Introduction to Computational Physics (PHYS105)

Lecture 8: Numeric Calculations

Carl Gwilliam
(C.Gwilliam@liverpool.ac.uk)
UNIVERSITY OF LIVERPOOL

- **Record + attendance code!**
- **Open poll!!!**
- Morning everyone
    - Please remember to register your attendance
- This week are are going to start looking at numeric techniques
    - As always, please feel free to interrupt at any time with questions

Lecture 7: Recap

- Last week we looked at the various types of python errors we can get
  - Code that is syntactically incorrect and therefore won't parse at all
  - Programs that start but give a run-time error and don't complete
  - Programs that run but produce incorrect results

Harder

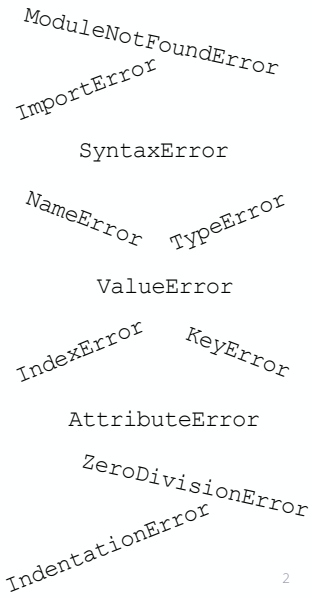
- We saw how to understand and debug python error messages:

```
TypeError                                 Traceback (most recent call last)
<ipython-input-39-2185f7667564> in <module>
     10
     11 w,l,h = 0.3,0.17,0.25
---> 12 V, A, s = rectPrismParams(w, l, h)
     13
     14 print(f"Volume = {V:.2f}")

<ipython-input-39-2185f7667564> in rectPrismParams(width, length, height)
      5     '''
      6     vol = width * length * height
----> 7     area = 2 (width * length + width * height + length * height)
      8     edge = 4 (width + length + height)
      9     return vol, area, edge

TypeError: 'int' object is not callable
```

Traceback: how the code got to the error point, with lines run indicated by --->

Read from bottom to top

The error type — The error message itself



2

- Before starting this week's material let's first recap last week, where we …
- **Which, as you no doubt saw in the problems class, get more difficult to debug as you go down**
- **In particular, we saw how to read and understand python errors and use them to debug code. As a reminder …**
  - You should read from the bottom up
  - The bottom left gives the type or class of the error; **we looked at several of these (listed on the right) interactively**
  - Next to that is the error message itself
  - Above this is the so-called traceback, which shows how the code got to the point of error, indicating the pertinent lines with an arrow
- **I also remind you that notebook we looked at interactively, both the buggy version and the solved version, is available in the lecture 7 CoCalc material if you want to recap**
- **Hopefully now you feel a bit more comfortable with python errors and can use these messages to help debug any issues you encounter in the remainder of the course**
  - **The demonstrators are still there to help of course but you should try to practice decoding the errors**

Lecture 7: Formative problem solutions

- Exercise 1: Naming and comments
  - Was not clear what the code does → look from inside out, printing each step if needed
    - `linspace` creates array of numbers 1-10
    - `cumprod` takes the cumulative product
    - We then take the last item of product array
    - Hence, factorial of number!
  - Simply changing the name would help clarify

```
import numpy as np
number = 10
nFactorial = int(np.cumprod(np.linspace(1, number, number))[number - 1])
print(f"Factorial of {number:d} = {nFactorial:d}.")
```

```
Factorial of 10 = 3628800.
```

  - Comment(s) would help further

```
import numpy as np
number = 10
# Perform cumulative product of array of numbers, then take the last value
nFactorial = int(np.cumprod(np.linspace(1, number, number))[number - 1])
print(f"Factorial of {number:d} = {nFactorial:d}.")
```

```
Factorial of 10 = 3628800.
```

- Exercise 2: Debugging ModuleNotFoundError
  - Simple typo in module name: pyploy → pyplot

```
ModuleNotFoundError: No module named
'matplotlib.pyploy'
```

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize = (1, 1))
```

- Exercise 3: Debugging NameError
  - Typo in line accessing index 1: L → l

```
littleArray = np.ones(3)
littleArray[0] = 2
LittleArray[1] = 1
littleArray[2] = 0
```

