Introduction to Computational Physics (PHYS105)

Lecture 8: Numeric Calculations

Carl Gwilliam

(C.Gwilliam@liverpool.ac.uk)

UNIVERSITY OF LIVERPOOL

Attendance code: 746400

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

ModuleNotFoundError

ImportError

SyntaxError

NameError

TypeError

ValueError

IndexError

KeyError

AttributeError

ZeroDivisionError

IndentationError

- 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 --->

The error type

The error message itself

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

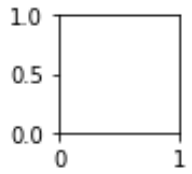
```
import numpy as np
number = 10
# Perform cumulative product of array of numbers, then take the last value
nFactorial = int(np.cumprod(np.linspace(1, number, number))[number - 1])
print(f"Factorial of {number:d} = {nFactorial:d}.")
```

```
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
```

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.
```

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

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

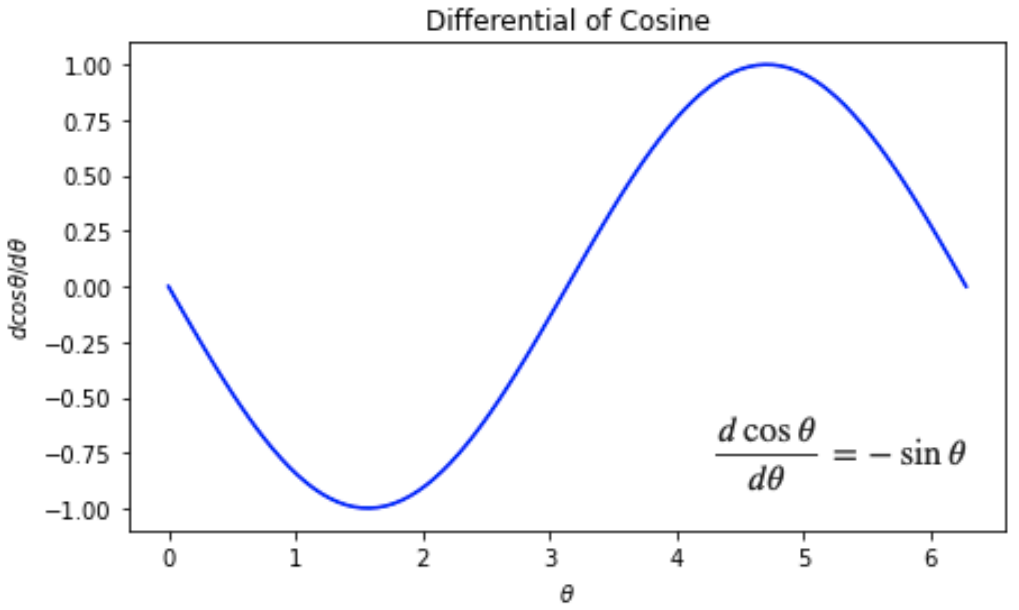
```
def cos_diff(theta, dtheta = 0.0001):
    return (np.cos(theta + dtheta) - np.cos(theta)) / dtheta
```

  - 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) )
```

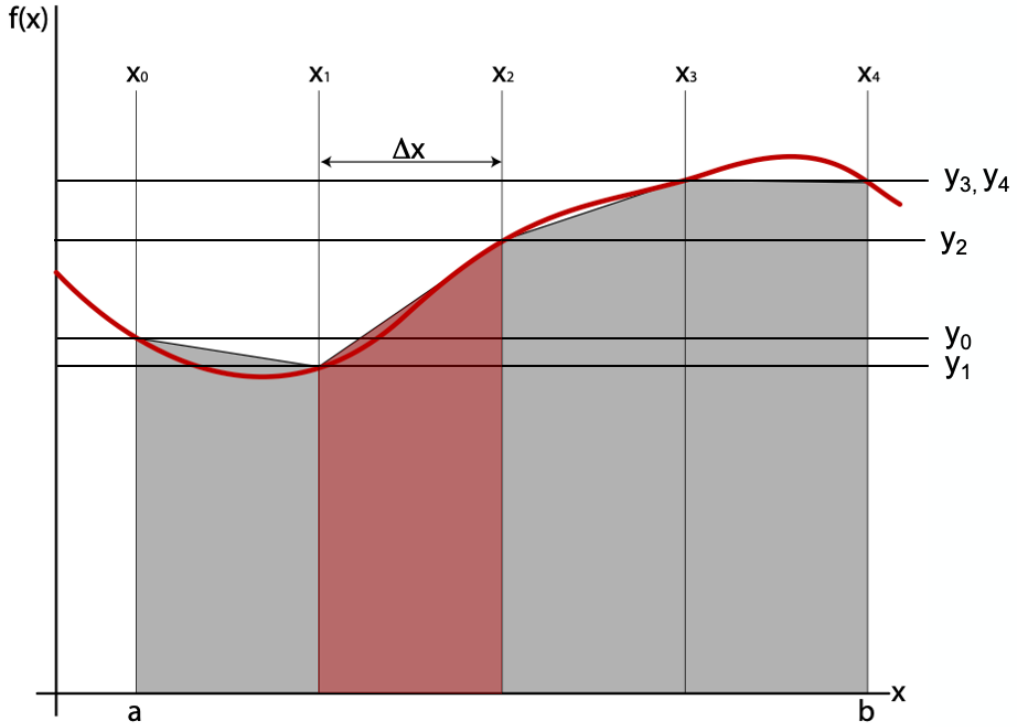
```
Difference (step=0.100000): 0.049958347219742905
Difference (step=0.010000): 0.004999958333473664
Difference (step=0.001000): 0.0004999999583255033
Difference (step=0.000100): 4.999999969612645e-05
```

```
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()
```



