Programming Solutions

Lu Vy

September 28, 2020

1 Summary

We compare three addition algorithms and demonstrate their theoretical error bounds. Using various sums, we verify that these sums are correct, and that they are stable, as assured by the mathematics. We also verify that addition is a well conditioned problem.

2 Problem Statement

We numerically verify (1) the conditioning upper bound on addition problems, and (2) the stability of three addition algorithms. The three algorithms in question are the recursive cumulative sum in single precision IEEE arithmetic, the binary fan-in tree, and the cumulative sum again in double precision (algorithms 1, 2, and 3 hereafter). Before the stability analysis, we show that the three algorithms produce correct results (as correct as they can be, given their limitations). The shortcomings of $\mathcal{F}(\beta, t, L, U)$ relative to $\mathbb{R}$ are reproduced analogously using the shortcomings of single precision floats relative to double precision floats.

3 Mathematics

We pose the problem in the following way: find $x \in \mathbb{R}$ that satisfies

$$F(x, \mathbf{d}) = (d_1 + d_2 + \cdots + d_n) - x = 0$$

given fixed $\mathbf{d} = (d_1, \ldots, d_n)^T \in \mathbb{R}^n$ [6]. Under the constraint $F(x, \mathbf{d}) = 0$, we may view x as a function of $\mathbf{d}$. This $x(\mathbf{d})$ is well conditioned because

$$\begin{aligned}\left|\frac{x(\mathbf{d}+\boldsymbol{\Delta}\mathbf{d}) - x(\mathbf{d})}{x(\mathbf{d})}\right| &= \left|\frac{\sum_i (d_i + \Delta d_i) - \sum_i (d_i)}{\sum_i (d_i)}\right| = \left|\frac{\sum_i \Delta d_i}{\sum_i d_i}\right| \\ &\leq \frac{\sum_i |\Delta d_i|}{\left|\sum_i d_i\right|} = \frac{\sum_i |d_i|}{\left|\sum_i d_i\right|}\frac{\sum_i |\Delta d_i|}{\sum_i |d_i|} = \kappa_r \frac{\|\boldsymbol{\Delta}\mathbf{d}\|_1}{\|\mathbf{d}\|_1}.\end{aligned}$$

One may bound the absolute error $\left|x(\mathbf{d}+\boldsymbol{\Delta}\mathbf{d}) - x(\mathbf{d})\right| < \kappa_a \|\boldsymbol{\Delta}\mathbf{d}\|_1$ in a similar way [2]. Here, $\kappa_r = \frac{\sum_i |d_i|}{\left|\sum_i d_i\right|}$ and $\kappa_a = \sum_i |d_i|$ denote the relative and absolute *condition numbers*.

The three algorithms merely approximate $x(\mathbf{d})$. In this investigation, we consider two sources of error that prevent us from attaining $x(\mathbf{d})$ exactly [1]:

1. finite precision representation of real numbers. These not only perturb the data $d_1, \ldots, d_n$, but also restrict the range of values that x may take.

2. accumulation of errors due to floating point arithmetic.

If $\hat{x}$ is an approximation of x, then two quantities of interest are $|\hat{x} - x|$ and $\left|\frac{\hat{x}-x}{x}\right|$. We call the former *absolute forward error* and the latter *relative forward error*. We derive upper bounds for each.

In fact, we show that all three algorithms are backwards stable. That is, the errors are no greater than

$$|\hat{x} - x| \leq \kappa_a p(n) u \qquad \text{and} \qquad \left|\frac{\hat{x} - x}{x}\right| \leq \kappa_r p(n) u,$$

for some $p(n)$, and for each approximate solution $\hat{x}$, there exists a vector $\Delta\mathbf{d}$ (small relative to $\mathbf{d}$) that satisfies $F(\hat{x}, \mathbf{d} + \Delta\mathbf{d}) = 0$. Here, $u = 2^{-24}$ denotes *unit round off*.

3.1 Algorithm 1

It has been shown [2] that if $\xi = (\xi_1, \ldots, \xi_n)^T$ is a vector of floating point numbers, then the errors have the approximate bounds

$$\begin{aligned} \left|\hat{x}_1(\xi) - x(\xi)\right| &\lesssim \kappa_a n u \\ \left|\frac{\hat{x}_1(\xi) - x(\xi)}{x(\xi)}\right| &\lesssim \kappa_r n u. \end{aligned}$$

If instead we require the data $\mathbf{d} = (d_1, \ldots, d_n)^T$ to be real valued, then this does not change the bounds (provided that they are within range, i.e. not too large). Indeed for each $d \in \mathbb{R}$ (within range), there exists $\xi \in \mathcal{F}(\beta, t, L, U)$ and $\delta \in (-u, u)$ for which $d = \xi(1 + \delta)$. Thus substitution of $\mathbf{d}$ into ξ does not change the order of $p(n) \in \mathcal{O}(n)$.

