PHY407: Lab 2

Date: September 24th. 2021

Lab Partners: Brendan Halliday and Nikolaos Rizos

Contributions:

- Q1. and Q3. Nikolaos Rizos
- Q2. Brendan Halliday

Q1.a.

PSEUDOCODE:

We will start by evaluating the STD using formula (1).

Start by looking at each value in the dataset.

Count how many elements exist in the dataset (start counting from 1, not from zero. Therefore, the first element would be 1, the second 2, etc.).

Now begin the first pass through the data.

Take the first value of the dataset and add to it the second value of the dataset. Then, take the result of this addition and add to it the third value of the dataset. Then, take the result of this second addition and add to it the fourth value of the dataset. Continue this process until and including the final value in the dataset.

Now, evaluate the sample mean by dividing the result of the above process, with the number of elements in the set. Record the result of this division.

Now begin the second pass through the data.

Start by looking at the first value in the dataset. Now, subtract the above mean from this first value. Square the result of the subtraction. Record the result of the squaring. Now move to the second value in the dataset. Now, subtract the mean from this second value. Square the result of the subtraction. Record the result of the squaring. Now, add these two squaring results, and record the result of this latest addition (we will name this result "of the latest addition", A for clarity purposes). Now move to the third value of the dataset. Now, subtract the mean from this third value. Square the result of the subtraction. Record the result of the squaring. Now add the result of the squaring, to A. Continue this process up to and including the final element in the dataset. Record the final result of this process (we will name it B).

Next, divide B by the number of elements at the dataset minus 1.

Next, take the square root of the above computation. The result of this square root is the STD of the data, evaluated using formula (1).

Next, subtract the actual value of the STD, from the above estimate of the STD, and divide the result of this subtraction by the actual value of the STD. The result of this computation is the relative error of the STD evaluated using formula (1).

We will continue by evaluating the STD using formula (2)

This method requires only one pass through the data.

Starting from the first value in the dataset, add to it the second value in the dataset. Record the result of this addition.

Divide this result with the number indicating the position of the second value that was added to the first value (here it is the number 2). Record the result of this division. This is the mean of the first 2 entries in the dataset. Proceed by multiplying the mean, by the above number indicating the position of the latter number.

Square the first value of the dataset, square the second value of the dataset and add the two squared quantities. Record the result of this addition of the two squared values.

Subtract the mean of the first 2 entries multiplied by the position of the latter number from the above result. Check that the result is non-negative, divide the result by the number indicating the position of the second element that was added to the first (again, here it is the number 2), minus the number 1. Take the square root of this operation (this is why the above result must be non-negative, as STD is a real number. If the above result is negative, stop the operation and issue a warning to the user). This result is now the STD of the first 2 elements evaluated using formula (2).

Continue this process for the second and third elements, with the biggest difference being the number indicating the position of the latter number, which is added to the previous number, now being 3 (because we are adding the 3^{rd} entry to the 2^{nd} entry). This gives the mean and STD of the first 3 entries.

Then continue this process for the third and fourth elements, (number indicating position is now 4 as we are adding the 4^{th} entry to the 3^{rd} entry), giving the mean and STD for the first 4 elements.

Continue this process for all the elements in the dataset. This will finally result in getting the value of the STD for all the elements in the set, while also allowing for the real time computation of the STD as data flows in.

Lastly, subtract the actual value of the STD, from the above estimate of the STD, and divide the result of this subtraction by the actual value of the STD. The result of this computation is the relative error of the STD evaluated using formula (2).

Q1.b.

Below are the printed outputs for the relative errors of the STD of Eq.(1) and Eq.(2) compared to the numpy.std() function, for Speed of light Data.

SPEED OF LIGHT DATA:

Relative error values:

Eq.(1): -1.7564474859226715e-16

Eq.(2): 3.740397602946237e-09

Larger Magnitude: Eq.(2): 3.740397602946237e-09

We can clearly observe that the relative error larger in magnitude is the one arising from Eq(2). It is larger in order of magnitude by a factor of 10^(7) compared to the relative error from Eq.(1).

Q1.c.

Below are the printed outputs for the relative errors of the STD evaluated using Eq.(1) and Eq.(2), first for a sequence of 2000 randomly distributed numbers with mean = 0 and standard deviation = 0.1 (this sequence is named "FIRST SEQUENCE' in the printed output), and second for a sequence of 2000 randomly distributed numbers with mean = 10^(7) and standard deviation = 0.1 (this sequence is named "SECOND SEQUENCE" in the printed output).

