PHY407 Lab 2 Assignment

Vincent Fan 1002563380
Satchel Page 1001695686

Question 1

a)

Pseudocode:

```
#load text file
#define function for standard deviation method 1
    #define mean as sum of data/length of data
    #initialize an int to store sum of differences
    #for each value, define sum of difference as (value – mean) ** 2
    #standard dev is the square root of (sum * 1/n-1)
    #return standard deviation
#define function for standard deviation method 2
    #initialize 2 int to store sum for mean and sum of squared
    #for each value, sum of square difference is value**2, and add value to sum for mean
    #check if sum is positive, if not, print warning and make it larger
    #standard deviation will be square root of
        ((sum of square – (sum for mean **2/length of data)) * 1/n-1)
    #return standard deviation
#define reference answer with numpy.std method
#relative error of results using eq (5) (y-x)/y
#relative error of results using eq (6) (y-x)/y
```

b)

The idea for this question is to look at some of the numerical issues that come from calculating the standard deviation of a data set using different methods within python. To begin, we look to contrast two distinct methods of calculating standard deviation for a sample, and one for the mean. The Method of calculating the mean is the standard sum of the data points divided by the number of data points in the sum. The first standard deviation formula is below, along with the formula for the mean. It takes the sum of the squared difference between each data point and mean, and divides it by n-1, and square roots the result.

$$\bar{x} \equiv \frac{1}{n}\sum_{i=1}^{n} x_i \quad \text{and} \quad \sigma \equiv \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

(We will call this method 1)

The second method of finding the standard deviation uses the formula below, which sums first the square of each data point, and then subtracts from the squared mean multiplied by the number of data points, with the rest remaining the same.

$$\sigma \equiv \sqrt{\frac{1}{n-1}\left(\sum_{i=1}^{n} x_i^2 - n\bar{x}^2\right)},$$

(We will call this method 2)

We wrote two distinct function in python to represent each of these methods, naming them std5 and std6 respectively, and applied them to a data set of values for the speed of light in 10^3km/s. To test the accuracy of these methods, we then wanted to contrast them with the known working method numpy.std function in python. To do this, we calculated the relative error using the formula (y – x)/y where y would be the numpy method, and x would be the returned value of std5 or std6 respectively.

Our output of these relative errors was:

```
The relative error of method 1:  3.512894971845343e-16
The relative error of method 2:  3.740397602946237e-09
```

Giving the first method of calculating standard deviation a decided advantage of accuracy over the second by a factor of 10^6.

c)

We now want to contrast the effectiveness of the two methods further. To do so, we generated two sets of numbers, both with 2000 entries and standard deviation equal to 1, but one set of numbers has a mean value of 0, and the other has a mean value of $1*10^7$. The idea here is to contrast the relative error of the two methods with the known accurate numpy.std function, and explore the effects of varying the mean value with the two methods. Defining the data set with the mean value of zero to be set 1 and the other to be set 2, we observe the output of our relative error using the two methods with data set 1 to be:

1st run

```
The relative error of method 1 of set 1:  7.86988877747e-16
The relative error of method 2 of set 1:  5.62134912676e-16
```

2nd run

```
The relative error of method 1 of set 1:  1.119130936150906e-16
The relative error of method 2 of set 1:  3.3573928084527175e-16
```

3rd run

```
The relative error of method 1 of set 1:   2.1869513989116554e-16
The relative error of method 2 of set 1:   6.560854196734967e-16
```

Both numbers outputted have extremely similar values and, more importantly, of the same magnitude, making It a bit redundant to isolate one to be better over the other. Note that we have ran the program several times since the data sets are randomized. Doing the same thing but using data set 2 instead, we obtain the following values:

1st run
The relative error of method 1 of set 2: 2.242155594169091e-16
The relative error of method 2 of set 2: 0.16222317688140392

2nd run
The relative error of method 1 of set 2: 6.596422862750849e-16
The relative error of method 2 of set 2: 0.062417238713495006

3rd run
The relative error of method 1 of set 2: 4.3943704239057196e-16
The relative error of method 2 of set 2: 0.12361875674316448

Here, when the mean value is of the order of 10^7, we can observe a much more dramatic difference in the outcome of the two methods. Clearly using method 1 is superior when checking the result relative to the known correct answer, because the relative error is on the magnitude of 10^{13} lower than the relative error using method 2.

