Lab Report 2

By Seunghyun Park 1003105855 & Juann Jeon 1005210166

Equation are given at the end of page for reference

Question 1 done by Seunghyun Park

Question 2 done by Seunghyun Park

Question 3 done by Seunghyun Park

Question 4 done by Juann Jeon

Lab Report done by Juann Jeon

Q1 a)

```
import numpy as np

data = np.loadtxt('filename.txt') #load the data
std = np.std(data, ddof = 1) #Calculate the correct standard deviation

#Define Equation (1)
def std_eq1 (data):
        x = data
        n = len(data)
        x̄ = (1/n) Σ_{i=1}^{n} x_i #Calculate the mean of data using for loop
        Calculate Σ_{i=1}^{n} (x_i − x̄)^2 using for loop
        return σ = sqrt( (1/(n−1)) Σ_{i=1}^{n} (x_i − x̄)^2 ) #Standard deviation Equation (1)

#Define Equation (2)
def std_eq2 (data):
        x = data
        n = len(data)
        x_square: Σ_{i=1}^{n} x_i^2 #Calculate the sum of squared data using for loop
        x̄ = (1/n) Σ_{i=1}^{n} x_i #Calculate the mean of data using for loop

        if the equation taking square root of negative number:
                return error message
        else:
                return σ = sqrt( (1/(n−1)) ( Σ_{i=1}^{n} x_i^2 − n x̄^2 ) ) #Standard deviation Equation (2)

relative_error_1 = | (std_eq1 − std) / std | #Calculate relative error of Equation (1)
relative_error_2 = | (std_eq2 − std) / std | #Calculate relative error of Equation (2)
```

Q1 b)

```
The standard deviation of speed of light using numpy.std is 0.07901054781905067
The standard deviation of speed of light using equation 1 is 0.07901054781905066
The standard deviation of speed of light using equation 2 is 0.07901054811458154

The relative error of the calculated standard deviation using equation 1: 1.7564474859226715e-16
The relative error of the calculated standard deviation using equation 2: 3.740397602946237e-09
```

Figure 1: Standard deviation and relative error output of Q1 b) from Lab02_Q1.py

From Figure 1, we can observe relative error by using Equation (2) is much larger than using Equation (1), by $\sim 10^7$ (i.e. there is a numerical error in Equation (2) starting 9th digit).

Q1 c)

```
The standard deviation of normal distribution with mean = 0 using numpy.std is 0.989408290967698
The standard deviation of normal distribution with mean = 0 using equation 1 is 0.9894082909676983
The standard deviation of normal distribution with mean = 0 using equation 2 is 0.9894082909676973

The standard deviation of normal distribution with mean = 1.e7 using numpy.std is 1.014936192066419
The standard deviation of normal distribution mean = 1.e7 using equation 1 is 1.0149361920664195
The standard deviation of normal distribution mean = 1.e7 using equation 2 is 0.848740349040886

The relative error of std of normal with mean 0 using equation 1: 3.3663242003135877e-16
The relative error of std of normal with mean 0 using equation 2: 6.732648400627175e-16
The relative error of std of normal with mean 1e7 using equation 1: 4.3755382192637455e-16
The relative error of std of normal with mean 1e7 using equation 2: 0.16375004096282825
```

Figure 2: Standard deviation and relative error output of Q1 c) from Lab02_Q1.py

From Figure 2, we can observe that when the mean is 0, relative error for both Equation (1) and Equation (2) stays approximately equal to the actual standard deviation found using np.std(). When mean is 10^7 which is far from 0, we can see that while relative error for Equation (1) has not changed much, relative error for Equation (2) became very large.

From this observation, we can conclude that Equation (1) is relatively accurate compared to Equation (2) at any value of mean, while Equation (2) quickly becomes unreliable as mean deviates further from 0.

