PHY407 Lab 2: Numerical Errors and Numerical Integration

Work Distribution:
Emma Jarvis: Q1, Q3
Lisa Nasu-Yu: Q2

September 24, 2021

1 Exploring Numerical Issues with Standard Deviation Calculations

1.a

We started by writing pseudocode to estimate the standard deviation using two methods given by equations 1 and 2. Then, we wrote peudocode to compute the relative error between each of these two methods and the correct answer which is taken to be `numpy.std`. When implementing the second method the number under the square root can be negative if $n\overline{x}^2$ is greater than the sum of all of the values squared. Thus, it is necessary to take the absolute value of this number to ensure the square root is operating on a positive number. This is not an issue when using method 1 because the value under the square root is squared first and is thus always positive.

```
"""
Pseudocode that calculates the standard deviation using equations (1) and (2)
from the second lab assignment. Then, the relative errors for this two methods
are computed. The correct value is taken to be the value of the standard
deviation computed using numpy.std. The relative error for each method is
computed by subtracting the correct value from the other value, and then
dividing by the correct value.
"""

# Pseudocode
# 1. Import numpy
# 2. Compute standard deviation using eq (1)
#    - Calculate the mean of the values
#    - Iterating over each value:
    #    - Subtract the mean
    #    - Square this value
    #    - Compute the sum of these values
#    - Divide the sum by 1 less than the total number of values
#    - Take the square root of this sum
# 3. Compute the standard deviation using eq (2)
#    - Sum over the square of all the values
#    - Subtract the mean*number of values from this sum
#    - Divide this value by 1 less than the total number of values
```

```
#    - Take the square root of the absolute value of this sum
# 4. Compute standard deviation using numpy.std
# 5. Compute the relative error of the values, x, to the true values, y,
#        computed using numpy.std as (x-y)/y
```

1.b

Using the pseudocode in 1.a as a guide, the standard deviations of the values of the Michelsen's speed of light data in the file `cdata.txt` were computed using Eq. 1 and Eq. 2, where $\overline{x}$ is the mean of the values. A summary of these values is given in Table 1.

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

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

Then, the relative error was computed. To do this, the standard deviation using `numpy.std` was computed and taken to be the correct value. This value is in row 1 of Table 1. Then the relative error was computed was in Eq. 3 where y is the value of the standard deviation computed with `numpy.std` and x is the values of the standard deviation computed with the methods from Equations 1 and 2. A summary of these values is given in Table 1. This table shows that the method for calculating the standard deviation using Eq. 1 has a relative error with a much smaller magnitude, by 7 orders of magnitude.

$$error = \frac{x - y}{y} \tag{3}$$

	Standard Deviation [10^3 km/s]	Relative Error Magnitude [10^3 km/s]
`numpy.std`	0.07901054781905067	
Method 1	0.0790105478190507	$3.512894971845343 \cdot 10^{-16}$
Method 2	0.07901054763832621	$2.2873460336752 \cdot 10^{-9}$

Table 1: Standard deviation and relative error values computed for the values of the Michelsen's speed of light data from cdata.txt using the two methods for computing the standard deviation given by equations 1 and 2.

`Printed Output`

```
The standard deviation using eq. (1) is  0.0790105478190507
The relative error using eq. (1) is  3.512894971845343e-16
The standard deviation using eq. (2) is  0.0786145022990487
The standard deviation using np.std is  0.07901054781905067
The relative error using eq. (2) is  -0.005012565169260565
```

1.c

Then, the same procedure was repeated with new values generated using `numpy.random.normal`. The first set of values had 2000 values with a mean of 0 and a standard deviation of 1. The second set of values also had 2000 values with a standard deviation of 1, but this time with a mean of 10^7. The relative error values with both methods for computing the standard deviation for each of these sets of data are summarized in Table 2. For the first sequence of values, generated with a mean of 0, the relative error is close to zero using both methods. However, for the second sequence with a mean of 10^7, method 1 yields a small relative error, but method 2 yields a relative error that is 14 orders of magnitude larder than method 1. By varying the mean, it can be seen that the relative error for the standard deviation computed using Eq. 1 is similar when the mean varies. The relative error for the standard deviation computed using Eq. 2 is 0 when the mean is 0, but increases as the mean increases.

