# PHY407 Lab 1

Monday, September 30, 2024

Jace Alloway - 1006940802 - alloway1

---

This assignment was compiled in VS Code using the Python and LaTeX extensions. Matplotlib does not produce inline plots in terminal, so the `%matplotlib inline` command was commented out. **Collaborators for all questions: none.**

**Equations:**

$$\sigma^{(1)} \equiv \left[\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2\right]^{1/2} \tag{1}$$

$$\sigma^{(2)} \equiv \left[\frac{1}{n-1}\left(\sum_{i=1}^{n}x_i^2 - n\overline{x}^2\right)\right]^{1/2} \tag{2}$$

$$\overline{x} \equiv \frac{1}{n}\sum_{i=1}^{n}x_i \tag{3}$$

$$\sigma^{(3)} = C\sqrt{N}\sqrt{\overline{x^2}} \tag{4}$$

$$\frac{\sigma^{(3)}}{\sum_{i=1}^{n}x_i} = \frac{C}{\sqrt{N}}\frac{\sqrt{\overline{x^2}}}{\overline{x}} \tag{5}$$

$$\%\text{ Difference} = \frac{|x-y|}{x} \tag{6}$$

## Problem 1

(**a**) To test the relative error found when estimating the standard deviation using equations (1) and (2), the equations were first defined, `cdata.txt` was loaded in using `np.loadtxt()`, and the base case determined by calling `np.std(..., ddof=1)` was computed. After computing the standard deviations from equations (1) and (2), they were compared against the `np.std(..., ddof=1)` call by defining the relative error function. The exact steps were outlined below and are included in the python file `q1.py`.

```
    PSEUDOCODE


        In computation:

INITIALIZE st_Dev1, st_Dev2 function calculations from eq(1) and eq(2)
INITIALIZE rel_Err function calculation as percentage error formula
OBTAIN cdata.txt with np.loadtxt() as the INIT input array
COMPUTE numpy standard deviation by calling np.std on input array
COMPUTE eq(1) and eq(2) standard deviations by calling st_Dev function
COMPUTE relative error by calling st_Dev function
COMPUTE difference of rel_Err output with np.std() output
PRINT relative error outputs and differences


        In initializing the st_Dev1, st_Dev2, rel_Err functions:

INITIALIZE st_Dev1 with 1 input array: x
CALCULATE length of x, the number of elements
IF len(x)<=1, raise a ValueError, since len(x) must be greater than 1
CALCULATE mean of x
INITIALIZE an array of 0's of len(x)
CALCULATE the (x[i] - mean(x))**2  elements, overwrite the 0's array
CALCULATE square root, multiply by (1/(n-1))
RETURN value

INITIALIZE st_Dev2 with 1 input array: x
CALCULATE length of x, the number of elements
IF len(x)<=1, raise a ValueError, since len(x) must be greater than 1
CALCULATE the square root element in one pass,
        sum(x[i]**2) - ((1/n)*(sum(x)))**2
IF the square root value is negative, raise a value error, cannot cast
        complex values into np.float64 type
CALCULATE square root, multiply by (1/(n-1))
RETURN value

INITIALIZE rel_Err with two input arrays, measured and actual
CALCULATE the absolute difference of measured - actual, divided by actual
RETURN value
```

**(b)** After computing the output(s) from (a), which are shown below, it was found that the relative error computed from equation (2) was larger.

```
Outputs from cdata.txt:
Relative Error (1): 0.0 Relative Error (2): 2.2873460336752e-09
Error from eq(2) is larger.
```

The explaination as to why the error is larger is written in part (c).