FIRST SEQUENCE:

Relative error values:

Eq.(1): 2.190451529260207e-16

Eq.(2): 4.380903058520414e-16

Larger Magnitude: Eq.(2): 4.380903058520414e-16

SECOND SEQUENCE:

Relative error values:

Eq.(1): 8.868532775112435e-16

Eq.(2): 0.12288090436429898

Larger Magnitude: Eq.(2): 0.12288090436429898

We observe that for the sequence of mean = 0, for the speed of light data (mean of order of magnitude around 10^(2)), and for the sequence with mean of order of magnitude 10^(7), the relative error using Eq.(1) seems to remain at the same order of magnitude of 10^(-16), which is miniscule. This might not even be the actual error deriving from the method used, but the rounding error of python arising from the relative error calculation, as it appears only at the very last digit python can register.

In contrast, the error arising from Eq.(2) seems to be highly dependent on the order of magnitude of the mean of the dataset.

In particular, we observe that for the sequence of mean = 0, Eq.(2) gives a relative error of order of magnitude 10^(-16).

For the speed of light data (whose mean has order of magnitude 10^2) the relative error from Eq.(2) has an order of magnitude of 10^(-9). Thus, for an increase in mean by a factor of 10^(2), the order of magnitude of the relative error increased by a factor of 10^(7).

For the sequence of mean = 10^(7), the relative error from Eq.(2) has an order of magnitude of 10^(-1). Thus, for an increase in mean by a factor of 10^(5) (increase measured from the order of magnitude of the speed of light data), the order of magnitude of the relative error increased by a factor of 10^(8).

The dependence of the relative error of Eq.(2) on the order of magnitude of the mean is thus made clear while looking at the above data.

This happens because python only expresses numbers up to a set number of digits. If we multiply a number with many decimal points as to make its non-decimal part bigger, the decimal part does not get maintained and instead gets truncated. Thus, when comparing two large numbers by subtracting them, the result is highly inaccurate as it only includes the subtraction of the large part of the two numbers, but not of their small decimal parts (as the decimal parts are omitted by the computer, even though they should be taken into account in the real-life subtraction).

The reason for the difference in accuracy between the two equations is the operations of Eq.(2) compared to Eq.(1). Eq.(1) first subtracts the mean value from the i-th datapoint and then squares the result of the subtraction. When calculating the STD using Eq.(2) we first square the two quantities (thus making them larger and truncating their decimal parts) and then we subtract them. This subtraction thus results in an inaccurate estimate of the decimal part of the STD. This is why when comparing the true STD to the value calculated using Eq.(2), the answer is only correct up to 1 decimal point (because of the truncation, the comparison of the rest of the decimal points was not even taken into account when Eq.(2) was implemented), thus resulting in an error with order of magnitude 10^(-1). This fact also explains why this phenomenon is not observed for small values, as squaring the small values does not make them that bigger, and thus does not force python to truncate the decimal parts, resulting in a miniscule error of the STD when compared to its true value.

Q2.a.i.

The analytical solution of the integral is π.

Q2.a.ii.

The following is the output for the program written in Lab02_Q2_a_ii.py. We output the approximations using both methods alongside their corresponding fractional error:

Exact Value of pi:

3.141592653589793
Trapezoidal rule:
3.1311764705882354
Fractional error:
0.003315574025695356

Simpson's rule:
3.14156862745098
Fractional error:
7.647757511045905e-06

Q2.a.iii.

The following is the output for the program written in Lab02_Q2_a_iii.py:

Number of slices required for Trapezoidal Rule is N = 16384
Trapezoidal rule time:
0.012973308563232422 s

Number of slices required for Simpson's rule is N = 32
Simpson's rule time:
0.0 s

Simpson's function is faster than the accuracy of the time() function, thus a value of 0.0 is returned.

Q2.a.iv.

