Phys105-Lecture06-student

September 23, 2025

1 Introduction to Computational Physics - Lecture 6: Least-squares line fitting

1.1 Table of contents

Recap of Lecture 5

Last week we looked at how to input and output information: * String formatting using f-strings or the `format` function * Taking input from the keyboard using the `input` function and from files using the `open` function * Writing output to files using the `write` function

An example of this is the following code, which reads in comma-separated (csv) data and prints it out as a formatted table, as you might want to for data from a lab experiment for example.

[2]:
```
# <!-- Student -->

# Print out the header
print ("| {:12s} | {:10s} | {:10s} | {:10s} | {:10s} |".format("Measurement",␣
  ↪"$x$ value", "$x$ error", "$y$ value", "$y$ error"))
print ("-"*68)

# Open the file
f = open("fitdata.csv")

# Print out the data, tabulated
```

```
imeas = 0
for line in f:
    imeas = imeas +1
    data = line.strip().split(",") # remove \n from end of line (strip) and␣
 ↪split by comma to get a list
    print (f"| {imeas:<12d} | {float(data[0]):^10.2f} | {float(data[1]):^10.2f}␣
 ↪| {float(data[2]):^10.1f} | {float(data[3]):^10.1f} |")

# Don't forget to close the file
f.close()
```

Measurement	x value	x error	y value	y error
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	3.58	0.13	36.5	1.1
5	4.08	0.17	42.7	0.9
6	4.76	0.18	47.1	1.1
7	5.62	0.15	52.9	1.5
8	6.02	0.19	78.8	0.9
9	8.45	0.17	85.2	1.2
10	9.65	0.11	99.4	2.9

Introduction to Lecture 6

We now have all the basic python we need to look at real physics use cases. We will start by looking this week at how to fit data i.e. find the parameters of a function that best describe it. You will use this frequently in labs (for fitting various data), replacing the black-box GUI you have used up to now to give you more control over the fit and the results.

1.2 Least squares fitting

It is very common in physics, and scientific fields in general, to have a set of data points (y_i, x_i) for $i = 0...N$ (where x is the independent variable and y is the dependent one) that we need to determine the relationship between i.e. $y = f(\beta, x)$ where β are the parameters of the function. I am sure you've already come across such cases at A-level and in your PHYS106 labs. Often we know the functional form of $f(\beta, x)$ (linear, polynomial, power, exponential, etc) but we need to determine the actual β parameters that best describe the data.

To do this we must minimise the (absolute) distance (often called the residual r_i) between the function prediction $f(\beta, x_i)$ and the data points y_i over the full set of points. The most common method to do this is the "least-squared" technique (that was briefly introduced in your PHYS106 lectures - if you do labs), so-called because it minimises the sum of the squares of the residuals

$$S = \sum_{i=0}^{N} r_i^2 = \sum_{i=0}^{N} [y_i - f(\beta, x_i)]^2$$

Of course, generally we don't measure the points (y_i, x_i) exactly, but with some uncertainty on

each, which also needs to be taken into account. This is done by dividing the residual by the (total) uncertainty on y_i, σ_{y_i}, to get the residual at each point in terms of standard deviations i.e

$$S = \sum_{i=0}^{N} r_i^2 = \sum_{i=0}^{N} \left[\frac{y_i - f(\beta, x_i)}{\sigma_{y_i}} \right]^2$$

For linear equations, this can be solved analytically (by setting the differential $dS/d\beta_i = 0$ to find the minimum). However, in the general, non-linear case it cannot. Instead we need to use a numerical method to find the solution iteratively, starting from an initial guess of the parameters. Luckily, python contains a powerful scientific python module called SciPy, which includes a least squares fitting routine as part of its optimization package.

We will now proceed to use the SciPy `least_squares` function, along with the NumPy `array` data structures and matplotlib `pyplot` methods we have already seen, to:

- Load experimental data
- Fit the data with a given functional form to find the best fit parameters
- Visualise the result.

Note: SciPy is a powerful module, that does much more than just fitting, and you will use other features of the module later in the course and in PHYS205 next year.

1.2.1 An imaginary experiment

Let's start with an imaginary experiment that performed 10 measurements 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	3.58	0.13	36.5	1.1
5	4.08	0.17	42.7	0.9
6	4.76	0.18	47.1	1.1
7	5.62	0.15	52.9	1.5
8	7.02	0.19	68.8	0.9
9	8.45	0.17	85.2	1.2
10	9.65	0.11	99.4	2.9

Table 1 *A number of data points which are expected to lie on a straight line.*

and are stored in the fitdata.csv file we printed above.