To show that $\hat{x}_1(\mathbf{d})$ is backwards stable, recall [3] that we may express $\hat{x}_1(\mathbf{d})$ in the form

$$\hat{x}_1(\mathbf{d}) = \sum_{i=1}^{n} d_i(1 + \eta_i),$$

where $|\eta_i| \lesssim nu$. Hence, $\hat{x}_1(\mathbf{d})$ actually solves the problem with data

$$\mathbf{d} + \mathbf{\Delta d} = \left(d_1(1 + \eta_1), \ldots, d_n(1 + \eta_n)\right)^T.$$

Our backward error (using the L_1 norm) is

$$\|\Delta\mathbf{d}\|_1 = \left\|(d_1\eta_1, \ldots, d_n\eta_n)^T\right\|_1 = \sum_{i=1}^{n} |d_i\eta_i| \leq \|\mathbf{d}\|_1 \|\eta\|_1$$

which is small relative to $\|\mathbf{d}\|_1$ because each η_i is small.

3.2 Algorithm 2

Assume that $n = 2^k$ for some $k \in \mathbb{N}$. If this is not the case, then take $\tilde{n} = 2^{\lceil \log_2 n \rceil}$ and let $d_{n+1} = \cdots = d_{\tilde{n}} = 0$. For floating point ξ, we claim that

$$\begin{aligned} \left|\hat{x}_2(\xi) - x(\xi)\right| &\lesssim \kappa_a (\log_2 n) u \\ \left|\frac{\hat{x}_2(\xi) - x(\xi)}{x(\xi)}\right| &\lesssim \kappa_r (\log_2 n) u. \end{aligned}$$

That is, $p(n) \in \mathcal{O}(\log_2 n)$ for the binary fan-in tree. For proof, see 1.3.c in the written homework solutions. As was the case before, real valued data does not change the order of $p(n)$; we may take $p(n) = 1 + \log_2 n$ if we wish to be conservative.

Just like algorithm 1, $\hat{x}_2(\mathbf{d})$ solves the problem with data $\left(d_1(1+\eta_1), \ldots, d_n(1+\eta_n)\right)^T$. We attain a similar expression for the backwards error (see homework problem 1.3.b.), with the η_i being different, but still small ($|\eta_1| \lesssim (1 + \log_2 n)\, u$).

3.3 Algorithm 3

Unit round off for double precision floats $u_{dp} = 2^{-53}$ is 2^{29} times smaller than that of single precision $u_{sp} = 2^{-24}$. We therefore neglect the accumulation of rounding error due to double precision addition. This may be problematic when $n \gg 2^{29}$, but we do not consider such sums here here. The algorithm in question is

$$\begin{aligned}
\sigma_{1:n} &= \mathrm{fl}\left(\mathrm{fl}(d_1) + \mathrm{fl}(d_2) + \cdots + \mathrm{fl}(d_n)\right) \\
&= \mathrm{fl}\left(d_1(1+\delta_1) + d_2(1+\delta_2) + \cdots + d_n(1+\delta_n)\right) \\
&= \left(d_1(1+\delta_1) + d_2(1+\delta_2) + \cdots + d_n(1+\delta_n)\right)(1+\delta_{n+1}) \\
&= \sum_{i=1}^{n} d_i(1+\delta_i)(1+\delta_{n+1}) \\
&= \sum_{i=1}^{n} d_i(1+\eta_i),
\end{aligned}$$

where

$$(1+\eta_i) = (1+\delta_i)(1+\delta_{n+1}) \implies |\eta_i| \lesssim 2u.$$

It follows that

$$\begin{aligned}
|\hat{x}_2 - x| &\lesssim \sum_{i=1}^{n} |d_i|\, 2u = 2\kappa_a u \\
\left|\frac{\hat{x}_2 - x}{x}\right| &\lesssim \frac{\sum_{i=1}^{n} |d_i|}{\left|\sum_{i=1}^{n} d_i\right|} 2u = 2\kappa_r u.
\end{aligned}$$

In particular, $p(n) \in \mathcal{O}(1)$.

As before, $\hat{x}_3(\mathbf{d})$ solves the problem with data $\mathbf{d} = \left(d_1(1+\eta_1), \ldots, d_n(1+\eta_n)\right)^T$, and the backward error is small: $|\eta_i| \lesssim 2u$.