The following is the output for the practical error estimation using the Trapezoidal rule. The program is written in Lab02_Q2_a_iv.py:

Trapezoidal Rule:
I_1 is :
3.140941612041389
I_2 is :
3.141429893174975
Practical error estimation:
0.00016276037786200348

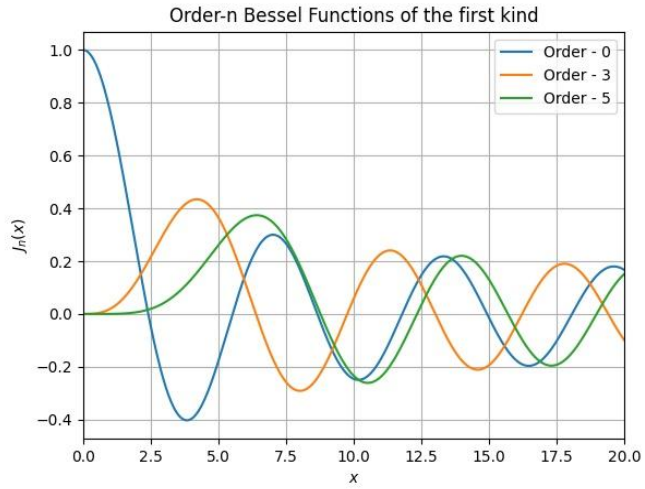
Q2.a.v.

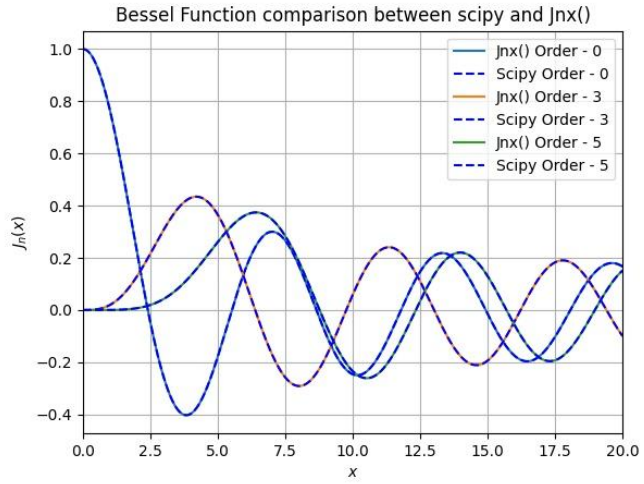
In our particular case, using the practical estimation of error for Simpson's rule, we are left with an error of O(-10). This is an underestimation of the error and the true order of magnitude should be $O(h_2^{\ 4}) = O(-7)$ where $h_2 = 0.03125$ in our case.

Q2.b.

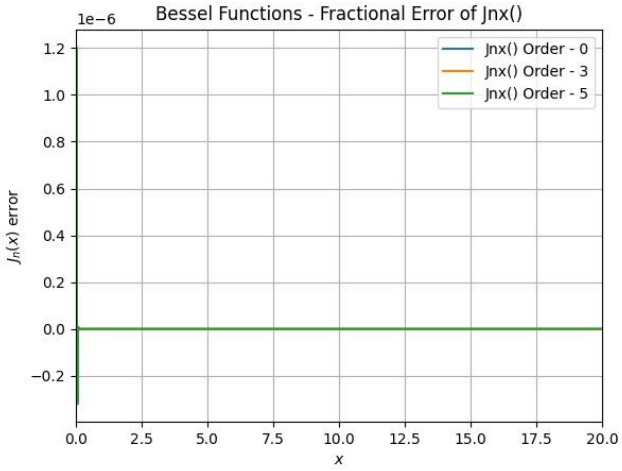
All code can be found in Lab02_Q2_b.py.

The first graph is the output for Jnx() which is the function I programmed to compute the nth order Bessel functions. The second graph is a comparison between my functions and the ones calculated by scipy on the same interval with the same number of steps. As shown, the two programs overlap thus it is reasonable to assume Jnx() is a good approximation to the true values.

