# CENG 371 - Scientific Computing
# 2023-1
# Homework 1

Anıl Eren Göçer
e2448397@ceng.metu.edu.tr

October 24, 2023

---

## Question 1

a) $f(n) = n\left(\frac{n+1}{n} - 1\right) - 1$ and $g(n) = f(n)/\epsilon$, where $n \in [1, 1000]$ and $\epsilon \approx 2.22$ x $10^{-16}$ in my computer

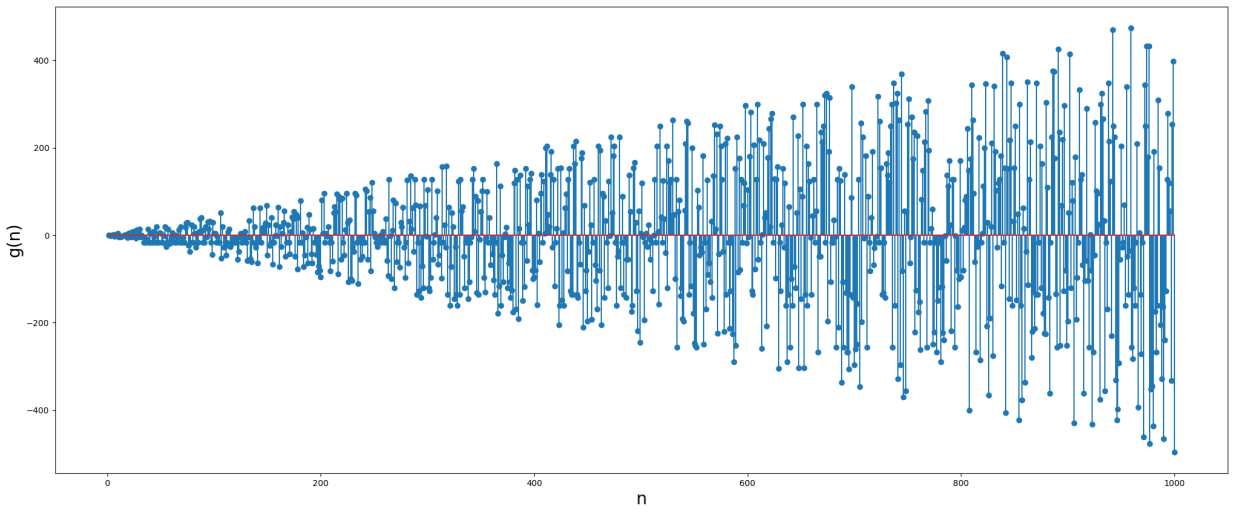


Figure 1: $n$ vs $g(n)$

b) $n$ values satisfying $g(n) = 0$ are $1, 2, 4, 8, 16, 32, 64, 128, 256, 512$

c) Remember that numbers are represent in base of 2 in IEEE 754 Floating Point Representation. That is any number $X = N + Z$ can be represented exactly in this representation if $N$ and $Z$ can be written in the following forms:

$$N = \alpha_0 \times 2^0 + \alpha_1 \times 2^1 + \alpha_2 \times 2^2 + ..... + \alpha_n \times 2^n$$
$$Z = \alpha_{-1} \times 2^{-1} + \alpha_{-2} \times 2^{-2} + \alpha_{-3} \times 2^{-3} + ..... + \alpha_m \times 2^{-m}$$

where $n$ and $m$ are finite integers decided by the IEEE 757 standard based on precision.

Also, recall the definition of **machine numbers**. Machine numbers are the numbers that can be represented exactly with respect to the representation described above. However, not all the numbers can be represented exactly in IEEE Floating Point Representation. As a result, these non-machine numbers are rounded to nearest machine when they take part in an operation. This results in **rounding errors**.

Now, observe the term $\frac{n+1}{n}$ which is equal to $1 + \frac{1}{n}$. The term $\frac{1}{n}$ yields **rounding errors** if it is not represented exactly. Moreover, it is not exactly representable if and only if $n$ is not a power of 2. Therefore, $\frac{1}{n}$ does not yield rounding error only for $1, 2, 4, 8, 16, 32, 64, 128, 256, 512$ in $[1, 1000]$. For the remaining $n$ values, it generates rounding errors and this leads to fact that $g(n) \neq 0$ for majority of $n$ values.

d) In the previous part, I stated that rounding error is generated for majority of the n values. Because of the term $n$ in the expression for $f(n)$ is multiplied with the expression $\left(\frac{n+1}{n} - 1\right)$, the rounding error is amplified as $n$ grows. As a result, $g(n)$ grows in size as n gets larger.