4 Algorithm and Implementation

Each algorithm takes in an array of length n and adds the first $k \leq n$ terms.

It is straightforward to program algorithm 1. All that is needed is an accumulator and a for-loop. By float, that is meant single precision floating point digit. For algorithm 2, we add in pairs until we reduce the array to a single number, which is the value returned. If there is an odd term, it is added to 0. Algorithm 3 is the same as algorithm 1, except the

accumulator is double precision, and every float added to it must also first be converted to double precision.

For the purposes of analysis, we need an algorithm that takes in double precision floats and outputs a double precision float. This is to represent the "exact" sum when one is not known in closed form. Because $u_{dp} \ll u_{sp}$, it does not matter which form of addition we use, but we opt for the binary fan-in tree because its error is $\mathcal{O}\left(\log_n nu_{dp}\right)$.

Algorithm 1: Single precision accumulation

Data: array of single precision floats $\xi_1, \ldots, \xi_n$, integer $k \leq n$ of terms to be added
Result: single precision float S
initialization;
float $S \leftarrow 0$;
for $i = 1$ **to** k **do**
  $S \leftarrow S + \xi_i$;
end
return S;

Algorithm 2: Binary fan-in tree

Data: array of single precision floats $\xi_1, \ldots, \xi_n$, integer $k \leq n$ of terms to be added
Result: single precision float S
initialization;
duplicate array $\mathbf{v} \leftarrow [\xi_n, \ldots, \xi_n]$;
while $k > 1$ **do**
  for $i = 0$ **to** $\lfloor \frac{k}{2} \rfloor$ **do**
    $\mathbf{v} \leftarrow \mathbf{v}[2i] + \mathbf{v}[2i+1]$;
  end
  if k *is even* **then**
    $k \leftarrow \frac{k}{2}$;
  else
    $k \leftarrow \lceil \frac{k}{2} \rceil$;
  end
end
return $\mathbf{v}[0]$;

Algorithm 3: Double precision accumulation

Data: array of single precision floats $\xi_1, \ldots, \xi_n$, integer $k \leq n$ of terms to be added
Result: single precision float S
initialization;
double $S \leftarrow 0$;
for $i = 1$ **to** k **do**
  double $d \leftarrow \xi_i$;
  $S \leftarrow S + d$;
end
float *out* $\leftarrow S$;
return *out*;

5 Experimental Design and Results

The three tasks at hand are: correctness, conditioning, and stability. We conduct two tests for each, for a total of six.

5.1 Correctness

Are the three algorithms accurate? To answer this, we consider Taylor series. Taylor's remainder theorem gives us a theoretical upper bound on their convergence rates: for each f with at least n derivatives in a neighborhood of 0, we have that

$$f(x) = f(0) + f'(0)\,x + \cdots + \frac{f^{(n-1)}(0)}{(n-1)!}x^{n-1} + \frac{f^{(n)}(c)}{n!}x^n$$

for some $c \in (0, x)$ [5]. Thus error of the $(n-1)^{th}$ term sum is of the order $\mathcal{O}\left(\frac{1}{n!}\right)$.

5.1.1 Test 1

Consider the series $\sum_{k=0}^{\infty} \frac{1}{k!} \to e$. We consider the first 15 terms, for anything after $k = 10$ is less than round off error: $\frac{1}{10!} > u > \frac{1}{11!}$. The mathematics suggests that

$$\text{err}_n = \left| e - \sum_{k=0}^{n-1} \frac{1}{k!} \right| = \frac{e^c}{n!} \leq \frac{e}{n!}, \qquad c \in (0, 1)\,.$$

An upper bound for the relative error is therefore $\text{err}_{\text{rel}} \leq \frac{1}{n!}$. Our experiment will verify whether this upper bound holds for $n \in \{2, \ldots, 15\}$. In accordance with lecture notes 4 [3], the cumulative sums will go from smallest to largest.

