PHY407 Lab02

Pierino Zindel (1002429703) and Hayden Johnson (1002103537)

September 24, 2018

Distribution of work: Questions 1 and 3 were completed by Pierino. Question 2 was completed by Hayden.

1 Numerical Standard Deviation

1.1 Part a):

Pseudo-code for comparing the relative error of two different methods of computing the standard deviation.

1. Import library to compute the 'true' standard deviation.
2. Define a function that takes a set of data points as input and then computes the standard deviation using a method that first computes the average value of the data and then uses it to compute the standard deviation and returns that.
3. Define a function that takes a set of data points as input and then computes the standard deviation using a method that only passes through the array of points once and computes the mean within the standard deviation computation. Return the computed standard deviation.
4. Define a function that takes the standard deviation, computes a standard deviation using a numpy function and returns the relative error between the two.
5. Import or generate a data set to be used to compute the standard deviations.
6. Pass the data set to the functions and return and print the relative error.

1.2 Part b):

The printed output from the code is:

- The relative error of method/eqtn one is: 3.512894971845343e-16
- The relative error of method/eqtn two is: 3.740397602946237e-09

The first value is the error related to using the method that passes over the array twice, and the second value is the error related to using the method that passes over the array only once. The result produced is a relative error that is larger for the one pass method than for the two pass method.

1.3 Part c):

The printed output is:

- Sequence one: method/eqtn one: The relative error is: 2.2209090062434743e-16
- Sequence one: method/eqtn two: The relative error is: 4.441818012486949e-16
- Sequence two: method/eqtn one: The relative error is: 0.008639747075393788
- Sequence two: method/eqtn two: The relative error is: 0.15108269089287008

From the returned relative errors for the controlled data set we see that as the mean of a data set the relative error of the two methods also increases given that the methods compute a difference between the errors and mean within the standard deviation calculation. Moreover we see that for the first sequence the relative error of both methods is within the same magnitude, but for the second sequence with a larger mean the one pass method produces a relative error that is two orders of magnitude larger than the one pass method. A reason for the possible increased relative error is that that the two pass method formula computes the difference in each data point and the mean, and then squares the result afterwards; whereas the one pass method formula squares the data points and mean first and then computes the difference afterwards. As a result the difference between the data and the mean is much more accented by the one pass method giving us a larger relative error as the mean is increased.

1.4 Part d):

A workaround for the issue of the difference between the data points and the mean being accented by the one pass method is to shift the entire data set by one of the points. As shown on the wikipedia page 'Algorithms for computing variance', we shift the entire data set by the first data point which works to decrease the mean and all of the points closer to zero which results in the square of the difference and the difference of squares becoming closer.

Using this workaround in our program for the data in part c we get a new relative error for the one pass method to be $2.175e-16$ for the first data sequence and 0.01186 for the second data sequence which is on par with the two pass method.

2 Trapezoid and Simpson's Rules for Integration

2.1 Part a)

2.1.1 Part i)

The values returned by the three different methods of computing erf(3) are listed in table 1. Considering scipy.special.erf to be the "true" value of the function, the two numerical integration methods agree to within 10^{-5} with only $N = 10$, which is quite good. Note also that Simpson's rule is more accurate, by almost an order of magnitude.

Table 1: Values of erf(3) as computed by different methods. The numerical integration methods used 10 divisions.

Method	Value
scipy.erf	0.999977909503
Trapezoid rule	0.999971912594
Spimson's rule	0.999977011298

2.1.2 Part ii)

The value of N was increased sequentially by an order of magnitude in order to produce the data in table 2. As can be seen from the table, the Trapezoid rule required N=10000 in order to make the error of order $O(-11)$, whereas Simpson's rule required only $N = 1000$.

Table 2: The relative error of, and time taken to integrate by, both the trapezoid and Simpson's methods for different sizes of N for the function erf(3).