1.2.2 Loading the data

The first thing we need to do is to load the data into NumPy arrays. We could do this manually by creating a NumPy array and copying the data from the file by hand e.g.

```
nPoints = 10
xData = np.zeros(nPoints)
xData[0] = 1.50
xData[1] = 2.31
```

etc, but this is error prone. Better is to read the data directly from the csv file, using the np.loadtxt function (which we covered in week 3) along with the optional `unpack` argument to transpose the data so that it can be read into separate arrays:

[3]:
```
# <!-- Student -->

import numpy as np
xdata, xerror, ydata, yerror = np.loadtxt("fitdata.csv", delimiter = ',',␣
 ↪unpack = True)
print(xdata)
print(xerror)
print(ydata)
print(yerror)
```

```
[1.5  2.31 2.78 3.58 4.08 4.76 5.62 6.02 8.45 9.65]
[0.21 0.11 0.43 0.13 0.17 0.18 0.15 0.19 0.17 0.11]
[14.3 20.2 30.1 36.5 42.7 47.1 52.9 78.8 85.2 99.4]
[2.1 1.7 3.3 1.1 0.9 1.1 1.5 0.9 1.2 2.9]
```

Once we have loaded the data, the first thing we should do is check it and the best way to do this is to plot it. This allows us to:

- Check that it is correct
- Confirm that it lies on the purported straight line

1.2.3 Lecture 6 formative exercise 1

Plot the data, including its errors, and check it looks sensible. Use `matplotlib.pyplot` (which we introduced in weeks 3 and 4) with the errorbar function. Does the data look like it lies on a straight line? If not, double check the values in the file compared to Table 1 above (which we assume is the definitive source) and fix any errors.

1.2.4 Fitting the data

The data should now look compatible with a straight line, as expected, so we are ready to fit it. As mentioned above, we will do this using SciPy's optimise.least_squares. Looking at the documentation the function definition is

```
scipy.optimize.least_squares(fun, x0, jac='2-point', bounds=(- inf, inf), method='trf', ftol=1e
```

This looks scary, but for now there are only a couple of arguments that are important to us: the first two, which have no default value, and the `args` tuple.

The first,`fun`, is a function that calculates the residuals r_i (with error) defined above, which will be minimised to find the best fit. Note that this function (which you will sometimes see referred to as a *cost* or *merit* function) defines the residuals, not the sum of the squares of them, since the `least_squares` method takes care of the latter automatically. The function must have the form

`fun(params, arg1, arg2, ...)` where `params` are the parameters to be minimised and `args` are the "data", which will be passed in indirectly via the `args` parameter of `least_squares`. In our case these are `args = (xdata, ydata, xerror, yerror)` so the minimisation function will have the form `fun(params, xdata, ydata, xerror, yerror)`. It should return an array of residuals, one for each point.

The second, `x0`, is simply the initial guess of the parameters that we will fit. It is a tuple/array of the same length as the number of parameters we will fit, 2 in our case: m and c.

Let's first define the minimisation function to calculate the residuals:

[5]:
```
# <!-- Student -->

def straight_line(params, xdata):
    """
    Function for a straight line
    - params    array of function parameters
    - xdata     array of x data points

    """

    f = params[0] + params[1]*xdata
    return f

def straight_line_diff(params, xdata):
    """
    Differential of function for a straight line
    - params    array of function parameters
    - xdata     array of x data points

    """

    df = params[1]
    return df

def minimise(params, xdata, ydata, xerr, yerr):
    """
    Calculation to minimise to find the best fit using chi2: difference between␣
 ↪each y point and its
    prediction  by the function, divided by the sum in quadrature of the error␣
 ↪on y, both from the
    y error and from the related error in x.
    - params    array of function parameters
    - xdata     array of x data points
    - ydata     array of y data points
    - xerr      array of x data errors
    - yerr      array of y data errors
    """
```

```
    residuals = (ydata - straight_line(params, xdata)) / (np.sqrt(yerr**2 +␣
↪straight_line_diff(params, xdata)**2 * xerr**2))

    return residuals
```

The minimisation function simply calculates the residuals defined in the second equation above, using NumPy's array-at-a-time functions we covered previously.