```
NameError: name 'LittleArray'
is not defined
```

```
littleArray = np.ones(3)
littleArray[0] = 2
littleArray[1] = 1
littleArray[2] = 0
```

3

- As usual, we will now look at the solutions to last week's formative problems and feedback on some common issues
- Ex1: was on naming and comments: the meaning of the code was not clear
  - **You can work it out by looking at it in the order python would execute i.e. from inner function call to outer (point)**
    - If you are unsure then printing each part is always a good strategy
  - So what's happening? We see…
    - First, calls np.linspace which we have used many times now and you should know creates array of numbers: here from 1->10 in unit steps
    - **Then, cumprod (which you can google) performs the cumulative product making an array where each element is the product of the items in the original array up to that point (show print)**
    - This produces a new array and we then use square brackets with the index 9 to take the last item
    - So this is the product of the numbers up to 10 -> simply factorial of number
  - Read remaining bullets
- Ex2: Solve the ModuleNotFoundError
  - Simple typo of pyplot as pyploy (point). Once fix that it imports fine.
- Ex3: Similar thing but for a NameError
  - Capitalisation, which is important in python: L not l. Once fix, runs fine

Lecture 7: Formative problem solutions (2)

- Exercise 4: Explaining SyntaxError
  - We have used round brackets rather than square brackets to access the list element
  - Python sees round brackets as us trying to call a function, hence complains as we can't give a function a value using assignment (=)

```
SyntaxError: cannot assign to function call
```

- Exercise 5: Debugging IndexError
  - We are trying to use a float as an index but index must be an integer (or slice etc)

```
IndexError: only integers, slices (`:`), ellipsis (`...`), numpy.
newaxis (`None`) and integer or boolean arrays are valid indices
```

```
index = 1.0
littleArray(index) = 3.2
```

```
index = 1.0
littleArray[index] = 3.2
```

```
index = 1
littleArray[index] = 7.2
```

- Exercise 6: Code runs but produces the wrong result
  - We have got the quadratic equation formula wrong
  - Should be -4ac not -3ac
  - Always check the results!
    - E.g. with known cases

```
nQuadEqs = 4
aArr = np.linspace(1.0, 4.0, nQuadEqs)
b,c = 2.0,-3.0

for n in range(0, nQuadEqs):
    discrim = b**2 - 4*aArr[n]*c
    assert (discrim > 0), "Bumped into negative discrim ({:5.3f}), can't take sqrt!".format(discrim)
    root1 = (-b + np.sqrt(discrim))/(2*aArr[n])
    root2 = (-b - np.sqrt(discrim))/(2*aArr[n])
    print("For a ={:.2f}, b = {:.2f} and c = {:.2f}, roots are {:.2f} and {:.2f}."
          .format(aArr[n], b, c, root1, root2))
```