	Mean = 0	Mean = 10^7
Method 1	$8.679668478488622 \cdot 10^{-16}$	$2.235009966364127 \cdot 10^{-16}$
Method 2	0.0	0.02178608454554426

Table 2: The magnitudes of the relative error values computed for two sets of values, one with a mean of 0 and the other with a mean of 10^7 using the two methods for computing the standard deviation given by equations 1 and 2.

```
Printed Output

The relative error using eq. (1) for mean = 0 is  -4.391378205261303e-16
The relative error using eq. (2) for mean = 0 is  -0.00025003125781490826
The relative error using eq. (1) for mean = 1e7 is  2.1817816388545653e-16
The relative error using eq. (2) for mean = 1e7 is  -0.013490383445111657
```

1.d

To workaround the problem with the one-pass method, we subtracted the mean from the values to bring the mean to zero. With this method, and using the data obtained in Q1.c that generated using numpy.random.normal with a mean of 10^7, the absolute value of the relative error was reduced to $1.1580821273761795 \cdot 10^{-16}$.

```
Printed Output

The relative error using eq. (2) for mean = 1e7 is  -1.1580821273761795e-16
```

2 Trapezoidal and Simpson's Rules for Integration

2.a

In this section we investigate 2 integration methods, trapezoid and Simpson, through the evaluation of the integral given in Eq. 4.

$$\int_0^1 \frac{4}{1+x^2} dx \tag{4}$$

2.a.1

The exact value of the integral is π.

2.a.2

We defined functions to compute the integral with the trapezoid and Simpson's rule using Eq. 5 and Eq. 6, respectively. These forms were derived through manipulation of the Taylor expansion of the Eq. 4, as shown in section 5.1 of Computational Physics by Mark Newman.

$$I(a,b) = h[\frac{1}{2}f(a) + \frac{1}{2}f(b) + \sum_{k=1}^{N-1} f(a+kh)] \tag{5}$$

$$I(a,b) = \frac{1}{3}h[f(a) + f(b) + 4\sum_{\text{k odd}}^{N-1} f(a+kh) + 2\sum_{\text{k even}}^{N-2} f(a+kh)] \tag{6}$$

Printed Output:
The value with trapezoidal rule and 4 slices is 3.1311764705882354.
The value with Simpson's rule and 4 slices is 3.14156862745098.

2.a.3

We compare the errors and computation times of each integration method.

The trapezoidal rule error was calculated with the Euler-Maclaurin formula for the trapezoidal rule,

$$\epsilon = \frac{1}{12}h^2[f'(a) - f'(b)] \tag{7}$$

The error for the Simpson's rule was calculated by taking the difference between the Simpson's rule value and the actual value. The formula for the approximation error of the Simpson's rule, Eq. 8, was not used because the 3rd derivative vanishes on both boundaries, resulting in an incorrect error of zero for any number of slices.

$$\epsilon = \frac{1}{90}h^4[f'''(a) - f'''(b)] \tag{8}$$

Printed Output:
2^{12} slices gives an error of order -9 for the trapezoidal rule.
The error for 2^{12} slices is 9.93e-09
It takes an average of 0.00109s to compute the integral for 2^{12} slices

2^4 slices gives an error of order -9 for the Simpson's rule.
The error for 2^4 slices is 2.36e-09
It takes an average of 9.96e-06s to compute the integral for 2^4 slices

It is considerably faster (by approximately 0.001s, compared to the Simpson's time of 9.96e-6) to compute the integral with the Simpson's rule than the trapezoid for the same order of error. This is because the number of slices, and thus the number of iterations performed, to sum the area under the curve is reduced.

2.a.4

$$\epsilon_2 = (I_2 - I_1)/3 \tag{9}$$

We apply the practical estimation of errors, defined by Eq. 9, to the trapezoidal rule to estimate the error of Eq. 4 with 32 slices.I_1, and I_2 are the values of the Simpson's rule integration for N_1 and N_2 slices, respectively.

