. Draft version September 23, 2022

PHY407 Lab-02 Report

Raina Irons[1] and Maxwell Fine[1]

[1] *University of Toronto*

QUESTION 1 - EXPLORING NUMERICAL ISSUES WITH STANDARD DEVIATION CALCULATIONS

PART A

The pseudo code for the relative error between the two methods used in Eq.1 and Eq.2,

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

$$\sigma = \sqrt{\frac{1}{n-1}\Big(\sum_{i=1}^{n} x_i^2 - n\bar{x}_i^2}, \tag{2}$$

can be found below.

```
# first calc eq 1, true value y
# not initially functions, but made it easier to work with to convert them to
# functions
# first we want to define a name, i just called it eq1 for simplicity
# then we want to have 2 for loops, one for calculating the mean, then another
# to use that calculated mean in order to find the
# std
# define some inital values for our loops to work with, like length of the data
# for how long the loop runs and a summation
# mean for loop
# update the mean sum depending on the index of the data
# exit the loop and divide by the total length of the data
# for the second loop we take the difference between the ith data point and the
# mean sum in the first loop, updating an inital
# value as we go
# use eq1 to calculate the std and return the value

# second calc eq 2, wrong value x
# define the similar initial values as eq1 function
# enter for loop to calculate the sum of squares of each of the data points and
# the sum of each of the data points
# exit loop and compute the square of the mean to use in the std
# calculate the std using eq2
```

PART B

Now we want to implement the above pseudo code into something useful we can use to analyse our data. The main attraction is the functions created that will intake an array of data and return the standard deviation. The first function,

```
def eq1(data):
    '''finds the std of a data set using eq 1 (two pass method)'''
    N = len(data)
```

```
    eq1_meansum = 0
    eq2_sum = 0
    eq1_stdsum = 0

    for i in range(N):
        eq1_meansum += data[i]
    eq1_mean = (1/N)*eq1_meansum # calculates the mean of the data set using eq 1

    for i in range(N):
        eq1_stdsum += (data[i] - eq1_mean)**2
    eq1_std = np.sqrt((1/(N-1))*eq1_stdsum)

    return eq1_std
```

utilizes the two pass method, and first calculates the mean of the data in the initial for loop, then uses that information to then go ahead and find the standard deviation in the second. I will also set here the function used to calculate the deviation using the single pass method.

```
def eq2(data):
    '''finds the std of a data set using eq 2 (one pass method)'''
    N = len(data)
    sum_1 = 0
    sum_2 = 0
    for i in range(N):
        sum_1 += data[i]**2
        sum_2 += data[i]
    sum_2 = (sum_2/N)**2
    term1 = sum_1
    term2 = N*sum_2
    diff = term1 - term2
    eq2_std = np.sqrt(diff/(N-1))

    return eq2_std
```

We can now go ahead and compute the standard deviation using the Numpy function numpy.std and calculate the relative error. For Eq.1, the relative error measured with respect to numpy.std was on the order of -10^{-16}. For the second function using the single pass method, the relative error was on the order of 3×10^{-9}. We can see here from a relatively small data set the standard deviations defer by a significant amount.

PART C

Now testing our functions on data with known means, we can define two normal functions with one mean equal to 1 and a second equal to 10^{-7}, and the same standard deviations. Given that both of our equations are functions, and so is numpy.std, we can freely call them upon these distributions to analyse their outputs. For the numpy.std function, we received values of 1.012 and 1.001 for the standard deviation. For Eq.1, we achieved values of 1.012 and 1.002 for the first and second normal distributions. And finally, we got 1.012 and 1.252 for the first and second distributions. All the information will be summarized in the table below.

Method	**Norm.1 std**	**Norm. 2 std**	**Rel. Error Norm. 1**	**Rel. Error Norm. 2**
numpy.std	1.0123865	1.0019765	0	0
Eq 1	1.0126397	1.0022271	0.0002500	0.0002500
Eq 2	1.0126397	1.2525112	0.0106421	0.2500404

PART D

For the two pass method, we want to make some corrections regarding its inaccuracy. We note that since the values in a data set can be large, we can subtract the median from each value in order to obtain an array with smaller values and not lose the information (this is the same as subtracting a constant term). In doing so, this allows the one pass to consider more decimal places and include more information in its calculation, leading to better results. The implementation is below.

```
def eq2_updated(data):
    '''finds the std of a data set using eq 2 (one pass method)'''
    N = len(data)
    median_data = np.median(data)
    data = data - median_data # here we correct for bigger values by
    # subtracting off the median of the data so our std can take
    # in smaller decimal place values to obtain a better std.
    sum_1 = 0
    sum_2 = 0
    for i in range(N):
        sum_1 += data[i]**2
        sum_2 += data[i]
    sum_2 = (sum_2/N)**2
    term1 = sum_1
    term2 = N*sum_2
    diff = term1 - term2
    eq2_std = np.sqrt(diff/(N-1))

    return eq2_std
```

Which results in standard deviations of 0.98280 and 1.002577 for the small and large mean normal distributions respectively.

