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/

1

- **Before starting today's lecture, I'd like to give you 5 mins to fill in the end of module questionnaire …**
  - **Which this time it is electronic rather than paper based.**

Introduction to Computational Physics (PHYS105)

Lecture 11: Modules and Fitting Revisited

Carl Gwilliam
(C.Gwilliam@liverpool.ac.uk)

Attendance code: 908651

- **Record and attendance code!**
- Morning everyone
  - **Don't forget to register your attendance**
- This week we are going to look at python modules and, in doing so, revisit fitting.
  - As usual, please feel free to interrupt with questions at any time

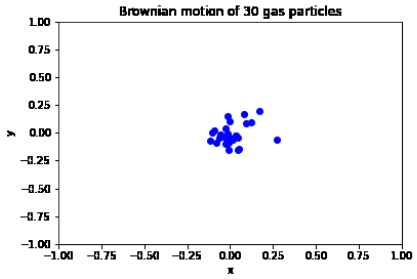
Lecture 10: Recap

- 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

```
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



3

- First though, lets recap last week, where we looked at … which use … experiments
- In doing so, …
  - First we setup the default random number generator by calling np.random.default_rng()
  - Then for uniform random numbers we simply call the random() function with the number of random numbers we want as input, in this case 10.
  - **For random numbers following a Poisson distribution, which remember describes discrete events occurring over an interval of space or time, we call the poisson function instead, which takes the mean and size as inputs**
  - **Finally, Gaussian- or normal-distributed random numbers are generated by calling the normal() function, with the mean, RMS and size as arguments, and describe many situations with large number of events due to the CLT**
- We then applied the Monte Carlo approach to several simple problems (sim + int), following a set of general steps
  - (read) .. in order to obtain the result.
- **in the summative problem you saw how to generate a single gas particle**
  - **This may seem simple but once you can generate one you can easily add others, such as the 30 shown here, …**
  - **… making them bounce of the walls (point)(and eventually each other) to model an ideal gas**
  - **Matplotlib even allows you to animate the results, though it is beyond this course.**

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

```
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

```
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

```
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]
```

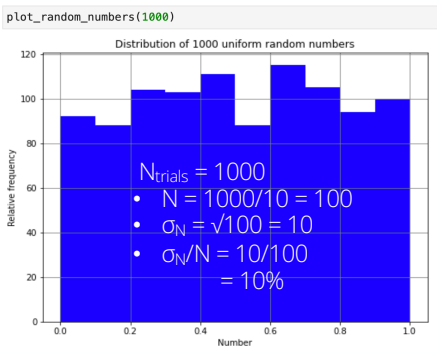
Diff. seed = diff. result

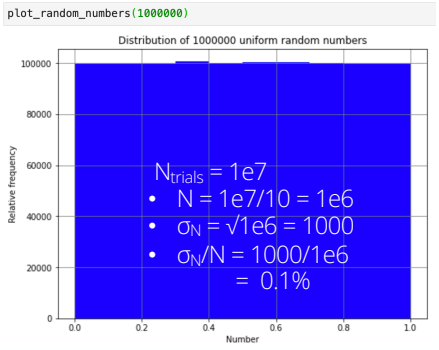
4

- Before we go on to this week's topic, lets as usual look at the solutions to last week's formative problems, and feedback on any common issues
- Ex1 was to check how the seed works by running the code with the same or different seeds multiple times
  - (read)
  - (read)
    - **… get the same sequence of random numbers, as shown here**
    - **… get a different *repeatable* sequence of random numbers, as shown here**
  - **Example:**
  - **Note how I used a function that takes the seed as input and simply called this to run the code multiple times**
  - If we don't …

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)





5

- Ex2 was to investigate … trials from 1000 (point) to 10 million (point)
  - Since the numbers are uniformly spread between 0 and 1 we expect the number in each bin to be the number of trials divided by the number of bins on average
    - **In our example, where we have 10 bins, we get 100 per bin in the first case (point) and 1 million per bin in the second case (point)**
  - **However, since the numbers are random they will fluctuate about this average, changing each time you run the code**
    - **But the more trails you run the less fluctuation about the average there is, or the smaller the uncertainty is, as you saw for the pi example in the notebook**
      - Example
      - **This is why we need to use samples of large sets of random numbers to get a good deterministic result**
  - **In fact, we can quantify this, and check it behaves as we expect, since the uncertainty on a given number N (assuming the CLT) is just root N**
    - **So, for the 100 events per bin the uncertainty is 10 and we indeed see fluctuations of this size (point)**
    - **For 1M events per bin on the other hand the uncertainty is 1000. While this is bigger, which seems contrary to what we expect, we have to consider the percentage error, which, dividing the uncertainty by the bin content, is 0.1% in this case compared to 10% in the previous case. Hence the fluctuations will be of the scale of only 0.1% which is very small.**

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

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

