Igbaria Ahmad 322751041

Q1.

Answer :

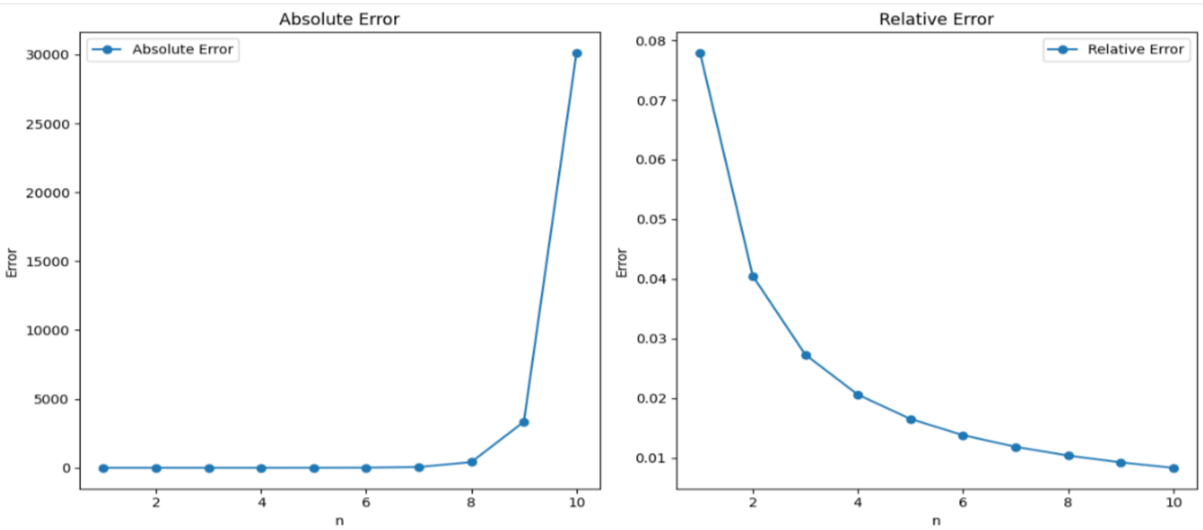
The absolute error is increases when n is gorws because the values of n! become very large and the difference between exact valute and estimeatd value using strigins formula also increases .

But in the Relative Error is decreases when n grows , although the abslute error is increases the estimated valu also grows. making the ratio between the absolute eeror and exact value becoming smaller .

we can to see is in the graphs below .

the code also in the python file .

```
import math
import matplotlib.pyplot as plt


def stirling_formula(n):
    approx_result=math.sqrt(2* math.pi * n)*(n/ math.e)**n
    return approx_result

def calculateAbsulte_Relative_errors(n_range):
    absolute_errors = []
    relative_errors = []

    for n in n_range: #for every n between 1 to 10
        true_value=math.factorial(n)
        approx_value= stirling_formula(n)
        absolute_error = abs(approx_value - true_value) #in order the forumla in the frist lecture
        relative_error = absolute_error /true_value #in order to the forumla in the first lecuture

        absolute_errors.append(absolute_error) #add the absoulte error to the list
        relative_errors.append(relative_error) #add the realtive erorr to the list

    return absolute_errors, relative_errors

n_range = list(range(1, 11)) # n 1 -> 10
absolute_errors, relative_errors = calculateAbsulte_Relative_errors(n_range)

#plot the result in grapth
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.plot(n_range, absolute_errors, 'o-', label='Absolute Error')
plt.xlabel('n')
plt.ylabel('Error')
plt.title('Absolute Error')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(n_range, relative_errors, 'o-', label='Relative Error')
plt.xlabel('n')
plt.ylabel('Error')
plt.title('Relative Error')
plt.legend()

plt.tight_layout()
plt.show()
```



Q2 . a

$$\epsilon_{mach} \approx |3 * \left(\frac{4}{3} - 1\right) - 1$$

This method is reasonable because it provides a quick way to estimate the machine epsilon

The idea is to use simple arithmetic operations that are prone to rounding errors due to the limited precision of floating-point representation.