In order to do this, it depends on the mathematical function we are fitting $f(\beta, x) = mx + c$, which we have coded in the python function `straight_line`. The `straight_line` function takes the parameters (params $= \beta$) as a array, in this case a two-component array whose first component `params[0]` is the intercept c and whose second component `params[1]` is the gradient m, along with the independent variable x

It also depends on the derivative of the straight line function, `straight_line_diff`. You should be able to work out why this is the case from the error analysis you do in the PHYS106 lectures or by thinking about the effect of an error on the x variable on the result of the function.

Note that we are following good practice and writing *doc strings* for our functions. In this case we have not only explained what the function does but also what the inputs are. Since you will be using this code often in your labs, this will help you remember what everything does.

Now, that we have the function we need an initial guess of the intercept and gradient, which from the plot above look like roughly 0 and 10, respectively.

[6]:
```
# <!-- Student -->
init_params = [0.0, 10.0]
```

Now we have the functions and initial parameters defined we can call the `least_squares` method, passing in the data as `args`:

[7]:
```
# <!-- Student -->
from scipy.optimize import least_squares
result = least_squares(minimise, init_params, args=(xdata, ydata, xerror,␣
 ↪yerror))
print(result)
```

```
     message: `ftol` termination condition is satisfied.
     success: True
      status: 2
         fun: [ 1.567e-01 -9.440e-01  5.743e-01  7.912e-01  1.247e+00
                -5.584e-02 -1.457e+00 -7.314e-01  8.735e-02  6.729e-01]
           x: [-1.535e+00  1.024e+01]
        cost: 3.2745972129027687
         jac: [[-3.327e-01 -5.068e-01]
               [-4.903e-01 -1.105e+00]
               …
               [-4.729e-01 -4.002e+00]
               [-3.214e-01 -3.110e+00]]
        grad: [ 2.254e-07 -6.923e-07]
```

```
  optimality: 6.923129177509927e-07
 active_mask: [ 0.000e+00  0.000e+00]
        nfev: 5
        njev: 5
```

OK, we have run the fit, now what? We have a results object, which contains several pieces of information documented here

The first thing is to check if the fit was successful, using the `success` option of the result, and the status at termination, using the `status`.

[17]:
```
# <!-- Student -->

if not result.success or result.status < 1:
    print ("ERROR: Fit failed with message {}".format(result.message))
    print ("Please check the data and initial parameter estimates")
else:
    print ("Fit succeeded")
```

```
Fit succeeded
```

OK, the fit succeeded, so we can then get the fitted parameters via `result.x`, which is an array containing the resulting intercept (index 0) and slope (index 1).

The first thing we should do with these is to check if they are a good fit to the data by calculating the chi-squared (χ^2) test statistic, that was introduced in your PHYS106 lectures and in this week's lecture. To do this, we first evaluate our residuals function `minimise` for the fitted parameters. We can do this manually by passing the `result.x` values into `minimise`

```
final_params = result.x
residual_array = minimise(final_params, xdata, ydata, xerror, yerror)
```

but the `least_squared` result already contains this as `result.fun`. We then square this to get the chi-squared per data point and sum to get the total chi-squared.

More useful, is the reduced chi-squared (χ^2/NDF) by dividing by the number of degrees of freedom (i.e. number of points - number of fit parameters). The closer the χ^2/NDF is to one the better the fit. Generally, for small numbers of degrees of freedom, values in the range $0.25 < \chi^2/\text{NDF}\ \$ < 4\$$ represent good agreement.

A somewhat more robust fit statistic is the chi-squared probability, which is the probability that we get the resulting chi-squared given the number of degrees of freedom and should be greater than 5% (ie. within 95% or $\approx 2\sigma$). The `scipy.stats.chi2` module has a function (called the "survival function" or "sf") to calculate this.

[9]:
```
# <!-- Student -->

from scipy.stats import chi2 as stats_chi2

# Get fitted parameters
final_params = result.x
c = final_params[0]
```

```
m = final_params[1]
nparams = len(final_params)

# Calculate chi2
chi2_array = result.fun ** 2
chi2 = sum(chi2_array)
npoints = len(xdata)
reduced_chi2 = chi2 / (npoints - nparams)
chi2_prob = stats_chi2.sf(chi2, (npoints - nparams))

# Print chi2
np.set_printoptions(precision = 3)
print("\n=== Fit quality ===")
print("chisq per point = \n",chi2_array)
print("chisq = {:7.5g}, ndf = {}, chisq/NDF = {:7.5g}, chisq prob = {:7.5g}\n".
 ↪format(chi2, npoints-nparams, reduced_chi2, chi2_prob))

if reduced_chi2 < 0.25 or reduced_chi2 > 4:
    print("WARNING: chi2/ndf suspiciously small or large.  Please check the␣
 ↪data and initial parameter estimates")

if chi2_prob < 0.05:
    print("WARNING: chi2 probability for given degrees of freedom less than 0.
 ↪05 .  Please check the data and initial parameter estimates")
```

```
=== 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 =  6.5492, ndf = 8, chisq/NDF = 0.81865, chisq prob = 0.58596
```

