End of module questionnaire

- This is your final chance to have a say on how the course is run and raise any issues you may have
  - Also, improving it for future student cohorts
- We will take your views on board
  - Reporting back on common issues
  - Making changes to alleviate them where possible
- If not done, please take 5 mins to fill this in now
  - https://liverpool.surveys.evasysplus.co.uk/

Introduction to Computational Physics (PHYS105)

Lecture 11: Modules and Fitting Revisited

Carl Gwilliam

(C.Gwilliam@liverpool.ac.uk)

UNIVERSITY OF LIVERPOOL

Attendance code: 908651

Lecture 10: Recap

- Last week we looked at at category of numeric techniques known as Monte Carlo (MC) Methods
  - Which use random sampling to obtain results by performing many trials or experiments

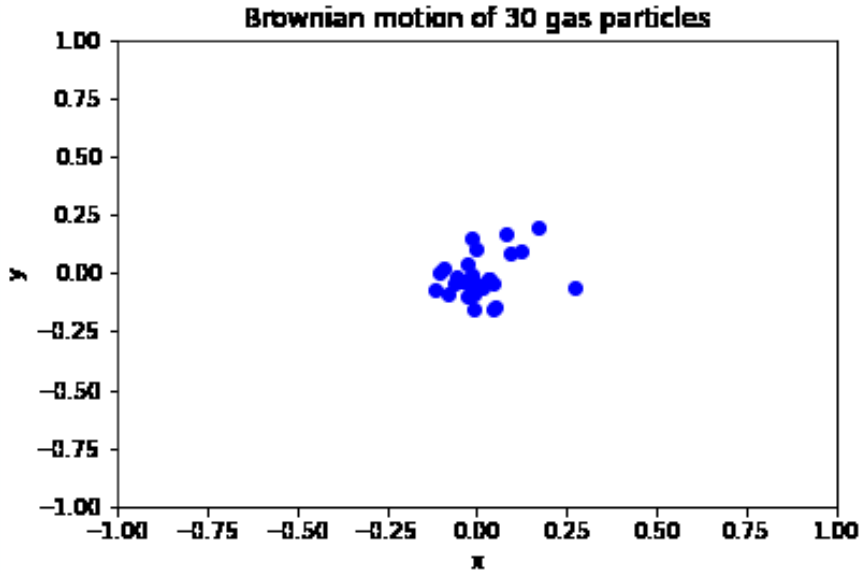
- In doing so, we saw how to generate random nums following a given distribution in Python

```python
import numpy as np
np.set_printoptions(2)

# initialise random number generator
rng = np.random.default_rng()

print("10 uniformly-distributed random numbers:")
print(rng.random(10))

print("\n10 Poisson-distributed random numbers:")
mean = 10
print(rng.poisson(mean, 10))

print("\n10 Gaussian-distributed random numbers:")
mean = 10
RMS = 1
print(rng.normal(mean, RMS, 10))
```

```
10 uniformly-distributed random numbers:
[0.02 0.71 0.31 0.12 0.49 0.07 0.5  0.37 0.87 0.24]

10 Poisson-distributed random numbers:
[12  9 13 11 10  9 15  4  9  4]

10 Gaussian-distributed random numbers:
[10.6   9.9  10.02  9.4  11.65  9.52  9.04  9.91 10.75  9.6 ]
```

- Applied MC approach to simple problems
  1. Define range over which inputs can lie
  2. Generate many random inputs over range
  3. Perform deterministic computation on these

- Can be easily extended to more complex cases



Lecture 10: Formative problem solutions

- Exercise 1: Check how the seed works by running with the same and different seeds
  - While the sequence of numbers is random, we often want to be able to repeat it again
    - E.g. so we can compare results, debug etc
  - The seed allows this by setting the point within the random sequence where we start
    - If the seed is the same, we start at the same point → get the same random sequence
    - If we change the seed, we start at a different point → hence get a different sequence
  - If we don't set a seed, we start at a random point in the sequence of random numbers
    - Hence getting a different set of random numbers each time

```python
def make_random_numbers(seed):

    # initalise random number generator with seed
    rng = np.random.default_rng(seed)

    # generate one random number
    x = rng.random()
    print("x =",x)

    # generate an array of 3 random numbers
    x1D = rng.random(3)
    print("x1D =",x1D, "\n")
```

Function to aid testing multiple times

```python
print("Random nums with seed = 1327:")
make_random_numbers(1327)
make_random_numbers(1327)
make_random_numbers(1327)
```

```
Random nums with seed = 1327:
x = 0.054832973632178206
x1D = [0.69034902 0.56377212 0.31681625]

x = 0.054832973632178206
x1D = [0.69034902 0.56377212 0.31681625]

x = 0.054832973632178206
x1D = [0.69034902 0.56377212 0.31681625]
```

Same seed = same result

```python
print("Random nums with seed = 3000:")
make_random_numbers(3000)
make_random_numbers(3000)
make_random_numbers(3000)
```

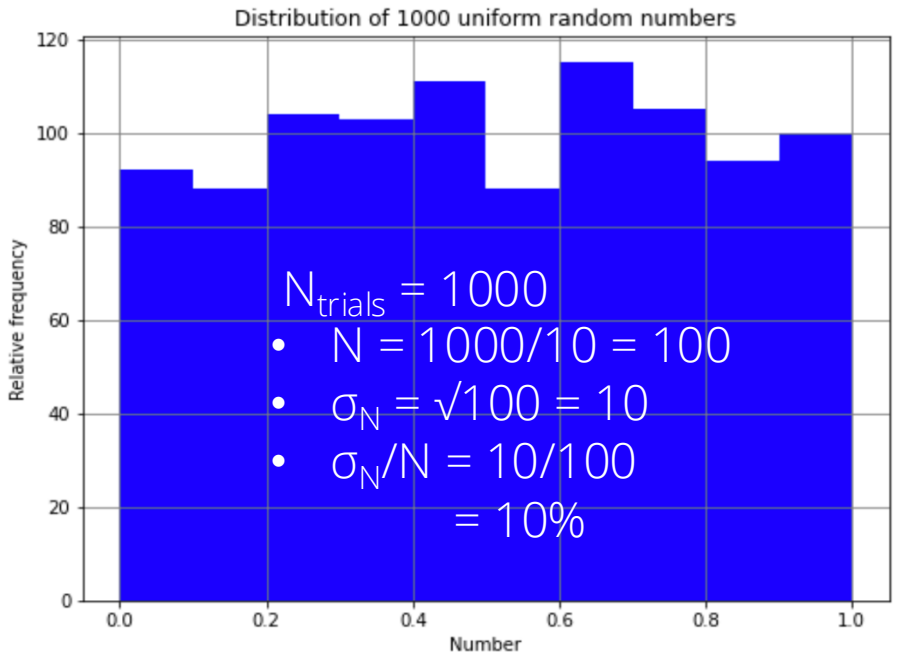
```
Random nums with seed = 3000:
x = 0.43651543692004446
x1D = [0.32807961 0.42113886 0.19423973]

x = 0.43651543692004446
x1D = [0.32807961 0.42113886 0.19423973]

x = 0.43651543692004446
x1D = [0.32807961 0.42113886 0.19423973]
```