# Question 2

a)

$$\sum_{n=1}^{10^6} \text{nums}[n] = \sum_{n=1}^{n} \left(1 + (10^6 + 1 - n) \times 10^{-8}\right)$$

$$= \sum_{n=1}^{10^6} 1 + 10^{-8} \sum_{n=1}^{10^6} \left(10^6 + 1 - n\right)$$

$$= \sum_{n=1}^{10^6} 1 + 10^{-8} \sum_{n=1}^{10^6} 10^6 + 10^{-8} \sum_{n=1}^{10^6} 1 - 10^{-8} \sum_{n=1}^{10^6} n$$

$$= 10^6 + 10^{-8}.(10^6.10^6) + 10^{-8}.(10^6) - 10^{-8}. \left(10^6.\frac{(10^6 + 1)}{2}\right)$$

$$= 10^6 + 10^4 + 10^{-2} - 10^{-2}.\frac{(10^6 + 1)}{2}$$

$$= 10^6 + 10^4 + 10^{-2} - \frac{10^4}{2} - \frac{10^{-2}}{2}$$

$$= 1000000 + 10000 + 0.01 - 5000 - 0.005$$

$$= 1005000.005$$

b) Pairwise summation algorithm works by dividing the array into two subarrays, calculating each subarray's sum seperately in such a recursive manner and adding each half's sum to get the result. Splitting the subarray is applied until the length of the array reaches to a relatively small constant (in my implementation, this constant is 1) and then sum of this atomic array is calculated by naive summation.

c) Results of the summation algorithms with single and double precision are given below table. Note that 10 digits after the decimal point are given.

| Algorithm | Single Precision | Double Precision |
|---|---|---|
| **Naive Summation** | 1002466.6875000000 | 1005000.0049999995 |
| **Compensated Summation** | 1005000.0000000000 | 1005000.0050000000 |
| **Pairwise Summation** | 1005000.0000000000 | 1005000.0049999999 |

Table 1: Algorithms vs Single and Double Precision results on nums[n]

d)

**Error Comparison**

I experimentally calculated the **absolute error** of each method by using the formula:

$$\text{A.E} = |f(n) - \hat{f}(n)|$$

where $f(n)$ is the correct value, which corresponds to the theoretical value I calculated in part a). Its numerical value is 1005000.005. $\hat{f}(n)$ is the value generated by the algorithm I employed. As a result, I have obtained the following absolute errors:

| Algorithm | Single Precision | Double Precision |
|---|---|---|
| **Naive Summation** | 2533.3175000000047 | 4.656612873077393e-10 |
| **Compensated Summation** | 0.005000000004656613 | 0.0 |
| **Pairwise Summation** | 0.005000000004656613 | 1.1641532182693481e-10 |

Table 2: Algorithms vs Single and Double Precision Absolute Errors

As shown in the Table 2, absolute errors for each algorithm can be compared as follows:

Naive S.P. > Pairwise S.P. = Compensated S.P. > Naive D.P. > Pairwise D.P. > Compensated D.P.

**Runtime Comparison**

I experimentally calculated the **average runtimes** of these three algorithms by running each of them 10 times on the given input ($n \in [1, 10^6]$). Therefore, I have obtained the following values. Note that following values are in seconds.

| Algorithm | Single Precision | Double Precision |
|---|---|---|
| **Naive Summation** | 0.10348036289215087 | 0.11036529541015624 |
| **Compensated Summation** | 0.34583773612976076 | 0.37045485973358155 |
| **Pairwise Summation** | 2.219097208976746 | 2.327226734161377 |

Table 3: Algorithms vs Single and Double Precision Runtimes

As shown in the Table 3, average runtimes of these three algorithms can be compared as follows:

Pairwise D.P > Pairwise S.P. > Compensated D.P. > Compensated S.P. > Naive D.P. > Naive S.P.

