of the equation (see *Numerical Recipes*):

$$\begin{pmatrix} \sum_{i=1}^n \frac{x_i^2}{\sigma_i^2} & \sum_{i=1}^n \frac{x_i}{\sigma_i^2} \\ \sum_{i=1}^n \frac{x_i}{\sigma_i^2} & \sum_{i=1}^n \frac{1}{\sigma_i^2} \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^n \frac{x_i y_i}{\sigma_i^2} \\ \sum_{i=1}^n \frac{y_i}{\sigma_i^2} \end{pmatrix} \tag{10}$$

which is a generalization of equation (2).

Exercise

Show that if all of the standard errors in the data points are the same then equation (10) reduces to the previous (unweighted) case described by equation (2).

(10) Write a function `wregress` that calculates the best fit coefficients a and b given the standard errors in the data points. The function definition should be of the form:

```
def wregress(x, y, sy):
    # your code
    return a,b
```

where `sy` is a vector containing the standard errors σ_i in the data points. Most of the code can be copied from your function `lregress` (written previously) since only minor changes will be needed. Test your code with the following script file:

```
x = np.arange(0, 5, 0.5)
y = np.array([2.75, 4.83, 5.05, 6.80, 8.83, 8.64, 11.03, 13.20, 13.08, 15.68])
sy = np.array([0.5, 0.6, 0.7, 0.8, 0.9, 0.9, 1.0, 1.1, 1.1, 1.2])

c = wregress(x, y, sy)
yhat = np.polyval(c,x)

plt.plot(x, yhat, x, y, 'o')
plt.errorbar(x, y, yerr=sy, fmt='o')
```

This should plot out the data, with $\pm\sigma$ error bars, together with your fitted line. Make sure you understand what `np.polyval` is doing. In your report, record the values of the fitted parameters and include your code for `wregress`.

Uncertainties in the parameters

Because the data points y_i are not known perfectly (only to within $\pm\sigma_i$), it should be clear that we cannot determine perfectly the fitting coefficients a and b of the least-square linear fit. There is a certain probability that a and b will deviate from the nominal values returned by `wregress`. In other words, a and b have their own uncertainties, $a \pm \sigma_a$ and $b \pm \sigma_b$. It is important to estimate those uncertainties: In many experiments we extract the parameter we are after through a linear fit, i.e the value of a (or b) is the ultimate result of the experiment, and we want to know how accurately it has been determined.

As *all* of the data points y_i are used in the determination of the fit parameters a and b, they each contribute some of their own uncertainty to the uncertainty of the fit parameters. In order to estimate the error in the fit parameters we can use the standard "error propagation" formula. The total error σ_a in parameter a is the quadrature-sum (square root of the sum of the squares) of the contributions of the errors in each data point to the error in a. The contribution is given by the general formula which is the basis for all error calculations:

$$\sigma_a^2 = \sum_{i=1}^{n} \left[\sigma_i^2 \left(\frac{\partial a}{\partial y_i} \right)^2 \right] \tag{11}$$

The error in parameter b is found in exactly the same way. Before equation (11) can be applied, equation (10) must be solved for a and b; $a = a(x_i, y_i, \sigma_i)$ and $b = b(x_i, y_i, \sigma_i)$. This can actually be done analytically and is easier than it looks: equation (10) is only a 2-by-2 linear system after all. This allows the derivatives in the above expression to be derived in closed form. When the result is simplified, we end up with (see *Numerical Recipes*):

$$\sigma_a^2 = \frac{1}{\Delta} \sum_{i=1}^{n} \frac{1}{\sigma_i^2} \qquad \text{and} \qquad \sigma_b^2 = \frac{1}{\Delta} \sum_{i=1}^{n} \frac{x_i^2}{\sigma_i^2} \tag{12}$$

where

$$\Delta = \sum_{i=1}^{n} \frac{1}{\sigma_i^2} \sum_{i=1}^{n} \frac{x_i^2}{\sigma_i^2} - \left(\sum_{i=1}^{n} \frac{x_i}{\sigma_i^2} \right)^2$$

(11) Write a function `cerror` that calculates the errors in the fitted coefficients a and b for a straight line fitting problem from the equations above. The function header should be of the form:

```
def cerror(x, y, sy):
    # your code
    return sa, sb
```

which returns a tuple containing the standard errors `sa` (σ_a) and `sb` (σ_b) in the fit parameters a and b respectively.

Test your function by adding a call to `cerror(x, y, sy)` at the end of the code you wrote for procedure (10) above. Report the uncertainties of the fitted parameters in your report. Also include the code of your `cerror` function.