Diff. seed = diff. result

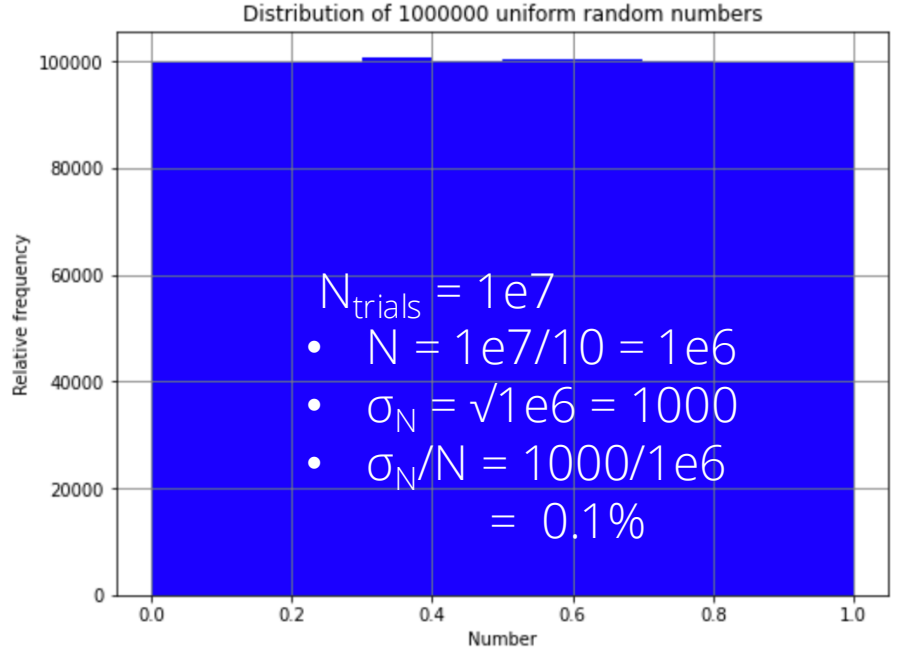
Lecture 10: Formative problem solutions (2)

- Exercise 2: Investigate the distribution of random numbers as we increase the number of trials
  - Numbers are spread uniformly between 0 and 1 → expect N_{trials} / N_{bins} in each bin on average
  - But, since they are random, there is of course fluctuation around this average
  - The more trials you run the less fluctuation there is
    - In fact, the error on a given bin content N is approximately $\sigma_N = \sqrt{N}$ (following the central limit theorem)

```
plot_random_numbers(1000)
```




```
plot_random_numbers(1000000)
```



Lecture 10: Formative problem solutions (3)

- Exercise 3: Determine the error on the MC estimate of π to see if it is consistent with the known value
  - For the error we simply code the derived formula into python using NumPy

$$\Delta\pi = 4\sqrt{\frac{pq}{N}}$$
$$= 4\sqrt{\frac{p(1-p)}{N}}$$
$$= 4\sqrt{\frac{N_C/N(1-N_C/N)}{N}}$$

p = probability of being within circle
q = probability of being outside circle

  - To check if it it consistent you need to use the consistency check we covered in lecture 6

$$\text{Significance} = \frac{|x_1 - x_2|}{\sqrt{\sigma_{x1}^2 + \sigma_{x2}^2}} < 3$$

```python
# Generate independent random x and y values in the range [0,1)
nGrains = 10000

print(f"Performing {nGrains:d} random trials ...")
rng = np.random.default_rng()
riceX = rng.random(nGrains)
riceY = rng.random(nGrains)

# Calculate the radial position using r**2 = x**2 + y**2
riceR = np.sqrt(riceX**2 + riceY**2)

# Find how many are inside the radius of the circle
nCircle = np.sum(riceR < 1)
print(f"Of which {nCircle:d} are within the circle")

# Calculate pi and it's error
pi = 4*nCircle/nGrains
dPi = 4*np.sqrt(nCircle/nGrains*(1 - nCircle/nGrains)/nGrains)
print(f"Giving pi estimate as {pi:8.7f} +- {dPi:8.7f}")

# Check the consistency by finding the number of standard
# deviations we are away and require to be less than 3 sigma
consistency = abs(pi - np.pi)/dPi
print("Deviation in terms of error is {:8.7f}.".format(consistency))
if consistency < 3:
    print("Calculated value of pi is consistent with expectation.")
else:
    print("Calculated value of pi is not consistent with expectation.")
```

```
Performing 10000 random trials ...
Of which 7891 are within the circle
Giving pi estimate as 3.1564000 +- 0.0163179
Deviation in terms of error is 0.9074296.
Calculated value of pi is consistent with expectation.
```

Lecture 10: Formative problem solutions (4)

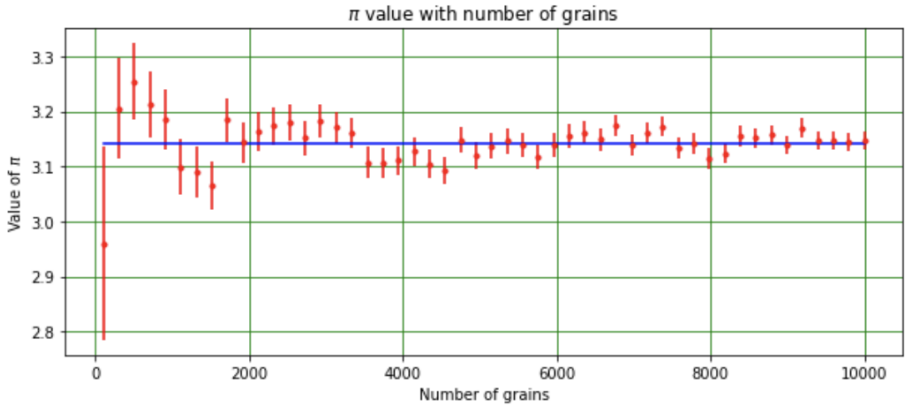
- Exercise 4: Fix the bug to plot the uncertainty on π versus the number of grains
  - The issue is that the `grainArr` contains floats, while `rng.random()` expects ints → convert
    - E.g. using `astype(int)` or `int()`

