Scientific Computing Homework 1

Erdem Şamlıoğlu

2448843

Question 1:

Here is the code for question 1 which plots g(n) for given n values and finding the values g(n)=0. (Commented the code to show the parts.)

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

n_values = np.arange(1, 1001)
epsilon = sys.float_info.epsilon

def f(n):
    return (n * (((n + 1) / n) - 1) - 1)

def g(n, epsilon):
    return f(n) / epsilon




g_values = [g(n, epsilon) for n in n_values]

#Plotting part of the question
plt.scatter(n_values, g_values, s=5)
plt.xlabel('n')
plt.ylabel('g(n)')
plt.title('Plot of g(n) for n in [1, 1000] and n is integer')
plt.grid(True)

plt.show()
valueforthis=f(1000)
```

```
#Finding zero values part
zero_values = [n for n, i in zip(n_values, g_values) if i == 0]
print("Values of n for g(n) is 0:", zero_values)
```

Part (a):

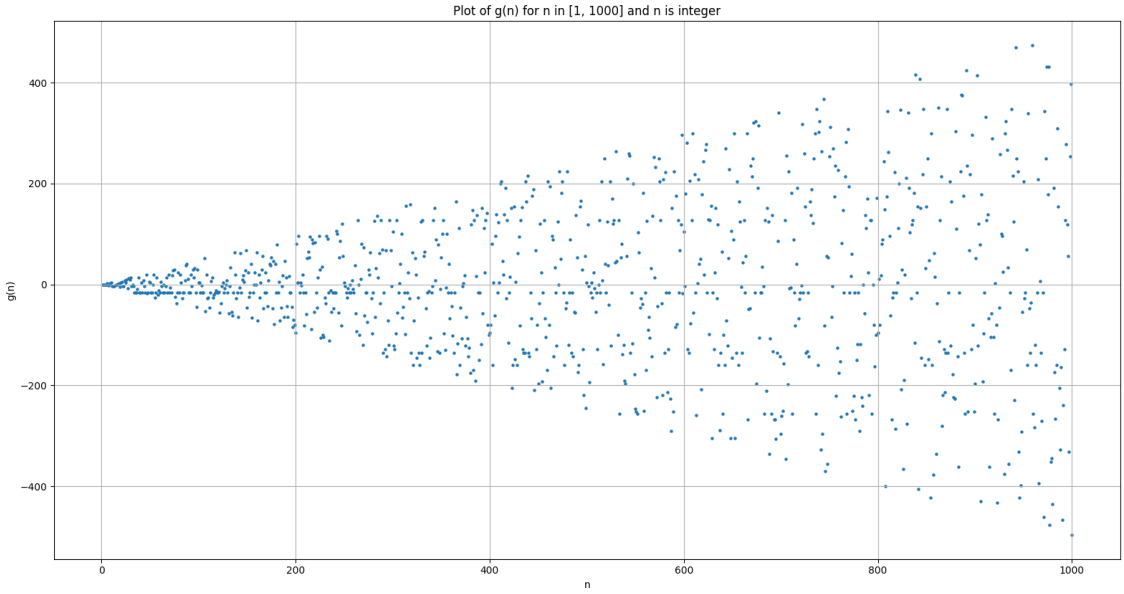

Figure 1: Plot of g(n)

The plot is shown in Figure 1.

Part (b):

The values that satisfy g(n)=0 are: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512. This is calculated in the code in the lines after the commented part as "Finding zero values part".

Part (c):

It is due to the precision loss in Python. Due to the way floating point numbers are represented in computers, there can be small rounding errors, which makes some errors in the calculations. For example the representation of 1/10: 0.0[0011] where the "0011" part is repeated infinitely many times. The reason is 1/10 isn't an multiple of power of two. Since we can't make the "0011" part infinitely in float point numbers, binary representation of 1/10 have to be truncated which could cause problems. In this example except the power of two, the other values of n will make it a value other than 0.
Here is an example code and the result of it:

```
def g():
return (((10+1)/10)-1)
print(g())
```

Which prints: 0.10000000000000009
Since there are only 10 values of power of two in [1,1000], the majority of ns will make g(n) a non zero value.

Part (d):

As I have mentioned in part c, except the values which makes g(n)=0, in the calculations there are errors due to float point arithmetic. As n increases the division $\frac{1}{n}$ gets closer to zero. The floating point representation have more errors for values which get closer to zero. So when $\frac{1}{n}$ is calculated for large values of n, and later multiplied by a large n, the inaccuracy gets higher too.

Since g(n) is related to f(n), and we divide it by ϵ, the errors are more visible to us and we can observe how they grow when n grows.

Question 2:

Part (a):

For the theoretical calculation, we can use arithmetic series summation formula. Which is $\frac{n(a_1+a_n)}{2}$, where n is the number of terms being added,a_1 is the first term, a_n is the last term. So we get,
$\frac{10^6(1+10^6 10^{-8}+1+10^{-8})}{2}$
$= 5 \times 10^5(2 + 0.01 + 0.00000001)$
$= 5 \times 10^5(2.01000001)$
$= 1005000.005$

Part (b):

Pairwise summation is a technique in numerical analysis to reduce the round off errors when we sum a sequence of finite precision floating point numbers by recursively breaking the sequence into two halves and adding the sums efficiently. It provides a very cost efficient summation which is close to the accuracy of compensated(Kahan) summation, while having a lower computational cost.

Part (c):

```
#Compensated
import numpy as np
import time
start_time = time.time()

n = 10**6
nums = [1 + (10**6 + 1 - i) * 10**(-8) for i in range(1, n + 1)]


def compensated_sum(nums, precision):
    sum_value = np.float32(0) if precision == 'single' else np.float64(0)
    compensation = np.float32(0) if precision == 'single' else np.float64(0)

    for num in nums:
        y = np.float32(num) if precision == 'single' else np.float64(num)
        y -= compensation
        t = sum_value + y
        compensation = (t - sum_value) - y
        sum_value = t

    return sum_value


compensated_single_precision = compensated_sum(nums, 'single')
compensated_double_precision = compensated_sum(nums, 'double')

print("--- %s seconds ---" % (time.time() - start_time))
print("Compensated Sum (Single Precision):", compensated_single_precision)
print("Compensated Sum (Double Precision):", compensated_double_precision)




#Pairwise
import numpy as np
```

```
import time
start_time = time.time()

n = 10**6
nums = [1 + (10**6 + 1 - i) * 10**(-8) for i in range(1, n + 1)]

threshold=2048
def pairwise_sum(nums, precision):
    if len(nums) <= threshold:
        if precision == 'single':
            result = np.float32(0)
        else:
            result = np.float64(0)
        for num in nums:
            result += num
    else:
        m = len(nums) // 2
        result = pairwise_sum(nums[:m], precision) + pairwise_sum(nums[m:], precision)
    return result


pairwise_single_precision = pairwise_sum(nums, 'single')
pairwise_double_precision = pairwise_sum(nums, 'double')

print("--- %s seconds ---" % (time.time() - start_time))
print("Pairwise Sum (Single Precision):", pairwise_single_precision)
print("Pairwise Sum (Double Precision):", pairwise_double_precision)
```

```
#Naive
import numpy as np
import time
start_time = time.time()


n = 10**6
nums = [1 + (10**6 + 1 - i) * 10**(-8) for i in range(1, n + 1)]

def naive_sum_single_precision(nums):
    result = np.float32(0)
    for num in nums:
        result += np.float32(num)
    return result
```

```
def naive_sum_double_precision(nums):
    result = np.float64(0)
    for num in nums:
        result += np.float64(num)
    return result

result_single_precision = naive_sum_single_precision(nums)
result_double_precision = naive_sum_double_precision(nums)


print("--- %s seconds ---" % (time.time() - start_time))
print("Naive Sum (Single Precision):", result_single_precision)
print("Naive Sum (Double Precision):", result_double_precision)
```

As I have commented the code lines, here are the compensated, pairwise and naive summations.

The Python code gives this as the result:
Compensated Sum (Single Precision): 1005000.0
Compensated Sum (Double Precision): 1005000.005
Pairwise Sum (Single Precision): 1005000.005
Pairwise Sum (Double Precision): 1005000.005
Naive Sum (Single Precision): 1002466.7
Naive Sum (Double Precision): 1005000.0049999995

I will comment on pairwise sum threshold value I have given in my code in part d and e.

Part (d):

The code I have implemented, to be able to find out the run time of the different summation algorithms:

```
import time
start_time = time.time()
print("--- %s seconds ---" % (time.time() - start_time)
```

pairwise: — 0.36741161346435547 seconds —
naive: — 0.7183852195739746 seconds —
compensated: — 1.1238110065460205 seconds —
These are the random taken ones from running the code to calculate the run time of it. I have run my code multiple times to observe the run times. The run times changes every time with a small amount of time. The most stable one is naive one whose value of run time doesn't change that much compared to others. 2nd is pairwise which also doesn't change that much compared to the compensated summation algorithm.

I have given a threshold value 2048 as threshold value for pairwise summation. I mainly tried values of power of 2, which had some inaccuracy unlike this result. I will comment on it in part e.

Comparing the errors:
For single precision:
Compensated has an absolute error of: 0.005
Pairwise has an absolute error of: 0.0
Naive has an absolute error of: 533.305

For single precision, pairwise has a higher accuracy than others, given a good threshold for the specific data set. Compensated has the 2nd accuracy among these 3. Naive summation is the 3rd in accuracy and has a very higher error compared to others.

For double precision:
Compensated has an absolute error of: 0.0
Pairwise has an absolute error of: 0.0
Naive has an absolute error of: 0.0000000005

For double precision, both pairwise and compensated summation methods have the same accuracy (if the pairwise is given a good threshold for this specific data). Naive summation has a lower accuracy, but still with a very small error which means it performs well in double precision.

Part (e):

As I have mentioned in part c and d that I would comment on the threshold, I had the chance to observe the logic and impact of the threshold on the algorithm.

Firstly, I have tried both very high and low numbers as thresholds. If it is set too high, then the algorithm is similar to naive summation, since the values in the array would be summed directly instead of being grouped. If it is set too low, then the recursion numbers can become high, which would result in function call number being high. This could increase the accuracy by grouping smaller numbers together and reducing the chances of large rounding errors, but it can also make the algorithm slower because of the recursion getting deeper. Here is an example for lower threshold (threshold=10):
— 0.8390569686889648 seconds —
Pairwise Sum (Single Precision): 1005000.0049999999
Pairwise Sum (Double Precision): 1005000.0049999999
Which has a worse run time than the previous one. The accuracy is also lower, which could be due to the float point arithmetic. Since it is not associative, the order is also important. Lower threshold means smaller groups, which could

affect the result.

As I have mentioned earlier, I have tried power of 2 for values. The value 2048 gave the highest accuracy, so I chose it. Later I have tried 4096,8192 etc. 4096 gave again a similar accuracy error like low threshold values, but 8192 gave the best accuracy like 2048. From these results, I have observed that for 4096,a mix of numbers with different magnitudes being summed together directly, leading to loss of precision. For 2048 and 8192,It is likely that the numbers being summed directly have more similar magnitudes, which leads to higher accuracy.

From the outputs I have observed:

Compensated Summation is the best since it has a very high accuracy, if the computational time isn't important and if one doesn't want to find the best threshold value for pairwise summation. Compensated Summation is especially good to effectively maintain the precision for a long list of floating point numbers.

Pairwise Summation is also very good with the accuracy. To optimize its' accuracy, one needs to find the best threshold for the given data set which could be challenging. As I have explained my observations and conclusions from it, for a given data, the optimal threshold could be more than 1 (2048 and 8192 in my case), and for a changing data set it, one should change the threshold every time according to the data change (data adding to the list, data subtracting from the list etc).

Naive Summation is faster compared to Compensated Summation, also it is faster compared to Pairwise Summation if recursion number is too high (not in threshold=2048 case.) but it is the least accurate, making it the worst compared to the other two. The reason naive accuracy being bad is the nature of floating point arithmetic. The other two summations has the same problem as well but they have an algorithm to reduce it, in case of naive summation there isn't any algorithm to prevent it.

The reasons that could be the reason making calculations less accurate:
Order of Operation((a+b)+c not being equal to a+(b+c).
Loss of Significance(With big data sets, many operations, very small numbers might disappear due to the limited precision of float point representation.
Rounding Errors(as I have mentioned in Question 1 part c, computers can't represent all decimal numbers.)

The reason Pairwise Summation doesn't get affected as much as Naive Summation is it recursively groups the list and sums the pairs of numbers, which generally add numbers with closer magnitudes. This provides reducing the loss of significance.

The reason Compensated doesn't get affected as much as Naive Summation is it keeps track of cumulative error and adjusts it for every addition.

Overall, it is speed vs accuracy. If one needs speed over accuracy, the best would be Pairwise Summation. If one needs accuracy over speed, the best would be Compensated Summation. If both are very important, finding an optimal threshold for pairwise and using it would be better.

Other Comments and Possible Improvements

For better accuracy, one should use double precision instead of single precision. As I have shown the results of the summation algorithms, double precision gave the better accuracy for all of the algorithms.

Pairwise Summation is great with both speed and accuracy when given a good threshold. Experimenting with different thresholds to find the best one for the given data set would make it better.

Also, for pairwise summation, instead of using recursion for the algorithm, we could consider an iterative algorithm which uses stack to prevent memory-intensity.

Another possible improvement for Pairwise Summation could be parallel processing to split and sum the data simultaneously.

For Compensated Summation, there is an improved version by Neumaier, which he calls "improved Kahan–Babuška algorithm". It covers the case when the next term to be added is larger in absolute value than the running sum, effectively swapping the role of what is large and what is small.