This is likely because method two squares values before takin the difference, meaning that both numbers are very close to each other. So, some values would be rounded off, thus creating round off errors. For example, if we take an input of 10,000,000.5 (which is in our range) and square it, we obtain 100,000,010,000,000.2, which when compared to squaring a value of just 10^7 produces a difference of 10,000,000.2. This is a huge difference, even with such a tight range of values, and such a dramatic effect from applying the square before takin the difference is very likely that caused the difference in relative errors here.

d)

We now look back to the eq (6) and examine the cause of larger relative error. For part c, we realized that if we compute the standard deviation with smaller numbers, i.e. set 1, with our method 2, the relative error is on the same magnitude as using method 1. Therefore, we will try to shift our light speed data close to 0 same as set 1. First, the algorithm will randomly pick an element from the data. Second, the algorithm will subtract the random element from each data point. Then, the rest will be the same as before. Finally, we have these results:

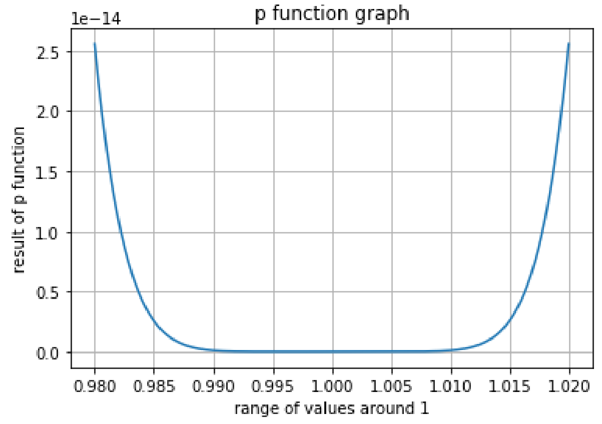
1st run
The relative error of method 6 with improved method: 1.7564474859226715e-16
2nd run
The relative error of method 6 with improved method: 8.782237429613357e-16
3rd run
The relative error of method 6 with improved method: 5.269342457768014e-16

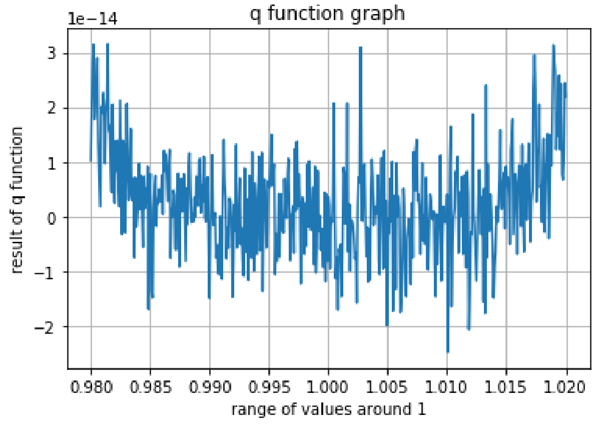
As we can see, the relative error is much smaller, and it is on the same magnitude as method 1. This works due to standard deviation does not change if we shift the data sets since it is a measure of how spread out the data is. Randomizing the choice of adjustment would also be more fair since we cannot handpick an element without bias.

Question 2 For this question, the idea is to further explore the different sources of numerical error which can come up by varying calculation methods on python. In this question, we want to compare an 8th degree polynomial to itself, only rewritten as either its compact form $p(u) = (1 - u)^8$ as well as in its expanded form
$q(u) = 1-8u + 28u^2 - 56u^3 + 70u^4 - 56u^5 + 28u^6 - 8u^7 + u^{8.}$
These are the same algebraic expressions, but not the same numeric expression, so to explore roundoff error, we are going to apply different tests to both these functions and explore the results.

a)

For the first part of the question, we want to graph both p(u) and q(u) very close to u=1, which we did by generating about 500 values of u in the range of 0.98 to 1.02, these were the plots that we obtained:





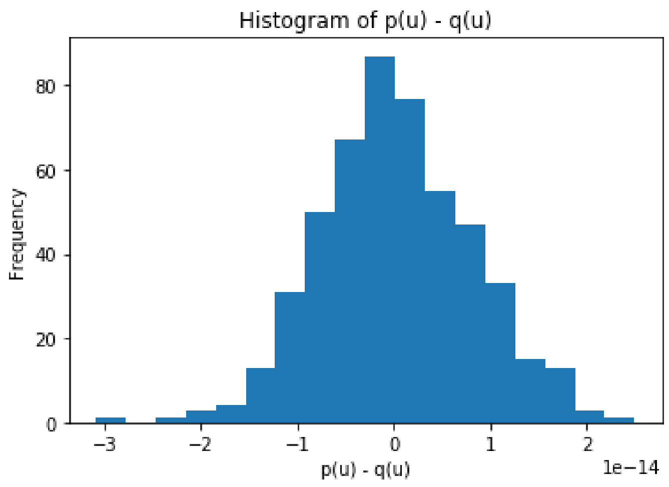
The q function certainly seems noisier. We believe these errors originates from rounding off values during the calculations. Since there are far more exponents in equation q, even though they are of the same degree, python is working through far more calculations which infers more likelihood of rounding off some numbers during the progress.