**(c)** To further explore the question of accumulating error with equation (2), two different random normal distribution arrays were defined with `np.random.normal()`. Both distributions were given with parameters

```
mean, sigma, n = (0., 1., 2000)
mean, sigma, n = (1.e7, 1., 2000)
```

so distribution two has the name number of points and standard deviation, however with just a larger mean. Using the same function as defined in part (a) to compare the equation (1) and (2) values with the `np.std(..., ddof=1)` call, along with their relative errors, the comparisons were computed. The sample of the written outputs are included below:

```
Outputs from distribution 1 - normal(0., 1., 2000):
Relative Error (1): 0.0 Relative Error (2): 0.0
Errors from (1) and (2) are identical.

Outputs from distribution 2 - normal(1.e7, 1., 2000):
Relative Error (1): 0.0 Relative Error (2): 0.008020432102293596
Error from eq(2) is larger.
```

It was found regularly that the standard deviations varied between trials, especially with the distribution 2 of larger mean. Over many trials it was found that the relative error for equation (1) was quite consistent with the `np.std(..., ddof=1)` call, with the maximum error being $O(10^{-16})$ for the first distribution. For the second distribution, it was found that equation (2) tended to always produce a relative error larger than equation (1) in comparison with the numpy call.
The reason this occurs is because error accumulates differently with equation (1) than it does with (2). In equation (1), error is accumulated just in the $(x_i - \overline{x})^2$ term, while equation (2) accumulates error from both the $x_i^2$ sum and the $n\overline{x}^2$ terms, which can blow up error for large floating numbers in the code. This will occur regardless of whether or not the data is passed through once or twice. Furthermore, I will note that this occurs generously with distributions with larger mean values, since the rounding error will quickly 'blow-up' the smaller values.

**(d)** The workaround to the one-pass method is to compute iterations for the mean and variance and pass through the array via a for-loop. This is known as Welford's algorithm (I took this from https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance - *This is NOT my thinking, but I can explain the logic behind it...* I had a hard time trying to find any other 'simple' method, although this algorithm isn't exactly 'simple'). It is derived as follows:

$$\begin{aligned}
(n-1)\sigma_n^2 - (n-2)\sigma_{n-1}^2 &= \sum_{i=1}^{n}(x_i - \overline{x}_n)^2 - \sum_{i=1}^{n-1}(x_i - \overline{x}_{n-1})^2 && (7)\\
&= (x_n - \overline{x}_n)^2 + \sum_{i=1}^{n-1}[(x_i - \overline{x}_n)^2 - (x_i - \overline{x}_{n-1})^2] && (8)\\
&= (x_n - \overline{x}_n)^2 + \sum_{i=1}^{n-1}[x_i^2 + \overline{x}_n^2 - 2x_i\overline{x}_n - x_i^2 - \overline{x}_{n-1}^2 + 2x_i\overline{x}_{n-1}] && (9)\\
&= (x_n - \overline{x}_n)^2 + \sum_{i=1}^{n-1}(2x_i - \overline{x}_n - \overline{x}_{n-1})(\overline{x}_{n-1} - \overline{x}_n) && (10)\\
&= (x_n - \overline{x}_n)^2 + (\overline{x}_{n-1} - \overline{x}_n)(n-1)(2\overline{x}_{n-1} - \overline{x}_n - \overline{x}_{n-1}) && (11)\\
&= (x_n - \overline{x}_n)^2 + (\overline{x}_{n-1} - \overline{x}_n)(n-1)\left(2\frac{n\overline{x}_n - x_n}{n-1} - \overline{x}_n - \frac{n\overline{x}_n - x_n}{n-1}\right) && (12)\\
&= (x_n - \overline{x}_n)^2 + (\overline{x}_{n-1} - \overline{x}_n)(\overline{x}_n - x_n) && (13)\\
&= (x_n - \overline{x}_n)(x_n - \overline{x}_n - \overline{x}_{n-1} + \overline{x}_n) && (14)\\
&= (x_n - \overline{x}_n)(x_n - \overline{x}_{n-1}) && (15)
\end{aligned}$$

where we have applied the recursion relation

$$\overline{x}_n = \frac{(n-1)\overline{x}_{n-1} + x_n}{n} \qquad (16)$$

and $\overline{x}_n$ is defined as in equation (3). In imposing (15) to the code, we calculate each following term in the series based on the previous terms. Let $M_n$ be the mean difference value, $M_n = \sum_{i=1}^{n}(x_i - \overline{x}_n)^2$, hence from (7)