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 $0.5\,(y_{i+1} + y_i)\,(x_{i+1} - x_i)$




```python
# 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)

Solving equations

- Even many seemingly simple equations cannot be solved analytically using algebraic methods

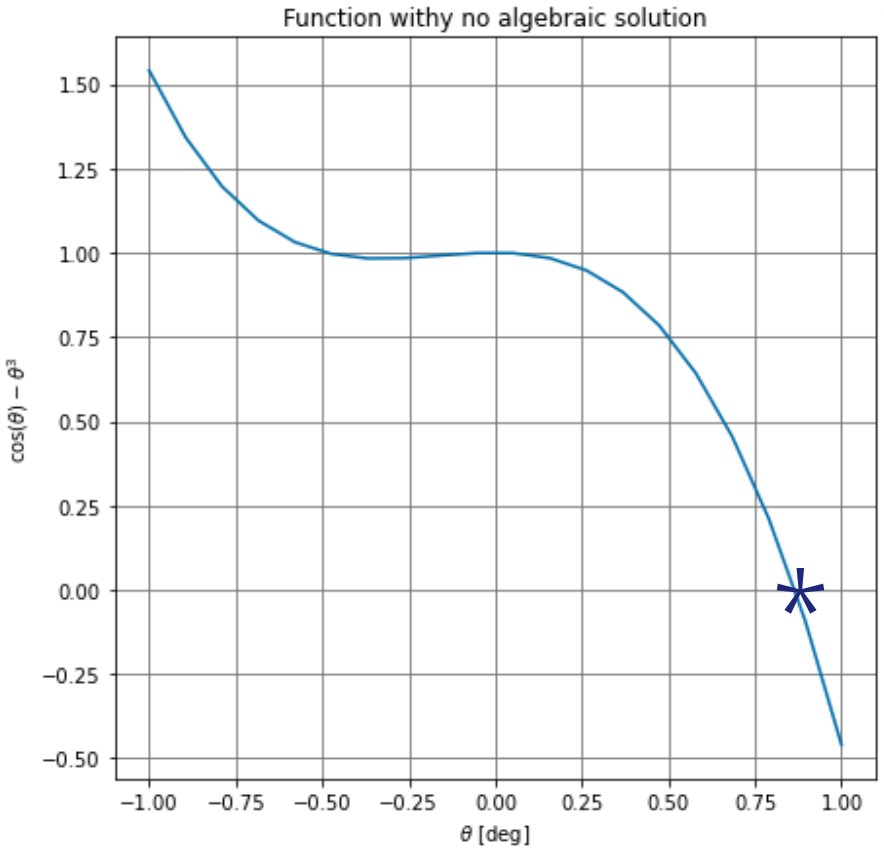
- Example: consider finding the solution(s) of $\cos(\theta) - \theta^3 = 0$
  - Plotting the function clearly shows a solution exists
  - But it you ask programs like MathWorks or SymPy to solve it, they are unable to provide an answer

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

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

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

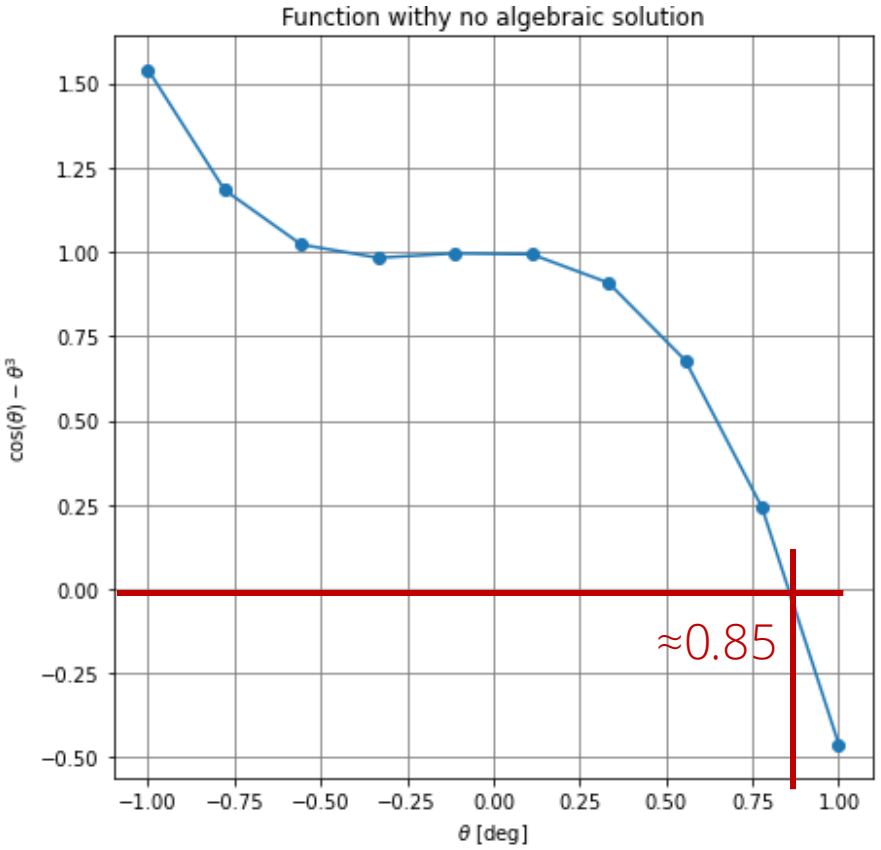
```
Warning: Unable to solve symbolically.
```



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

Solving equations

- Graphical approach: a naïve method is to draw a plot and simply read off the value



- More accurate the more points you have but still relies on eyeballing the plot

- NumPy approach: store values as arrays and do equivalent manipulation programmatically
  - Create array of angles & corresponding values