```
# 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.
```

6

- **Ex 3 continued unc theme by asking you to determine the … value**
  - Reminder of how calculated pi numerically: throw 10000 uniform random numbers in x and y, calculate the radius and sum those < 1, then divide by the total number and times by 4
  - For the error we simply code the formula that was derived in the notebook, which can be written in terms of just the number of trials and the number within the circle by substituting the probability of being inside (p) or outside (q)
  - To check if it is consistent with the known value, which we can take from np.pi, we need to use the consistency check that you have used in labs and which we saw in week 7
  - **I remind you that this calculates how significant the difference is by taking the absolute difference between the two values and dividing by the total uncertainty (add in quad)**
    - In our case the known value of pi is very exact and so has no significant uncertainty, hence we just divide the difference by the error we have calculated on our estimate
    - I got pi as 3.156 with an error of 0.016 or so, which is less than 1 sigma away from the known value and hence they are consistent since this is less than our usual bound of 3

Lecture 10: Formative problem solutions (4)

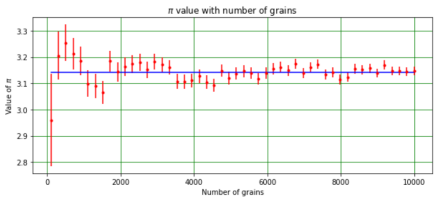
- Exercise 4: Fix the bug to plot the uncertainty on π versus the number of grains
  - The issue is that the `grainArr` contains floats, while `rng.random()` expects ints → convert
    - E.g. using `astype(int)` or `int()`

```
---------------------------------------------------------------------------
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]
```



7

- **Ex4 gave you some practice at debugging by fixing the error in the code to plot the uncertainty on pi as we increase the number of trials**
  - Convert the code into a function
  - Create an array of the number of grains from 100 to 10000 in steps of 50
  - Loop over this and call the function for each, storing the value and the error in arrays. Finally plot this but got an error
- **Remembering how we read the error starting from the bottom, we can see that we have a TypeError (point), which tells us that we are using a float somewhere python expects an int**
  - **If we then look at the traceback it tells us that the code calls the pimc() function on L33 (point) and then gives an error when trying to call rng.random() with the nGrains as an argument within that function (point)**
  - **This is because, as we can see in the printout here (point) the way we have setup our array using linspace gives floating point (or decimal) numbers while np.random expects an integer (or whole) number of events of course**
  - **So we need to convert the floats to ints and the easiest way to do this is to use numpy's astype() function (point) with int as an argument to convert the whole array (as saw in earlier week)**
    - **Another, equally good method, that several people used, was to simply put the int() function around the nGrains in the call to the pimc() function or random() (point to error)**

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()
```

8

- Finally, Ex5 asked you to add comments to the code, explaining each line of the for loop in particular
    - The amount of comments is a bit a matter of taste and here I have probably over-commented to be didactic
    - For the for loop, the first line (point) calculates the number of times one or more patient has no bed.
    - We use numpy's array at a time method to check when each element in our patient array is greater than the number of beds under study, getting an array of False (if not) and True (if so)
    - We then sum this up to find the number of days it occurs, remembering True and False are treated as 1 and 0, respectively
    - And store the result at the correct point in the array
    - We use our remainder operator to print out the results every time the number of beds is divisible by 5
- **Any questions on the formative exercises?**
- Grades and feedback on Week 9 assignments have been posted on canvas