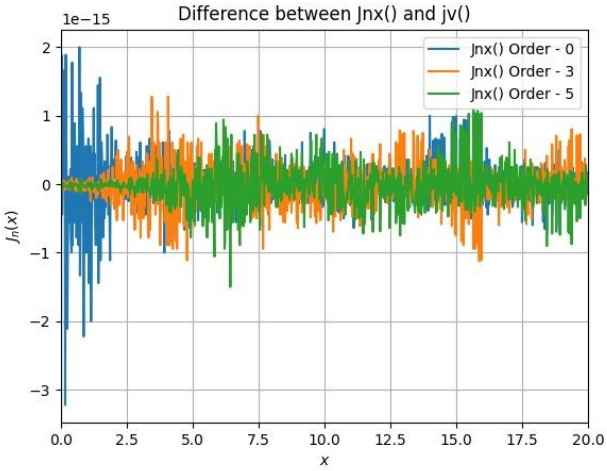




The fractional error can be calculated between Jnx() and special.jv() using jv() as the expected value. The following is a graph of the fractional error of Jnx() for each value of x. The fractional error is on the order of O(-6) which is reasonably small and we can conclude that Jnx() is a good approximation to the expected value. The entries where special.jv() is 0.0, that entry was replaced with 1.0 to avoid a math error.

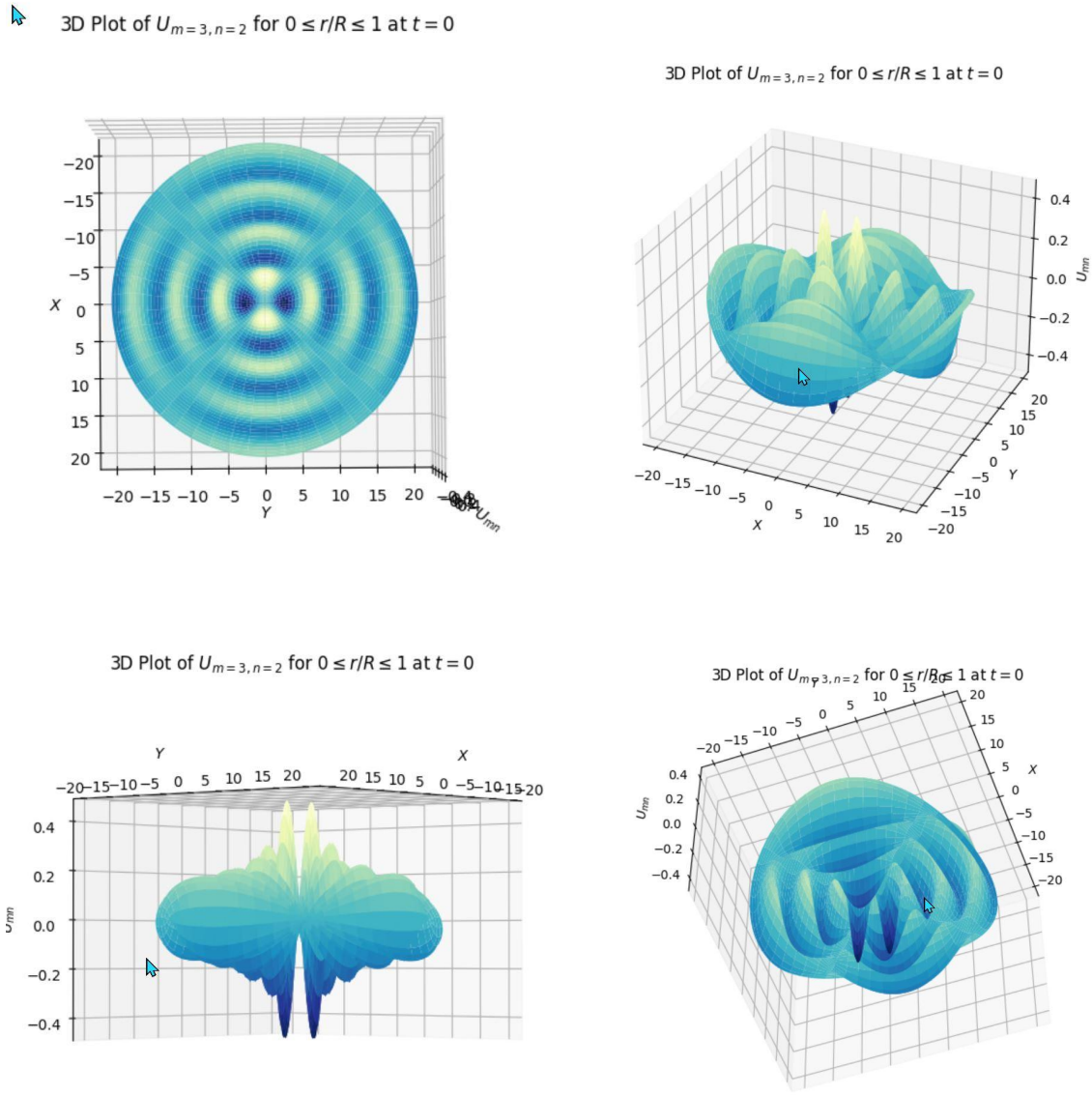


The raw difference is computed as well and is shown below. The order of magnitude is on the order os O(-15) or smaller (This exceeds the machines accuracy, thus the difference is basically zero):