QUESTION 2 - TRAPEZOIDAL AND SIMPSONS RULES FOR INTEGRATION

PART A

For Q2A, we are asked to find the exact value of $\int_0^1 \frac{4}{1+x^2}\mathrm{d}x$. We recall from elementary calculus, that this integral takes the form of a trigonometric substitution, the solution is $4\tan(x)^{-1}$ evaluated from 0 to 1 which is π.

PART B

For Q2B, we are tasked with numerically integrating and comparing the Trapezoidal vs Simpson's rule for numerical integration, using $N = 4$ slices. The solutions are Trapezoidal = 3.1311764705882354, Simpson = 3.8082352941176465, and π = 3.141592653589793. Trapezoids rule is closer to the true value of π, perhaps because of the low amount of slices, and the shape of the integrand might be better approximated by Trapezoids rather then parabolas over the.

PART C

Question Q2C asks us to find N, or n such that with $N = 2^n$ slices the error is less than 10^{-9}, additionally we are tasked with finding how long does this take to compute the integral to have such a precision.

For Simpson's rule, Time = 1.892207670211792[s], N = 107388928.0, and n = 30, while the Trapezoidal rule is Time = 0.0025503158569335936[s], N = 16384.0, and n = 14. It appears that despite Simpson's rule being more accurate in general that the Trapezoidal rule is better for this integrand.

PART D

For Q2D, we are tasked with implementing the practical estimation of errors for the Trapezoidal rule, and use it to estimate the error. Using $N_2 = 32$, $N_1 = 16$. For the Trapezoid rule, the leading order of approximation error is $\epsilon = \frac{h^2[f'(a)-f'(b)}{12}$, where $h = (b-a)/N$. In order to calculate this we need to find f', which is $\frac{\mathrm{d}}{\mathrm{d}x}(\frac{4}{1+x^2}) = -\frac{8x}{(1+x^2)^2}$. For our $f(x)$, this becomes $h^2\frac{-1}{6}$, for N_1, $\epsilon = \frac{1}{1536} \approx 0.0006$. For N_2, $\epsilon = \frac{1}{6144} \approx 0.0002$.

PART E

For Q2E, we are tasked with explaining why we cannot use the practical estimation for Simpson's rule for our integrated. For Simpson's rule, the leading order of approximation error is$\epsilon = \frac{h^4[f'''(a)-f'''(b)}{12}$, where $h = (b-a)/N$. It happens that our $f'''(x)$ is 0 at both $x = a$, and $x = b$. This means our error estimation is.. 0, which is not a physical amount of error to have.

QUESTION 3 - STEFAN-BOLTZMANN CONSTANT

PART A

In question 3A, we are tasked with showing that the integration of black body radiation curve (B), can be written as:

$$W = \pi \int_0^\infty B d\nu = C_1 \int_0^\infty \frac{x^3}{e^x - 1} dx \tag{3}$$

where W is the total energy per unit area emitted, ν is the wavenumber, and C_1 is a yet undetermined constant. Recall that $B = \frac{2hc^2\nu^3}{e^{\frac{hc\nu}{kT}} - 1}$, where the constants have their standard meaning in astronomy. This can be solved by using a simple substitution of $x = \frac{hc\nu}{kT}$. This value of x means that $\mathrm{d}\nu = \frac{KT}{hc}\mathrm{d}x$. The question is now to find the value of C_1.

The next step is to plug our value of x, and $\mathrm{d}x$ into 3 and then solve algebraically for C_1:

$$W = \int_0^\infty \pi B \mathrm{d}\nu = \pi \int_0^\infty \frac{2hc^{-2}\nu^3}{e^{\frac{hc\nu}{kT}} - 1} \mathrm{d}\nu, \tag{4}$$

$$\rightarrow \pi \int_0^\infty \frac{\frac{2k^4T^4}{h^3c^2}x^3}{e^x - 1} \mathrm{d}x. \tag{5}$$

From 5, it is clear that $C_1 = \frac{\pi 2k^4T^4}{h^3c^2}$.

PART B

In question 3B, we are asked with writing a program to calculate the value for W given a temperature T. For our solution, we choose to use `scipy.integrate.quad` to numerically solve. `quad` not only gives the solution, it also returns an estimate of the absolute error in the result. In order to use this `scipy` function, we need to write a function for the integrand and give boundary values to integrate for. In our case, the lower bound is 0, and the upper bound is ∞, we used `numpy.inf` to represent this boundary as recommendation by the `scipy` documentation.