N	scipy.erf time (s)	Trapezoid error	Trapezoid time (s)	Simpson's error	Simpson's time (s)
10	2.1×10^{-5}	6.0×10^{-6}	2.8×10^{-5}	9.0×10^{-7}	2.3×10^{-5}
100	2.5×10^{-5}	6.3×10^{-8}	2.6×10^{-4}	1.1×10^{-10}	1.6×10^{-4}
1000	2.5×10^{-5}	6.3×10^{-10}	3.7×10^{-3}	9.4×10^{-15}	3.5×10^{-3}
10000	1.8×10^{-5}	6.3×10^{-12}	1.6×10^{-2}	1.1×10^{-15}	1.9×10^{-2}

2.1.3 Part iii & iv)

Table 3 shows the data for parts iii and iv; the errors calculated are in all cases for the integrals calculated with $N_2 = 2N_1$ steps. For both methods of integration, the two error estimates agree more closely for larger values of N_2, which makes sense since the error itself is smaller for these integrations with a smaller stepsize. The two methods for error estimation do not agree exactly because they are both accurate to some order in h (h^2, say, for the both errors for the trapezoid), the additional higher-order terms are not accounted for and introduce small differences between the values calculated by the two methods.

Table 3: The estimated size of error for both the trapezoid and Simpson's methods as computed by the practical error formulas (ϵ_2) and the Euler-Maclaurin formulas ($\epsilon_{\text{E-M}}$).

N_1	N_2	Trapezoid ϵ_2	Trapezoid $\epsilon_{\text{E-M}}$	Simpson's ϵ_2	Simpson's $\epsilon_{\text{E-M}}$
10	20	1.48×10^{-6}	1.57×10^{-6}	5.54×10^{-8}	7.05×10^{-8}
100	200	1.566×10^{-8}	1.567×10^{-8}	7.03×10^{-12}	7.05×10^{-12}
1000	2000	1.56659×10^{-10}	1.56660×10^{-10}	7.11×10^{-16}	7.05×10^{-16}

2.2 Part b)

2.2.1 Exercise 5.4.a)

The first three Bessel functions were computed on $x \in [0, 20]$ using the code in Lab02_Q2_b_a.py and are plotted in figure 1, along with the values of the functions as computed by scipy.special.jv. As can be seen from this figure, the two methods agree as closely as can be determined by graphical inspection.