Moving beyond Jupyter notebooks

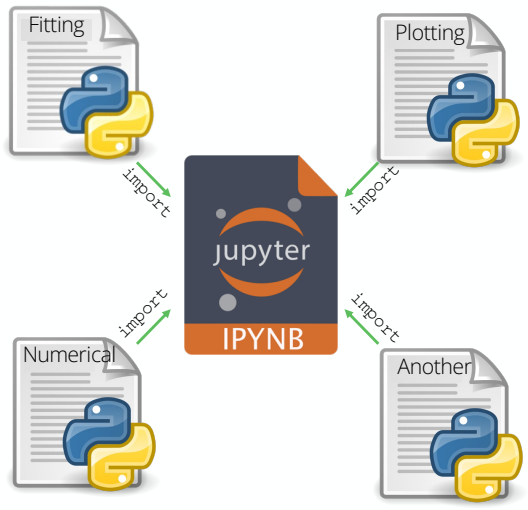
- During the course, we have been writing our Python code exclusively in Jupyter notebooks
  - Which are a great resource for learning + developing code interactively and disseminating the results
- However, they also have some disadvantages, especially for writing longer programs
  - Large notebooks 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**

9

- **If not, let's move on to today's topic …**
- So far … the results, as we have seen
- (Read)
  - As I am sure you have noticed …
  - … as you have almost certainly encountered
  - (read others)
- (Read)

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



10

- (Read first bullet) … modules, hence the name
  - **For example, you might want a module for your fitting code as we will make today (point)**
  - **But you might also want a module for your plotting code, a module for numerical analysis, and potentially others (point)**
  - **Each of which you can import and use in any notebook (point)**
- (Read second bullet)
  - Firstly, simplicity …
  - Second, maintainability, since …
  - Finally, reusability …
- You have already used many different types of modules
  - (Read)
  - If you do labs, …
- (Read)

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

11

- (Read first bullet)
  - **Written in any editor you like, which can also be directly in cocalc, and saved with a file extension .py, which stands for python**
- It can contain essentially any valid python code such as … and can also use other modules too by importing them in the usual way
- **Let's take a look at an example python file, which I have called mymodule.py:**
  - **Here, I create several variables (point): a string, a number and a numpy data array, exactly as I would in a jupyter notebook, remembering for the latter to import numpy**
  - **I also define a function, imaginatively called func (point), which for sake of example, takes one argument and prints that out**

- Once we have written our module we simply place it in our current working directory where the notebook is and then import it into the notebook just like we would any other module we have used (point)
    - Note, that the name of the module we import doesn't include the .py file suffix
- **We can then access the information in our module in the same way as numpy etc by using the module name, which I have shortened to mod for convenience, followed by the variable or function name separated by a dot**
    - So we can call our variables string, number and data like this and print the results
    - We can also call our function with the required argument
    - In each case, we see the expected output.
- **ANY QUESTIONS on the basics of modules?**

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

12

- Now of course, having …
- (Read bullet)
  - **Which is specified by the path variable in the sys module we have encountered before**
- By default … python list of paths to look in from your python installation
  - **Which on cocalc looks like the following for me (point), where you can see I have simply imported the sys module and printed sys.path**
  - **Each of these locations is searched in turn to find the requested module**
- However, as this is just a list ([]), we can easily add to the list of locations searched for modules by simply appending paths to this list
- So, for example, if we put our module say in a directory called libs inside our current directory we simply append the relative path libs to sys.path, as shown here (point) and python will now look for any module there too

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

13

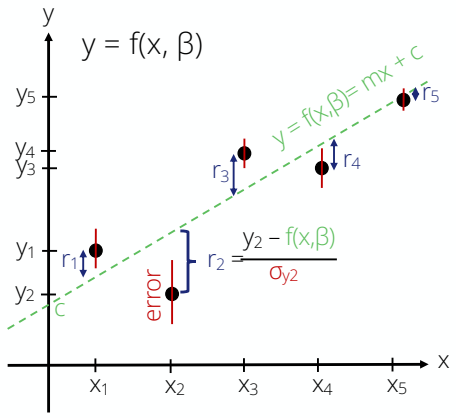
- **If the libs directory was not under our current directory but somewhere else, say in our home directory, we can just append the full, or absolute, path to the list as shown here**
- So, the idea is that we can move useful code … want, without having to copy-paste the code
- **Any questions on creating and using python modules?**