Although these three algorithms have different numerical values for their runtimes, they all have the same algorithmic time complexity, which is **O(n)**. The reason behind is that all algorithms have to complete $n - 1$ addition for $n$ numbers in the input. Since addition is the core operation which makes up most of the computational cost, other computations can be neglected and cost of addition operation will dominate as $n$ grows.

In order to prove this fact, I calculated the runtimes for each algorithms with input sizes (range of $n$) 10, $10^2$, $10^3$, $10^4$, $10^5$, $10^6$ and $10^7$. As a result, I observed the **linear** relationship between input size and elapsed time as shown in the below figures.

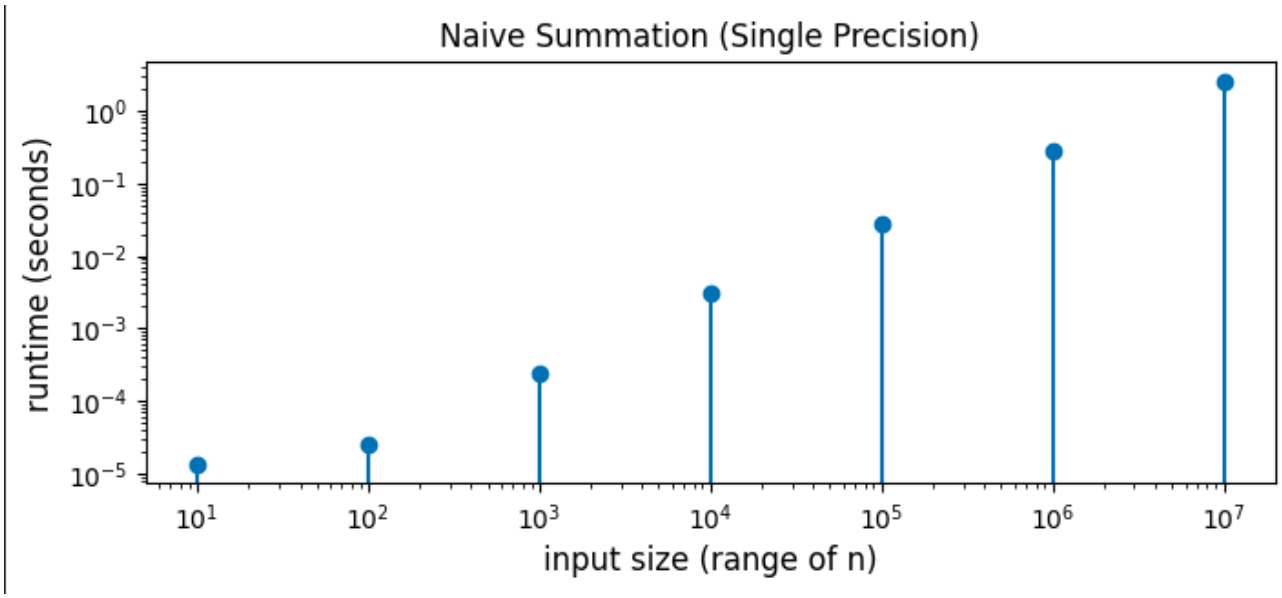


Figure 2: Runtime of Compensated Summation (Single Precision) Algorithm with different input sizes