$$M_n = M_{n-1} + (x_n - \overline{x}_n)(x_n - \overline{x}_{n-1}) \tag{17}$$

so the squared, unbiased variance is $\sigma_n^2 = \dfrac{M_n}{n-1}$ from (7). We therefore implement the algorithm to first initialize the $M_0 = 0$ variable, then compute recursions as followed until $M_n$ is obtained. Then one can evaluate for $\sigma_n$. The pseudocode is then

```
    PSEUDOCODE

INITIALIZE reduceErrEq2 function, which takes in an array x
COMPUTE iterable variance difference with regards to each term according
                    to Welford's algorithm (single-pass)
RETURN value


In initializing reduceErrEq2 function:

DEFINE AND INITIALIZE new variables for accumulated mean M, accumulated
                    variance (squared) S
COMPUTE length of x, INITIALIZE as n
COMPUTE iterable terms 'sample' and 'previous mean' via FOR loop
                    iterating through x
COMPUTE iterated (squared) variance, iterated mean
COMPUTE square root of (squared) variance over n-1, for non-biased variance
RETURN value
```

In programming the Welford algorithm, it was found that the error was significantly reduced:

```
The relative error with np.std from the Welford algorithm to dist1, dist2 is:
(RE.dist1, RE.dist2) = (3.346586027127443e-16, 8.910293341740408e-11)
```

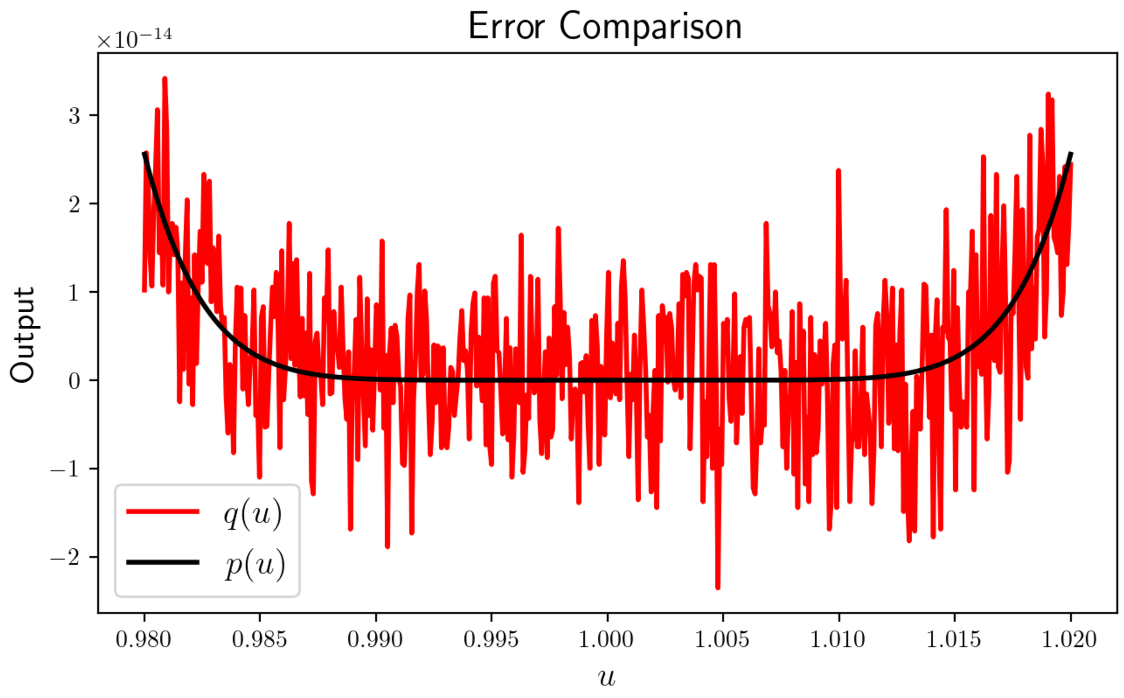
which was expected, because error was not accumulated for a single-pass while calculating the mean of the larger distribution.