5.1.2 Test 2

Consider the series $\sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)!} \to \sin(1)$. We consider the first 10 terms, for the same reason specified in test 1. The mathematics suggests that

$$\text{err}_n = \left| \sin(1) - \sum_{k=0}^{n-1} \frac{(-1)^k}{(2k+1)!} \right| = \frac{\sin(c)}{2n!} \leq \frac{1}{2n!}.$$

5.2 Conditioning

How much will perturbations in the data affect the solution? The mathematics says that summation is a well conditioned problem, so any change in the data should result in a proportionate change in the solution. To verify this, we generate fix n and generate a noise vector $\mathbf{\Delta d}$ with IID random components $u_i \sim \mathcal{U}(-M, M)$. The mathematics predicts

$$\left| \frac{x(\mathbf{d} + \mathbf{\Delta d}) - x(\mathbf{d})}{x(\mathbf{d})} \right| \leq \kappa_r \frac{\|\mathbf{\Delta d}\|_1}{\|\mathbf{d}\|_1} = \frac{\sum_{i=1}^{n} |u_i|}{\sum_{i=1}^{n} |d_i|} \leq \frac{nM}{\sum_{i=1}^{n} |d_i|},$$

and we verify this with $M \in \left\{2^{-5}, 2^{-4}, \ldots, 2^{5}\right\}$.

We seek sums for which the solution can be stored exactly in the computer. For an algorithm, we choose the "exact" sum specified in section 4. Since we are evaluating the conditioning of the problem, as opposed to the stability of an algorithm, we do not use algorithms 1, 2, or 3.

5.2.1 Test 3

Consider the sum $\sum_{n=1}^{100} n = 5050$. To each n, we add $u_i \sim \mathcal{U}(-M, M)$ and evaluate whether the new sum, $x(\mathbf{d} + \mathbf{\Delta d})$ satisfies

$$\left| \frac{x(\mathbf{d} + \mathbf{\Delta d}) - 5050}{5050} \right| \leq \frac{M}{101}.$$

5.2.2 Test 4

Consider the sum $\sum_{p=-5}^{5} 2^p = 2^6 - 2^{-5}$. This is a number that can be exactly represented in a computer. We perturb each term and see if the new sum still satisfies

$$\left| \frac{x(\mathbf{d} + \mathbf{\Delta d}) - \left(2^6 - 2^{-5}\right)}{2^6 - 2^{-5}} \right| \leq \frac{11}{2^6 - 2^{-5}} M.$$

5.3 Stability

How sensitive is the computed solution to round off error? We have shown in section 3 that all three algorithms are backwards stable, so the computed solution should never be too far from the exact solution. We shall compute each sum in single precision, and compare them to the "exact" sum in double precision. We expect to see the upper bound $\left|\frac{\hat{x}-x}{x}\right| \leq \kappa_r p(n) u$. Recall from section 3 that $p(n)$ is approximately n, $1 + \log_2 n$, and 2 for algorithms 1, 2, and 3 respectively.

5.3.1 Test 5

To demonstrate accumulated round off error, we seek sums whose terms cannot be represented exactly in a computer. Consider the sum $\sum_{i=1}^{n} i \sin\left(\left(\frac{i}{2} + \frac{1}{i}\right)\pi\right)$ for $n \in \left\{2, 2^2, \ldots, 2^{15}\right\}$.

5.3.2 Test 6

Consider the sum $\sum_{i=1}^{n} i \sin\left(\frac{i}{2} + \frac{1}{i}\right)$ for $n \in \left\{2, 2^2, \ldots, 2^{15}\right\}$.

5.4 Results

We present our findings through graphs, for they most clearly illustrate our findings.