b)

Continuing from part a), we are now curious about the relationship p(u)-q(u) in terms of a histogram form, and then compare the information provided by the histogram in relation to the equation:

$$\sigma = C\sqrt{N}\sqrt{\overline{x^2}}, \quad \text{(Call this equation 1)}$$

This is an equation from the computational physics UofT Lab 2 write-up, which describes the standard deviation of a data set with respect to the number of calculations (N) and the mean of x squared as well as the constant C = e-16. First, we observe the histogram:



The distribution is fairly normally distributed about zero, suggesting at least that the noisier q function is on average tending to agree with the p function. Checking the standard deviation of this quantity using the numpy.std method, we obtain a value of:

numpy std method: 7.9030969279e-15

Now using the estimate equation for standard deviation offered a moment ago, we calculate the standard deviation to be:

Eq7 method: 1.13446022407e-14.

This is obtained using N = 44 which is the sum of the exponents and 8 additions (1+2+3+4+5+6+7+8) +8. And the mean of x squared was found by taking only the coefficients since our range of values were close to 1, we can assume all x goes to 1. Then we just take the sum of each term's coefficient squared. Note that if we further reduce the equation by canceling the N and the N from calculating the mean, we can just use the sum of x squared.

$$\sigma = C\sqrt{\sum x_i^2}$$

We can see that our value is about 50% off from the numpy.std method which is still considered to be about the same magnitude.

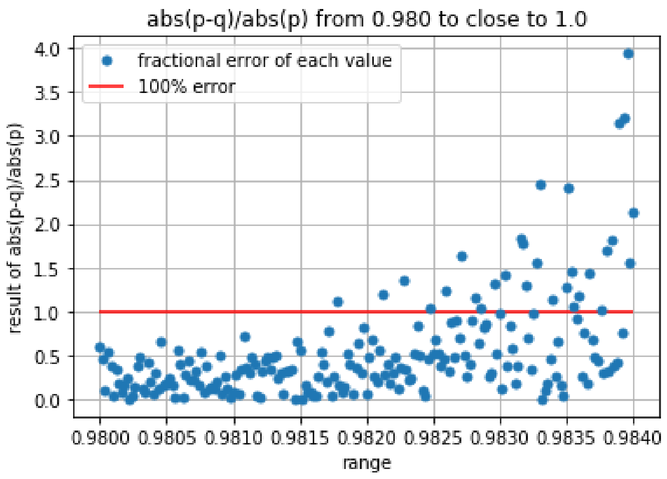
For a normal distribution, we expect around 68.3% of the data to lie within one standard deviation of the mean, and around 95% to lie within 2. This is precisely what we observe if we take the standard deviation to equal the statistical quantity of equation 7, suggesting that this is very closely related to the true standard deviation of the difference p(u) – q(u). This may also suggest that python has a random distribution of when it will round off values during calculation.

c)

Here we are interested in the magnitude of error produced by the fractional error formula:

$$\frac{\sigma}{\sum_i x_i} = \frac{C}{\sqrt{N}} \frac{\sqrt{\overline{x^2}}}{\overline{x}}$$

For x (or in our case u) values between 0.98 and 1.0. Our initial plots of these values diverge quickly around 0.984, so we plotted the values as a function for the full range of values between 0.98 and 0.984, representing 100% error with a red horizontal line. The plot produced is the following:



Note that there are several points that are really close to the red line, suggesting some values of u has a relative error close to 100% or 1.0 in our graph. Included in the code is also a function displaying the value in the range (98% - 102%) where the fractional error is about 100%, the outputted is as follows:

factional error is about 100% at 0.983000000000003
factional error is about 100% at 0.9832600000000032

d)

The last thing we want to do in this question is explore the effects of roundoff error in situations where we have products or quotients rather than series. First, we describe a function f = u**8/((u**4)*(u**4)) which equals to 1, but is built off of both a product a quotient and some exponents. To explore the effects roundoff error might have on the function, we call a range of values between 0.98 and 1.02 and check the resulting values produced by the function f-1. While the results should be linear all equaling 1, we found an error of 2.220446049250313e-16. Comparing this to the value offered in the textbook and ignoring the sqrt2 in front of the equation and approximating x to equal 1, we expect the value to be proportionate to C, which is 1.e-16. This is exactly the magnitude of the result we got when calculating the roundoff error, suggesting that the method of calculating standard deviation works correctly with products and quotients as well as series.