```
For a =1.00, b = 2.00 and c = -3.00, roots are 1.00 and -3.00.
For a =2.00, b = 2.00 and c = -3.00, roots are 0.82 and -1.82.
For a =3.00, b = 2.00 and c = -3.00, roots are 0.72 and -1.39.
For a =4.00, b = 2.00 and c = -3.00, roots are 0.65 and -1.15.
```

4

- Ex4: Explain the syntax error, which said …
  - Can only assign the result of the function call to a variable on the LHS (not RHS)
- Ex5: After fixing this to square brackets there was still an error, this time an IndexError
  - Because, as the message says, arrays can only take certain types as an index e.g. integer and slices (as we saw) + a few we have not covered
  - We try to give it float (anything with decimal in is a float even if fractional part is 0)
  - Simply replace by an integer 1 not 1.0 (or, as some people did, could convert index to integer after using int() – both are fine)
- Ex6: Code to calculate the solution to the quadratic equation, which runs w/o error but gives the wrong result
  - Error is simple: Quadratic equation wrong: -3ac -> -4ac
  - **But, as you will have seen, these kind of bugs, were the code runs without error messages, can be very hard to debug (esp. if buried in long code)**
  - **Are very important though as if we get the wrong result the code is pretty much useless**
  - **Luckily, we are not likely to deal with critical code but a mistake could literally be the difference between life and death in some cases**
  - **We need to test our code e.g wrt known results e.g via calculator (and in official production code developers often spend many hours writing several test programs for different scenarios)**
- **QUESTIONS: Any questions on the formative problems?**
- Grades for lecture 6 problems released in canvas with an average mark of XX

Physics calculations

- Solving physics problems requires calculations that often involve evaluating equations
  - Finding the roots
  - Solving sets of equations
  - Integrating
  - Differentiating
  - Etc
- To solve these problems computationally we need to be able to deal with them in python
- For some equations we are lucky enough to know the analytic form of the solution
  - Even in this case, solving by hand quickly becomes time consuming
  - Several dedicated programs have been developed to automate algebraic calculations
    - E.g. MathWorks and even a python module SymPy, although we will not cover this in our course

- For most equations this is not the case, so we need to solve them using numeric techniques
  - Employing various approximate methods, such as graphical, discretisation, iteration, etc
- We will look at such methods over the next few weeks
  - Building up our python skills and solving real-world problems

5

- Let's now move on to the topic of this week's lecture …
- By their very nature …
- **For some … exception rather than rule, particularly if we look at real-world problems (as we shall see)**
- If an analytical solution exists we can solve it algebraically
  - **However, in many cases still … so easier and more efficient to solve using a computer**
  - e.g. … mathworks online or even the sympy python module (though we won't cover this)
- **If no analytic solution exists, we need to solve numerically, which just means using one or other approximations**
  - **And we shall see several different methods …**
  - **This can be done in python combining the numpy and scipy modules we have encountered already and we will look at these over the next few weeks**
  - Practicing our python in realistic programs as we go

Differentiation

- Numerical methods may sound daunting but they are not really anything special
  - You have almost certainty come across them before, just not called them this
- Differentiation is simply the change in y over the change in x as the size of the x-step tends to 0

$$\frac{dy}{dx} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

- Can calculate a differential numerically using this
  - Example: differentiation of cosθ

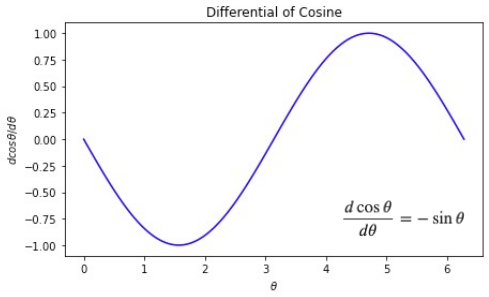
```
def cos_diff(theta, dtheta = 0.0001):
    return (np.cos(theta + dtheta) - np.cos(theta)) / dtheta
```

  - Gets closer to the correct value the smaller dθ

```
for s in [0.1, 0.01, 0.001, 0.0001]:
    print(f"Difference (step={s:5f}):", np.sin(0) - cos_diff(0, s) )
```

```
Difference (step=0.100000): 0.049958347219742905
Difference (step=0.010000): 0.004999958333473664
Difference (step=0.001000): 0.0004999999583255033
Difference (step=0.000100): 4.999999969612645e-05
```

```
angles = np.linspace(0, 2*np.pi, 100)
diff = cos_diff(angles)
plt.figure(figsize = (7, 4))
plt.title("Differential of Cosine")
plt.xlabel(r"$\theta$")
plt.ylabel(r"$dcos\theta/d\theta$")
plt.plot(angles, diff, color = 'b', linestyle = '-')
plt.show()
```



6

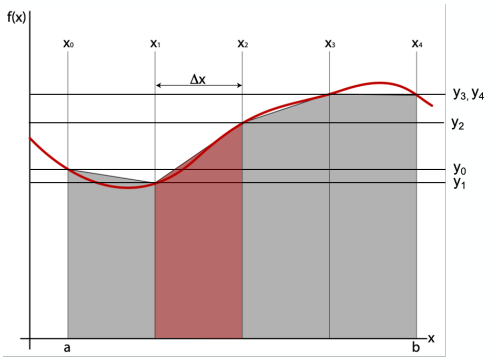
- **For example, when you learnt differentiation I am sure you started off from the change in the y value of a function over the change in the x value (i.e. gradient) for a given step dX**
  - **Then made this step dx smaller and smaller**
  - **Finding that the differential is just the limit as the step size dx tends to 0, as written here**
- **Since numeric methods are nothing more than approximations, we can use this equation for the difference in a function at x+dx compared to x divided by the step size dx to calculate a differential numerically**
- So, lets do that. For example, lets look at the differentiation of cos(theta), which of course we can know the answer analytically to be –sin(theta), numerically
  - We write a python function, which I have called cos_diff, that takes two args: theta and the size of our step (now dtheta with default)
  - **Inside the function we simply calculate cos(theta + dtheta) – cos(theta) divided by the step size dtheta following our equation defining a differential above (point)**
  - **We can call this for theta 0 and we get something close to the**

expected 0
    - As we reduce the step size we get closer and closer to actual differentiation and hence to to the real value of 0 (point)
    - (remember variables are stored with limited precision so may not be exactly 0)
- Going beyond just evaluating this at one value (0), we could create a numpy array of theta angles from 0 to 2pi (point) and pass this into our numeric approximation
- Due to numpy's array-at-a-time calculations, the result is the array of the corresponding differential at each theta point
- If we plot this with matplotlib, we see it is indeed –sin(theta) as expected (or a sine wave flipped over)

- We have just done our first numeric calculation! And it hopefully wasn't so scary.

Integration

- We know that physically, integration is simply calculating the area under the given function
  - Between a given set of bounds in the discrete case
- Can be calculated numerically by simply splitting into a series of small slices & summing their areas
  - Each slice can be approximated by a trapezium whose area is given by $0.5\,(y_{i+1} + y_i)\,(x_{i+1} - x_i)$