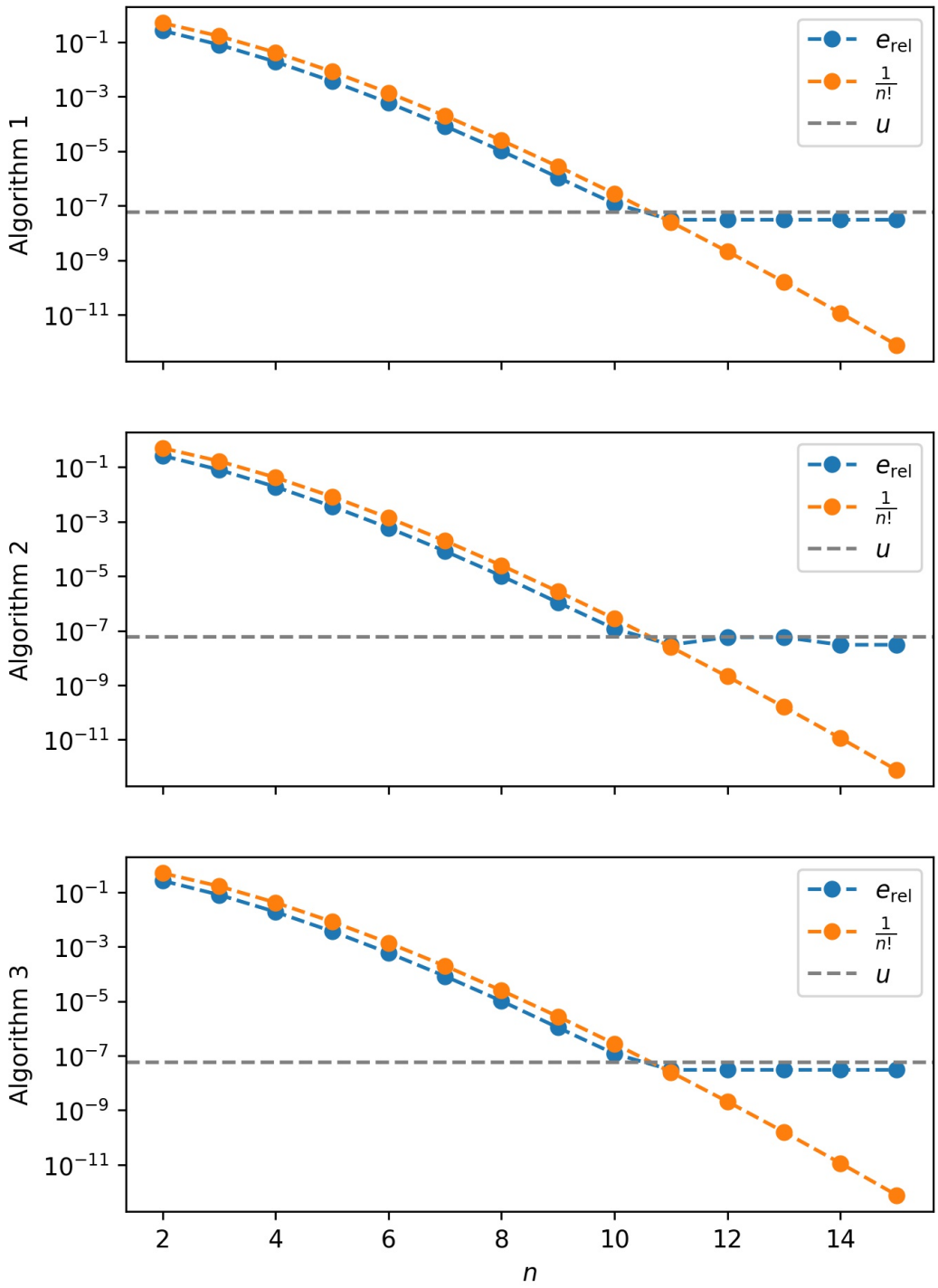

Figure 1: Results of test 1. Relative errors of partial Taylor series against upper bounds.