Q1 d)

```
import numpy as np

data = np.loadtxt('filename.txt') #load the data
std = np.std(data, ddof = 1) #Calculate the correct standard deviation

def fix_std_eq2 (data):
```

x = data

n = len(data)

$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ #Calculate the mean of data using for loop

$x = x - \bar{x}$ #Make x close to 0 so it becomes more accurate

x_square: $\sum_{i=1}^{n} x_i^2$ #Calculate the sum of corrected squared data using for loop

$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ #Calculate the mean of corrected data using for loop

if the equation taking square root of negative number:

return error message

else:

return $\sigma = \sqrt{\frac{1}{n-1}(\sum_{i=1}^{n} x_i^2 - n\bar{x}^2)}$ #Standard deviation Equation (2)

```
The standard deviation of normal distribution with mean 1.e7 using fixed equation 2 is 1.01493619206642
The relative error of std of normal with mean 1e7 using fixed equation 2:  8.751076438527491e-16
```

Figure 3: Standard deviation and relative error output of Q1 d) from Lab02_Q1.py

From Figure 3, we can observe that standard deviation after changing x relatively close to 0 by subtracting $\bar{x}$ lead to very small relative error compared to Figure 1, and the same digit of error of relative error using Equation (1).

Q2 Pseudocode

```
import numpy as np
import time

#Define a function we want to calculate
def f (x):
        return 4/(1+x^2)
#Define a trapezoidal method
def trapezoidal (a, b, N):
        # a = Lower bound of integral
        # b = Upper bound of integral
        # N = number of slice
        h = ((b-a)/N) # Width of slice
        Calculate sum_{k=1}^{N-1} f(a + kh) term using for loop
        Calculate the integral using I(a,b) = h[1/2 f(a) + 1/2 f(b) + sum_{k=1}^{N-1} f(a + kh)]
        return I(a,b)

#Define a simpson method
def simpson (a, b, N):
        # a = Lower bound of integral
        # b = Upper bound of integral
        # N = number of slice
        h = ((b-a)/N) # Width of slice

        Calculate sum_{k odd} f(a + kh) term using for loop

        Calculate sum_{k even} f(a + kh) term using for loop
        Calculate the integral using

          I(a,b) = 1/3 h[f(a) + f(b) + 4 sum_{k odd} f(a + kh)) + 2 sum_{k even} f(a + kh)]
        return I(a,b)

#For Q 2b)
# a = 0, b = 1, N = 4
trape_4 = trapezoidal(0, 1, 4) #Calculating the integral with N = 4 using Trapezoidal's
rule
simp_4 = simpson(0, 1, 4) # Calculating the integral with N = 4 using Simpons's rule

#For Q 2c)
n_trape = #slices required to approximate an error of O(10^-9) for trapezoidal
n_simp = #slices required to approximate an error of O(10^-9) for simpson

Find the n_trape value using 'if'
```

```
        π - trapezoidal(0,1,n_trape) < 10**(-9):
        print(n_trape)

Find the n_simp value using 'if'
        π - trapezoidal(0,1,n_simp) < 10**(-9):
        print(n_simp)

t1 = time.time() #Start time measure for trapezoidal method
trape_error = π - trapezoidal(0, 1, n_trape) #Find an error of O(10^-9) for trapezoidal
t2 = time.time() #End time measure for trapezoidal method
t_trape = t2 - t1 #Find time it took for trapezoidal method
t3 = time.time() #Start time measure for simpson method
simp_error = π - simpson(0, 1, n_simp) #Find an error of O(10^-9) for simpson
t4 = time.time() #End time measure for simpson method
t_simp = t4 - t3 #Find time it took for simpson method

#For Q 2d)
def error_trapezoid(i1,i2):
        return (i2-i1)/3

i1 = trapezoidal(0,1,16) #Error estimate of N_1 = 16
i2 = trapezoidal(0,1,32) #Error estimate of N_2 = 32
```

Q2 a)

$$\int_0^1 \frac{4}{1+x^2} dx = [4arctan(x)]_0^1 = \pi$$

Q2 b)

```
The exact value of the integral is 3.141592653589793
The value of the integral using Trapezoidal's rule is 3.1311764705882354
The value of the integral using Simpson's rule is 3.14156862745098
```

Figure 4: Q2 b) from Lab02_Q2.py

From Figure 4, we can see that Simpson's rule gives a more accurate outcome compared to Trapezoidal rule.

Q2 C)

```
For the Trapezoidal method, the slices to approximate the integral with an error of O(10^-9) is 16384 and the error is
6.208913383431991e-10
For the Simpson method, the slices of approximate the integral with an error of O(10^-9) is 32 and the error is
3.695665995451236e-11

The times takes to compute the integral with an error of O(10^-9) for Trapezoidal is 0.007979393005371094
The times takes to compute the integral with an error of O(10^-9) for Simpson is 0.0
```

Figure 5: Q2 c) from Lab02_Q2.py

From Figure 5, we can see that Trapezoidal rule required $2^{14} = 16384$ number of slices to have an error of $O(10^{-9})$ while Simpson's rule only required $2^{5} = 32$ number of slices to have an error of $O(10^{-9})$.