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

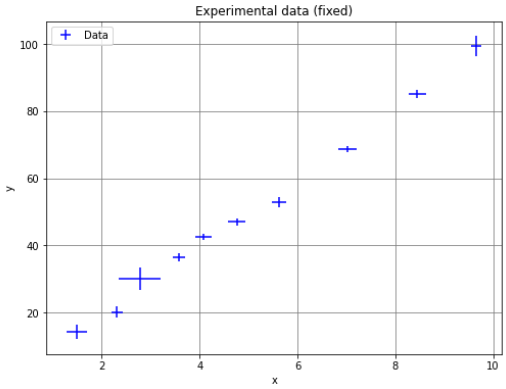
Tuple of input data (x, y, σ_x, σ_y) to minimise with respect to

14

- **Given we going to make a fitting module, it is prudent to remind ourselves about the fitting method we covered in lecture 6 …**
- **To determine the relationship between the independent and dependent variables, x and y, of our data we have to fit it with a functional form**
  - **Often we know the expected functional form f(x) from some model or theory and want to see if it describes our data**
  - **In doing so we want to find the values of the unknown parameters of the model, which we called beta (point), that best describe our data**
- We saw in lecture 6 that we determine the best fit by … residuals, which are just the … (point) … error, **such that more uncertain points have less influence in the fit**
- In general, we actually minimise the sum of the square of the residuals (point), **which is nothing more than that chi2 statistic that you use in PHYS106 if you do labs**
  - We do this using the so-called least-squared fitting method, which we took from scipy's optimise module
- For our purposes this took three arguments:
  - The function …, remembering that scipy took care internally of squaring and summing these
  - The initial guess of any free parameters, which we can find from a plot
  - And the tuple of input data, in the form of x and y plus the corresponding errors, that internally are passed into the residuals function (fun) to minimise it 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]
```

15

- **Let's now remind ourselves briefly the steps we need to preform to do a straight-line fit …**
- Firstly, we must read in the data, values and errors, which we usually do from a file using the np.loadtxt() function (point)
  - We then plot this to check that it is sensible (point) and to determine the initial parameters, which here are an intercept of around 0 and a gradient of around 10

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
```

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

14

- We then needed to define the functions that implement the mathematical function of the model and it's derivative (point)
  - Which we use to calculate the point-by-point residuals following the previous formula (point)

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)
```

14

- **We then actually run the fit by calling the least_squares method (point), passing in our residual function, the initial guess of the parameters and the data along with its uncertainties**
  - **This is the actual FIT part!**
  - **This provides a result object with several pieces of information**
  - The first thing we must check is that the fit succeeded, using result.success (point).
  - If so we can extract the best-fit parameters using result.x (point)

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
  - $0.25 < \chi^2/NDF < 4$ and/or χ^2 prob > 0.05

```
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
```

14

- We then need to check if the fit is a good description of the data, which we do using our chi2 test statistic
  - **This is calculated from the resulting residuals function, accessed using result.fun, squared and summed**
  - **We can divide this by the number of dof to get the chi2 per d.o.f, which should lie roughly in the range ¼ - 4**
  - **We can also calculate the probability of this chi2 value, which should be above 5%**

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
  - $0.25 < \chi^2/NDF < 4$ 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
```

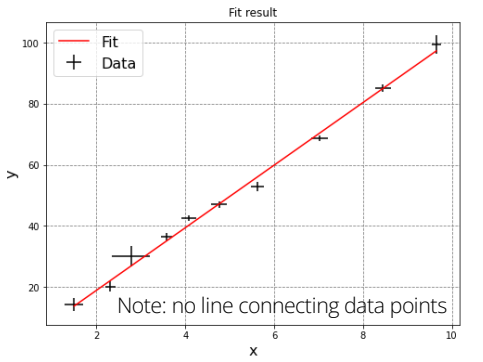
14

- **Following this, we need to determine the uncertainties on the fitted parameters, which we extract from the Jacobian matrix received by result.jac**
  - **You will understand the calculation for this after looking at matrices in PHYS108 next semester**
- We can then print out the best fit parameters along with this associated error
  - **Remembering to include the units if there are any**