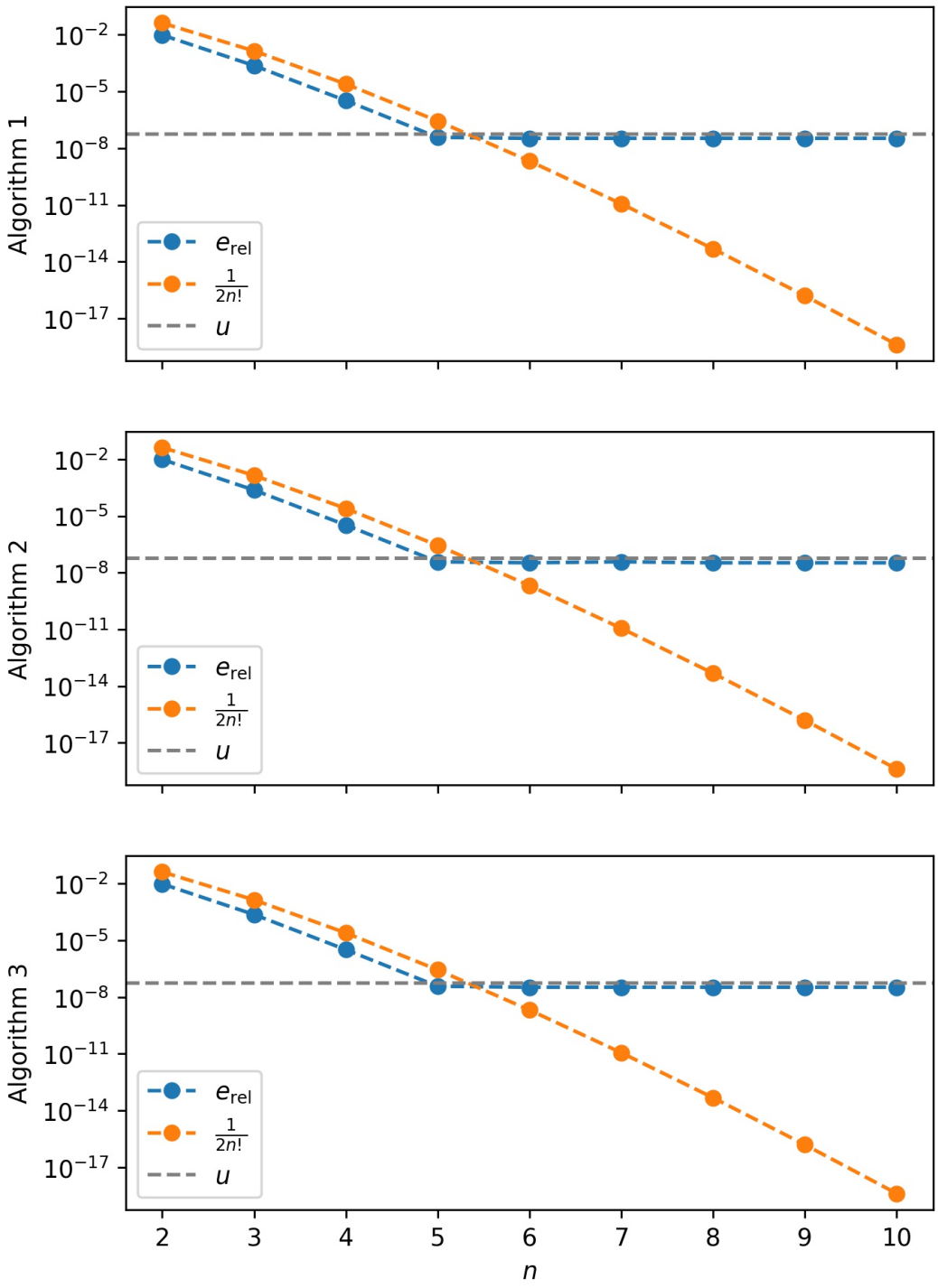

Figure 2: Results of test 2. Similar to test 1.

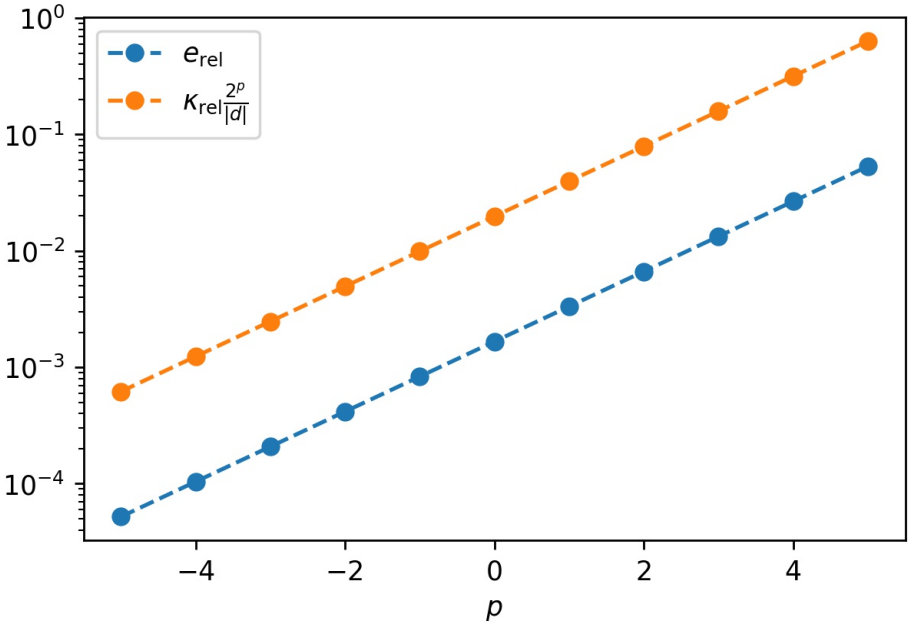

Figure 3: Results of test 3. Relative changes in solution against "scale" of perturbation.