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-12-0b935c68467d> in <module>
     31
     32 for n in range(0, nSteps):
---> 33     pi, dPi = pimc(grainArr[n])
     34     piArr[n] = pi
     35     dPiArr[n] = dPi

<ipython-input-12-0b935c68467d> in pimc(nGrains)
      5
      6     # Generate independent random x and y values in the range [0,1) for nGrains of rice
----> 7     riceX = rng.random(nGrains)
      8     riceY = rng.random(nGrains)
      9

_generator.pyx in numpy.random._generator.Generator.random()

_common.pyx in numpy.random._common.double_fill()

TypeError: 'numpy.float64' object cannot be interpreted as an integer
```

- Once fixed then we see that, as we should expect from Ex2, the more grains we throw the better our precision on the estimate of π
  - i.e smaller uncertainty

```
# Create array of grains from 100 to 1000 in 50 steps
minGrains = 100
maxGrains = 10000
nSteps = 50

# Force grainArr to be an array of integers.
grainArr = np.linspace(minGrains, maxGrains, nSteps)
print(grainArr[:10])
grainArr = grainArr.astype(int)

# Calculate pi for each step and store the result
piArr = np.zeros(nSteps)
dPiArr = np.zeros(nSteps)

for n in range(0, nSteps):
    pi, dPi = pimc(grainArr[n])
    piArr[n] = pi
    dPiArr[n] = dPi

# Plot the result
plt.figure(figsize = (10,4))
plt.title("$\pi$ value with number of grains")
plt.xlabel("Number of grains")
plt.ylabel("Value of $\pi$")
plt.errorbar(grainArr, piArr, yerr = dPiArr, color = 'r', marker = '.', linestyle = '')
plt.plot(grainArr, np.pi*np.ones(nSteps), color = 'b', marker = '', linestyle = '-')
plt.grid(color = 'g')
plt.show()
```

```
[ 100.          302.04081633  504.08163265  706.12244898  908.16326531
 1110.20408163 1312.24489796 1514.28571429 1716.32653061 1918.36734694]
```



Lecture 10: Formative problem solutions (5)

- Exercise 5: Add comments to the code, explaining each line of the for loop in particular

```
# Set maximum number of beds
maxBedNumber = int(2.5*nPatsDay)

# Array of numbers of times there are no beds available
noBedArr = np.zeros(maxBedNumber)
print("Number of beds available \t No. days one or more patients has no bed")

# Loop over possible numbers of beds
for nBeds in range(0, maxBedNumber):
    # Calculate the number of times one or more patients has no bed. The condition
    # patsPerDay > nBeds is True whenever this occurs. Adding up the number of Trues
    # in the arrays (True = 1) gives the number of times this has occurred in the
    # winter under study.
    nWithoutBed = np.sum(patsPerDay > nBeds)

    # Save the value so it can be plotted
    noBedArr[nBeds] = nWithoutBed

    # Provide a printout every time nBeds is divisible by 5.
    if nBeds%5 == 0:
        print("\t\t {:d} \t\t\t\t {:d}".format(nBeds, nWithoutBed))

# Create an array containing the numbers of beds available for the plot.
bedNumberArr = np.linspace(0, maxBedNumber - 1, maxBedNumber)

# Plot the resulting distribution
plt.figure(figsize = (6, 5))
plt.title("Study of A&E capacity")
plt.xlabel("No. of beds")
plt.ylabel("No. of days patient has no bed")
plt.plot(bedNumberArr, noBedArr, color = 'b', label = "No. times patient has no bed")
plt.plot(nPatsDay, 0.0, color = 'r', marker = '^', label = "Avg. number patients per day")
plt.legend()
plt.grid(color = 'g')
plt.show()
```

Moving beyond Jupyter notebooks

- During the course, we have been writing our Python code exclusively in Jupyter notebooks
  - Which are a great resource for learning + developing code interactively and disseminating the results
- However, they also have some disadvantages, especially for writing longer programs
  - Large notebooks can be messy and are slow to both open and run
  - There is a large potential for conflicts between piece of code e.g. overwriting names
  - You often need to run all the cells and the results may depend on the order in which they're run
  - If you want to reuse code elsewhere you have to copy-and-paste it into a new notebook
- As you write more complex problems you will want to split things up into modular chunks
  - Each of which does a specific task, allowing you to focus on the problem at hand
- Also, going forward, you will increasingly want to reuse code you have developed
  - Both code from within PHYS105 (e.g. fitting code) and also future courses
- This doesn't mean we need to give up on Jupyter notebooks entirely
  - We can have the best of both worlds by moving reusable pieces of code into **modules**

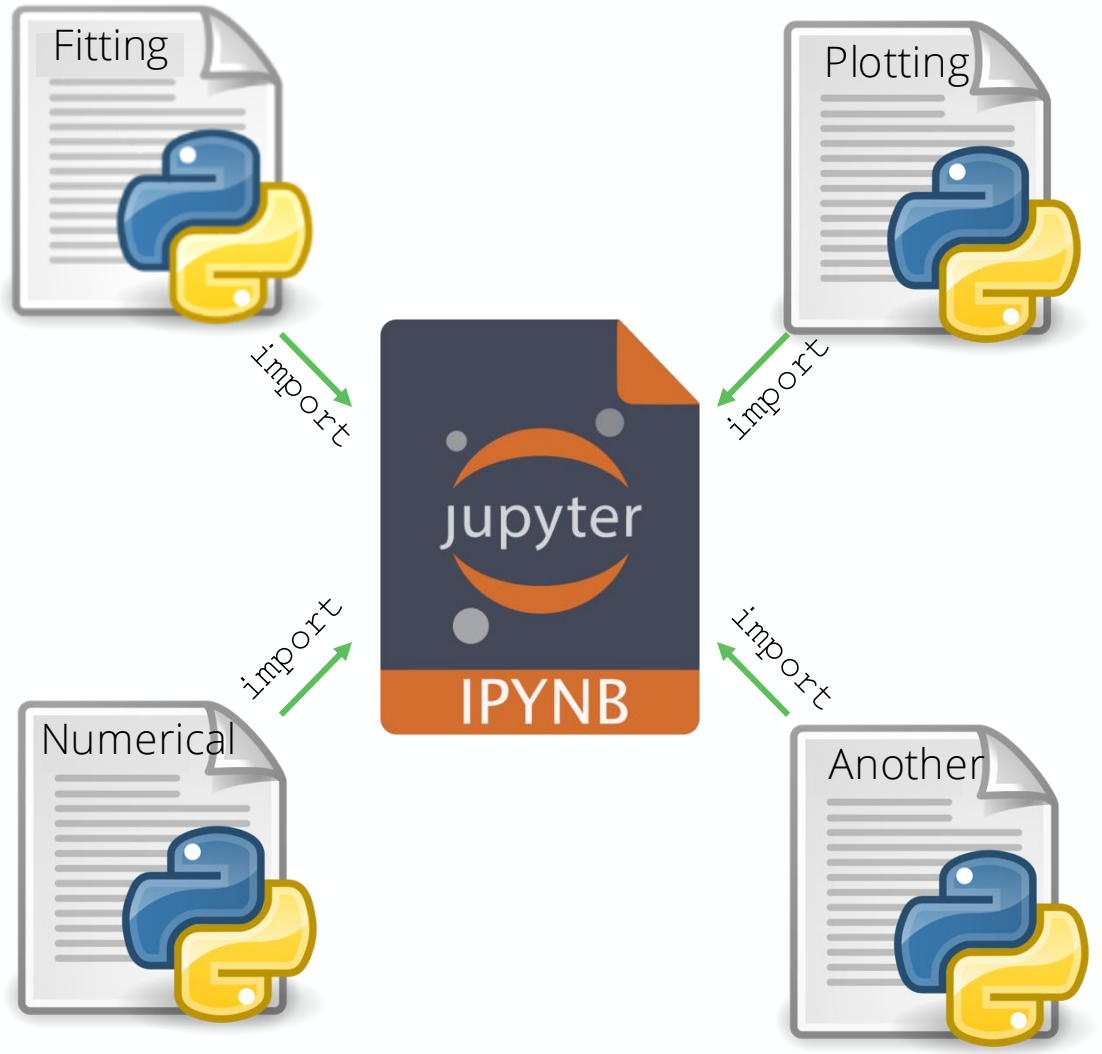
Python modules

- Python allows us to write programs in a modular way by separating code into modules

- This module approach has several advantages
  - Simplicity: a module can focus on a particular problem or a small portion of a larger problem
  - Maintainability: smaller chunks are easier to maintain and reduce the possibility of conflicts
  - Reusability: the code we have written can be reused in many notebooks without duplicating

- You have already used many Python modules
  - Built-in modules e.g. `math`
  - Third-part modules e.g `numpy, scipy`
  - Modules I wrote e.g. `LabModule`

- Will now see how you can write your own modules
  - That you can add to and reuse in the future



Creating your own module

- Creating a python module is actually very easy
  - It is just a file containing valid python code
  - Written in any editor you like (inc. CoCalc)
  - Saved with a .py file extension

- It can contain variables, data structures, functions etc
  - And can import and use other modules too

mymodule.py

