Instructor: Dr. S. Hughes

Due date: Sun, Jan 21 (midnight)

Lab 1 - Numerical Derivatives

Keywords: forward-difference, central-difference, SymPy, numerical errors in finite-difference, Richardson-extrapolated finite-difference, Legendre Polynomials, Rodrigues' formula, recursion

Task: go through all the questions and coding exercises below and hand in a single modularized Python code where each part runs each part of the question(s). Remember and include your surname at the beginning of the code name.

Code Submissions on onQ: Attach a single .py (called LabX_YourLastName.py) [or max one per question, but 1 preferred] and single .py.txt (same code, same name, just a different extension).

Marks: Coding Skills, Efficiency, Correct Results and Clear Graphs and Good Presentation of Results (6); Good use of Comments (2); Good Coding Structure and Readability (2). Total: (10)

Reminder: All codes must run under Python 3.9.x. and the Spyder IDE (check this before submitting). Section off the different parts of your code by using cells, i.e., #%% (and label section in code, e.g., Q1(b) $\cdots$), and add useful comments, so we can step through it easily.

Acknowledgements: Please comment on any help that you received from others for completing the Lab assignment.

Some Background (in addition to lecture notes):

Rubin Landau's Lecture on Numerical Differentiation
Numerical differentiation - Wikipedia
SymPy (symbolic mathematics in Python)
Legendre Polynomials - Wikipedia

Question 1 [6 pts]

Consider the following function:

$$\boxed{f(x) = \exp[\sin(2x)]} \tag{1}$$

(a) Using `SymPy`, obtain the analytical first-order derivative ($f'(x)$), the second-order derivative ($f''(x)$), and the third-order derivative ($f'''(x)$), and print these out with your Python codes[A]. Also create *lambda functions* from these derivative functions, so we can use them later for an error analysis with finite differencing. Now create finite arrays from these functions, where you can explicitly pass a value of x, where x goes from $x = 0$ to $x = 2\pi$, evenly, with exactly 200 points. Print the following to confirm these are the correct array (assuming your array is called `xs`: `print(len(xs),xs[0],xs[len(xs)-1] -2*m.pi)` [if you import the `math` module as `m`]. It should read as: 200 0.0 0.0. Plot (f, f', f'', and f''') versus x, label your axes, and label the various curves with an informative legend. Make those labels, and any curves/symbols nice and clear and big (no tiny fonts that hurt our eyes!).

(b) Compute the numerical finite-difference first-order derivative using the standard (our first) forward-difference (f-d) and central-difference (c-d) schemes, and plot the answer as a function of x, using the same x range as in (a). Show your answers for a step size of $h = 0.15$ and $h = 0.5$[B], and

[A] This is also easy to do analytically of course, but the point is to know how to use `SymPy` to do this for you

[B] We are using units of x, so you do not need to specify units here compare these with the analytical solution (on the same graph). Again, label your axes, and the curves with a legend.

(c) Next, at the fixed $x = 1$ value, show the f-d and c-d *absolute errors* as a function of h, when h is allowed to vary over 16 orders of magnitude, from 10^{-16} to 1. Label your axes, and the curves with a legend. Also show the analytical estimates using the estimated numerical error formulas in the lecture notes (note, you will also need to compute the machine precision). Use a sensible scaling for the step size, so it changes iteratively by an order of magnitude, for each iteration, e.g., 0.01, 0.001, etc.

Question 2 [4 pts]

This question deals with computing the Legendre Polynomials (LPs). First we will motivate the use of LPs, which appear frequently in the solution to physics and engineering problems, and give some simple background. The LP functions are complete and orthogonal polynomials[C], with a number of useful mathematical properties. For example, their orthogonality is represented by:

$$\int_{-1}^{1} dx P_m(x) P_n(x) dx = \frac{2}{2n+1}\delta_{mn}, \tag{2}$$

where δ_{mn} denotes the Kronecker delta, equal to 1 if $m = n$ and to 0 otherwise.