Now that we have checked that it is a good fit, we can print out the parameters and their errors. We have the parameters already but we need to calculate the errors from the jacobian matrix, which is the matrix of the first order partial derivatives. We square this and take the inverse to get the covariance matrix, whose diagonal elements are the variance (i.e. error squared) of the parameters. In doing so, we check that the squared jacobian is not singular (i.e. has a finite determinant). You will learn the details of this in PHYS108 in semester 2 so for now you will just have to trust me. Note: NumPy has all the functions we need to deal with the matrix calculations.

[10]:
```
# <!-- Student -->

# Calculate errors
jacobian = result.jac
jacobian2 = np.dot(jacobian.T, jacobian)
determinant = np.linalg.det(jacobian2)

if determinant < 1E-32:
```

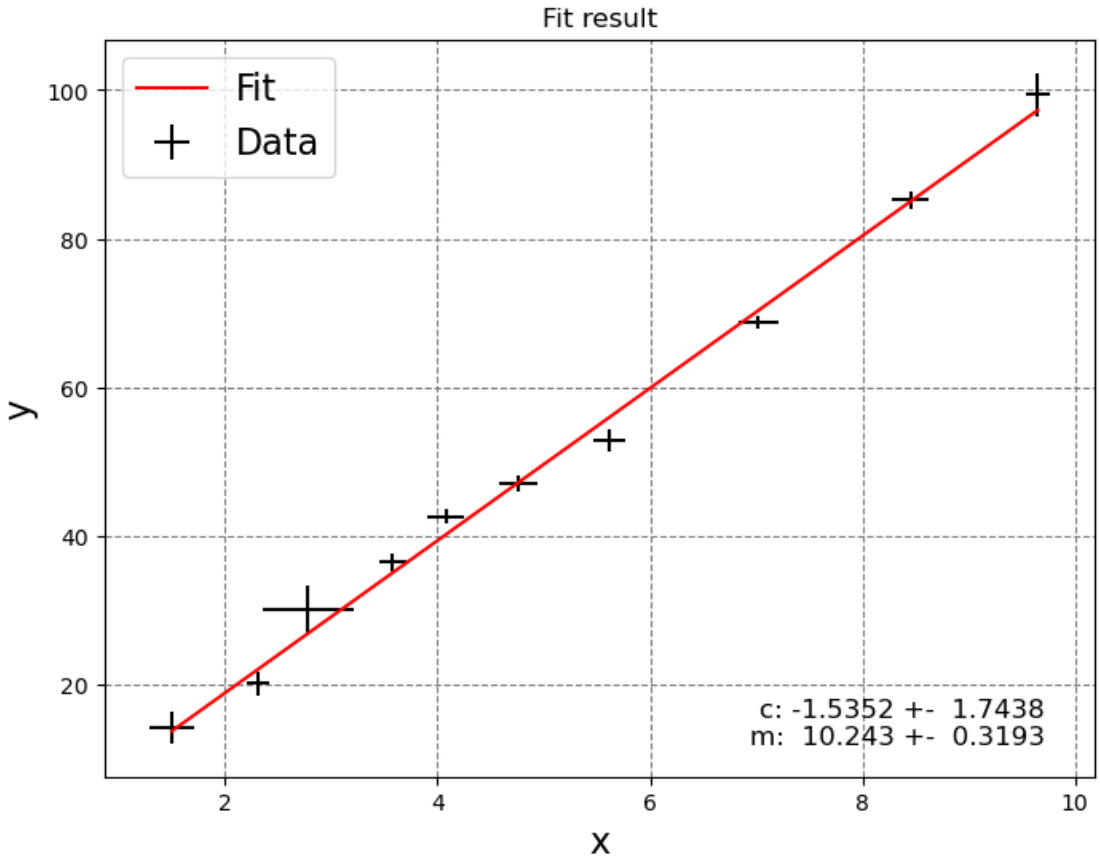
```
    print(f"Matrix singular (determinant = {determinant}, error calculation␣
 ↪failed.")
    param_errors = np.zeros(nparams)
else:
    covariance = np.linalg.inv(jacobian2)
    param_errors = np.sqrt(covariance.diagonal())

print ("c = {:7.5g} +- {:7.5g}".format(final_params[0], param_errors[0]))
print ("m = {:7.5g} +- {:7.5g}".format(final_params[1], param_errors[1]))
```

```
c = -1.5352 +-  1.7438
m =  10.243 +-  0.3193
```