At this point, you are probably thinking that equations (12) may well return those uncertainties your demonstrators will be looking after in the reports for your other experiments, but they are not exactly the kind of expressions you will easily remember from the top of your head. The good news is that (1) the calculation of uncertainties in the fit parameters can be expressed more easily using the matrix formalism introduced above, and (2) that PYTHON has some built-in fitting functions that will return those uncertainties if you know how to ask and what to look for.

The weighted linear least-square fitting problem can be expressed through a generalization of the matrix equations (5) and (6). All one has to do is to replace the matrices A and y in these expressions by the corresponding *weighted* matrices A_w and y_w that read

$$A_\mathrm{w} = \begin{pmatrix} x_1/\sigma_1 & 1/\sigma_1 \\ x_2/\sigma_2 & 1/\sigma_2 \\ \vdots & \vdots \\ x_n/\sigma_n & 1/\sigma_n \end{pmatrix}, \qquad y_\mathrm{w} = \begin{pmatrix} y_1/\sigma_1 \\ y_2/\sigma_2 \\ \vdots \\ y_n/\sigma_n \end{pmatrix} \tag{13}$$

Notice how, in comparison to the unweighted case, each row of these matrices is simply divided by the standard error of the corresponding data point and this can be easily implemented through a straightforward change to your `mregress` function. Note that this can be further generalized to any linear combinations of any number of functions (i.e. columns) as described with the radioactive decay fitting. Rewriting equation (6) with the new matrices, the fitting parameters `c` are then solutions of the normal equations:

$$A_\mathrm{w}^t A_\mathrm{w}\, c = A_\mathrm{w}^t\, y_\mathrm{w} \tag{14}$$

Note that for the straight-line fitting problem, this is simply another way of writing equation (10). From this, one can define the *covariance* matrix, which is given by

$$\mathrm{cov} = \left[A_\mathrm{w}^t A_\mathrm{w}\right]^{-1}$$

This matrix — which can be calculated very easily using numpy matrix operations — is defined because its diagonal elements are actually the *square* of the uncertainties in the fitting parameters (i.e. the *variance* of the fitting parameters; see *Numerical Recipes* for a simple proof). When applied to the straight-line fitting problem considered here, this leads to a much simpler way of writing down the results of equation (12)

$$\sigma_a^2 = \mathrm{cov}_{11} \qquad \sigma_b^2 = \mathrm{cov}_{22}$$

The second good news is that some PYTHON fitting functions will return that covariance matrix. It is then straightforward to access the uncertainties in the fitting parameters. This is illustrated below for polynomial fitting.

To get a visual sense of the magnitude of those uncertainties, you *could* superimpose on the graph with all the data a line with a slope of $a + \sigma_a$ and $a - \sigma_a$ so as to delineate some *confidence limits*, a bit like what we have done in procedure (4). The parameter b has uncertainties of its own however, and it is not so clear how to combine deviations in slope a with deviations in y-intercept b. In fact, you may realize that the uncertainties in those two parameters are *not* independent: they are after all both derived from the *same* set of data, $(x_i, y_i \pm \sigma_i)$. This is where the off-diagonal element of the covariance matrix, $\mathrm{cov}_{12} = \mathrm{cov}_{21}$, comes into the picture: it is actually equal to the *co-variance* σ_{ab} between the two fitting parameters, and expresses how statistical variations in the fitting parameters are correlated (or not). For example, a positive σ_{ab} indicates that a and b will likely vary in the same direction.

To draw some confidence limits for the fit, we can now proceed this way: Recall that the fitting parameters can be used to model the data as $\widehat{y} = ax + b$, and this can be applied to any value of x (not just the x_i). Using error propagation, we can then calculate *uncertainties in these modelled data* as

$$\sigma_{\widehat{y}}^2 = \sigma_a^2 x^2 + 2\sigma_{ab} x + \sigma_b^2$$

which is the generalization of the error-propagating formula to the case of non-independent variables. We can then use $\widehat{y} \pm \sigma_{\widehat{y}}$ as our confidence limits. Again, the above equation can be written in matrix form for generalization to any linear combination of functions and for simpler implementation

$$\sigma_{\widehat{y}}^2 = \mathrm{diag}\left(A\ \mathrm{cov}\, A^t\right) \tag{15}$$

where A has the usual *un-weighted* form as defined before [the same form as that used by `mregress` in procedure (7)], with each row corresponding to a particular value of x (not just the initial x_i) for which the modelled data uncertainty is sought after, so that $\widehat{y} = Ac$. In the final procedure below, we will use this equation to illustrate graphically how the uncertainties in the data affect the modelled fit.

Use of `polyfit`

`np.polyfit`, a part of the standard `numpy/scipy` library, fits polynomials of any degree to sets of data, returning the best fit parameters. It can do a weighted fit, with the weights of the points specified by the optional extra calling parameter `w=weight-array`, where the weights are to be understood as a vector whose elements are $1/\sigma_i$. By specifying `cov=True` in the calling parameters, it will also return the covariance matrix as defined above, the diagonal elements of which are the variances of the parameters. In truth, `np.polyfit` is simply an implementation of the general weighted least square fit in matrix format as described above. It simply solves equation (14), no more, no less.