## Problem 2

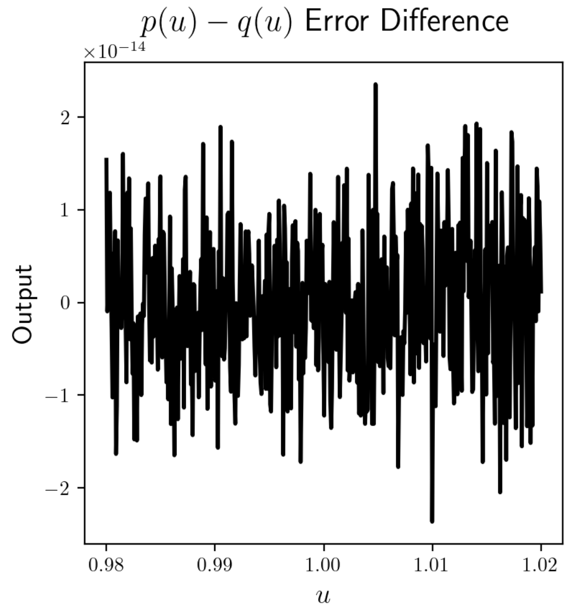
**(a)** $p(u)$ and $q(u)$ were plotted using `matplotlib.pyplot`.

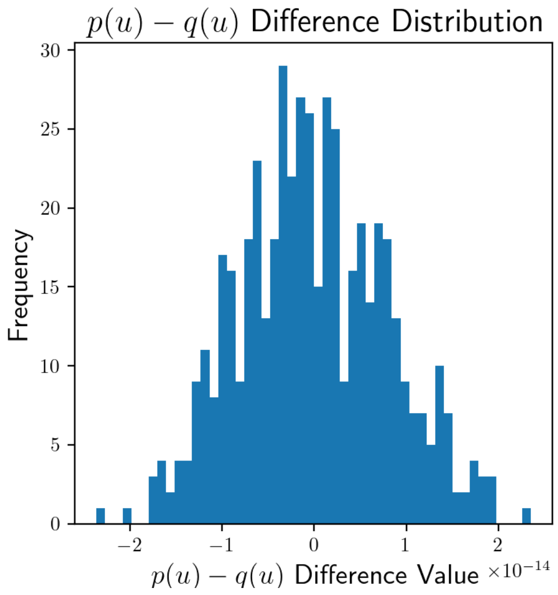
$$p(u) = (1-u)^8 \tag{18}$$
$$q(u) = 1 - 8u + 28u^2 - 56u^3 + 70u^4 - 56u^5 + 28u^6 - 8u^7 + u^8 \tag{19}$$

While the plot of $p(u)$ appears smooth, the error accumulating from $q(u)$ is significantly greater since there is an error associated with each term, which is then added as $\sigma = \sqrt{\sigma_1^2 + \sigma_2^2 + \ldots}$.



**(b)** The error was then isolated and plotted by examining the quantity $p(u) - q(u)$, along the same domain $0.98 < u < 1.02$.





From the histogram, it can be seen that the distribution of error differences follow a Gaussian, hence containing a standard distribution, which aligns with equation (4). To verify this, the numpy standard deviation was calculated with `np.std(..., ddof=1)` and using equation (4).