Since there are many slices to calculate, the Trapezoidal method took about 8ms to finish calculation while Simpson's rule took it immediately.

Q2 d)

```
The error estimation of Trapezoidal Method is  0.00016276037786200348
```

Figure 6: Q2 d) from Lab02_Q2.py

Figure 6 shows the error estimation of Trapezoidal rules using $N_1 = 16$ and $N_2 = 32$. The error estimation of Trapezoidal rule can be obtained by the equation $\epsilon_2 = ch^2 = \frac{1}{3}(I_2 - I_1)$.

Q2 e)

Notice from Q2 b) and Q2 c) that Simpson's rule has very high accuracy even at low number of slices (i.e. N is small). Because of this, the method from Textbook p.153 would not work for Simpson's rule as there is small to no error for error estimation, as precision for Simpson's rule could easily pass beyond $O(10^{-16})$ where numerical error often appears from computation. So to make Simpson's rule accurate on computation, we need not to adapt error estimation but simply raise the number of slices (i.e. make N larger).

Q3 a)

$B = \frac{2hc^{-2}v^3}{e^{\frac{hv}{kt}}-1}$

Let $x = \frac{hv}{kt}$, then $dx = \frac{h}{kt}dv$

$W = \pi\int_0^\infty B dv = \pi\int_0^\infty \frac{2hc^{-2}k^4T^4x^3}{h^4(e^x-1)}dx = \frac{2\pi k^4T^4}{h^3c^2}\int_0^\infty \frac{x^3}{e^x-1}dx$

$\therefore C_1 = \frac{2\pi k^4T^4}{h^3c^2}$

Q3 b)

import numpy as np
from scipy import constants

h = #planck constant
c = #speed of light
k = #boltzmann constant
σ = #Stefan-Boltzmann constant
T = #Any temperature in Kelvin
N = #Number of slice
a = #Lower bound of integral, it is close to 0 but not 0 to avoid division by 0
b = #Upper bound of integral, it can be sufficiently large number to replace infinity

#Define an Equation $\frac{x^3}{e^x-1}$
def f (x):
    return $\frac{x^3}{e^x-1}$

#Integral using simpson method
def simpson (a, b, N):
    # a = Lower bound of integral
    # b = Upper bound of integral
    # N = number of slice
    $h = (\frac{b-a}{N})$ # Width of slice

    Calculate $\sum_{k\ odd} f(a + kh)$ term using for loop

    Calculate $\sum_{k\ even} f(a + kh)$ term using for loop
    Calculate the integral using

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

    return $I(a,b)$

#Function to calculate W given T
def W (T):

return $\frac{2\pi k^4 T^4}{h^3 c^2}$ * simpson(a,b,N) * T^4

accuracy = $\frac{\sigma T^4}{W(T)} \times 100\%$ #Calculate the accuracy of W(T)

```
The calculated W(T) when T = 273.15K is 315.6578273744114 Wm^-2
The actual W(T) when T = 273.15 K is  315.6578223008046 Wm^-2
The accuracy of the calculated value is 99.99999839268779 %
```

Figure 7: Outcome of Q3 b) from Lab02_Q3.py

Notice that for large x, Equation $\frac{x^3}{e^x-1}$ converges to zero. (i.e. $\lim_{x \to \infty} \frac{x^3}{e^x-1} = 0$). Also, the lower bound of the integral cannot be equal to zero since the Equation $\frac{x^3}{e^x-1}$ becomes undefined at a = 0. Therefore, for numerical computation, with sufficiently large x it can substitute the place of the upper bound of integral in Equation $\frac{x^3}{e^x-1}$ and sufficiently small x it can replace the lower bound a = 0. For integration, Simpson's rule is used as it has higher precision than the Trapezoidal rule which was shown from Q2. Figure 7 shows the accuracy of integral using the upper bound of integral b = 500, which shows it has high precision with reasonably large b.

Q3 c)

```
The calculated Stefan-Boltzmann constant is 5.670374510140622e-08 Wm^-2K-^4
The Stefan-Boltmann constant is 5.670374419e-08 Wm^-2K-^4
```

Figure 8: Outcome of Q3 c) from Lab02_Q3.py

Figure 8 shows the outcome of Stefan-Boltzman constant by numerical computation, which shows accurate value of Stefan-Boltzmann constant from scipy upto 7 significant digits.