The syntax for calling `np.polyfit` for a straight-line weighted fit is

```
c, cov = np.polyfit(x, y, 1, w=1/sy, cov=True)
```

where `x`, `y`, and `sy` are vector containing the data (x_i, y_i) to fit and the corresponding uncertainties σ_i as before. The parameter `1` is there to specify fitting to a *first*-degree polynomial (straight-line, $ax + b$). Return values include the fitting coefficients `c = [a,b]` and the covariance matrix `cov`.

The line of code above is the one-line that you will want to use to calculate uncertainties for many other Advanced Lab experiments. Keep it in mind!

(12) Extend the code written for procedure (10) and (11) to check your fitting parameters and their uncertainties by using `np.polyfit` on the same data set. Extract the uncertainties in a and b from the diagonal elements of the covariance matrix. Include your code and list all the values you obtain in your report. Comment on whether they match or not with the ones obtained previously. Notice that `np.polyfit` is typically a bit more conservative in the uncertainties it returns (it scales the covariance matrix by a certain factor).

(13) Now consider an extended range of x values, `xext`, for which we are going to use the fitted model and the covariance matrix to calculate uncertainties in modelled values to delineate confidence limits.

```
xext = np.arange(-1, 6, 0.1)
```

Further extend your previous code to calculate the corresponding `yext` fitted values and their uncertainties `sigma_yext` [using equation (15) and the function `np.diag` that extracts the diagonal of a matrix]. For that, you will need to define the matrix `A` based on `xext` in a way similar to that in procedure (7). To calculate `yext`, you can either use another call to `np.polyval` or the expression Ac. Both should give you the same results.

Then, complete the graph generated in procedure (10) by adding shaded areas illustrating the $\pm\sigma_{\widehat{y}}$ and $\pm 2\sigma_{\widehat{y}}$ confidence limits of the fit with calls to

```
plt.fill_between(xext, yext+2*sigma_yext, yext-2*sigma_yext, alpha=0.25)
plt.fill_between(xext, yext+sigma_yext, yext-sigma_yext, alpha=0.25)
```

You should have, on the same graph, the data points with error bars, the line of best fit, and the confidence limits. Comment on the shape of the confidence limits. Is this what you expect? Play around by changing uncertainties `sy` for selected points of the data (one or two at a time), making them either very large or very small. Comment on the corresponding changes in the fit confidence limits, in particular in the location of the narrowest portion. Is this what you expect? Also comment on the position of the original data points with respect to the $1\sigma_{\widehat{y}}$ and $2\sigma_{\widehat{y}}$ confidence limits.

(14) Further add to your script file from procedure (10) so that it calculates the chi-squared value for the fit, as given by equation (9). Record this value, and consult a chi-squared table. The appendix of the pamphlet of experiment 211 describes how to calculate the number of degrees of freedom. Is the chi-squared value for this fit reasonable (i.e. does it fall into the range expected from 90% of possible experiments)? Interpret the significance of the value of chi-squared that you have obtained. Make sure to include the full code of your script.

Note: When using `np.polyfit`, you should **always** obtain the uncertainties on your parameters, plot the data (with error bars) and the fitted curve, and also calculate the chi-squared value. `np.polyfit` is an extremely useful linear fitting function that can fit polynomials of any order to data. Furthermore, many functions can be written as polynomials, for instance $a + b\cos^2\theta + c\cos^4\theta$ is a polynomial of degree 2 in $\cos^2\theta$.

Common errors

1. PYTHON is a multi-purpose language. We are using it for scientific data analysis, but there are many other applications. Two data structures are similar, which are the *list* and the *numpy array.* To illustrate the difference, try following:

```
a = [1,2,3]       # this is a list
print(3*a)        # could be surprising
b = np.array(a)   # convert to an numpy array
print(3*b)        # that is more familiar
```

2. Two versions of Python coexist, Python 2.x and 3.x. One important difference is how integer division is treated. In Python 2.x, when dividing two integers (like `4/3`), Python will make the result an integer, and therefore round the result to the nearest integer (in this case 1). To obtain the full floating point result (`1.333`), at least one of the numbers must be a float, which can be done by adding a zero floating part, like in `4.0/3`. In contrast, Python 3.x will always generate a floating result when dividing. This can lead to hard-to-spot errors. In doubt, write your numbers with an explicit zero floating part.

List of Equipment

1. Personal computer with an installation of `Python`, `scipy`, `numpy` and `matplotlib`

S.M. Tan

M.I.J. Fleming

A.R. Poletti

M.D. Hoogerland

S. Coen, this version — July 25, 2016.