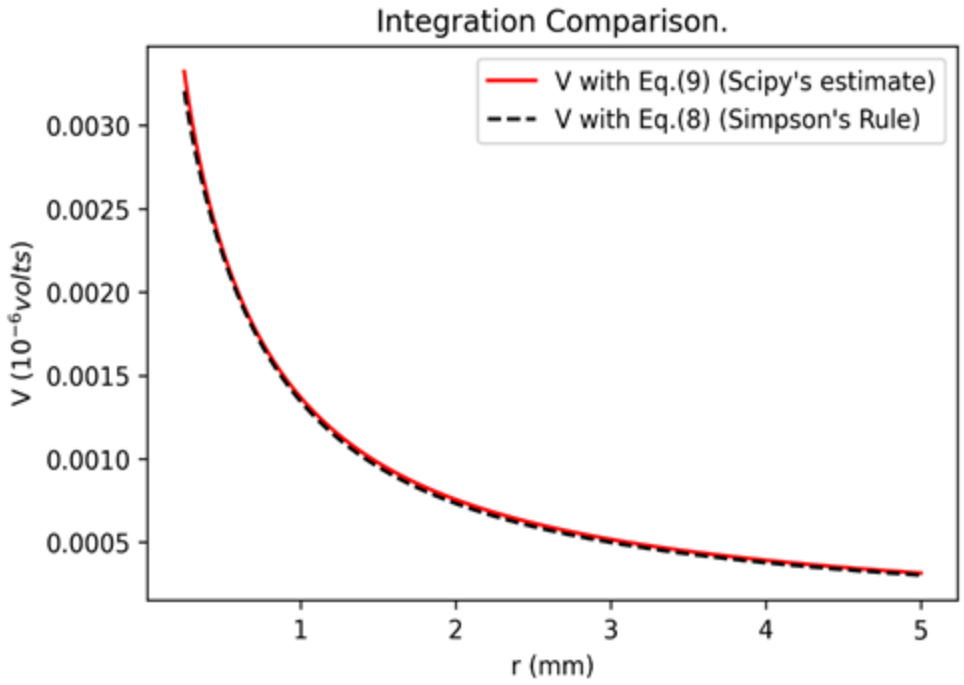
Q2.c.

The following screenshots are from the 3D plot of U_{m=3, n=2} where R = 20 , a = 1, and c = 1:



Q3.a.

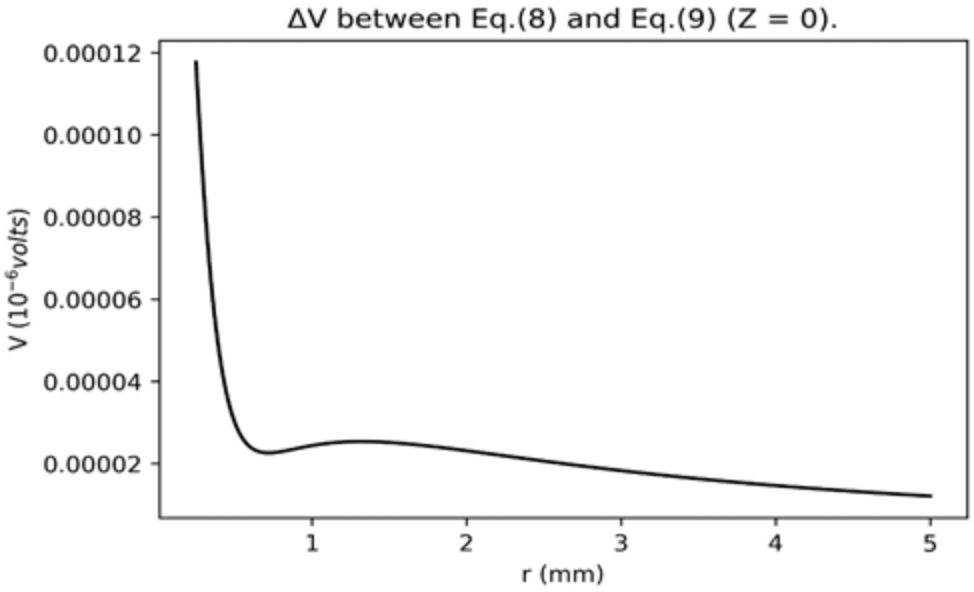
The results of using Eq.(8) and Simpson's rule in order to approximate the integral in this equation for Z = 0, overlayed on the correct result for V(r, Z = 0) given by Eq.(9), appears below.