For example :

$$\left(\frac{4}{3} - 1\right) = \frac{1}{3}$$

$$\frac{1}{3} * 3 = 1 \ \rightarrow 1 - 1 = 0$$

But the result is not be exactly zero , but a small value representing the machine epsilon.

Q2 . b

```
[2] import numpy as np


    #SP metohod
    sp_value = np.float32(4) /np.float32(3)
    epsilon_sp = np.abs(np.float32(3) * (sp_value - np.float32(1)) - np.float32(1))
    print(f"Unit roundof SP: {epsilon_sp}")

    #DP method
    dp_value =np.float64(4) /np.float64(3)
    epsilon_dp = np.abs(np.float64(3) * (dp_value - np.float64(1)) - np.float64(1))
    print(f"Unit roundoff DP): {epsilon_dp}")
```

```
Unit roundof SP: 1.1920928955078125e-07
Unit roundoff DP): 2.220446049250313e-16
```

Q2 , c

The method for calculating the unit roundoff will not work the same way in a system with base 3because the rounding errors accumulate differently. In base 3, the errors accumulate in a way that leads to an inaccurate ϵ_{mach} Therefore, the method is not as effective in base 3.

Q3.

In SP:

The precision is limited to 7 decimal digits , the accumulated erros in sum will cause to the value to eventually stabiliz and not continue to grow as expected .