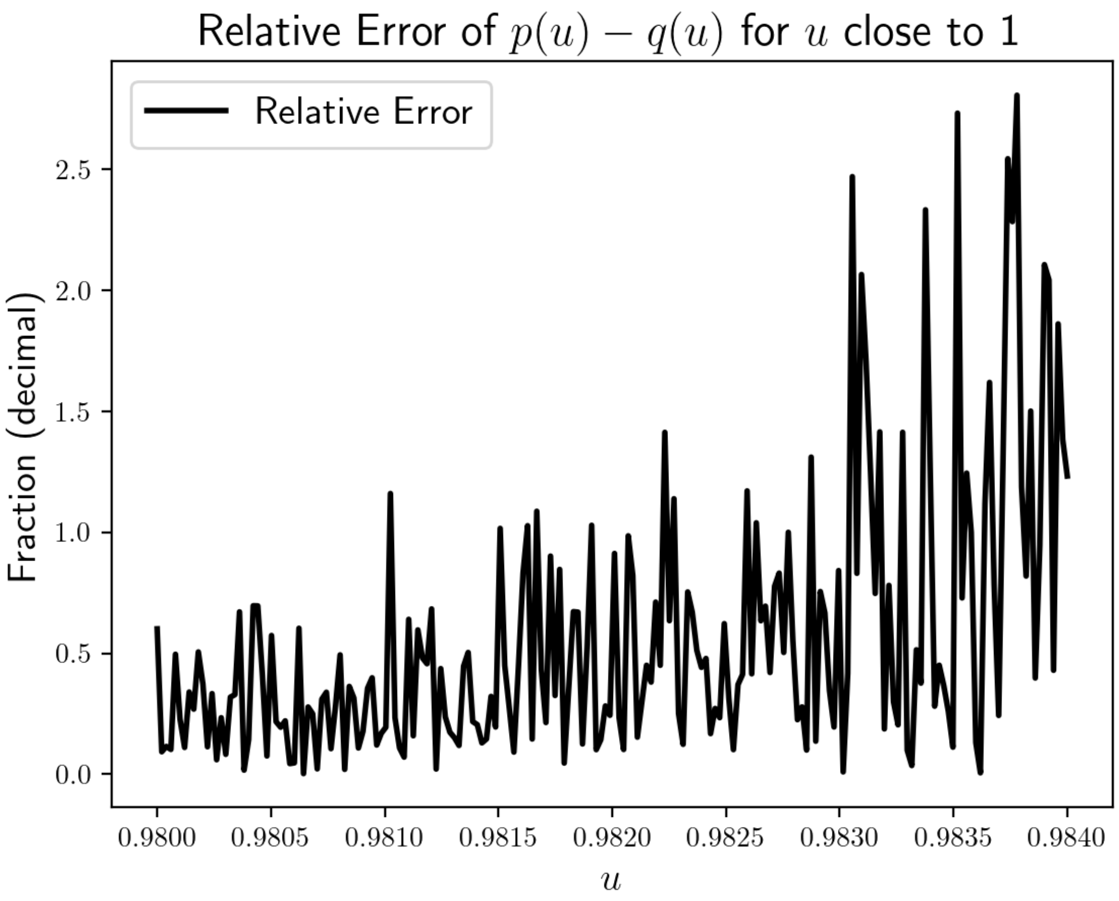
```
The deviation error estimate for the difference p(u) - q(u) is
(8.000296869673734e-15, 6.732769984159082e-15, 0.15843498137167802),
the numpy standard deviation, the manually determined
polynomial deviation, and the percentage error between the two.
```

First note that the relative error between the two methods is approximately $16\%$. In applying equation (4), some error needed to be propagated between both $p$ and $q$ functions. First note that python rounds after computing exponents, so the error associated with $p(u)$ is just $Cu^8$, since the '$-1$' has no error. The error associated with $q(u)$ is then $\sqrt{|Cu^0|^2 + |Cu^1|^2 + \cdots + |Cu^8|^2} * q(u)$. These errors were then added, as previously described, and the squared mean in equation (4) was calculated with `np.sqrt(np.mean(np.square(error)))` from which this was multiplied by $C\sqrt{N}$.