1.2.5 Visualising the result

Finally, we plot the data with error bars (using the routine `plt.errorbar` from `matplotlib.pyplot`) and draw the fitted line, which we obtain by evaluating our `straight_line` function with the `final_params` on the plot (using `plt.plot`). We also use the plt.text function to print the fitted parameter values on the plot so everything is in one place.

[11]:
```
# Calculate fitted function values
yfit = straight_line(final_params, xdata)

fig = plt.figure(figsize = (8, 6))
plt.title('Fit result')
plt.xlabel('x', fontsize=16)
plt.ylabel('y', fontsize=16)
plt.grid(color = 'grey', linestyle="--")

# Plot data (no line connecting the points!)
plt.errorbar(xdata, ydata, xerr = xerror, yerr = yerror, fmt='k', linestyle =␣
 ↪'', label = "Data")

# Plot fit
plt.plot(xdata, yfit, color = 'r', linestyle = '-', label = "Fit")

# Add legend and fit params
plt.legend(loc = 2, fontsize=16)

text = "c: {:7.5g} +- {:7.5g}\n".format(final_params[0], param_errors[0])
text += "m: {:7.5g} +- {:7.5g}\n".format(final_params[1], param_errors[1])
plt.text(0.95, 0.0, text, transform = fig.axes[0].transAxes, ha = "right", va =␣
 ↪"bottom", fontsize=12)

plt.show()
```



The data look to be reasonably well described by a straight line. The value of the statistic χ^2/NDF and the χ^2 probability backs this up.

1.2.6 Lecture 6 formative exercise 2

Fit a straight line to the data in Table 2 (which is the same as Table 1 with two new points added). Do this by first adding the new data points to the arrays, then copying the relevant fitting cells above and pasting them below this cell (use the *Edit* menu). Unlike above, where we have done the fit in several cells so that we can explain what is happening as we go, bring the code together in a **single** cell.

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	3.58	0.13	36.5	1.1
5	4.08	0.17	42.7	0.9
6	4.76	0.18	47.1	1.1
7	5.62	0.15	52.9	1.5
8	7.02	0.19	68.8	0.9

Measurement	x value	Error in x	y value	Error in y
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 2 *Extended range of data points*

Notes: * To extend the data you can either make a new csv file, and read it in like above, or you can use the np.append method. * You don't need to copy the definitions of `straight_line`, `straight_line_diff` and `minimise` since they are already defined once you have run the corresponding cells and do not change.

1.2.7 Lecture 6 formative exercise 3

Does the straight line still describe the data? Have the values of the gradient or the intercept changed significantly? Be quantitative in your answer.

Note: two values can be said to be consistent if the difference between them is less than 3 times their combined error (this is called the "consistency check" in PHYS106)

Since you will use this code many times, it is useful to package it up into a function (see below) that we can call, which simply takes in the x and y data and their errors, along with the initial parameters. From now on whenever you need to do a fit you can just copy this function and call it with the correct arguments).