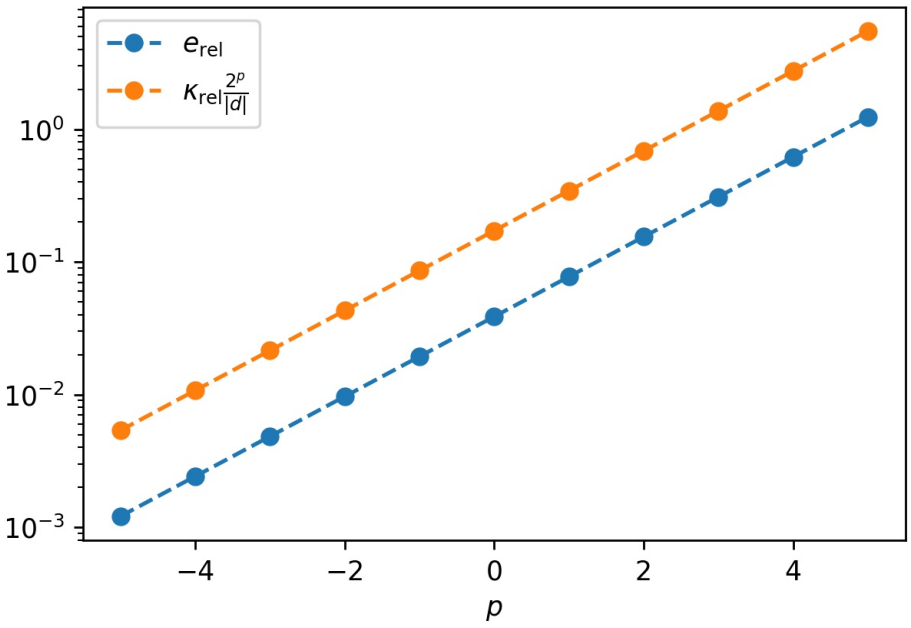

Figure 4: Results of test 4. Similar to test 3.

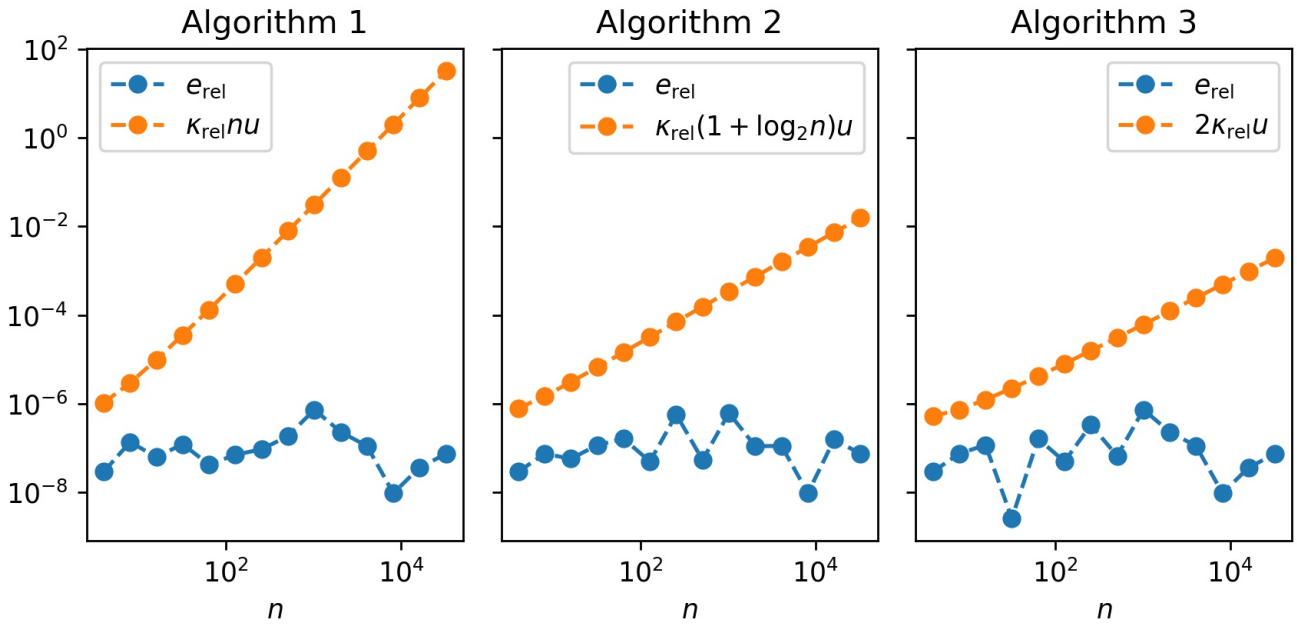

Figure 5: Results of test 5. Accumulated rounding error versus upper bounds.