I choose limit to be 1000000

The reason the series stops is that the floating point system reaches its precision limit , after a certain point adding very small values does not change the total sum due to the rounding of the numbers

```python
import numpy as np

def harmonic_series_sum_sp(limit):
    sum_value = np.float32(0.0)
    for n in range(1, limit + 1):
        sum_value += np.float32(1.0 / n)
        if n % 100000 == 0:
            print(f"Iteration {n}: Sum (SP) = {sum_value}")
    return sum_value

limit = 10000000

print("Calculating sum with SP:")
sp_sum = harmonic_series_sum_sp(limit)
print(f" sum with SP: {sp_sum}")
```

```
Calculating sum with Single Precision (SP):
Iteration 100000: Sum (SP) = 12.090850830078125
Iteration 200000: Sum (SP) = 12.782756805419922
Iteration 300000: Sum (SP) = 13.195324897766113
Iteration 400000: Sum (SP) = 13.481427192687988
Iteration 500000: Sum (SP) = 13.690691947937012
Iteration 600000: Sum (SP) = 13.881426811218262
Iteration 700000: Sum (SP) = 14.071255683898926
Iteration 800000: Sum (SP) = 14.16662311553955
Iteration 900000: Sum (SP) = 14.261990547180176
Iteration 1000000: Sum (SP) = 14.357357978820 8
Iteration 1100000: Sum (SP) = 14.452725410461426
Iteration 1200000: Sum (SP) = 14.54809284210205
Iteration 1300000: Sum (SP) = 14.643460273742676
Iteration 1400000: Sum (SP) = 14.7388277053833
Iteration 1500000: Sum (SP) = 14.834195137023926
Iteration 1600000: Sum (SP) = 14.92956256866455
Iteration 1700000: Sum (SP) = 15.024930000305176
Iteration 1800000: Sum (SP) = 15.1202974319458
Iteration 1900000: Sum (SP) = 15.215664863586426
Iteration 2000000: Sum (SP) = 15.31103229522705
Iteration 2100000: Sum (SP) = 15.403682708740234
Iteration 2200000: Sum (SP) = 15.403682708740234
Iteration 2300000: Sum (SP) = 15.403682708740234
Iteration 2400000: Sum (SP) = 15.403682708740234
Iteration 2500000: Sum (SP) = 15.403682708740234
Iteration 2600000: Sum (SP) = 15.403682708740234
Iteration 2700000: Sum (SP) = 15.403682708740234
Iteration 2800000: Sum (SP) = 15.403682708740234
Iteration 2900000: Sum (SP) = 15.403682708740234
Iteration 3000000: Sum (SP) = 15.403682708740234
Iteration 3100000: Sum (SP) = 15.403682708740234
Iteration 3200000: Sum (SP) = 15.403682708740234
Iteration 3300000: Sum (SP) = 15.403682708740234
Iteration 3400000: Sum (SP) = 15.403682708740234
Iteration 3500000: Sum (SP) = 15.403682708740234
Iteration 3600000: Sum (SP) = 15.403682708740234
Iteration 3700000: Sum (SP) = 15.403682708740234
Iteration 3800000: Sum (SP) = 15.403682708740234
Iteration 3900000: Sum (SP) = 15.403682708740234
Iteration 4000000: Sum (SP) = 15.403682708740234
Iteration 4100000: Sum (SP) = 15.403682708740234
Iteration 4200000: Sum (SP) = 15.403682708740234
Iteration 4300000: Sum (SP) = 15.403682708740234
Iteration 4400000: Sum (SP) = 15.403682708740234
Iteration 4500000: Sum (SP) = 15.403682708740234
Iteration 4600000: Sum (SP) = 15.403682708740234
Iteration 4700000: Sum (SP) = 15.403682708740234
Iteration 4800000: Sum (SP) = 15.403682708740234
Iteration 4900000: Sum (SP) = 15.403682708740234
Iteration 5000000: Sum (SP) = 15.403682708740234
Iteration 5100000: Sum (SP) = 15.403682708740234
```

```
Iteration 5600000: Sum (SP) = 15.403682708740234
Iteration 5700000: Sum (SP) = 15.403682708740234
Iteration 5800000: Sum (SP) = 15.403682708740234
Iteration 5900000: Sum (SP) = 15.403682708740234
Iteration 6000000: Sum (SP) = 15.403682708740234
Iteration 6100000: Sum (SP) = 15.403682708740234
Iteration 6200000: Sum (SP) = 15.403682708740234
Iteration 6300000: Sum (SP) = 15.403682708740234
Iteration 6400000: Sum (SP) = 15.403682708740234
Iteration 6500000: Sum (SP) = 15.403682708740234
Iteration 6600000: Sum (SP) = 15.403682708740234
Iteration 6700000: Sum (SP) = 15.403682708740234
Iteration 6800000: Sum (SP) = 15.403682708740234
Iteration 6900000: Sum (SP) = 15.403682708740234
Iteration 7000000: Sum (SP) = 15.403682708740234
Iteration 7100000: Sum (SP) = 15.403682708740234
Iteration 7200000: Sum (SP) = 15.403682708740234
Iteration 7300000: Sum (SP) = 15.403682708740234
Iteration 7400000: Sum (SP) = 15.403682708740234
Iteration 7500000: Sum (SP) = 15.403682708740234
Iteration 7600000: Sum (SP) = 15.403682708740234
Iteration 7700000: Sum (SP) = 15.403682708740234
Iteration 7800000: Sum (SP) = 15.403682708740234
Iteration 7900000: Sum (SP) = 15.403682708740234
Iteration 8000000: Sum (SP) = 15.403682708740234
Iteration 8100000: Sum (SP) = 15.403682708740234
Iteration 8200000: Sum (SP) = 15.403682708740234
Iteration 8300000: Sum (SP) = 15.403682708740234
Iteration 8400000: Sum (SP) = 15.403682708740234
Iteration 8500000: Sum (SP) = 15.403682708740234
Iteration 8600000: Sum (SP) = 15.403682708740234
Iteration 8700000: Sum (SP) = 15.403682708740234
Iteration 8800000: Sum (SP) = 15.403682708740234
Iteration 8900000: Sum (SP) = 15.403682708740234
Iteration 9000000: Sum (SP) = 15.403682708740234
Iteration 9100000: Sum (SP) = 15.403682708740234
Iteration 9200000: Sum (SP) = 15.403682708740234
Iteration 9300000: Sum (SP) = 15.403682708740234
Iteration 9400000: Sum (SP) = 15.403682708740234
Iteration 9500000: Sum (SP) = 15.403682708740234
Iteration 9600000: Sum (SP) = 15.403682708740234
Iteration 9700000: Sum (SP) = 15.403682708740234
Iteration 9800000: Sum (SP) = 15.403682708740234
Iteration 9900000: Sum (SP) = 15.403682708740234
Iteration 10000000: Sum (SP) = 15.403682708740234
 sum with SP: 15.403682708740234
```