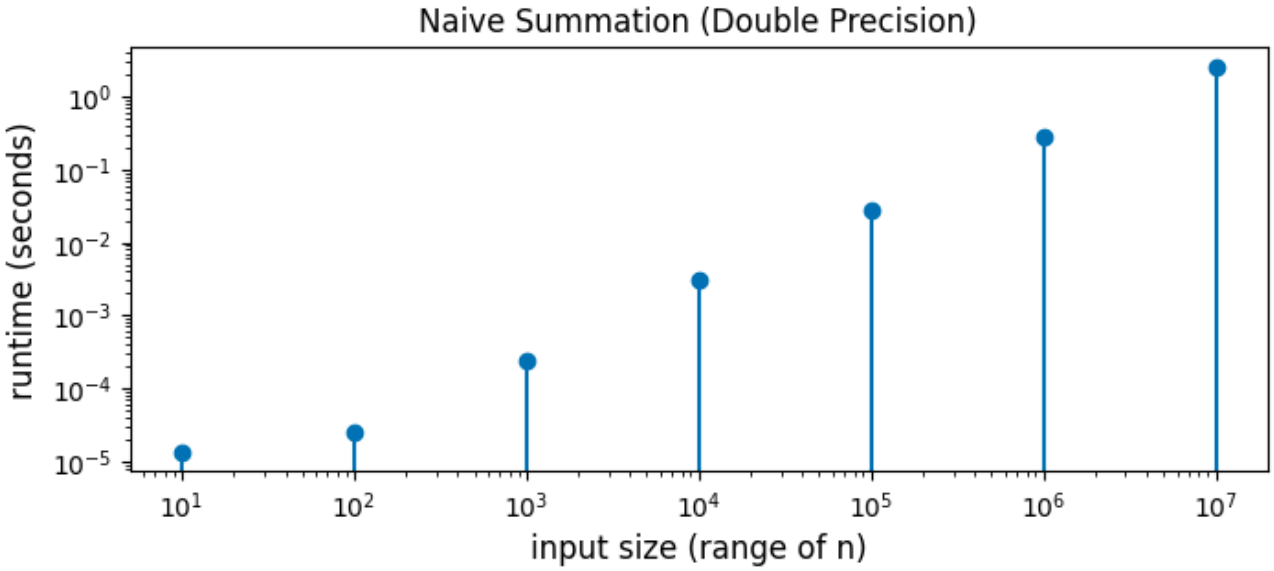


Figure 3: Runtime of Compensated Summation (Double Precision) Algorithm with different input sizes

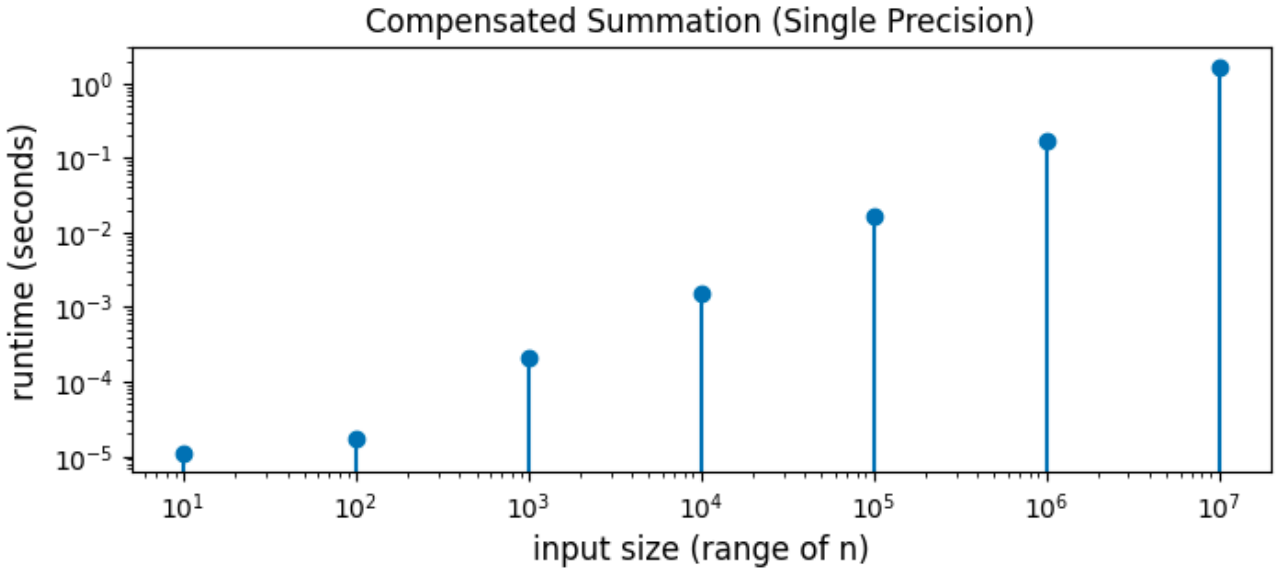


Figure 4: Runtime of Compensated Summation (Single Precision) Algorithm with different input sizes