```
1  import numpy as np
2
3  # Variables
4  string = "This is my first module"
5  number = 10
6  data = np.linspace(0, 10, 10)
7
8  # Functions
9  def func(arg):
10     print (f"The argument was {arg}")
11     return
```

```
import mymodule as mod
```

```
print(mod.string)
print(mod.number)
print(mod.data)
```

```
This is my first module
10
[ 0.    1.11  2.22  3.33  4.44  5.56  6.67  7.78  8.89 10.  ]
```

```
mod.func(100)
```

```
The argument was 100
```

- To use our module, we just place it in our current working directory
  - I.e. same directory as the notebook

- We then simply import it, just like any of the other modules we have used
  - Note: don't include .py suffix

Finding the module

- Having to keep our modules in the same directory as our notebook is not very useful
  - Particularly if we want to be able to reuse these in other notebooks or places

- Luckily, python has a way of specifying where it looks for files not in the current directory
  - Which is specified by the `sys.path`

- By default, it contains a system-dependent list of paths from your python installation
  - But can append new paths, either relative or absolute, to it to add new locations

- E.g. If we put our module in a libs directory we simply add this directory to the `sys.path`
  - Then can import is as if it's in the current dir.

Default CoCalc sys.path

```
1 import sys
2 sys.path
```

```
['/home/user/Phys105 Introduction to Computational Physics',
 '/ext/anaconda2020.02/lib/python37.zip',
 '/ext/anaconda2020.02/lib/python3.7',
 '/ext/anaconda2020.02/lib/python3.7/lib-dynload',
 '',
 '/ext/anaconda2020.02/lib/python3.7/site-packages',
 '/ext/anaconda2020.02/lib/python3.7/site-packages/IPython/extensions',
 '/home/user/.ipython']
```

Adding a new relative path

```
1 sys.path.append("libs")
2 sys.path
```

```
['/home/user/Phys105 Introduction to Computational Physics',
 '/ext/anaconda2020.02/lib/python37.zip',
 '/ext/anaconda2020.02/lib/python3.7',
 '/ext/anaconda2020.02/lib/python3.7/lib-dynload',
 '',
 '/ext/anaconda2020.02/lib/python3.7/site-packages',
 '/ext/anaconda2020.02/lib/python3.7/site-packages/IPython/extensions',
 '/home/user/.ipython',
 'libs']
```

- So can move useful code into a central module and simply import it into any notebook we want

Finding the module

- Having to keep our modules in the same directory as our notebook is not very useful
  - Particularly if we want to be able to reuse these in other notebooks or places

- Luckily, python has a way of specifying where it looks for files not in the current directory
  - Which is specified by the `sys.path`

- By default, it contains a system-dependent list of paths from your python installation
  - But can append new paths, either relative or absolute, to it to add new locations

- E.g. If we put our module in a libs directory we simply add this directory to the `sys.path`
  - Then can import is as if it's in the current dir.

Default CoCalc sys.path

```
1 import sys
2 sys.path
```

```
['/home/user/Phys105 Introduction to Computational Physics',
 '/ext/anaconda2020.02/lib/python37.zip',
 '/ext/anaconda2020.02/lib/python3.7',
 '/ext/anaconda2020.02/lib/python3.7/lib-dynload',
 '',
 '/ext/anaconda2020.02/lib/python3.7/site-packages',
 '/ext/anaconda2020.02/lib/python3.7/site-packages/IPython/extensions',
 '/home/user/.ipython']
```

Adding a new absolute path