Printed Output:
The error estimation for the trapezoidal rule with $N_2 = 32$ slices is 0.000163.

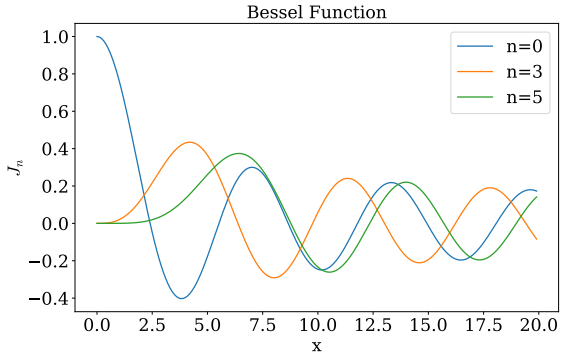
2.a.5

We cannot use the method of practical estimation of errors for this particular case because the third derivative of the integrand vanishes at both boundaries of the integral. The practical estimation of errors relies on the assumption that the Simpson's rule accurate up to $O(h^3)$. In most cases, we can prove this by manipulating the Taylor expansion of the integrand about an endpoint of a single slice, resulting in Eq. 8. However, in our particular case, the third derivative of the integrand is zero at both boundaries, giving us an approximate error of zero for any number of slices - a result that is obviously incorrect. In order to make this method viable for our case, we can shift the centre our Taylor expansion. This would give non-zero third order values at the shifted boundaries, allowing us to make the approximation in error of order h^4.

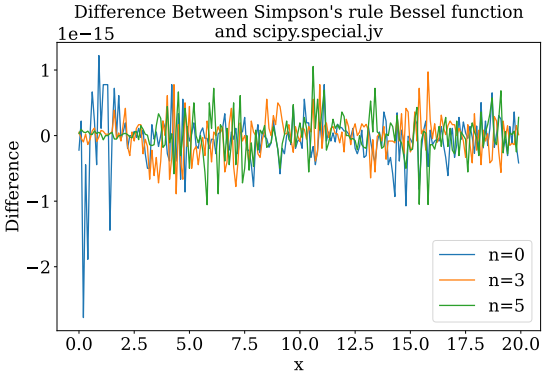
2.b

We defined a function to evaluate a Bessel function using the Simpson's rule and compared it to the Bessel function included in the scipy.special module. The formula of the Bessel function is given in Eq. 10.

$$J_n(x) = \frac{1}{\pi}\int_0^\pi cos(n\phi - xsin\phi)d\phi \tag{10}$$



(a) Bessel function (Eq. 10) with Simpson's rule



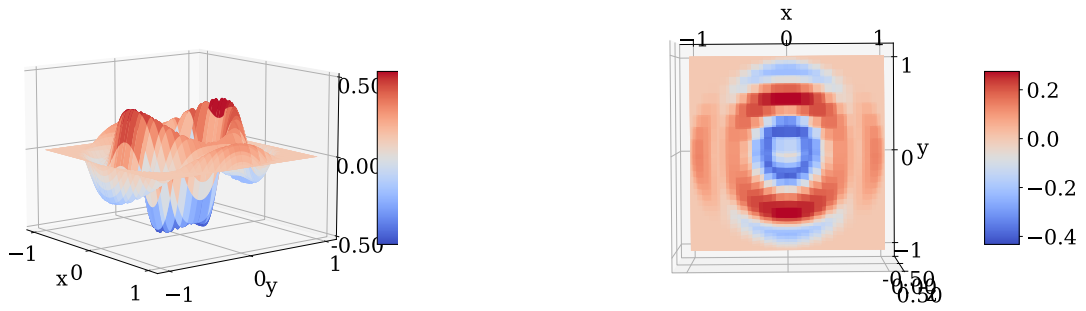
(b) Difference between Bessel functions from Eq. 10 and Simpson's rule, and from the scipy.special package.

Figure 1: Figure (a) depicts our Bessel function, defined by Eq. 10 and integration with Simpson's rule for $n = 0$, 3, 5. Figure (b) plots our Bessel functions (shown in Figure (a)) subtracted from scipy.special.jv, the SciPy Bessel function.