```
# Number of slices
nslices = 100

# Create array of angles and cosines for each step
angles = np.linspace(-np.pi/2, np.pi/2, nslices+1)
cosines = np.cos(angles)

# Loop over the slices to find the total area
total_area = 0
for i in range(nslices):
        # Find coordinates of current + next point
        x1 = angles[i]
        x2 = angles[i+1]
        y1 = cosines[i]
        y2 = cosines[i+1]

        # Calculate area of slice and sum
        trapezium_area = 0.5 * (y1+y2) * (x2-x1)
        total_area = total_area + trapezium_area

print(f"The area is {total_area:0.2f}")
```

```
The area is 2.00
```

- E.g: integral of cosθ from $-\pi/2$ to $\pi/2$
  - The smaller the slices the more accurate the answer (see notebook)

7

- We can also perform an integral numerically. We know …
- … of small discrete slices **(akin to the method we used for differentiation when we went from a continuous derivative to discrete steps)** and summing their areas as you see here
  - We can approximate each slice by a trapezium whose area is given be width x average height
- Let's do this in python for e.g cos theta between –pi/2 and +pi/2
  - We will try 100 slices and create an array of the angles between –pi/2 and pi/2 using np.linspace (note the edges are the nslices + 1)
  - We use numpy's array at a time functionality to get the corresponding cosines
  - We now need to loop over the range of slices (which is one less than entries) starting from 0 (default) …
    - For each we get the x values at that point in the angles array and the next one (I+1), plus the corresponding y values from the cosines array
    - Then we calculate the area of the trapezium
    - Then sum the area, using a variable defined as 0 to start with outside the loop
- If we print the area we get the expected value of 2
  - **Of course, the move slices the more accurate the result and you will investigate this in the notebook**
- **Any questions on numeric methods?**

Solving equations

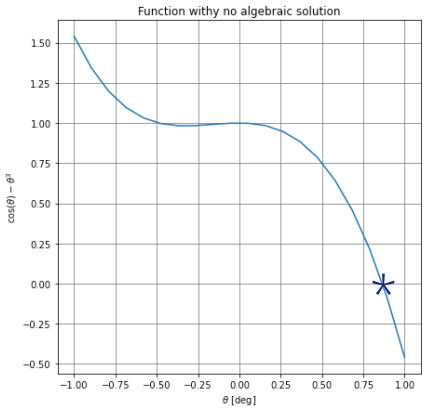
- Even many seemingly simple equations cannot be solved analytically using algebraic methods
- Example: consider finding the solution(s) of $\cos(\theta) - \theta^3 = 0$
  - Plotting the function clearly shows a solution exists
  - But it you ask programs like MathWorks or SymPy to solve it, they are unable to provide an answer