But in DP The sum will continue to grow for a longer period due to the higher precision of about 16 significant decimal digits. The output will show that the sum of the series continues to increase for a longer time before stabilizing at a certain value.

```python
def harmonic_series_sum_dp(limit):
    sum_value = np.float64(0.0)
    for n in range(1, limit + 1):
        sum_value += np.float64(1.0 / n)
        if n % 100000 == 0:
            print(f"Iteration {n}: Sum (DP) = {sum_value}")
    return sum_value
print("Calculating sumDP:")
dp_sum = harmonic_series_sum_dp(limit)
print(f"sum with DP: {dp_sum}")
```

```
Calculating sum with Double Precision (DP):
Iteration 100000: Sum (DP) = 12.090146129863335
Iteration 200000: Sum (DP) = 12.78329081042982
Iteration 300000: Sum (DP) = 13.188755085205663
Iteration 400000: Sum (DP) = 13.476436740991026
Iteration 500000: Sum (DP) = 13.699580042305627
Iteration 600000: Sum (DP) = 13.881901432432876
Iteration 700000: Sum (DP) = 14.036051993212334
Iteration 800000: Sum (DP) = 14.16958329655086
Iteration 900000: Sum (DP) = 14.28736626276287
Iteration 1000000: Sum (DP) = 14.392726722864989
Iteration 1100000: Sum (DP) = 14.488036857214839
Iteration 1200000: Sum (DP) = 14.575048196325563
Iteration 1300000: Sum (DP) = 14.655090871947937
Iteration 1400000: Sum (DP) = 14.729198816629237
Iteration 1500000: Sum (DP) = 14.798191664306565
Iteration 1600000: Sum (DP) = 14.862730164611069
Iteration 1700000: Sum (DP) = 14.923354768045108
Iteration 1800000: Sum (DP) = 14.980513165545224
Iteration 1900000: Sum (DP) = 15.034580372195501
Iteration 2000000: Sum (DP) = 15.085873653425047
Iteration 2100000: Sum (DP) = 15.13466380568964
Iteration 2200000: Sum (DP) = 15.181183810502088
Iteration 2300000: Sum (DP) = 15.22563556319151
Iteration 2400000: Sum (DP) = 15.268195168552333
Iteration 2500000: Sum (DP) = 15.309017154739198
Iteration 2600000: Sum (DP) = 15.34823786020014
Iteration 2700000: Sum (DP) = 15.385978181060597
Iteration 2800000: Sum (DP) = 15.422345818617647
Iteration 2900000: Sum (DP) = 15.457437132271345
Iteration 3000000: Sum (DP) = 15.491338678199934
Iteration 3100000: Sum (DP) = 15.524128495646433
Iteration 3200000: Sum (DP) = 15.555877188920611
Iteration 3300000: Sum (DP) = 15.58664884285524
Iteration 3400000: Sum (DP) = 15.61650180154596
Iteration 3500000: Sum (DP) = 15.64548933421745
Iteration 3600000: Sum (DP) = 15.673660207215658
Iteration 3700000: Sum (DP) = 15.701059177650066
Iteration 3800000: Sum (DP) = 15.72772742117618
Iteration 3900000: Sum (DP) = 15.753702904205538
Iteration 4000000: Sum (DP) = 15.779020708984671
Iteration 4100000: Sum (DP) = 15.803713318526546
Iteration 4200000: Sum (DP) = 15.827810867201743
Iteration 4300000: Sum (DP) = 15.851341361843357
Iteration 4400000: Sum (DP) = 15.874330877425393