Figure 1b shows that the difference between our Simpson's rule Bessel function and scipy.special.jv decreases with increasing order, n. For all plotted n, the difference in the 2 Bessel functions is of order -15 while the actual value is of order -1. With a fractional error on the order of -14, our Simpson's rule Bessel function reproduced the SciPy routines very well. This was to be expected, because we discovered in Q2a part iii, that a relatively small number of slices (and thus a relatively large slice width) was sufficient to have very small errors in the Simpson's integration.

2.c

The general solution to the wave equation of a vibrating membrane was depicted on a 3-D surface plot. No code submission was required for this question.



(a) Diagonal view

(b) XY plane

(c) XZ plane

(d) YZ plane

Figure 2: Various views of the 3-D plot depicting the general solution to the wave equation for a vibrating membrane. We defined the radius of the membrane to be R = 1, with the wave equation equal to zero outside of this region. We approximated the wave phase speed, c, to be 1, but a solution involving the true wave phase speed can easily be recovered by dimensional analysis.

3 Electrostatic Potential for a Line of Charge

3.a

The Simpson's rule was then used to numerically integrate the electrostatic potential for a line charge given by Eq. 11. This was done using 8 slices. For this integral, Q was set to 10^{-13} C, l was set to 0.001 m, z was set to 0 and r was set to be a range of 100 linearly spaced numbers from 0.25mm to 5mm. Then this integral was compared to the known value of the integral given by Eq. 12. These results were then plotted as seen in Fig. 3

$$V(r,z) = \int_{-\pi/2}^{\pi/2} \frac{Qe^{-(\tan u)^2}}{4\pi\epsilon_0(\cos u)^2\sqrt{(z - l\tan u)^2 + r^2}}\, du \tag{11}$$

$$V(r, z=0) = \frac{Q}{4\pi\epsilon_0 l} e^{\frac{r^2}{2l^2}} K_0\left(\frac{r^2}{2l^2}\right) \tag{12}$$

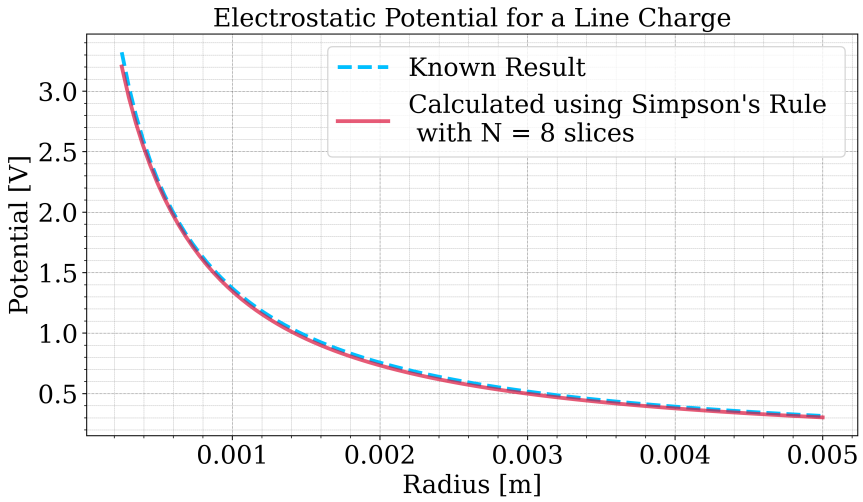


Figure 3: The electrostatic potential for a line of charge computed numerically using the Simpson's rule with 8 slices (red line) compared to the known solution given by Eq. 12 (blue dotted line).

Then, the differences in these two solutions to the integral were calculated. To make the fractional error one part per million, the number of slices when using the Simpson's rule was increased from 8 to 50. These difference values were then plotted and can be seen in Fig. 4.