```
1 sys.path.append("/home/user/libs")
2 sys.path
```

```
['/home/user/Phys105 Introduction to Computational Physics',
 '/ext/anaconda2020.02/lib/python37.zip',
 '/ext/anaconda2020.02/lib/python3.7',
 '/ext/anaconda2020.02/lib/python3.7/lib-dynload',
 '',
 '/ext/anaconda2020.02/lib/python3.7/site-packages',
 '/ext/anaconda2020.02/lib/python3.7/site-packages/IPython/extensions',
 '/home/user/.ipython',
 'libs',
 '/home/user/libs']
```

- So can move useful code into a central module and simply import it into any notebook we want

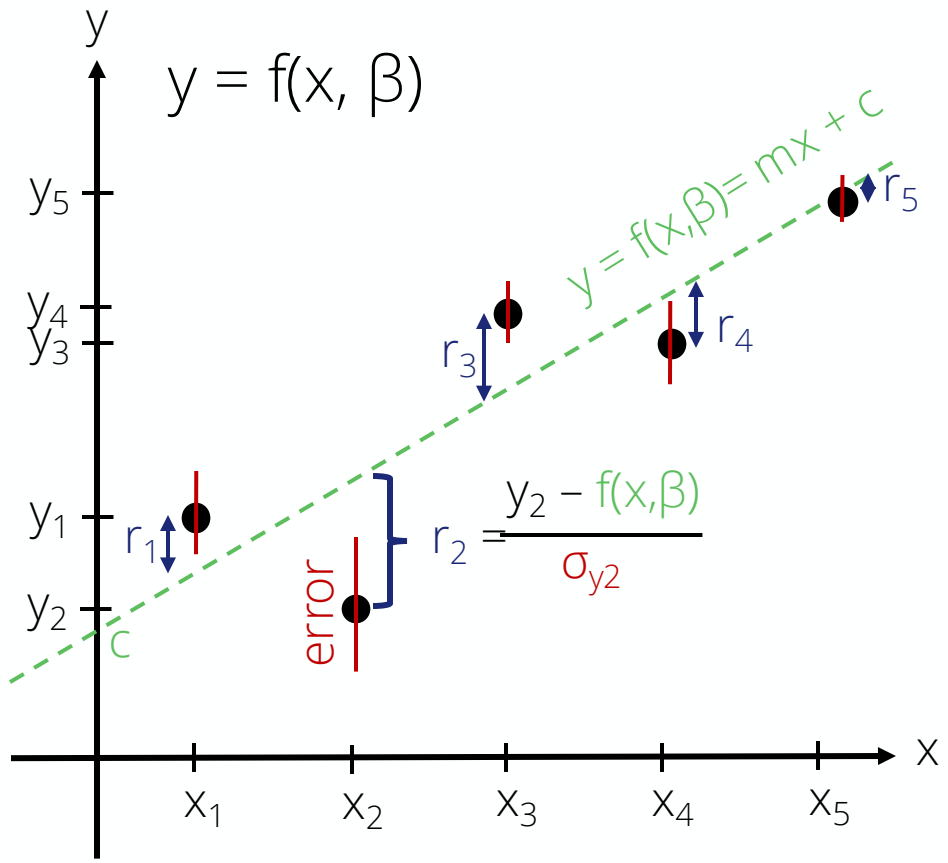
Reminder of fitting

- We perform fits to experimental data to determine the relationship between x and y
  - Given the functional form, we can find the values of the unknown params that best describe the data

- We determine the best fit by minimising the residuals
  - i.e. the distance between the data point and the model prediction at that x value, divided by the point's error

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

- We performed the fit using the least-squares method
  - Which is available from the `scipy.optimize` module




```
scipy.optimize.least_squares(fun, x0, args=())
```

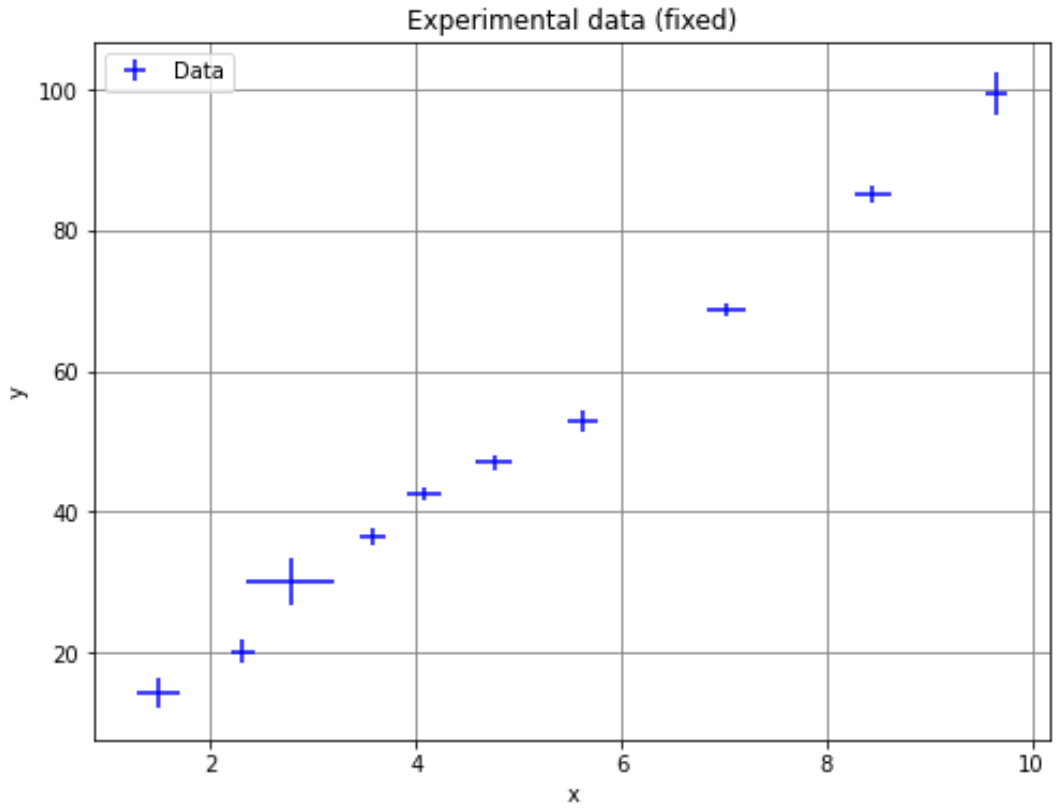
Function computing point-by-point residuals to be minimised

Initial guess of free parameters

Tuple of input data (x, y, σ_x, σ_y) to minimise with respect to

Fitting procedure steps

- Read in your data, usually from a file
  - Plot to check sensible
  - And find initial params




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

```
(array([1.5 , 2.31, 2.78, 3.58, 4.08, 4.76, 5.62, 7.02, 8.45, 9.65]),
 array([0.21, 0.11, 0.43, 0.13, 0.17, 0.18, 0.15, 0.19, 0.17, 0.11]),
 array([14.3, 20.2, 30.1, 36.5, 42.7, 47.1, 52.9, 68.8, 85.2, 99.4]),
 array([2.1, 1.7, 3.3, 1.1, 0.9, 1.1, 1.5, 0.9, 1.2, 2.9]))
```