**(c)** Equation (5) was then implemented to determine the error relative to $C$, using all of the same methods in calculating error.

```
The fractional error of the difference p(u) - q(u) is
                    1.3460272650682308*C, approximately 1.0*C.
```

To double check that the error approaches 100% around the u = [0.980, 0.984] range, the relative error between $p$ and $q$ was plotted according to equation (6):

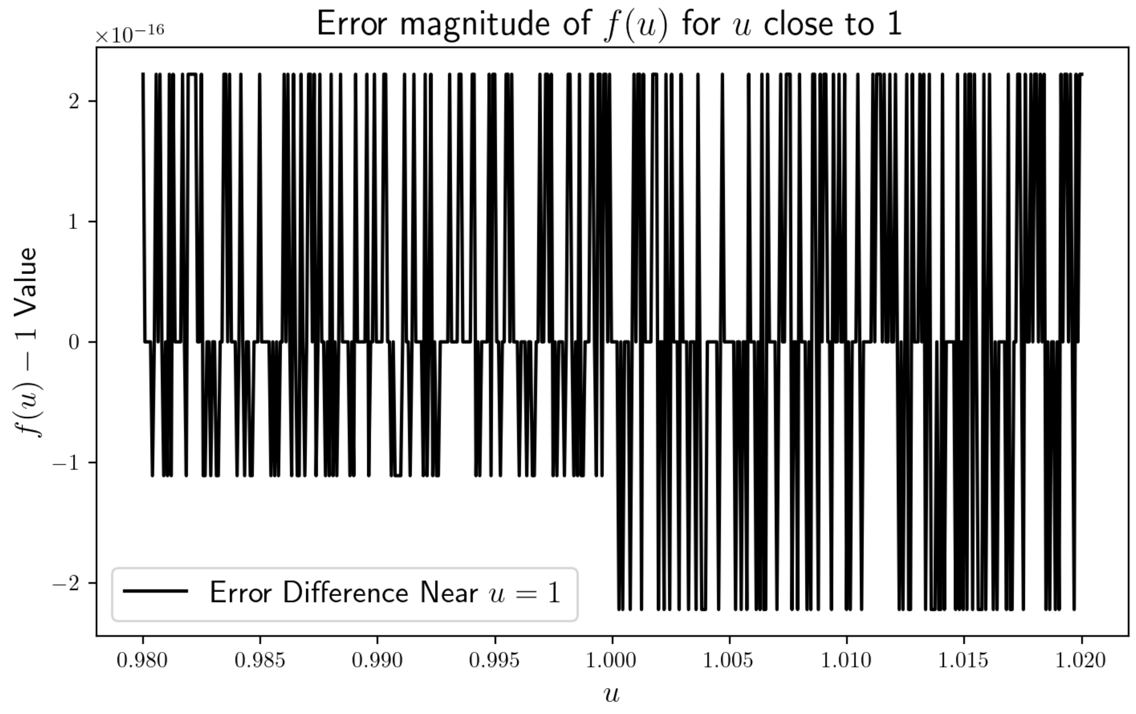


from which one can see the error approaching over 100% between the (0.9810, 0.9830) interval.

**(d)** Error accumulation was also investigated by examining the function

$$f(u) = \frac{u^8}{u^4 \cdot u^4} \tag{20}$$

Using the same error functions defined through parts (a) - (b), the error was propagated using the manually-determine calculation, the `np.std(..., ddof=1)` call, and with the equation (4) representation of the deviation.
The manual error was propagated as described before, being $Cu^8$ in the numerator and $\sqrt{2}Cu^4 * u^4$ in the denominator (we are multiplying two terms in the denominator, hence the $\sqrt{2}$). The error were then concatenated according to the multiplication rule.
The relation $f(u) - 1$ was then plotted to show the error magnitudes. Note that differences for points $u < 1$ appear to only be half the magnitude of the rest of the errors, and the reason why this occured is unknown to me. If I had to guess, it would be because for $u < 1$, the $\frac{1}{u^4 \cdot u^4}$ term dominates, which can be considered as more "unstable" ($\sim \sqrt{2}C$) since error will accumulate in a different way, while for $u > 1$, the $u^8$ term dominates, which is more "stable" in producing error ($\sim C$).