Iteration 4500000: Sum (DP) = 15.896803730752515
Iteration 4600000: Sum (DP) = 15.918782635055628
```

```
Iteration 4800000: Sum (DP) = 15.961342244945342
Iteration 4900000: Sum (DP) = 15.98196153002207
Iteration 5000000: Sum (DP) = 16.002164235298594
Iteration 5100000: Sum (DP) = 16.021966860633672
Iteration 5200000: Sum (DP) = 16.041384944605607
Iteration 5300000: Sum (DP) = 16.060433137762097
Iteration 5400000: Sum (DP) = 16.07912526902734
Iteration 5500000: Sum (DP) = 16.09747440601179
Iteration 5600000: Sum (DP) = 16.11549290989073
Iteration 5700000: Sum (DP) = 16.133192485423876
Iteration 5800000: Sum (DP) = 16.150584226623213
Iteration 5900000: Sum (DP) = 16.167678658521222
Iteration 6000000: Sum (DP) = 16.184485775426072
Iteration 6100000: Sum (DP) = 16.201015076011387
Iteration 6200000: Sum (DP) = 16.217275595561112
Iteration 6300000: Sum (DP) = 16.233275935627393
Iteration 6400000: Sum (DP) = 16.2490242913555
Iteration 6500000: Sum (DP) = 16.264528476689737
Iteration 6600000: Sum (DP) = 16.27979594765469
Iteration 6700000: Sum (DP) = 16.294833823888503
Iteration 6800000: Sum (DP) = 16.3096489085765
Iteration 6900000: Sum (DP) = 16.32424770693144
Iteration 7000000: Sum (DP) = 16.33863644334839
Iteration 7100000: Sum (DP) = 16.352821077334756
Iteration 7200000: Sum (DP) = 16.36680731833111
Iteration 7300000: Sum (DP) = 16.380600639512124
Iteration 7400000: Sum (DP) = 16.39420629064256
Iteration 7500000: Sum (DP) = 16.407629310073748
Iteration 7600000: Sum (DP) = 16.420874535946453
Iteration 7700000: Sum (DP) = 16.433946616659316
Iteration 7800000: Sum (DP) = 16.446850020662605
Iteration 7900000: Sum (DP) = 16.45958904562878
Iteration 8000000: Sum (DP) = 16.47216782704436
Iteration 8100000: Sum (DP) = 16.484590346271396
Iteration 8200000: Sum (DP) = 16.496860438110453
Iteration 8300000: Sum (DP) = 16.50898179790833
Iteration 8400000: Sum (DP) = 16.520957988237235
Iteration 8500000: Sum (DP) = 16.53279244518387
Iteration 8600000: Sum (DP) = 16.54448848426301
Iteration 8700000: Sum (DP) = 16.556049305995653
Iteration 8800000: Sum (DP) = 16.567478001166386
Iteration 8900000: Sum (DP) = 16.57877555781645
Iteration 9000000: Sum (DP) = 16.589950855755458
Iteration 9100000: Sum (DP) = 16.601000691331667
Iteration 9200000: Sum (DP) = 16.611929761266516
Iteration 9300000: Sum (DP) = 16.622740676786258
Iteration 9400000: Sum (DP) = 16.63343596533133
Iteration 9500000: Sum (DP) = 16.64401807410216
Iteration 9600000: Sum (DP) = 16.65448937342126
Iteration 9700000: Sum (DP) = 16.664852159918908
Iteration 9800000: Sum (DP) = 16.675108659560088
Iteration 9900000: Sum (DP) = 16.68526103050885
Iteration 10000000: Sum (DP) = 16.695311365857272
Final Harmonic series sum with Double Precision (DP): 16.695311365857272
```