```
import matplotlib.pyplot as plt
plt.figure(figsize = (8, 6))
plt.title("Experimental data (fixed)")
plt.xlabel("x")
plt.ylabel("y")
plt.errorbar(xdata, ydata, xerr=xerror, yerr=yerror, color = 'b',
             linestyle = '', label = 'Data')
plt.grid(color = 'grey')
plt.legend(loc = 2)
plt.show()
```

```
init_params = [0.0, 10.0]
```

Fitting procedure steps

- Read in your data, usually from a file
  - Plot to check sensible
  - And find initial params
- Define the necessary functions
  - Functional form and its differential
  - Point-by-point residuals

```python
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
```

```python
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
```

```python
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
    - func      function describing data
    - diff      differential of function
    """

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

    return residuals
```

$$\frac{y_i - f(\beta, x_i)}{\sigma_{y_i}}$$

Fitting procedure steps

- Read in your data, usually from a file
  - Plot to check sensible
  - And find initial params
- Define the necessary functions
  - Functional form and its differential
  - Point-by-point residuals
- Run `least_squares()` from scipy.optimize
  - Check fit succeeds and get fitted params

```
from scipy.optimize import least_squares
result = least_squares(minimise, init_params,
                       args=(xdata, ydata, xerror, yerror))

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
```

```
final_params = result.x
c = final_params[0]
m = final_params[1]
nparams = len(final_params)
```

Fitting procedure steps

- Read in your data, usually from a file
  - Plot to check sensible
  - And find initial params
- Define the necessary functions
  - Functional form and its differential
  - Point-by-point residuals
- Run `least_squares()` from scipy.optimize
  - Check fit succeeds and get fitted params
- Calculate χ^2 test statistic to check fit is good
  - $0.25 < \chi^2/\mathrm{NDF} < 4$ and/or χ^2 prob > 0.05

```python
from scipy.stats import chi2 as stats_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
```

Fitting procedure steps

- Read in your data, usually from a file
  - Plot to check sensible
  - And find initial params
- Define the necessary functions
  - Functional form and its differential
  - Point-by-point residuals
- Run `least_squares()` from scipy.optimize
  - Check fit succeeds and get fitted params
- Calculate χ^2 test statistic to check fit is good
  - $0.25 < \chi^2/\text{NDF} < 4$ and/or χ^2 prob > 0.05
- Calculate parameter errors from Jacobian
  - Print best-fit parameters and associated error

```
jacobian = result.jac
jacobian2 = np.dot(jacobian.T, jacobian)
determinant = np.linalg.det(jacobian2)

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

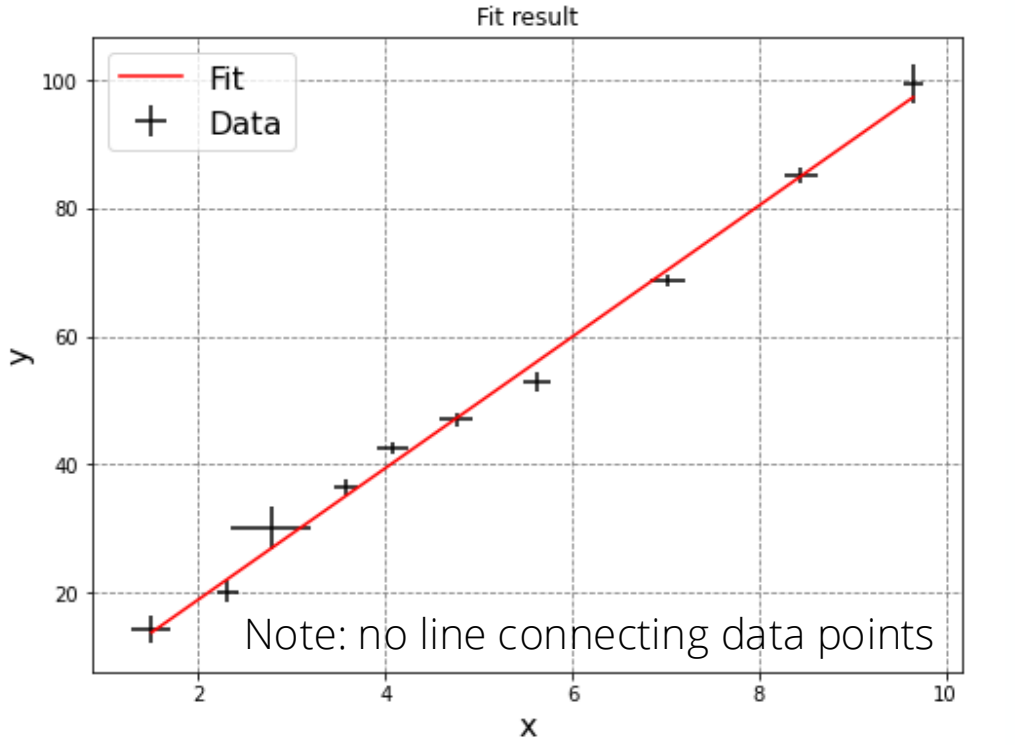
```
print ("c = {:7.5g} +- {:7.5g}"
       .format(final_params[0], param_errors[0]))
print ("m = {:7.5g} +- {:7.5g}"
       .format(final_params[1], param_errors[1]))
```

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

Fitting procedure steps

- Read in your data, usually from a file
  - Plot to check sensible
  - And find initial params
- Define the necessary functions
  - Functional form and its differential
  - Point-by-point residuals
- Run `least_squares()` from scipy.optimize
  - Check fit succeeds and get fitted params
- Calculate χ^2 test statistic to check fit is good
  - $0.25 < \chi^2/\text{NDF} < 4$ and/or χ^2 prob > 0.05
- Calculate parameter errors from Jacobian
  - Print best-fit parameters and associated error
- Plot data and fitted function
  - Passing best fit params to functional form