[15]:
```
# <!-- Student -->

def fit(xdata, ydata, xerror, yerror, init_params,
        xlabel = "x", ylabel = "y", title = "Fit result"):
    """
    Function to perform least-squares fit for a straight line.
    - xdata          array of x data points
    - ydata          array of y data points
    - xerr           array of x data errors
    - yerr           array of y data errors
    - init_params    array of function parameters
    - xlabel         x-axis label
    - ylabel         y-axis label
    - title          plot title
    """

    # Run fit
    result = least_squares(minimise, init_params, args=(xdata, ydata, xerror,␣
 ↪yerror))

    # Check fit succeeds
    if not result.success or result.status < 1:
```

```
        print ("ERROR: Fit failed with message {}".format(result.message))
        print ("Please check the data and inital parameter estimates")
        return 0,0
    else:
        print ("Fit succeeded")

    # Get fitted parameters
    final_params = result.x
    c = final_params[0]
    m = final_params[1]
    nparams = len(final_params)

    # Calculate chi2
    chi2_array = result.fun ** 2
    chi2 = sum(chi2_array)
    npoints = len(xdata)
    reduced_chi2 = chi2 / (npoints - nparams)
    chi2_prob = stats_chi2.sf(chi2, (npoints - nparams))

    # Print chi2
    np.set_printoptions(precision = 3)
    print("\n=== Fit quality ===")
    print("chisq per point = \n",chi2_array)
    print("chisq = {:7.5g}, ndf = {}, chisq/NDF = {:7.5g}, chisq prob = {:7.
↪5g}\n".format(chi2, npoints-nparams, reduced_chi2, chi2_prob))

    if reduced_chi2 < 0.25 or reduced_chi2 > 4:
        print("WARNING: chi2/ndf suspiciously small or large.  Please check the␣
↪data and initial parameter estimates")

    if chi2_prob < 0.05:
        print("WARNING: chi2 probability for given degrees of freedom less than␣
↪0.05 .  Please check the data and initial parameter estimates")

    # Calculate errors
    jacobian = result.jac
    jacobian2 = np.dot(jacobian.T, jacobian)
    determinant = np.linalg.det(jacobian2)

    if determinant < 1E-32:
        print(f"Matrix singular (determinant = {determinant}, error calculation␣
↪failed.")
        param_errors = np.zeros(nparams)
    else:
        covariance = np.linalg.inv(jacobian2)
        param_errors = np.sqrt(covariance.diagonal())
```

```
    print ("=== Fitted parameters ===")
    print ("c = {:7.5g} +- {:7.5g}".format(final_params[0], param_errors[0]))
    print ("m = {:7.5g} +- {:7.5g}".format(final_params[1], param_errors[1]))

    # Calculate fitted function values
    yfit = straight_line(final_params, xdata)

    # Visualise result
    fig = plt.figure(figsize = (8, 6))
    plt.title(title)
    plt.xlabel(xlabel, fontsize=16)
    plt.ylabel(ylabel, fontsize=16)
    plt.grid(color = 'grey', linestyle="--")

    plt.errorbar(xdata, ydata, xerr = xerror, yerr = yerror, fmt='k', linestyle␣
↪= '', label = "Data")
    plt.plot(xdata, yfit, color = 'r', linestyle = '-', label = "Fit")

    plt.legend(loc = 2, fontsize=16)

    text = "c: {:7.5g} +- {:7.5g}\n".format(final_params[0], param_errors[0])
    text += "m: {:7.5g} +- {:7.5g}\n".format(final_params[1], param_errors[1])
    plt.text(0.95, 0.0, text, transform = fig.axes[0].transAxes, ha = "right",␣
↪va = "bottom", fontsize=12)

    plt.show()

    # Return arrays of parameters and associated errors
    return final_params, param_errors
```

1.2.8 Lecture 6 summative exercise

So far the data we have been using is imaginary, as we concentrated on the tools to do the fitting. In this exercise you will use them on real data from an experiment on Boyle's law. The data, consisting of a series of measurements of pressure (P) [in $Pa = N/m^2$] and volume (V) [in m^3] and their uncertainties (σ_P and σ_V) taken at fixed temperature, can be found in the file `boyle.csv` (in the order V, σ_V, P, σ_P). Read this data and use the `fit` function to show that it conforms to Boyle's law i.e. "PV = k", finding the best-fit proportionality constraint, k. Comment on the goodness of fit.

Notes: * Think about how to make this into something linear that is easy to fit and don't forget to propagate the errors correctly following what you have leant in PHYS106 and/or the formulae here. All steps should be done in python. * You will likely need to plot the data first to find the initial parameters

1.3 Summary

That's the end of Week 3. You should now know to use the `least_squares` function from SciPy, along with NumPy arrays, to fit experimental data, visualising the results using Matplotlib. You will use this technique and the code we have developed in PHYS106 and throughout your labs in later years. We concentrated on simple examples of fitting straight lines, since this is what you will mostly encounter in PHYS106 and many problems can be made linear by a change of variables (as in Ex 4). However, the code we have developed is much more powerful: it can be used to fit any function as long as we can write down the functional form and its derivative. We will come back to fitting later in the course.

Now that our programs are getting more complex, next week we will take a step back and look at good coding practice and debugging errors, before going on to look at more real physics use cases.