Q4. a

$x^2 - y^2$: More accurate calculation because the operations are simpler and less sensitive to rounding errors.

(x-y)(x+y): Less accurate due to cancellation error when x and y are close to each other.

The subtraction and addition operations can yield very small results, leading to significant rounding errors in the multiplication.

Therefore, according to what we learned, the expression$x^2 - y^2$ is more accurate in floating-point arithmetic than (x-y)(x+y).

Q4.b

There will be a significant diifernece between two expressions when x,y are close to each other , especially when x is much bigger than y .

There will be a significant difference between the two expressions when x and y are close in magnitude, especially when x is much larger than y . the difference (x-y) may be very smallleading to a significant loss of precision in the multiplication (x-y)(x+y) in floating-point arithmetic.

The difference (X-y) MAy be very small leading to significant loss of precesion in the (x-y)(x+y in the floating point arithmetic .

We can to see in the code when x=1 , y=0.9999999 (close to each other) the difference is :

```
[8]  def method1(x, y):
         return x**2 - y**2

     def method2(x, y):
         return (x - y) * (x + y)

     def demonstrate_difference(x, y):
         result_1 = method1(x, y)
         result_2 = method2(x, y)
         print("method 1:", result_1)
         print("method 2:", result_2)
         print("Difference:", abs(result_1 - result_2))

     x = 1.0
     y = 0.999999999
     demonstrate_difference(x, y)
```

```
method 1: 1.999999943436137e-09
method 2: 1.999999942436137e-09
Difference: 1.0000001492112815e-18
```

But if x=4 , y =1 :

```
[11] def method1(x, y):
         return x**2 - y**2

     def method2(x, y):
         return (x - y) * (x + y)

     def demonstrate_difference(x, y):
         result_1 = method1(x, y)
         result_2 = method2(x, y)
         print("method 1:", result_1)
         print("method 2:", result_2)
         print("Difference:", abs(result_1 - result_2))

     x = 4
     y = 1
     demonstrate_difference(x, y)
```

```
method 1: 15
method 2: 15
Difference: 0
```

Q5. A

Method 2 $x_k = a + kh\,, \quad k = 0, \dots, n$ is a better because it minimizes the errors

In method 1 the error accumulates at eatch step of calculating the values , but in method 2 each value is valaclated alone from the initial value (a)

So , resuling in less accumulation of computational erros in method 2

Q5.B

We in the range [0,1] , A=0 , B=1 (Rahel sent the email about this)

```python
import numpy as np
import matplotlib.pyplot as plt

def method1(a, b, n):
    h = (b-a)/ n
    x = [a]
    for k in range(1, n + 1):
        m=x[k-1]
        x.append(m + h)
    return x

def method2(a, b, n):
    h = (b - a) / n
    x = [a + k * h for k in range(n + 1)]
    return x

a = 0.0
b = 1.0
n = 10

method1_results= method1(a, b, n)
method2_results = method2(a, b, n)

print(f" the result of method 1: {method1_results}")
print(f"the result of method 2: {method2_results}")
```

```
the result of method 1: [0.0, 0.1, 0.2, 0.30000000000000004, 0.4, 0.5, 0.6, 0.7, 0.7999999999999999, 0.8999999999999999, 0.9999999999999999]
the result of method 2: [0.0, 0.1, 0.2, 0.30000000000000004, 0.4, 0.5, 0.6000000000000001, 0.7000000000000001, 0.8, 0.9, 1.0]
```

We can to see that the method 2 is better than method 1 because that it minimzed accumulation error .

Q6.