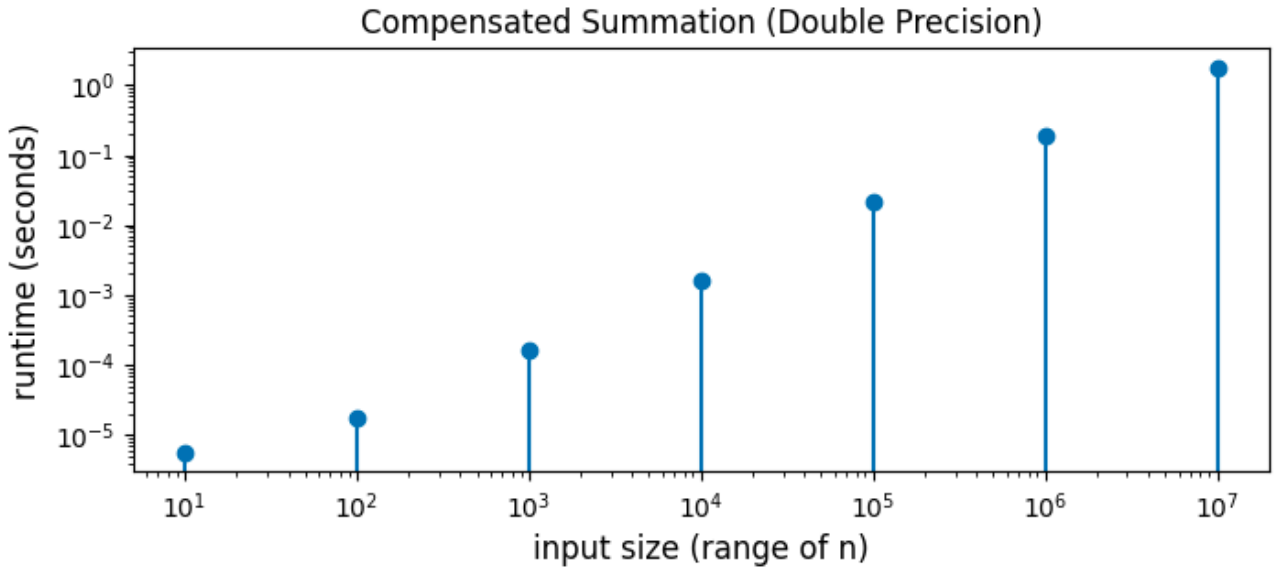


Figure 5: Runtime of Compensated Summation (Double Precision) Algorithm with different input sizes

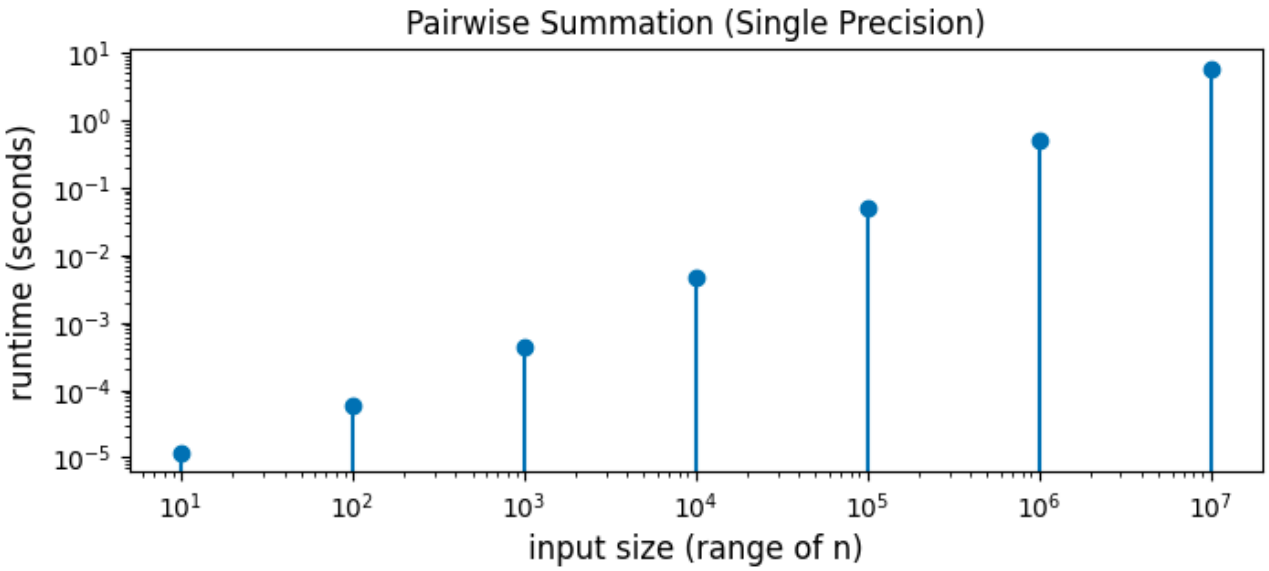


Figure 6: Runtime of Pairwise Summation (Single Precision) Algorithm with different input sizes