Question 3

a) We found C1 to equal to $\frac{2\pi k^4 T^4}{h^3 c^2}$

b)

In our program, we took advantage of the Simpson's method of integral approximation. Since we cannot integrate from 0 to infinity in python because we would encounter overflow issues. We took a really small number to approximate as 0 and a really large number to be infinite. We took N to be 10,000, a to be 0.00001, b to be 700 (e^{700} is reaching python's limit) and defined a variable h to be (b-a)/N. This method is similar to the example in chapter 5 of the textbook *Computational Physics* by Mark Newman. We defined temperature (T) to be 100 degrees and used the equation above in part a) to write out C1, and multiplied the integral built from the Simpson's method by C1 in order to find the value of W. The error we estimate can be approximated by the degree of h^4 because the Simpson's rule is a third-degree integration rule. Since h is of the order e-4, we can estimate the degree of error to be around e-16.

Printing the described integral estimated by Simpson's rule, we obtained a value equal to 6.49394, which is very close to what wolfram alpha has calculated.

c)

To ensure the accuracy of this method is to a reasonable high degree, we now want to take our result and apply it to an equation for an already known constant. The constant we will try to accurately evaluate is the Stefan-Boltzmann constant, which can be derived from the formula:

$$W = \sigma T^4,$$

Taking the value of W obtained from the integration and dividing it out by 100^4 for T^4, we get a constant equal to 5.670367165497756e-08. Checking this against, the scipy.constant version, we compare it to 5.670367e-08. This results in an accuracy level of 99.99999708135724%, an extremely successful result.

Question 4

a)

The goal in this question is to calculate a complex integral using the Simpson's rule approximation on python, and then using a known solution to check the accuracy of our results. Most of the work for this question has to do with applying Simpson's rule to the integral for the electric potential of a line of charge. The integral we represented is:

$$V(r,z) = \int_{-\pi/2}^{\pi/2} \frac{Qe^{-(\tan u)^2}\,du}{4\pi\epsilon_0 \cos^2 u\sqrt{(z-l\tan u)^2 + r^2}},$$

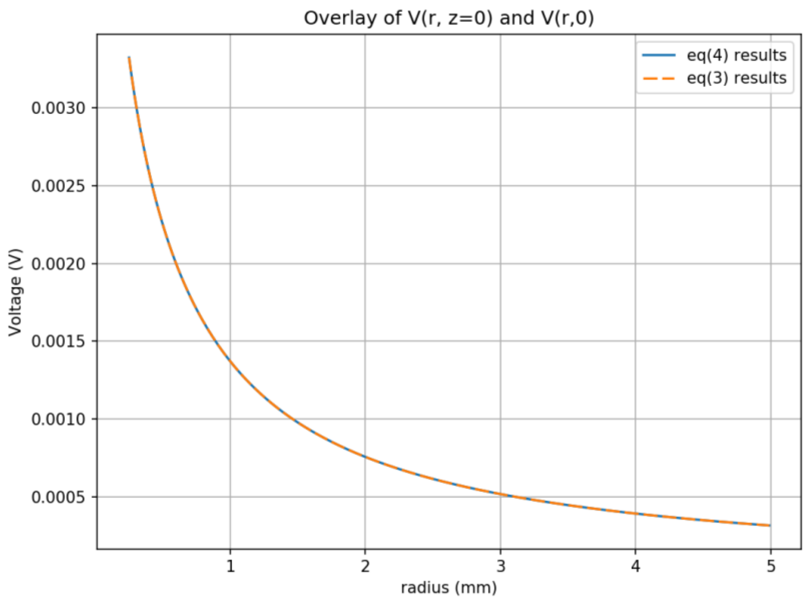
And the solution we get will be compared to the known solution for z=0, which is:

$$V(r, z=0) = \frac{Q}{4\pi\epsilon_0 l} e^{r^2/2l^2} K_0(r^2/2l^2),$$

Which can be called with the scipy.special.kn(0,x) for the Bessel function that the end.
After creating a function to describe the integral with Simpson's method with help from the computational physics text book, we set our variables Q = 10^{-13} Coulombs, l = 1mm and imported epsilon_0 from scipy. The range for r was 0.25-5 mm with steps defined to be 0.02mm. The error between the Simpson's method solution and the Bessel function was pretty small starting at a low N value of8, so we increased N up to 60 where we found the solution to agree with the Bessel function to the degree of 10e-7, or one part in a million a little better than desired by the question:

Simpson's approximation with N=60: 0.200448577331
Using known solution for z=: 0.200448650429

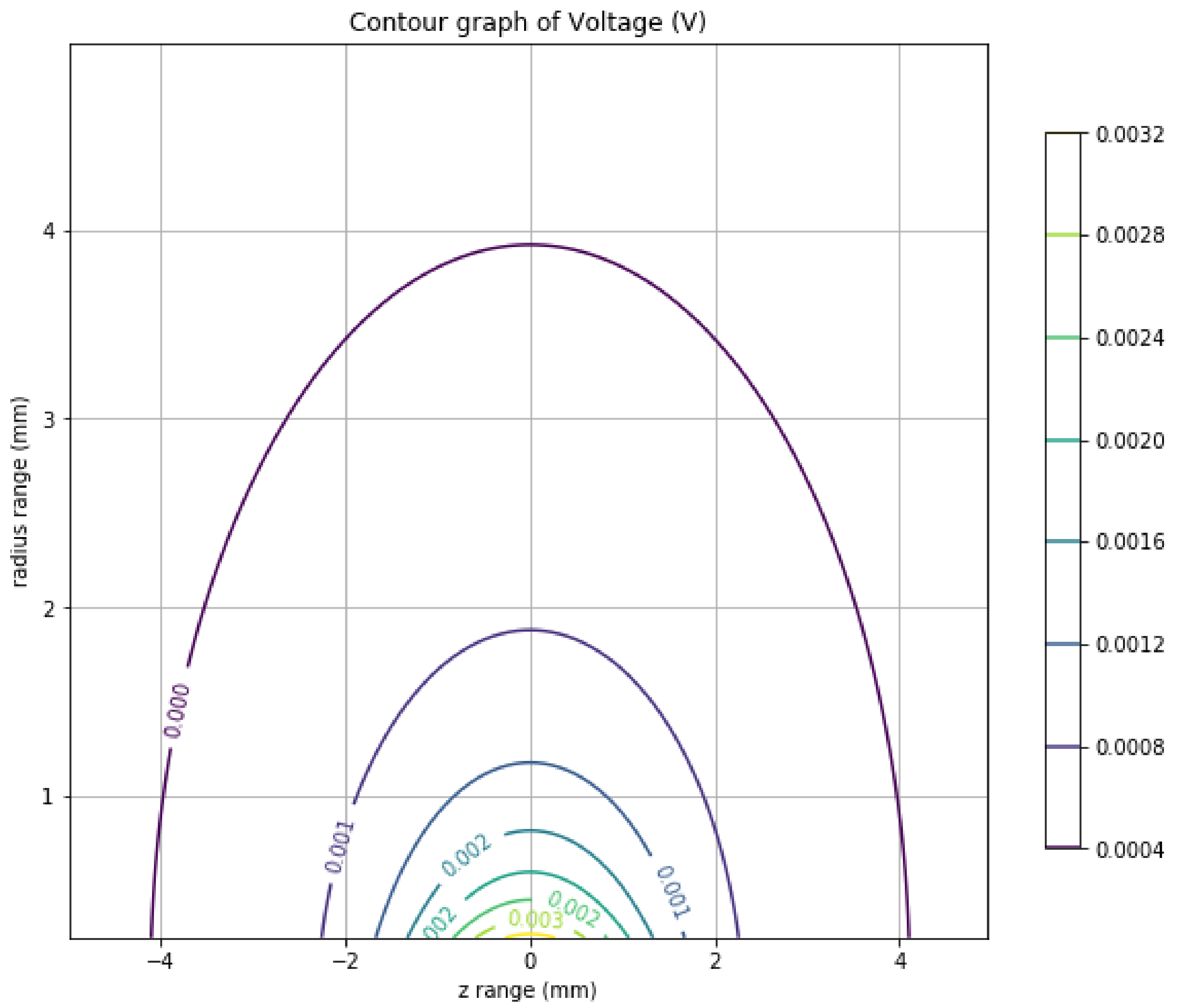
Plotting the two solutions on the same set of axis for contrast yielded the following results:



The fit here looks good, with N=60 we maintain fractional error less than one part in 10,000,000 as desired. Note that the eq (4) results are the blue line and eq (3) results are the orange dash line. As we can see, they are almost the same match.

b)

This question was aimed at generating a gradient field plot with a range of both variables r and z. Setting the range of z to be -5mm < z < 5mm, and keeping the same range for r that we used in part a), we find the following plot for the contours of V:



Note that the inline indicators are not accurate. Instead, please refer to the color bar on the right of the graph for more accurate contour values as the voltage.