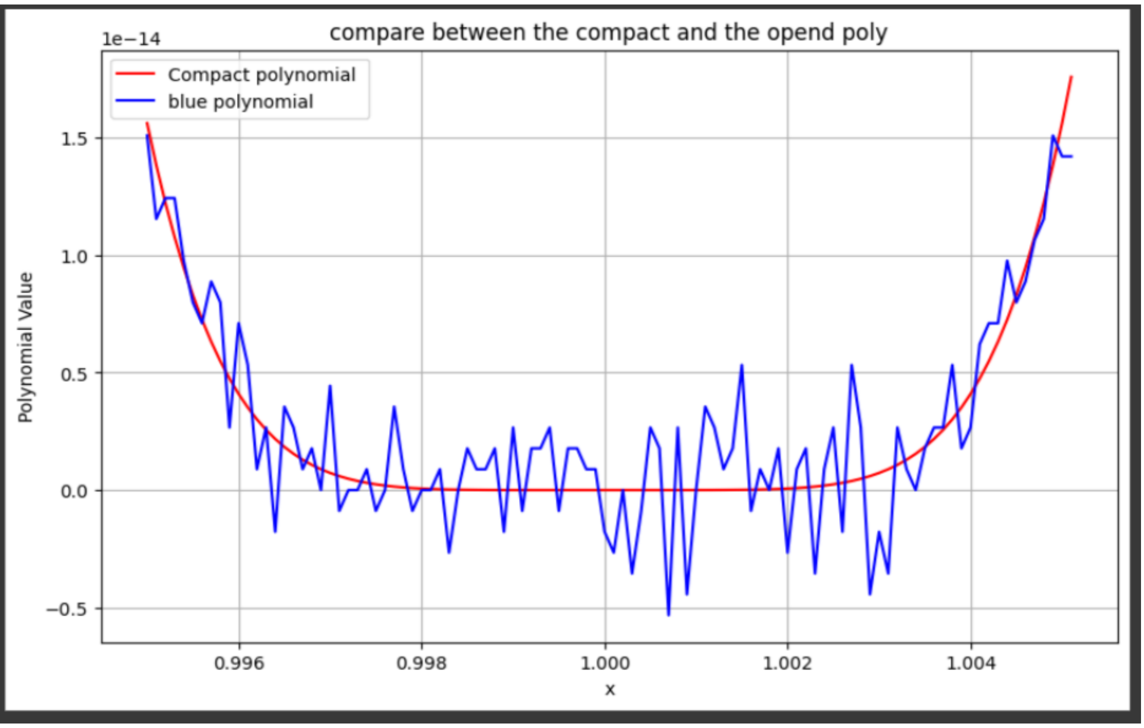
The x_values in the range about 0995 to 1.005)

```
[1]  import numpy as np
     import matplotlib.pyplot as plt

     def compact_polynomial(x):
         return (x - 1)**6

     def open_polynomial(x):
         return x**6 - 6*x**5 + 15*x**4 - 20*x**3 + 15*x**2 - 6*x + 1

     x_values = np.arange(0.995, 1.0051, 0.0001)
     compact_values = [compact_polynomial(x) for x in x_values]
     expanded_values= [open_polynomial(x) for x in x_values]

     # ציור גרפים
     plt.figure(figsize=(10, 6))
     plt.plot(x_values, compact_values, label='Compact polynomial ', color='red')
     plt.plot(x_values, expanded_values, label='blue polynomial ', color='blue')
     plt.xlabel('x')
     plt.ylabel('Polynomial Value')
     plt.title('compare between the compact and the opend poly')
     plt.legend()
     plt.grid(True)
     plt.show()
```

The Output of the code :

The Red is the compactform polynomial

The blue is the opened polynomial



In the compact polynomial , the polynomial equals zero at the point x=1 and remains positive for all other values .

The values of the compact polynomial are very close to zero when x is close to 1 , so they are near zero on the graph

However in the open polynomial the values may change more dramatically near this point (x=1) you can see on the graph that the values of the opend polynomial near (x=1) show small differences compared to the compat . THIS Is doue to calculate erorr from the sub and the sum operation involved in the expanded polynomial

The compact polynomial is moree accurate of calction because there are fewer and simpler operations , but the opend polynmail involves more sum and sub of large values , that is cause to calcation erros .

Therefore the compact polynomial is more accurate of calculation accuracy around the point x==1 .