In executing the script, the error was compared to equation (4.5) in the Newman text, which are written below:

```
The np.std value, the eq(4.5) value from text, and my manually computed
deviation in error is
1.4033833856493925e-16, 1.414213562373095e-16, 1.7320508075688774e-16,
respectively.
```

The key idea here is that the errors are nearly identical, with very little differences between the three of them. This implies that, in calculating errors from computed values in python, either three of these methods are valid and are on the same order of magnitude as each other (and of the proportionality constant $C$).

## Problem 3

(**a**) Evaluate the integral $\int_0^1 dx\, \frac{4}{1+x^2}$:

$$4\int_0^1 \frac{dx}{1+x^2} = 4\int_{\arctan(0)}^{\arctan(1)} d\theta \frac{\sec^2\theta}{1+\tan^2\theta} \qquad (x=\tan\theta;\ dx=\sec^2\theta\, d\theta) \tag{21}$$
$$= 4\int_{\arctan(0)}^{\arctan(1)} d\theta \frac{\sec^2\theta}{\sec^2\theta} \tag{22}$$
$$= 4\left[\arctan(1)-\arctan(0)\right] \tag{23}$$
$$= 4\cdot\frac{\pi}{4} \tag{24}$$
$$= \pi. \tag{25}$$

(**b**) The trapezoidal rule and Simpson's rule were both written into the python code as functional functions (they take in the functions we wish to integrate as well as their derivatives to estimate the error produced). The conditions used was that an even number of slices needed was required to execute the Simpson's call. Pseudocode was written and is included below for defining these functions: The trapezoidal and Simpsons slice integral values were computed as outline in Newman's text, along with their expansion correction errors ($A[k]$ represents the area of the k-th slice, the bounds are denoted as $(L,U)$, and $n$-th order corrections as $\delta^{(n)}$):

$$A_T[k] = \frac{h}{2}\left[f(L+(k-1)h)+f(L+kh)\right] \tag{26}$$
$$\delta^{(1)}I_T = \frac{h^2}{12}\left[f'(L)-f'(U)\right] \tag{27}$$
$$A_S[k] = \frac{h}{3}\left[f(L+2kh)+4f(L+(1+2k)h)+f(L+2(1+k)h)\right] \tag{28}$$
$$\delta^{(3)}I_S = \frac{h^4}{180}\left[f'''(L)-f'''(U)\right] \tag{29}$$

The pseudocode for the computation is then

```
    PSEUDOCODE

    For trapezoidal:
INITIALIZE trapezoidal functional, taking in func, bounds, num_slices,
        and the first derivative of func
IF len(bounds) != 2, RAISE ValueError, since we should only work with
         an upper and lower bound
DEFINE constants: lower and upper bounds, h (width)
COMPUTE slice width h based on num_slices and bounds
INITIALIZE array of 0's of length num_slices - 1, the number of area elements
FOR each entry in empty array, COMPUTE the trapezoidal value of the
        k-th slice
COMPUTE sum of aray
COMPUTE approximation error by CALLING derivative of func
COMPUTE rounding error by CALLING integral value * C
COMPUTE total error by applying adding rules
```

```
RETURN integral value, error value

    For Simpsons:
INITIALIZE Simpson's functional, taking in func, bounds, num_slices,
        and the third derivative of func
IF len(bounds) =! 2, RAISE ValueError, since we should only work with
         an upper and lower bound
IF num_slices %2 != 0, RAISE ValueError, since the number of slices must be
         even to execute
DEFINE constants: lower and upper bounds, h (width)
COMPUTE slice width h based on num_slices and bounds
INITIALIZE array of 0's of length int(num_slices/2), the number of area
        elements (we iterate through every other element)
FOR each entry in empty array, COMPUTE the area element based on the left,
        center, and rightmost elements
COMPUTE sum of array
COMPUTE approximation error (3rd order) by CALLING third derivative of func
COMPUTE rounding error by CALLING integral value * C
COMPUTE total error by applying adding rules
RETURN integral value, error value

    For integral computation:
INITIALIZE rat_func, rat_funcprime (1st deriv), rat_funcppprime (3rd deriv)
CALL trapezoidal / simpsons functionals, CALL rat_func and derivatives
COMPUTE integrals
RETURN values and relative error compared to np.pi
```

