CENG 371 - Scientific Computing
Fall 2022
Homework 1

Sezgin, Mustafa
e2380863@ceng.metu.edu.tr

October 27, 2022

Question 1

a)

The graph n vs $g(n)$ is shown in Figure 1.

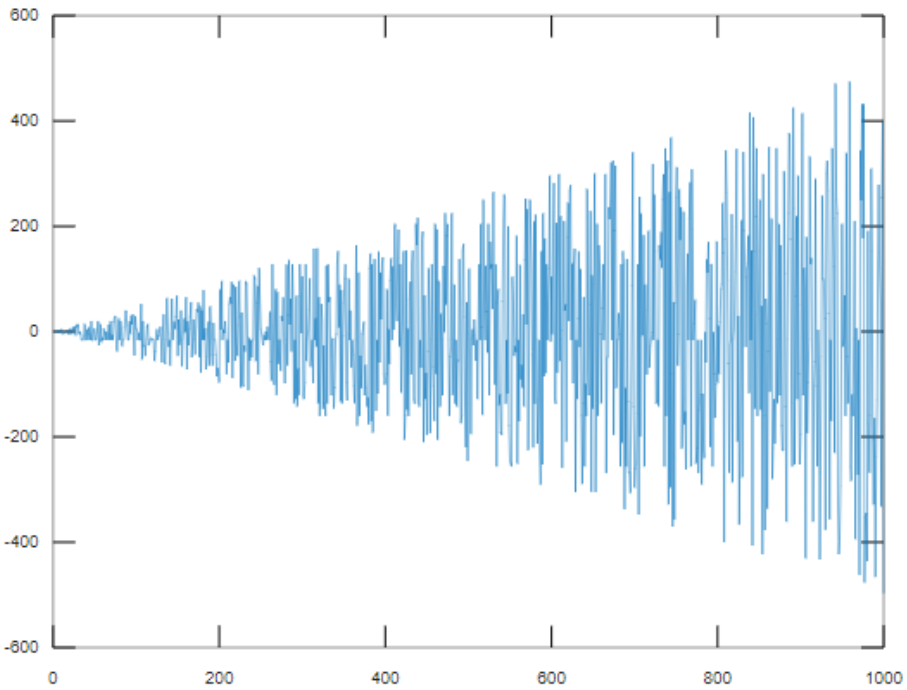

Figure 1: n vs $g(n)$

b)

$g(n) = 0$ if $n \in \{1, 2, 4, 8, 16, 32, 64, 128, 256, 512\}$

c)

$\frac{1}{n}$ is a machine number if n is a power of 2 and $n \leq \frac{1}{\epsilon}$. Therefore, floating point representation of $\frac{1}{n}$ for such a number does not have a rounding error. Majority of $n \in [1, 1000]$ are not powers of 2, and $\frac{1}{n}$ loses its bits since it has infinite bits in mantissa. This causes rounding errors and $g(n)$ becomes non-zero.

d)

Calculated solution of $f(n)$ is as follows.

$$\begin{aligned}
fl(f(n)) &= \left(n\left(\frac{(n+1)(1+\delta_1)}{n}(1+\delta_2) - 1\right)(1+\delta_3)(1+\delta_4) - 1\right)(1+\delta_5) \\
&= (((n+1)(1+\delta_1)(1+\delta_2) - n)(1+\delta_3)(1+\delta_4) - 1)(1+\delta_5) \\
&= ((n+1)(1+\delta_1)(1+\delta_2) - n)(1+\delta_3)(1+\delta_4)(1+\delta_5) - (1+\delta_5) \\
&= (n+1)(1+\delta_1)(1+\delta_2)(1+\delta_3)(1+\delta_4)(1+\delta_5) - n(1+\delta_3)(1+\delta_4)(1+\delta_5) - (1+\delta_5) \\
&= (n+1)(1+\delta_1') - n(1+\delta_2') - (1+\delta_3') \\
&= n(\delta_1' - \delta_2') + \delta_1' - \delta_3' \\
&= n\delta_1'' + \delta_2''
\end{aligned}$$

where $|\delta_{1,2,3,4,5}| \leq \epsilon$, $|\delta_1'| \leq 5\epsilon$, $|\delta_2'| \leq 3\epsilon$, $|\delta_3'| \leq \epsilon$, $|\delta_1''| \leq 8\epsilon$, and $|\delta_2''| \leq 6\epsilon$.

Calculated solution of $g(n)$ is as follows.

$$fl(g(n)) = \frac{n\delta_1'' + \delta_2''}{\epsilon}$$

where

$$|fl(g(n))| = \left|\frac{n\delta_1'' + \delta_2''}{\epsilon}\right| \leq \frac{|n||\delta_1''| + |\delta_2''|}{\epsilon} \leq 8|n| + 6$$

This depends on n and grows in size as n gets larger.