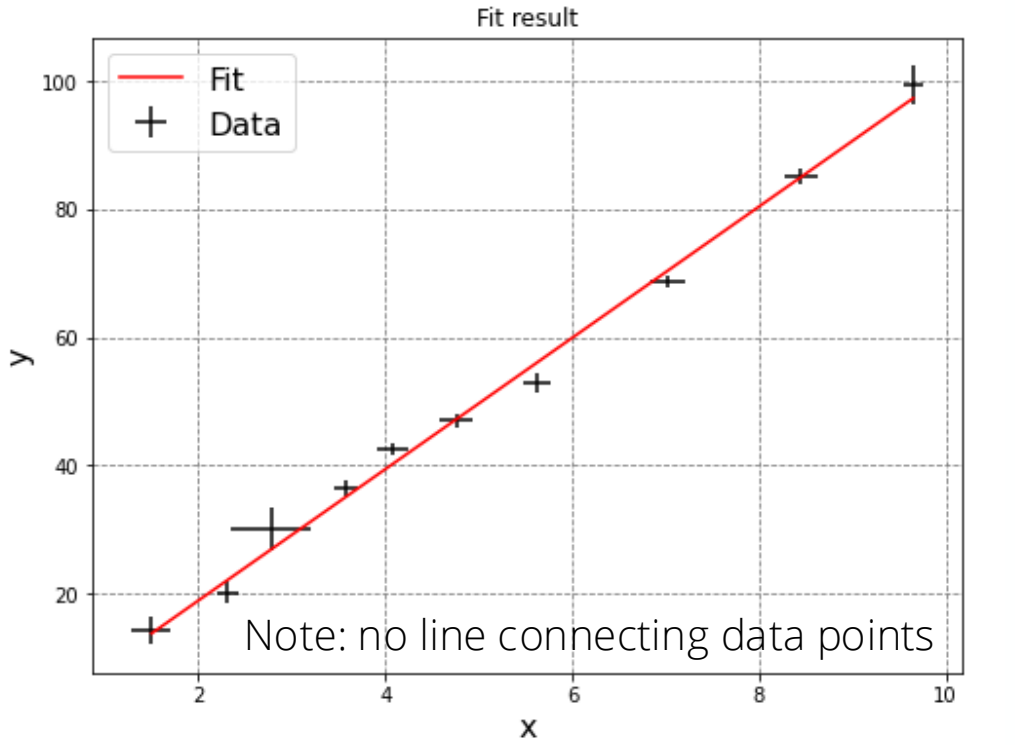
```
yfit = straight_line(final_params, xdata)

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="--")
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)
plt.show()
```



Fitting procedure steps

- Read in your data, usually from a file
  - Plot to check sensible
  - And find initial params
- Define the necessary functions
  - Functional form and its differential
  - Point-by-point residuals
- Run `least_squares()` from scipy.optimize
  - Check fit succeeds and get fitted params
- Calculate χ^2 test statistic to check fit is good
  - $0.25 < \chi^2/\text{NDF} < 4$ and/or χ^2 prob > 0.05
- Calculate parameter errors from Jacobian
  - Print best-fit parameters and associated error
- Plot data and fitted function
  - Passing best fit params to functional form
- Encapsulated into a reusable fit function in lecture 6

```
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="--")
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)
plt.show()
```



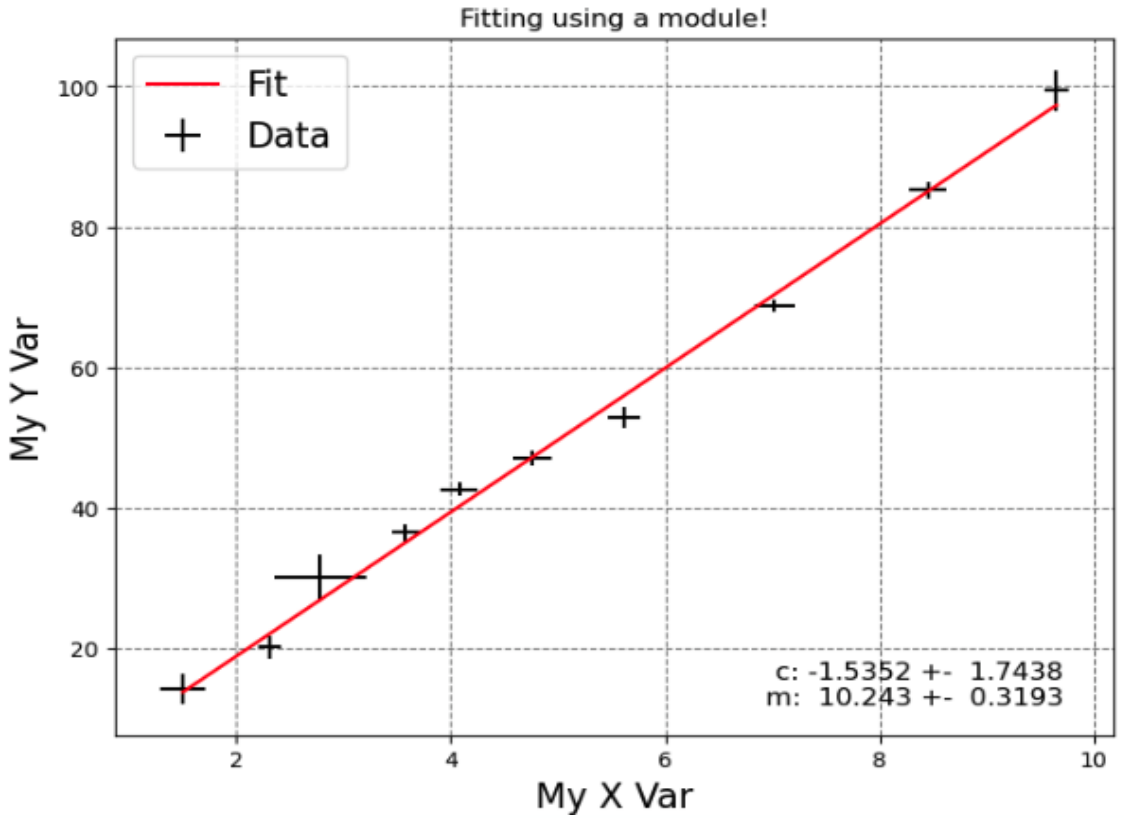
Fitting procedure steps

- Read in your data, usually from a file
  - Plot to check sensible
  - And find initial params
- Define the necessary functions
  - Functional form and its differential
  - Point-by-point residuals
- Run `least_squares()` from scipy.optimize
  - Check fit succeeds and get fitted params
- Calculate χ^2 test statistic to check fit is good
  - $0.25 < \chi^2/\mathrm{NDF} < 4$ and/or χ^2 prob > 0.05
- Calculate parameter errors from Jacobian
  - Print best-fit parameters and associated error
- Plot data and fitted function
  - Passing best fit params to functional form
- Encapsulated into a reusable fit function in lecture 6 → now move to a module for future use !

modules/fitting_line.py

```
# Add module location to sys path
import sys
sys.path.append("modules")

# import fit function from module
from fitting_line import fit

# Call with your data and initial params
fit(xdata, ydata, xerror, yerror, init_params,
    xlabel = "My X Var", ylabel = "My Y Var", title = "Fitting using a module!")
```

```
Fit succeeded

=== 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

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



Extending fitting

- So far, we've only looked at fitting straight lines, but it's straight forward to extend to other cases
  - Just need to know the function and its derivative

- For example, we can easily extend our code to be able to fit a generic polynomial:

$$f(x) = p_1 1x^0 + p_2 2x^1 + p_3 3x^2 \ldots$$