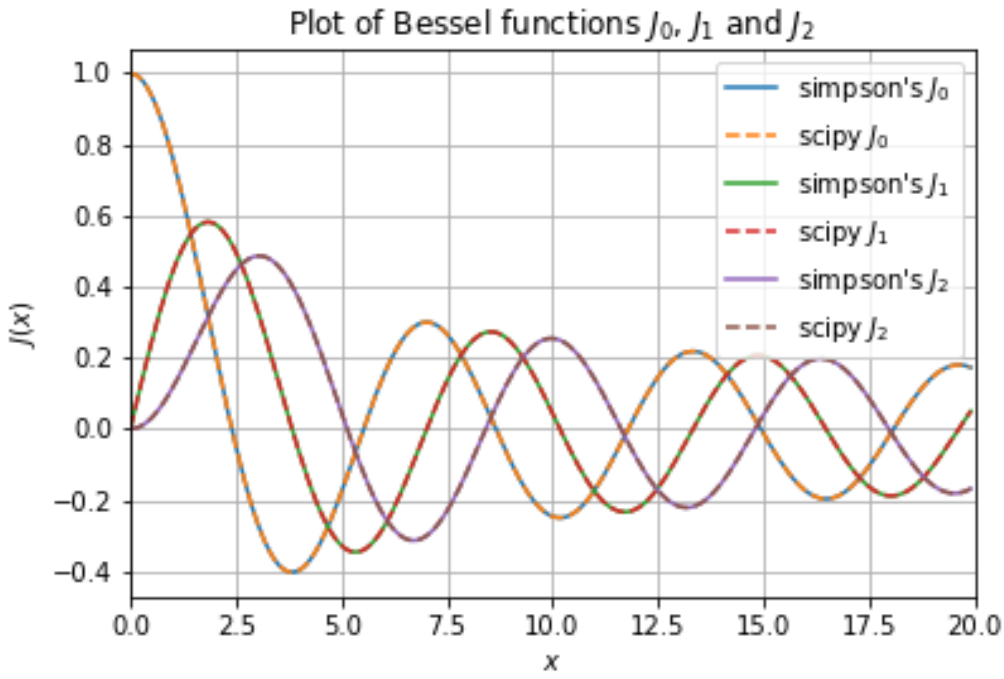

Figure 1: Graph showing plots of the first three Bessel functions for $x \in [0, 20]$ as computed by the code in Lab02_Q2_b.py (solid lines) and by scipy.special.jv (dashed lines).

2.2.2 Exercise 5.4.b)

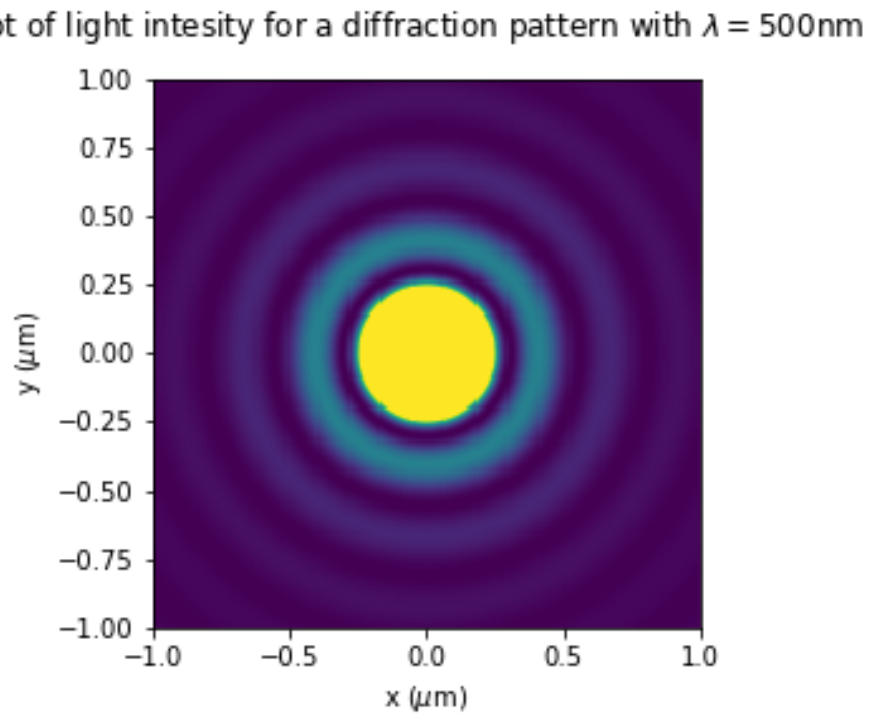

Figure 2: Plot created with pcolormesh showing the intensity of a diffraction pattern of $\lambda = 500$nm over a 2μm grid. Note that the color scale is saturated at the central maximum in order to make the outer fringes more easily differentiable.

3 Stefan-Boltzmann Constant

3.1 Part a):

By plotting the function that we intend to integrate, namely $f(x) = \frac{x^3}{e^x - 1}$, we can see that the function is smooth and well behaved on the interval from 0 to ∞. Therefore we can easily use the trapezoidal method of integration with sufficient bins and reach a very accurate result. Computing the integral gave an answer of 6.49394 which can be compared to the analytic value noted on the the HyperPhysics website (http://hyperphysics.phy-astr.gsu.edu/hbase/thermo/stefan2.html) to be $\frac{\pi^4}{15} \approx 6.49394$ which is identical.

3.2 Part b):

Using the scipy define values for the Boltzmann constant, speed of light, Planks constant, and pi, we determine the value of the Stefan-Boltzmann constant using our integral to be $\sigma_{computed} = 5.67x10^{-8}$ and compare it to the value defined in the scipy module as $\sigma_{defined} = 5.67x10^{-8}$, which agrees with our results.