Q4 a)

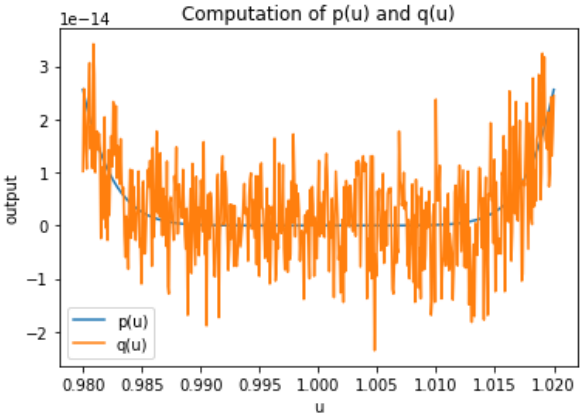

Figure 9: Plot of p(u) and q(u)

Figure 9 shows the plot of p(u) and q(u), where:

$$p(u) = (1 - u)^8$$

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

Where $u \in [0.98,\ 1.02]$ (in python, it is done by `np.linspace(0.98, 1.02, num = 500)`). Mathematically $p(u)$ is equal to $q(u)$, but as shown on Figure 9, in python $q(u)$ shows many numerical errors compared to p(u). It is because the equation $q(u)$ has much more operations during the calculation compared to the equation $p(u)$. The roundoff error arises in each step of operation which makes the $q(u)$ plot noiser.

Q4 b)

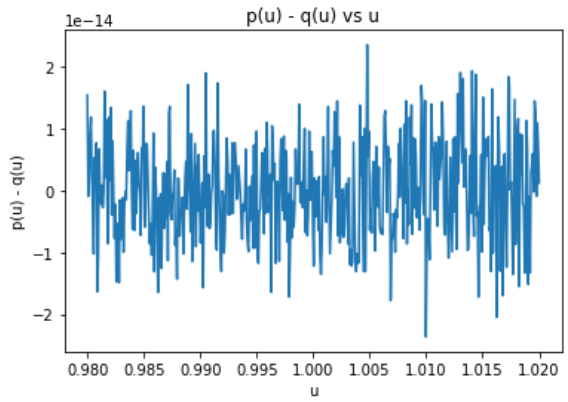

Figure 10: Plot of p(u) - q(u)

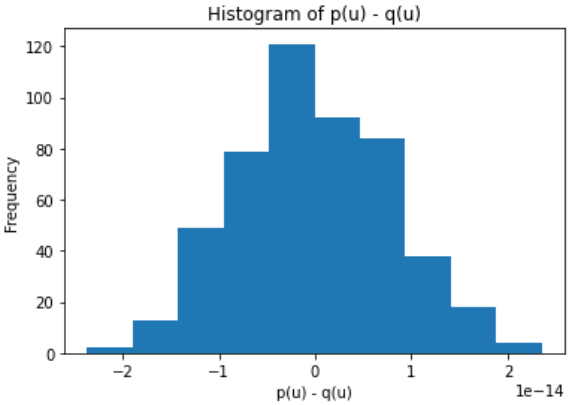

Figure 11: Histogram of p(u) - q(u)

Figure 10 shows $p(u) - q(u)$ and Figure 11 shows histogram of $p(u) - q(u)$. Notice that Figure 11 has a normal distribution, which shows there is a random error, hence there is an error propagation.

```
standard deviation of p(u) - q(u) using np.std(): 8.000009349796112e-15
Using Equation (3): 1.789017587364345e-29
standard deviation of p(u) - q(u) without C * sqrt(N): 8.000729876578688e-15
Digits of C * sqrt(N): 2.23606797749979e-15
```

Figure 12: Standard deviation of $p(u) - q(u)$ with $x = p(u) - q(u)$

In this question, we calculate two different standard deviation of $p(u) - q(u)$ by treating it in two different ways. The Figure 12 shows the standard deviation of $p(u) - q(u)$ when we use the equation (3) with $x = p(u) - q(u)$. The result shows that when using $x = p(u) - q(u)$, error piles up and results in much smaller standard deviation compared to the `np.std()` method. If we check the term $C \times \sqrt{N}$ separately, it has the same size of e-15, which shows error is around 100%.

```
The standard deviation when treat (p-q) as the sum of a series of terms: 1.1342783595586094e-14
```

Figure 13: Standard deviation when treating (p-q) as the sum of a series of terms, where each term is a power of u or of (1-u)

The second way we calculate the standard deviation of $p(u) - q(u)$ is by treating $p(u) - q(u)$ as the sum of a series of terms, where each term is a power of u or of $(1 - u)$. The standard deviation is about 40% larger in magnitude than using `np.std()`, but it still stays relatively similar to `np.std()` unlike using Equation (3).