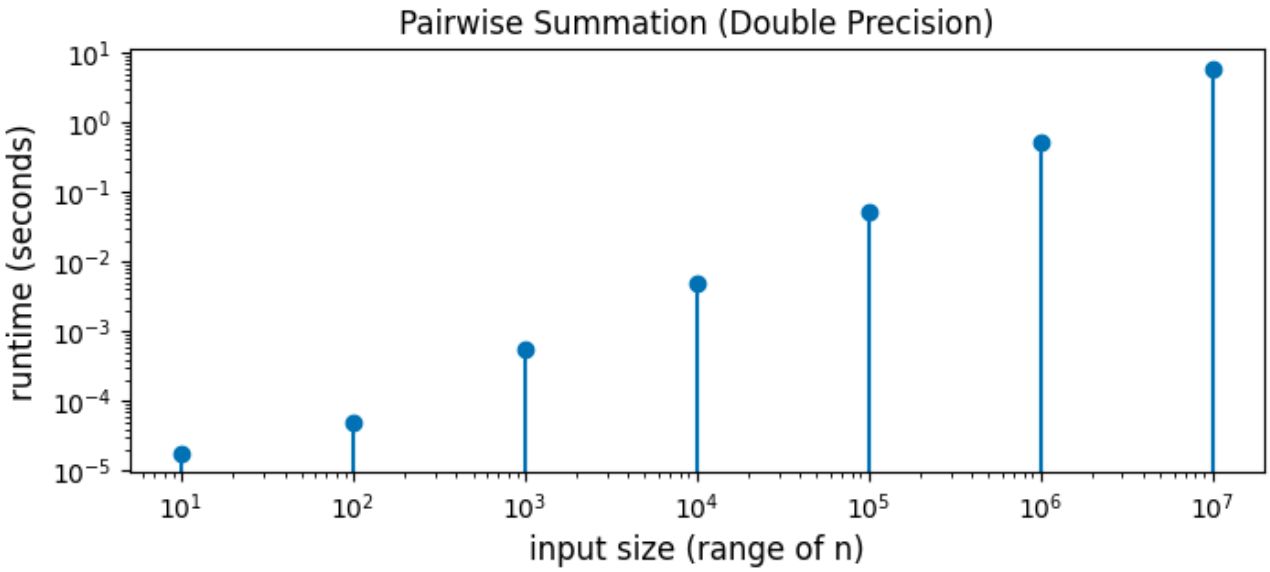


Figure 7: Runtime of Pairwise Summation (Double Precision) Algorithm with different input sizes

e)

**i)** In the second question, my first observation is that numerical value of runtime of Pairwise Summation Algorithms is approximately 5-10 times higher compared to other algorithms. One of the reasons behind this is the fact that Pairwise Summation Algorithm is implemented in a recursive manner. Because lots of function call is made to calculate pairwise summation, runtime is increased substantially. If I implement Pairwise Algorithm in an iterative fashion, I would get less numerical value of runtime.

**ii)** Furthermore, in the second question, we obtained smaller results for all three algorithms by using double precision. The main cause of this is that rounding errors would be smaller as the precision increases. This is because of the following fact:

The worst case for absolute error is the case in which $x$ sits at the middle of the interval between two consecutive machine numbers $x_1$ and $x_2$

$$\text{Absolute Error} = |x - rd(x)|$$
$$\text{A.E} \leq \frac{1}{2}|x_1 - x_2|$$

If we use double precision, difference between consecutive machine numbers gets smaller. As a result, expression $|x_1 - x_2|$ given above gets smaller and error would be smaller.

**iii)** In the first question, my comment is about reducing errors. Observe that if we calculate $f(n)$ algebraically, it would be equal to 0 for all values of $n$ regardless of its size. By taking this example into account, we should look for a different formulation of the problem that are mathematically equivalent but numerically not in order to eliminate errors.