```
syms x
eqn = cos(x) - x^3 == 0
```

$$eqn = \cos(x) - x^3 = 0$$

```
S = solve(eqn, x)
```

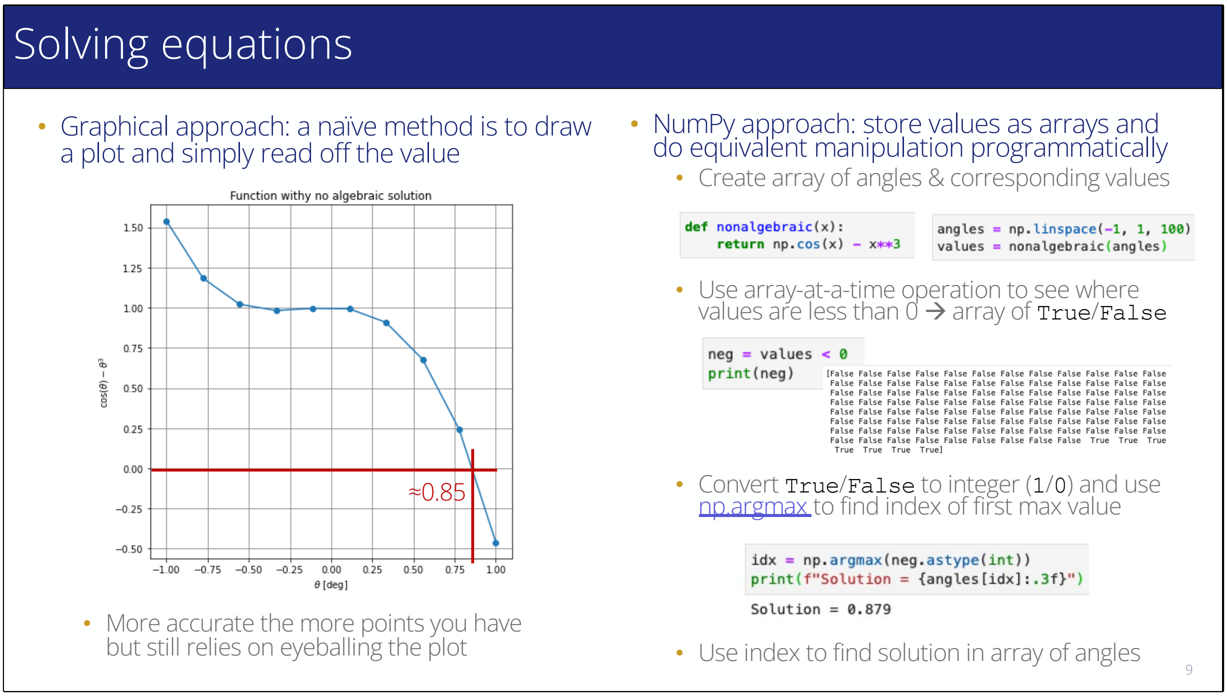
```
Warning: Unable to solve symbolically.
```



- Rather than just throwing our hands up, we must instead solve it using numerical methods
  - Which can be achieved by various methods …

8

- OK, what about solving equations?
- For example …
  - **If we plot the function we clearly see that a solution exists i.e. the function passes through 0**
  - But if we try to ask … it is unable to provide answer
    - **Don't need to understand this**



- The simplest method we can think of to solve this is to do it graphically
- **Here is the plot of our function again but were this time I have used somewhat less points and labelled them with a marker**
    - **You can see that matplotlib interpolates between the points with a straight line**
- If we read off where the line graph crosses zero we get something like 0.85 as the solution
    - Of course, the more points we use the less far matplotlib has to interpolate and in principle the more accurate our result
    - But we are still limited by the fact we are eyeballing the plot
- **A step up from this is to do the equivalent thing programmatically, rather than by eye, by storing the data as a numpy array not a graph**
- **First, let's write our equation as a python function using numpy so it works with an array**
    - Then we create an array of angles using linspace, over the range of our plot -1, 1
    - Here is the code doing this in ten steps and making the plot above.
    - **Since we are now doing it programmatically we can now use many more points e.g. 100**

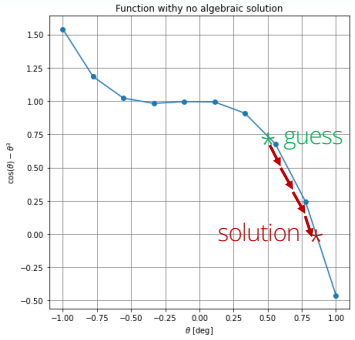
    - We then pass it to our function to get the corresponding array of our equation values
- We can use our array-at-a-time operations to require the result to be < 0
    - Which gives us an array of booleans, as you can see, which are True where it is < 0 and False where it is not
- **We then simply need to find the point at which it switches from False to True to see where It crosses 0**
    - **To do this we convert the array from booleans to integers, where True becomes 1 and False 0, using the astype() function**
    - **We then use numpy's argmax() function (show docs), linked here, which finds the index of the max value or, if there are more than one of the max value the first of these**
    - **So in our case this will give us the index of the first 1 (which was the first True i.e the first point < 0).**
    - **We simply use the result as the index to our angle array to find the corresponding angle where this occurs.**
    - **Here we get ≈0.88, which seems sensible given our eyeballed guess but is likely more accurate.**
    - **Of course, if we make our array bigger, adding more points, we will get a more accurate result (try 1000)**
- **QUESTIONS: Any questions on the numeric solutions or what we have done so far ?**

Solving equations (2)

- Iterative approach: start from initial guess & step left/right if function value less/greater than 0
  - Revaluate function and keep stepping left/right till value is 0 within a certain tolerance