Where the approximation is added to the rounding error. The obtained values for the relative errors with $\pi$ were

```
Trapezoidal Percentage Error: 0.10498749649498862
Simpsons Percentage Error: 7.647757510904547e-06
Trapezoidal Error: 0.010416666666666666
Simpsons Error: 3.1415686274509803e-16
```

**(c)** To determine an approximate number of slices until the error reaches $O(10^{-9})$, a for loop was created to iterate through values using a toggleable counter until the number of slices computed an integral with an error reduced to $O(10^{-9})$.

```
PSEUDOCODE

INITIALIZE array of powers for 2^n
INITIALIZE t-range and s-range toggles to toggle if statements on or off
         once a condition is satisfied
FOR n in powers:
    INITIALIZE order (10^-9)
    COMPUTE N = 2**n
    COMPUTE error from trapezoidal integration using correction formula
    COMPUTE error from simpsons integration using correction formula
    COMPUTE time differences for integrations

    IF the t-range counter is FALSE and the trapezoidal error is
             less than the order:
        RETURN the 2**n-value, error, and time
        SET t-range = TRUE to not return to this if statement

    IF the s-range counter is FALSE and the simpsons error is
             less than the order:
        RETURN the 2**n-value, error, and time
        SET s-range = TRUE to not return to this if statement

    IF s-range AND t-range are both TRUE:
        BREAK the for-loop
```

In general, it was found that the amount of time elapsed to integrate over the Simpson's rule was on the $O(10^{-5})$ seconds, and $O(10^{-2})$ seconds for the trapezoidal integral. This was calculated my importing and calling `time.time()`. I will note that the times recorded were functions of $n$, and that the time elapsed is not accumulated for all $n$ calculations. The outputs were as followed:

```
Simpsons: num_slices 4 ( n =  2 ) with error 3.1415686274509803e-16
        integrated at a time of 5.1975250244140625e-05
Trapezoidal: num_slices 4096 ( n =  12 ) with error 9.934107462565104e-09
        integrated at a time of 0.011295080184936523
```

Note that the approximation error is $0$ for the Simpson's calculation, and this follows from the way the third derivatives of the rational function operate:

$$f(x) = \frac{4}{1+x^2} \tag{30}$$

$$f'(x) = \frac{-8x}{(1+x^2)^2} \tag{31}$$

$$f'''(x) = \frac{96x(1-x^2)}{(x^2+1)^4} \tag{32}$$

(evaluating (29) from (32) between $(0,1)$ implies $\frac{h^4}{180}(0-0) = 0$).

**(d)** The practical estimation of errors was applied to the trapezoidal rule by defining a function to take in the integral values (I didn't write pseudocode for this - it is straightforward define and call):

$$\text{trapezoidal\_errEst} = \frac{1}{3}(I_2 - I_1) \tag{33}$$

where $I_2$ represents the integral at the finer $N$ resolution, and $I_1$ represents the integral taken at a coarser resolution ($N_1 < N_2$). The error estimate for the $N = 32$ slices was then determined by just calling this function:

```
Trapezoidal Practical Error Estimation Value of 32 slices:
        0.004422873203949074
```

**(e)** As previously mentioned in $(c)$, calculating errors for Simpson's rule according to $(29)$ is not possible in this case because our bounds, $(0, 1)$, will always yield $0$ for the error, regardless of the number of slices. Hence the only error on the integral would be that of the rounding error, which will always be $\approx \pi \times 10^{-16}$. To find a workaround, the best option would be to break the integral into two different intervals (ie, $(0, 0.5)$ and $(0.5, 1)$), calculate the error for them both individually, and then add them.