Instructor: Dr. S. Hughes

Due date: Sun, Jan 28 (midnight)

Lab 2 - Numerical Integration

Keywords: Rectangular Rule, Trapezoidal Rule, Simpson's (1/3) Rule, Error Function, 2D Simpson's Rule, `dblquad` from `scipy.integrate`, and `quad` from `mpmath`

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 (preferable, or no more than several) .py (called LabX_YourLastName.py) 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 code by using #%% (and label section in code), and add useful comments, so we can step through it easily.

Acknowledgements: Please comment on any help that you received from your group members or others concerning this Lab assignment.

Some Background (in addition to lecture notes):

Simpson's Rule - Wikipedia
Python Matrices
Rub Landau - Lecture on Numerical Integrations
Error Function - Wikipedia

Question 1

(a) In the lecture notes, I gave a "bad" example of the standard *rectangular method* for numerical integration. First fix the error in the code. Then rewrite this as an efficient subroutine, where you pass the function of interest, the start and end points of the integration, and the total number of points, n, that will represent the function on a finite size grid. As in the lecture notes, use the definition of the error function, $erf(z)$, which appears frequently in Physics and Engineering problems (often as part of the solution), and can be defined from the following integral:

$$erf(z) = \frac{2}{\sqrt{\pi}} \int_0^z \exp(-x^2). \tag{1}$$

(b) Write two more Python subroutines to carry out the integral, now using the *trapezoidal method* and *Simpson's method*, and evaluate the solution to the integral at $z = 1$. Using a total number of steps, $n = 100$ and $n = 101$, print out the percentage *relative error*, by comparing with Python's `m.erf(1)` (e.g., from the `math` module, if importing as m), comparing all three numerical integration methods. The relative error in percent is:

$$\text{RE} = \left| \frac{(ans_{\text{num}} - ans_{\text{exact}})}{ans_{\text{exact}}} \right|, \tag{2}$$

and if we want in percent we can multiply that by 100.

Briefly comment in your code or write out to the screen an explanation of how the errors change for the two different n. Use this knowledge to make some of your subroutines more robust and sensible (e.g., with a warning or a clean exit under certain conditions – probably a "user warning" is better, at least for testing).

(c) It can be tedious and potentially problematic, when carrying out numerical integration and just guessing it works, and then checking that the result is probably accurate for large n (numerical cells to evaluate). We can do better than this with 'adaptive step techniques'.

Implement a subroutine `adaptive_step` (adaptive step approach), which will evaluate the integration of the same function in (b) to a precision of less than 10^{-10} relative error (not in percent, so the RE given above).

Starting with $n = 3$[A], and then doubling and subtracting 1 (i.e., $n' = 2 * n - 1$), every iteration, use the trapezoidal and Simpson methods of numerical integration to see how many step sizes (and iterations) are required to achieve this level of accuracy (print out your solutions). Ideally you want to pass the named subroutines that you have already written, and this subroutine should be as compact and efficient as possible.

Question 2

(a) Write a 2D version of Simpson's rule for numerical integration, where you will pass lower and upper bounds for each dimension (e.g., a, b and c, d), the total number of points for each dimension (say n, m), and a 2d function (such as: $f2d$). Thus your subroutine will look something like this: `simp2d(f2d,a,b,c,d,n,m)`. For efficiency, and readability, set up a "weight matrix" and use `meshgrid` for efficient vectorization.

Then apply your subroutine to compute the following integral:

$$A = \int_0^{\pi} dx \int_0^{\pi/2} dy \, \sqrt{x^2 + y} \, \sin(x) \cos(y), \tag{3}$$

and test how the results change for $n, m = 101, 101$, $n, m = 1001, 1001$, and $n, m = 51, 101$. Using $n, m = 1001, 1001$, I obtain an area of $A = 3.5389940350753895$, and for $n, m = 51, 101$, I obtain $A = 3.5389937460658536$.

(b) Check your answer using an alternative approach in Python, specifically now using a `lambda` function and the `quad` integration routine from the excellent `mpmath` *link*) module[B]. Notably, this method should only take two lines of code! Using this method, I obtain $A = 3.53899403553488$.

(c) Let's do one more way for fun! Next use the `dblquad`[C] *link* routine from `scipy.integrate`, which can be imported with: `from scipy.integrate import dblquad`. Specifically, this routine uses a technique from the Fortran library QUADPACK.

Using this final approach, check the answer again, in 1 line of code (since we already have the `lambda` function). I obtain: $A = 3.538994035624072$, and note this function also returns the estimated error of $err \approx 4.e{-}08$ (second argument). This order of magnitude error is expected

[A] Perhaps it is more natural to start with $n = 2$ here, but we want to use the Simpson's approach with an *odd* number of points. We could also implement a Simpson subroutine that adds 1 if n is even, and in fact that is probably better!

[B] For example, you can read in the module in the standard way through: `import mpmath as mp`. Also note the same module can be used for the trig functions like `sin(x)`, though you can also use `pympy`.

[C] This can be changed to also do triple integrals: `tplquad`.

as it is around the default setting for the relative error (see docs), but this can also be reduced by passing an extra argument: `epsrel`, e.g., `epsrel=1.e-10`. Note that this method also works on functions that have been created with `lambdify` (which returns a number, not a symbolic function), since it is basically an efficient numerical integration based on quadrature.