As an example application, a common way of representing the solution to a point charge in electrostatics is through the LPs. For a single point charge q_0, located at $\mathbf{r}_0$, the electrostatic potential at any position $\mathbf{r}$ is

$$\phi_0(\mathbf{r}) = k\frac{q_0}{|\mathbf{r}-\mathbf{r}_0|}, \tag{3}$$

which can be problematic to compute. A more useful series expansion can be obtained in terms of LPs, from the relation

$$\frac{1}{|\mathbf{r}-\mathbf{r}_0|} = \frac{1}{r}\sum_{n=0}^{\infty}\left(\frac{r_0}{r}\right)^n P_n(\cos\theta_0), \tag{4}$$

which assumes $r > r_0$. Defining $u \equiv r_0/r$ and $x \equiv \cos\theta_0$, then we can write

$$\frac{1}{\sqrt{1-2(r_0/r)\cos\theta_0 + (r_0/r)^2}} = \sum_{n=0}^{\infty} u^n P_n(x), \tag{5}$$

and the function on the LHS is called the *generating function* of LPs.

The first five LPs are given as:

$$P_0(x) = 1,\ P_1(x) = x,\ P_2(x) = \frac{1}{2}(3x^2-1), \tag{6}$$

$$P_3(x) = \frac{1}{2}(5x^3-3x),\ P_4(x) = \frac{1}{8}(35x^4-30x^2+3). \tag{7}$$

However, all the LPs can be obtained by a recurrence relation known as *Bonnet's recurrence relation*:

$$P_{j+1}(x) = \frac{(2j+1)xP_j(x) - jP_{j-1}(x)}{j+1}. \tag{8}$$

Indeed there is a special function for this in Python under `SymPy`. Below I show some code that could use this function, using the example of $n = 3$:

[C] So any function on that interval can be expressed as a linear combination of the functions in the set - this is crazy useful ('lit'), as you will see later in your more advanced courses!

```python
from scipy.special import legendre
Pn = legendre(3)
nsteps = 200
xs = [i/nsteps for i in range (-nsteps+1,nsteps)]
y = Pn(xs)
```

We will not use this ‘black box’ approach, but instead a sample code ‘legendre.py’ and ‘call_legendre.py’ is available on the course web pages, so you can see how this works. The latter code plots several LPs versus x, and shows you how to import the functions (modules). Note that the following line in python

```python
if __name__ == '__main__':
```

prevents an external code, that uses this function, from executing the lines below this; thus such lines are usually included in reusable codes containing functions that you might input later, to other codes. Note also that this code can compute P_n' (the LP derivative) as well, with a separate recurrence expression,

$$P_n'(x) = \frac{nP_{n-1}(x) - nxP_n(x)}{1-x^2}. \tag{9}$$

(a) To the question: An alternative way to obtain the LPs is through the *Rodrigues' formula*

$$P_n(x) = \frac{1}{2^n n!}\frac{d^n}{dx^n}[(x^2-1)^n], \tag{10}$$

which is the one you will code up in Python, using our c-d derivative formula.

For a step size of $h = 0.01$, implement the Rodrigues' formula using standard c-d subroutines, and compare the results for the first 4 non-trivial LPs (using the sample code, so with $n = 1$ to $n = 4$). Plot these together, for different values of n (so a separate graph per n, to make it clear). Some example *code snippets* for this is given below:

```python
def calc_cd_1(f,n,x,h):
    cd = (f(n,x+h/2) - f(n,x-h/2))/h
    return cd

def calc_cd_2(f,n,x,h):
    cd = (calc_cd_1(f,n,x+h/2,h) - calc_cd_1(f,n,x-h/2,h))/h
    return cd

def calc_cd_3(f,n,x,h):
    cd = (calc_cd_2(f,n,x+h/2,h) - calc_cd_2(f,n,x-h/2,h))/h
    return cd
...
```

Bonus Question - for question 2 [say 2(b)] - if attempting, include in the same code, with different cells, clearly labelled (2pts)

Obviously the code above is not efficient for large n. Imagine doing this for $n = 50$! Thus, improve your algorithm to avoid the need to write multiple functions, by recursively calling the main c-d function, or using iteration, and verify it works by comparing again with the LPs from the sample code, now going easily from say $n = 1$ to $n = 8$ (i.e., without having to change any code at all, apart from the desired n). To be clear, your improved subroutine should only need a few lines of code. Note also the factorial function is available within the `math` module.