```
def polynomial_diff(params, xdata):
    "Differential of a polynomial of order `len(nparams)`"

    df = 0
    npol = len(params)

    # Loop over num params (skipping param 0)
    for i in range(1, npol):
        # Add a term p_i * i * x^(i-1)
        df += params[i] * i * xdata ** (i - 1)

    return df
```

$$r_i = \frac{y_i - f(\beta, x_i)}{\sigma_{y_i}}$$

$$f(x) = p_0 x^0 + p_1 x^1 + p_2 x^2 + p_3 x^3 \ldots$$

```
def polynomial(params, xdata):
    "Function for a polynomial of order `len(nparams)`"

    f = 0
    npol = len(params)

    # Loop over num params
    for i in range(npol):
        # Add a term p_i * x^i for each
        f += params[i] * xdata ** i

    return f
```

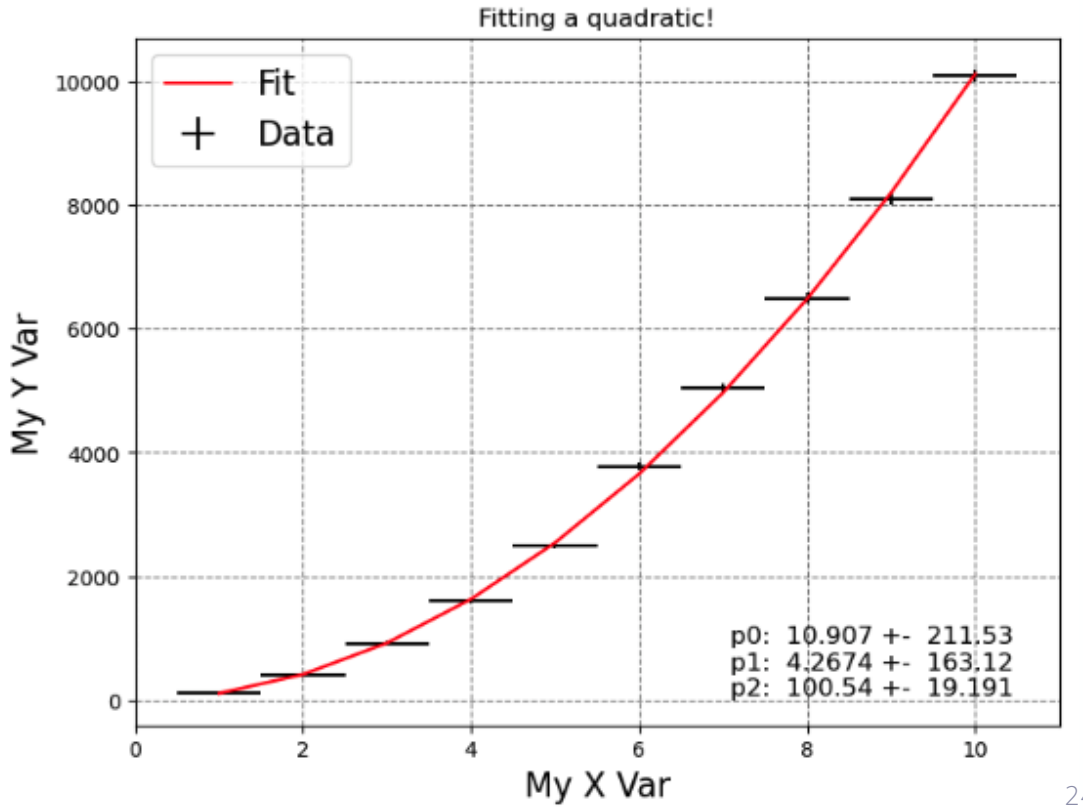
```
def residuals(params, xdata, ydata, xerr, yerr):
    """
    Calculation to minimise to find the best fit using chi2: difference between each y point and its
    prediction  by the function, divided by the sum in quadrature of the error on y, both from the
    y error and from the related error in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

    residual_array = (ydata - polynomial(params, xdata)) / (np.sqrt(yerr**2 + polynomial_diff(params, xdata)**2 * xerr**2))

    return residual_array
```

Polynomial fit

- We can then update our module with these polynomial functions to make it more general
  - `modules/fitting_poly.py`
- Call it exactly as before
  - Just remember to give the correct number of initial parameters for the functional form using
- E.g. fit a quadratic:

```
# Add module location to sys path
import sys
sys.path.append("modules")

# import updated fit function from module
from fitting_poly import fit

# Load quadratic data
import numpy as np
xdata2, xerror2, ydata2, yerror2 = np.loadtxt("fitdata_nonlinear.csv",
                                              delimiter = ',', unpack = True)

# 3 initial values now: p0 + p1 x + p2 x^2
init_params2 = [1.0, 10.0, 100.0]

# Call with your data and initial params
fit(xdata2, ydata2, xerror2, yerror2, init_params2,
    xlabel = "My X Var", ylabel = "My Y Var", title = "Fitting a quadratic!")
```

```
Fit succeeded

=== Fit quality ===
chisq per point =
 [1.541e-04 9.134e-06 3.370e-04 6.079e-03 1.004e-02 4.080e-02 1.018e-02
 2.247e-04 1.362e-02 4.286e-04]
chisq = 0.081868, ndf = 7, chisq/NDF = 0.011695, chisq prob =       1

WARNING: chi2/ndf suspiciously small or large.  Please check the data and initial
parameter estimates
=== Fitted parameters ===
param 0 =  10.907 +-  211.53
param 1 =  4.2674 +-  163.12
param 2 =  100.54 +-  19.191
```



How confident do you feel with programming in python (v2)?

Not confident at all - I am still struggling to understand the basics

0%

Somewhat confident - I understand the majority of the python features we have covered

0%

Confident - I understand the features and am happy writing simple programs

0%

Very confident - I am happy bring together different features into more complex programs

0%

Summary

- This week, we saw how to organise our code into modules, splitting it up & allowing it to be reused
  - At the same time, we also reminded ourselves about fitting
- That's pretty much it for the PHYS105 course
  - I hope it was enjoyable and useful!
- You now know the basics of programming in python
  - One of the most popular programming languages
- Used this to model real-world (physics) problems
  - Which you will use throughout your degree and beyond
- As a bonus, you go away with a fitting module
  - That you can reuse in labs + add to further

Merry Christmas!

Made in python using matplotlib
`MerryXmas.ipynb`

- The second year Computational Physics (PHYS205) module will build on what you've learnt
  - Allowing you to use python to undertake a project of your choosing.