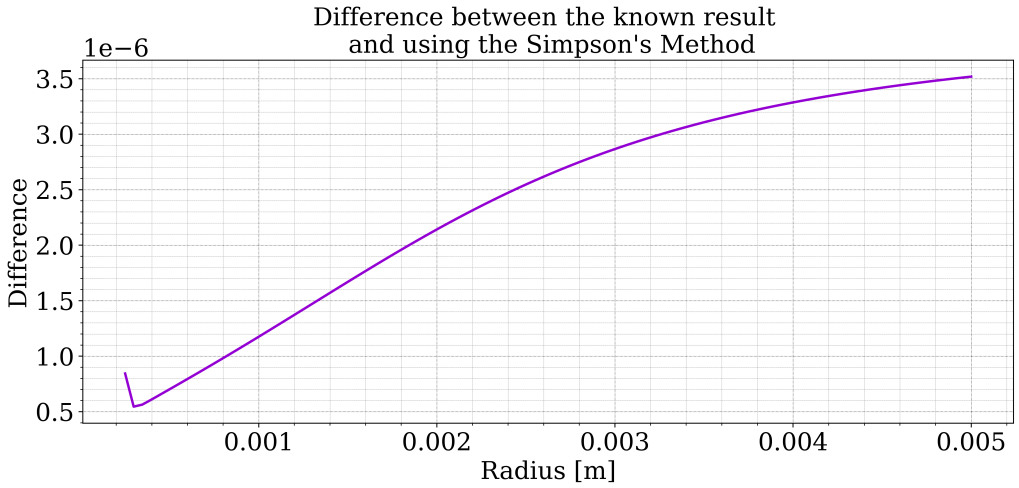
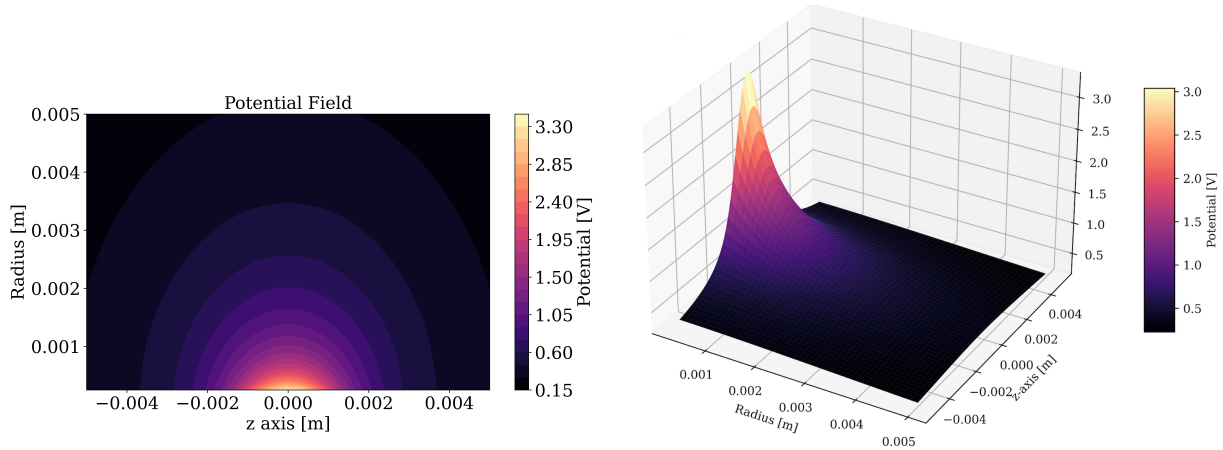


Figure 4: The difference between the known result of the electrostatic potential for a line of charge with $z = 0$ given by Eq. 12 and the values computed using the Simpson's rule with 50 slices. Increasing the number of slices to 50 allowed the differences to be approximately 1 part per million.

3.b

Next, continuing to use $N = 50$, the potential integral was again solved using the Simpson's rule with all the same parameters except for the z values which were changed from 0 to a range of 100 linearly spaced values between -5mm to 5mm. Fig. 5a depicts the 2d contour of this potential field and Fig. 5b depicts the 3 dimensional plot of the potential field.



(a) Contour

(b) 3D Plot

Figure 5: Figure (a) depicts the 2 dimensional contours of the potential field. Figure (b) depicts the 3 dimensional surface plot of this same potential field.