```
def iter_solve(xguess, step = 0.0001, tolerance = 0.001):
    "Function to iteratively solve cos(theta) - theta^3 = 0"

    nsteps = 0

    # Loop while abs value of function is above tolerance
    while abs(nonalgebraic(xguess)) > tolerance:

        if nonalgebraic(xguess) > 0:
            # If value > 0, step to right to decrease
            xguess = xguess + step
        elif nonalgebraic(xguess) < 0:
            # If value < 0, step to left to increase
            xguess = xguess - step
        else:
            # if exactly 0 stop
            break

        # Count number of iterations
        nsteps += 1

    return xguess, nsteps

sol, nsteps = iter_solve(0.5)
print (f"Found solution = {sol:.3f} in {nsteps} iterations")
```

```
Found solution = 0.865 in 3652 iterations
```



- In fact, scipy.optimize has a sophisticated iterative root-finding function called fsolve

```
from scipy.optimize import fsolve
sols = fsolve(nonalgebraic, 0.5)
print(f"Solution:\n{sols[0]:.3f}")
```

```
Solution:
0.865
```

10

- Another category of numerical methods is to take an iterative approach i.e. to make an initial guess and keep refining it
  - Which is what the least-squared fitting we saw in week 7 did, starting from our initial parameter guesses
- In our case we can start from a guess by eye and find the value of the function at that point
  - If it is > 0 we have to move to the right (point)
  - if it is < 0 we would move to the left (point starting from end)
- We then re-evaluate the function at the new point and continue moving left/right, depending which side of 0 the result is, until we get the solution, in our case 0, within a certain tolerance
- Let's write a function to do this in python that I have called iter_solve
  - It takes our initial guess for the solution, the size of the step we want to take left/right in x each time and the tolerance we want for the solution
  - We then have a while loop that evaluates our function at the starting guess and checks if it is more than tolerance away from 0
    - Note the use of abs for the modulus since at this point

we don't care which side of 0 it is, just if it is more than tolerance away we need to keep going
    - If the function is > 0 we add step to our initial guess to move right, else if it < 0 we subtract step from our initial guess to move left
        - Note, there is a probability we hit exactly 0 by luck on one of the iterations, in which case we obv want to stop so we use a break statement
    - The while loop then continues round, moving a step left/right each time, until the condition is satisfied i.e. the function is 0 within the specified tolerance
    - At that point it returns the result and also, just to see how many iterations it takes, the number of steps
    - We call this with e.g. 0.5 as the starting guess and we get a solution of 0.865 which is similar to our previous guesses but more accurate and it took ~3600 iterations.
- The nice thing about the iterative method is we can control explicitly the tolerance we want
    - Of course, at the expense of needing more iterations and hence longer CPU time if we want it more accurate
- We could improve on this approach in many ways **(can anyone suggest any?)** for example making the size of the step be dynamic so it is is bigger if we are further from 0 and smaller if we are closer
    - But we don't need to since scipy.optimise module…
    - We just import the fsolve function and then call it with the python function representing our equation (**which it expects to be rearranged in a form equal to 0**) and the initial guess (can also take other args such as the tolerance but here I have gone with the default)
    - In this simple case we get the same result as our iter_solve function, 0.865, so we did a pretty good job ☺
- Any questions on numeric methods?

- As mentioned, I am happy to recap topics you struggle on as we go through the remaining lectures
    - To help with this I have put together a poll to see which topics to go through again
    - Please drag them into the order with the things you struggle with an would like to see again most at the top

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

Which python topic do you most struggle with and would like recapped?
https://www.polleverywhere.com/ranking_polls/2pddkyEfYeUf1OS2PYe8q

Summary

- This week we have started to look at how to tackle problems that cannot be calculated analytically
  - In particular: integration, differentiation and solving equations
- In doing so we introduced the concept of numerical methods
  - Which are nothing more than a variety of approximations
  - Such as discretization, iteration and graphical approaches
- The notebook will give you chance to try out these methods
  - While continuing to further practice your python coding

- Next week, we will see how to solve differential equations numerically
  - And apply this to a real-world example, namely projectile motion taking into account air resistance

12

- **Any final questions?**