Q4 c)

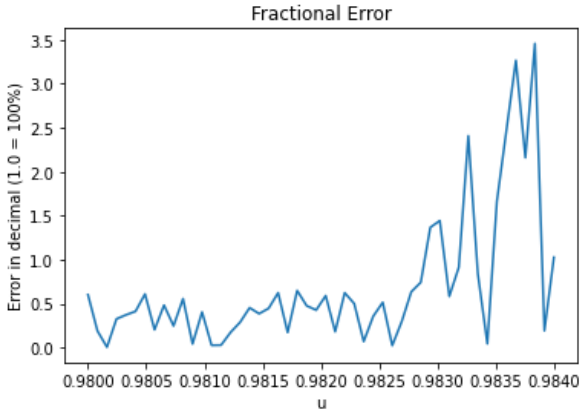

Figure 14: Error using abs(p - q)/abs(p), upto 0.9840

```
The error approaches 100% at 0.982938775510204
```

Figure 15: Point where error approaches 100%

Figure 12 shows the plot of Error from 0.98 to 0.984. From Figure 14 we can observe that as u gets larger the Error diverges to positive infinity very quickly. Figure 15 shows the point where error approaches to 100%, which is approximately 0.983. Using Equation (4), by plug in values we can observe that decimal figures change from e-15 to e-17, which shows error is about 100% at u = 0.983.

Q4 d)

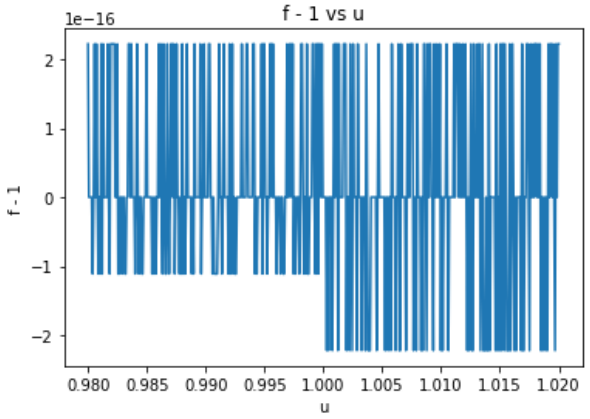

Figure 16:Plot of $f - 1$ over u

Figure 16 shows the plot of $f - 1$ over u, where $f = \frac{u^8}{u^4 \times u^4}$. Mathematically we can see that $u^8 = u^{4+4} = u^4 \times u^4$ so f should always equal to 1, but as Figure 16 shows on computation it has a numerical error. Equation (4.5) from textbook gives $\sigma = \sqrt{2}Cx$. Without factor of $\sqrt{2}$, $\sigma = Cx =$ 1.02e-16 , and with $\sqrt{2}$, $\sigma = \sqrt{2}Cx$ = 1.4142702442794026e-16

```
standard deviation of f using np.std(): 1.4087150935148697e-16
```

Figure 17: Standard deviation of $f = \frac{u^8}{u^4 \times u^4}$

Figure 17 shows the standard deviation of Figure 15 using `np.std()`, which is very similar to the value of $\sigma = \sqrt{2}Cx$ = 1.4142702442794026e-16

Appendix

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

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

Equation (3): $\sigma = C\sqrt{N}\sqrt{\bar{x}^2}$, where $C = 10^{-16}$

Equation (4): $\frac{\frac{\sigma}{N}}{\sum_i x_i} = \frac{C}{\sqrt{N}}\frac{\sqrt{\bar{x}^2}}{\bar{x}}$

Equation (5): $B = \frac{2hc^{-2}v^3}{e^{\frac{hv}{kT}}-1}$

Equation (6): $W = \sigma T^4$, σ here is a Stefan-Boltzmann constant not standard deviation