`scipy.integrate.quad` was chosen because of its ability to very accurately calculate integrals of high degree polynomials, as the integrand can be expressed as a Taylor series. Additionally, for its ease in error estimation and calculating integrals with ∞ boundaries.

```
'''# pseudo code

# import scipy, numpy
# import constants from scipy (in SI units)

# write a function to calculate integrand (using the new integrand IE x)
    # Uses given T (in kelvin) to calculate a C_1 value
    # then returns value of integrand

# numerically integrate using scipy quad
    # from 0 to np.inf
    # answer and accuracy estimate is included in quad
    # print values

'''
```

PART C

We can use our program from Q3B, to estimate a value for the Stefan-Boltzmann constant. Recall that $W = \sigma T^4$, where σ is the Stefan-Boltzmann constant. All that is needed to find σ is to multiply by T^{-4}. The associated error is $\delta\sigma$, is given by $\delta\sigma = (\delta W)T^{-4}$ (standard error propagation). Our reported value is $\sigma = 5.670374419 * 10^{-08} \pm 2 *$

10^{-17} $[\frac{w}{m^2K^4}]$, the value in `scipy.constants` is $\sigma = 5.670374419 * 10^{-08}$ $[\frac{w}{m^2K^4}]$. Our value matches the true value, until the limit of precision in the true value[5]

[5] Given the precision of this integration method, it is important to take note of how many significant figures there are in the physical constants used, and the limit of floating point accuracy in python. In our case, the error in the integral is close to the precision of the other physical constants but does not exceed them.

QUESTION 4 - EXPLORING ROUNDOFF ERROR

PART A

We have the two functions

$$p(u) = (1-u)^8 \tag{6}$$

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

Our first goal is to examine these functions close to the value 1. This will allow us to see how they differ numerically, even though they are mathematically the same function. We can see that difference below. We can see from 1 that

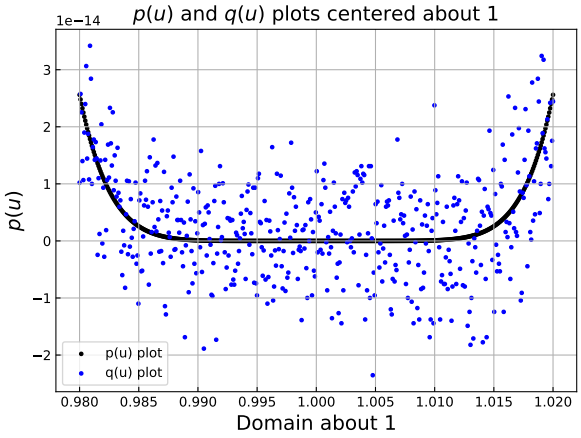

Figure 1: We can see here that when we zoom in on the region centered about 1 the numerical values of the functions differ significantly.

$q(u)$ is much noisier than $p(u)$. This could be explained by the fact that $q(u)$ has many more terms than $p(u)$, given way to decimal values playing a larger role in the overall value of the function.

PART B

Now we we take the difference of our two functions and look at their distribution and histogram. We can look at

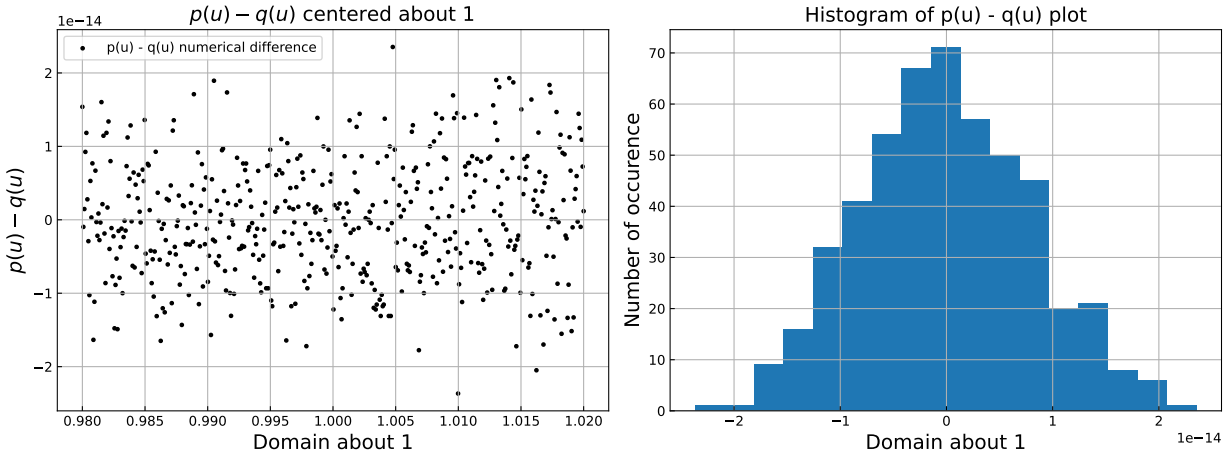

Figure 2: *Left*: The numerical difference between the two functions $p(u)$ and $q(u)$ plotted out. We see the spread is centered about 0, but is not extact. *Right*: The histogram showing the distribution of the difference.

the standard deviation (using Numpy) of this distribution, and find that it is on the order of 8×10^{-15}. This is an extremely small spread. We can compare this to the error

$$\sigma_r = C\sqrt{N}\sqrt{\bar{x^2}}, \tag{8}$$

with $C = 1 \times 10^{-16}$ as the error constant, $N = 10$ for the number of terms in the polynomial, and the mean of the summation of squares. Which gives us a value of 2×10^{-15}. We can note that these values are beginning to border on the limit of decimal precision.