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
  - $0.25 < \chi^2/NDF < 4$ 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()
```



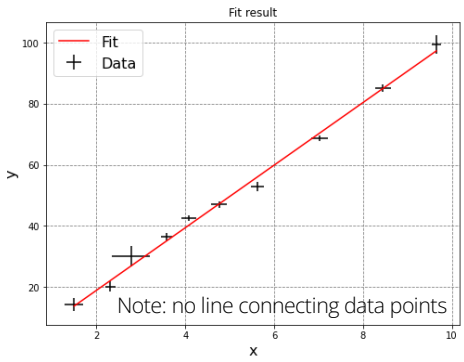
14

- Finally, we pass the best fit parameters and the x data back to the definition of our straight line (point) to determine the corresponding y values
  - This allows us to plot the resulting best-fit model (as a line) along with the data, which are drawn as points with error bars and w/o a line
  - **This is not the FIT it is just showing the result of the FIT compared to our data**

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()
```



14

- **At the end of lecture we encapsulated all this into a function that we could call to reuse**

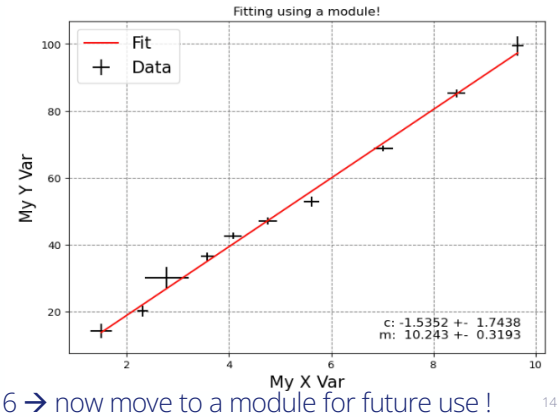
Fitting procedure steps

- 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
  - $0.25 < \chi^2/NDF < 4$ 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
```



14

- Now we take the fit function and the other necessary functions and put it into a module (show)
  - In future you can just point the sys.path to this module's location
  - Then import it and call with your data and initial params etc
- This is all you need to do when you need to do a fit from now on

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
```

15

- **In doing so we can also make our fitting a bit more general …**
- (Read first bullet)
- **One relatively simple thing we can do is to extend our code from just a straight line to any polynomial**
- **To do this we first need to write the python functions for a polynomial and it's derivative**
- We know that a polynomial is just a constant p0 + p1 times x + p2 times x^2 etc (point) and in fact the constant can be written as p0 times x^0, since anything to the power 0 is just 1, making the structure of all the terms the same
  - So we need to code this up in a function like our previous straight_line() function, which I have now called polynomial(). Let's start by setting our function f to 0 (point).
  - Since we know that the number of polynomial terms is just the length of our array of input parameters we can write a

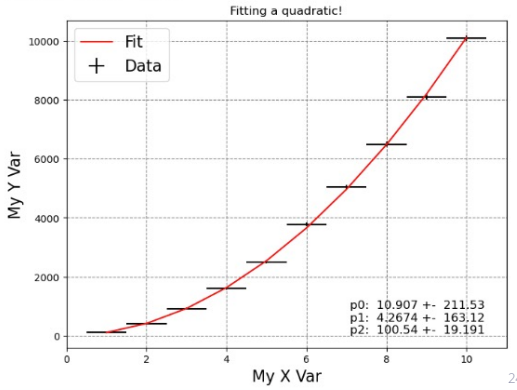
loop over that number using for i in range and add each term to our function (point)
        - For each parameter we simply add the current value, taken from the ith element of the parameter array, multiplied by x to that power (point)
- For the differential wrt x we, of course, multiply each term by the current power and then subtract one from the power (point)
    - This can be coded up in the same way as the polynomial itself but with each term now being the parameter times the index i times x to the power i-1 (point)
    - In this case, we start the loop at 1 (point, not 0) since the 0^{th} term was a constant and of course gives no differential
- Once we have those functions, it is just a case of replacing the straight_line and straight_line_diff functions in our residuals calculation with the new polynomial versions (point)
    - And updating the fit function to call that instead