Question 2

a)

$$\begin{aligned}
\sum_{n=1}^{10^6} \text{nums}[n] &= \sum_{n=1}^{10^6} \left(1 + (10^6 + 1 - n) \cdot 10^{-8}\right) \\
&= \sum_{i=1}^{10^6} 1 + 10^{-8} \left(\sum_{n=1}^{10^6} (10^6 + 1) - \sum_{n=1}^{10^6} n \right) \\
&= 10^6 + 10^{-8} \left(10^6(10^6 + 1) - \frac{10^6(10^6 + 1)}{2} \right) \\
&= 10^6 + 10^{-8} \frac{10^6(10^6 + 1)}{2} \\
&= 10^6 + \frac{10^4 + 10^{-2}}{2} \\
&= 1005000.005
\end{aligned}$$

b)

Pairwise summation is a divide and conquer algorithm that divides the input array into two halves, sums them recursively, and returns summation of two summations.

c)

The outputs of the summation algorithms are as follows.

Single precision:

- naive summation = 1002466.687500
- compensated summation = 1005000.000000
- pairwise summation = 1005000.000000

Double precision:

- naive summation = 1005000.005000
- compensated summation = 1005000.005000
- pairwise summation = 1005000.005000

d)

Single precision:

- naive summation: absolute error is 2533.3175, relative error is 0.2521%, and run time is 2.17 seconds.

- compensated summation: absolute error is 0.005, relative error is nearly 0%, and run time is 4.28 seconds.
- pairwise summation: absolute error and relative error are the same as compensated summation, and run time is 25.42 seconds.

Double precision:

- all summation types: absolute error and relative error are 0.
- naive summation: run time is 2.20 seconds.
- compensated summation: run time is 4.36 seconds.
- pairwise summation: run time is 25.23 seconds.

e)

All elements of `nums` are positive since

$$1 \leq n \leq 10^6$$

$$-10^6 \leq -n \leq -1$$

$$1 \leq 10^6 + 1 - n \leq 10^6,$$

which implies

$$\frac{\sum_{i=1}^{n} |x_i|}{|\sum_{i=1}^{n} x_i|} = 1.$$

In our case $n = 10^6$ and the exact sum is 1005000.005.

1. For naive summation

$$\text{rel. error} \leq (n-1)\epsilon$$

The upper bound of relative error is $(10^6 - 1)\epsilon$, which is roughly

- 0.119 in single precision,
- 2.22×10^{-10} in double precision.

The upper bound of absolute error is roughly

- 119805 in single precision,
- 0.000223 in double precision.

These agree with the calculated sums I obtained.
Additionally, this is a linear algorithm, and the time complexity is $O(n)$.

2. For compensated summation

$$\text{rel. error} \leq \epsilon + O(n\epsilon^2)$$

The upper bound of relative error is $\epsilon + 10^6\epsilon^2 c$ for some constant c, which is roughly

- 1.19×10^{-7} in single precision,
- 2.22×10^{-16} in double precision.

The upper bound of absolute error is roughly

- 0.12 in single precision,
- 2.23×10^{-10} in double precision.

These boundaries may increase depending on the value of c. However, these are much smaller than those of naive summation algorithm.
Additionally, this is also a linear algorithm, and the time complexity is $O(n)$ similar to naive summation algorithm, but has a greater run time due to the higher number of operations in each iteration.

3. For pairwise summation

$$\text{rel. error} \leq \frac{\epsilon \log_2 n}{1 - \epsilon \log_2 n}$$

The upper bound of relative error is $\frac{\epsilon \log_2 10^6}{1 - \epsilon \log_2 10^6}$, which is roughly

- 2.38×10^{-6} in single precision,
- 4.43×10^{-15} in double precision.

The upper bound of absolute error is roughly

- 2.39 in single precision,
- 4.45×10^{-9} in double precision.

These are slightly greater than those of compensated summation algorithm, but still much smaller than those of naive summation algorithm.
Additionally, this is also a linear algorithm since it has to visit each element, and the time complexity is $O(n)$ similar to the previous algorithms. This is a divide-and-conquer algorithm with less accumulation of rounding errors. I implemented it as a recursive function, and this is why it has a higher run time compared to others.

In summary, the naive summation algorithm is the worst of all and produces an output with the highest error margin, but it is the fastest due to its simplicity. The compensated summation algorithm is the best of all with a negligible error margin, and it is the second fastest one. The pairwise summation algorithm also produces an output with a low error margin close to that of compensated summation algorithm, but it is the slowest in my implementation due to recursive structure. Using double precision significantly decreases errors. In our case, using double precision produces results with no errors and has almost no cost in terms of execution time.