(In the above graph, the Simpson approximation was evaluated for N = 8 slices)

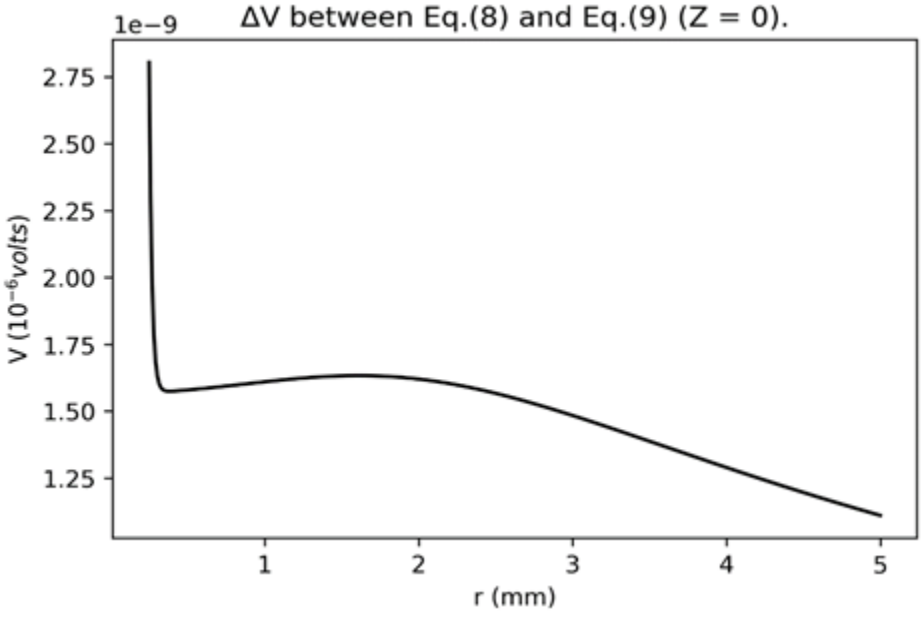
The graph consists of 1000 approximations of Eq.(8) using Simpson's method and 1000 values of Eq.(9), corresponding to 1000 equally spaced values of . We see that the two plots almost perfectly coincide, even for this small number of slices.

Below appears the plot of the difference between the estimate of the potential given by Eq.(8) with N = 8 slices, and the true value of the potential given by Eq.(9) for each of the 1000 above values of r.



We see that the difference between the two plots is in the order of magnitude of . This is consistent with the question's prompt that even for N = 8 slices the approximation should be quite accurate.

Next, we evaluate the maximum fractional error between the estimate and the true value of the potential as we increase the number of slices. The goal is to get this fractional error to be about 1 part in a million, or in the order of magnitude of 1.e-6. The printed value of the maximum fractional error (for the given range of r values) is 3.5186399379934197e-06, for N = 50 slices. Now the same plot of ΔV for the given range of r values appears below.



It is clear from the above graph that the estimates of the potential given by the Simpson's rule approximation of Eq.(8) for N = 50 slices now reduces the difference between the true value and the estimated value to an order of magnitude of 1.e-9. The two values (for any given value of r in the required range) now differ by about 1 part in a billion. The approximation of the potential using Eq.(8) is thus very accurate for this number of slices.

Q3.b.

Using the number of slices determined above, required for the accurate estimation of V using Simpson's rule to approximate Eq.(8), we now plot the gradients of V(r, z) for 1000 pairs of (r, z) values lying in the domain of 1000 equally spaced values of r in the interval and 1000 equally spaced values of z in the interval . The plot appears below.