```
def nonalgebraic(x):
    return np.cos(x) - x**3
```

```
angles = np.linspace(-1, 1, 100)
values = nonalgebraic(angles)
```

  - Use array-at-a-time operation to see where values are less than 0 → array of `True`/`False`

```
neg = values < 0
print(neg)
```

```
[False False False False False False False False False False False False
 False False False False False False False False False False False False
 False False False False False False False False False False False False
 False False False False False False False False False False False False
 False False False False False False False False False False False False
 False False False False False False False False False False False False
 False False False False False False False False False False False False
 False False False False False False False False False  True  True  True
  True  True  True  True]
```

  - Convert `True`/`False` to integer (1/0) and use np.argmax to find index of first max value

```
idx = np.argmax(neg.astype(int))
print(f"Solution = {angles[idx]:.3f}")
```

```
Solution = 0.879
```

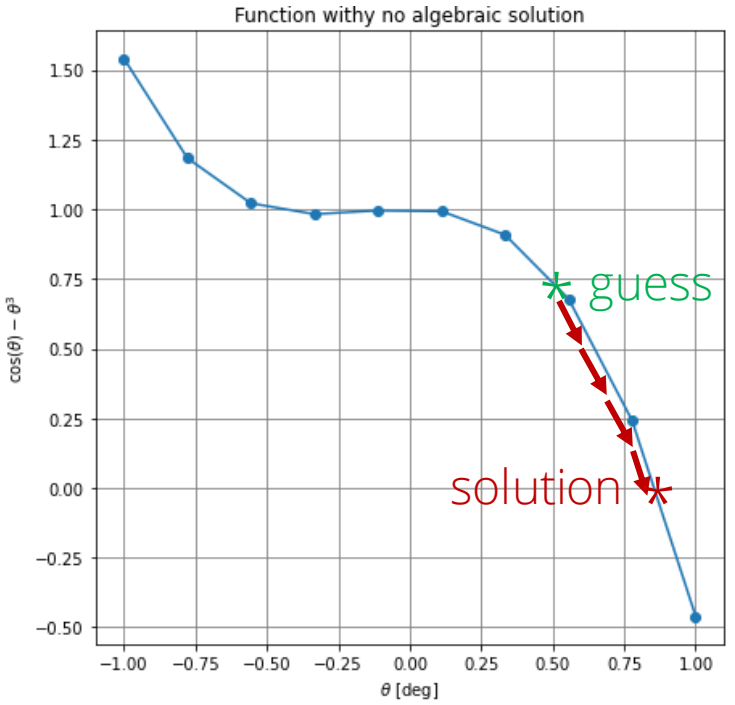
  - Use index to find solution in array of angles

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

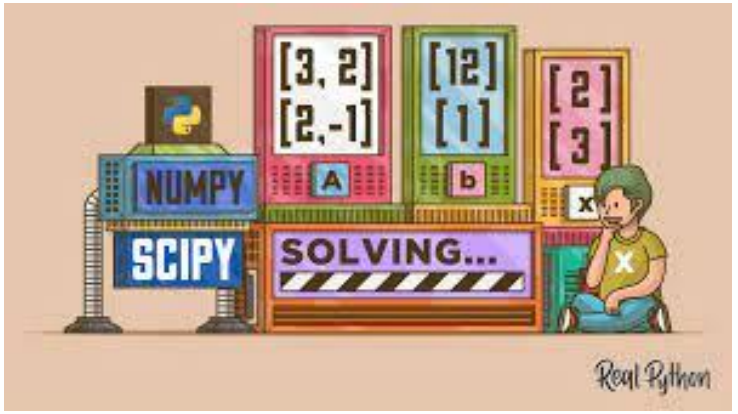
```
from scipy.optimize import fsolve
sols = fsolve(nonalgebraic, 0.5)
print(f"Solution:\n{sols[0]:.3f}")
```

```
Solution:
0.865
```

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