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:

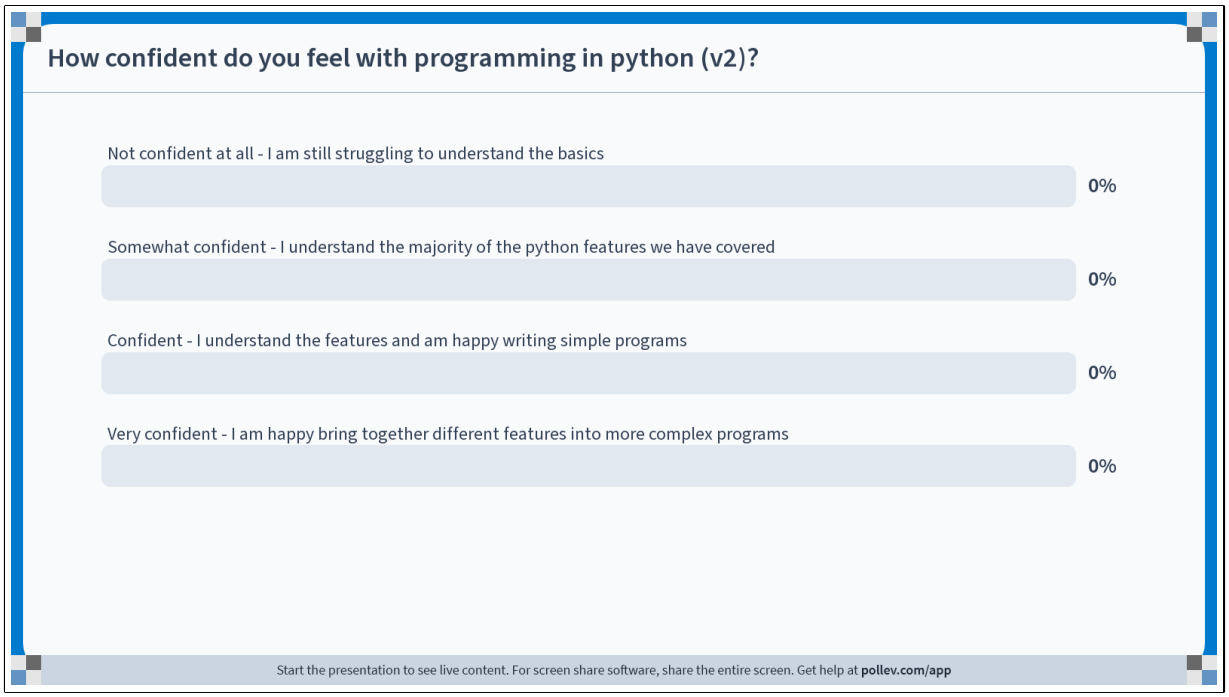
```
# Add module location to sys path
import sys
sys.path.append("modules")

# 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
```



24

- Show new module
  - Highlight the lines that changed: polynomial call + loop to print params and errors
- We can then use this to fit a new set of data that is quadratic rather than linear



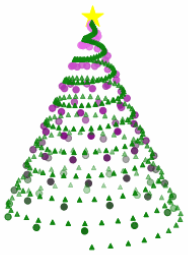
Do not modify the notes in this section to avoid tampering with the Poll Everywhere activity.
More info at polleverywhere.com/support

How confident do you feel with programming in python (v2)?
https://www.polleverywhere.com/multiple_choice_polls/YMdt8VmjJEM0pUgltFs8V?state=opened&flow=Default&onscreen=persist

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.

16

- So in summary …
  - … both within science fields and generally
  - … using a variety of techniques
  - … we finally made a fitting module
- (Read)
- **Any final questions ?**
- **If not, Merry Christmas and I hope you have a good holiday.**
  - **Here's a Christmas tree I made in python using matplotlib, the code for which is also in the CoCalc for this week**