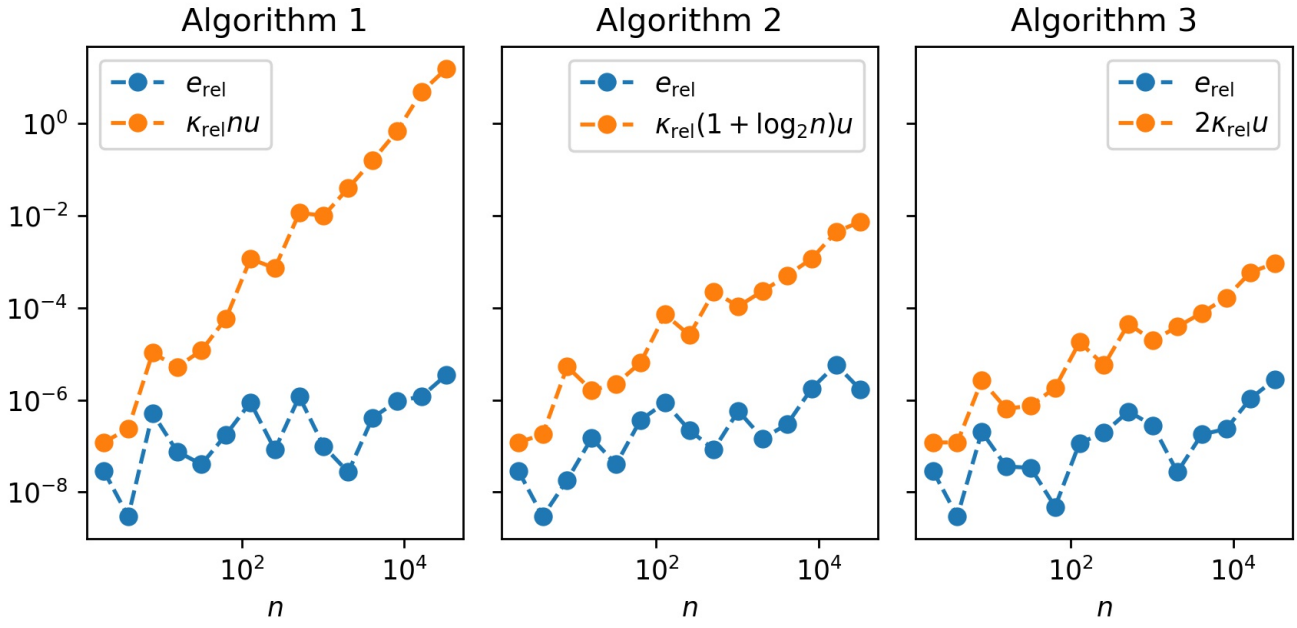

Figure 6: Results of test 6. Similar to test 5.

6 Conclusions

Our algorithms are accurate. Figures 1 and 2 compare the relative error on each term of the sum against the upper bound given by Taylor's remainder theorem. The error consistently falls below the bound until they drop below unit round off u, at which point the sum becomes fixed, for new terms are too small to change it. In actuality, we have the more accurate bound $\text{err}_{\text{rel}} \leq \max\left\{C\frac{1}{n!}, u\right\}$.

Summation as a computational problem enjoys well conditioning. Bounded changes in the summands result in bounded changes in the solution. The mathematics of section 3 shows this, and the results of tests 3 and 4 agree with this. In test 3, we added random, but bounded, noise to the summands $1, 2, \ldots, 100$ and showed that the solution 5050 does not change (in relative value) by more than $\kappa_r \frac{M}{\|\mathbf{d}\|}$. This is depicted in figure 3, which plots the change in the solution against the "scale" of perturbation, given by p in $M = 2^p$. For small ($p = -5$) and large ($p = 5$) perturbations alike, the upper bound we have proves to be reliable. Figure 4 argues the same conclusion, except with the summands $2^{-5}, 2^{-4}, \ldots, 2^5$.

The three algorithms we propose are all backwards stable. We have shown the mathematics in section 3, yet tests 5 and 6 only verify that our algorithms are weakly stable. Figures 5 and 6 demonstrate that the upper bounds κ_{rel} are accurate for each of the three algorithms. In fact, they are quite loose, for the relative error grows far slower than the bounds even for large n. This implies that the computed solution is always near the exact solution. That the computed solution solves a "nearby" problem has only been shown mathematically.

7 Program Files

There are three computational units, all written in `C++`. These are entitled `main.cpp`, `tests.cpp`, and `functions.cpp`. Once in the `Code` directory, each `cpp` file may be compiled using `gcc` with the terminal commands

```
g++ main.cpp -c
g++ tests.cpp -c
g++ functions.cpp -c
```

Then, the resulting object files can be linked using the command

```
g++ main.o tests.o functions.o -o program
```

which produces an executable file named `program`. The executable can be ran with the line

```
program.exe
```

The code to produce the graphics is written in `Python` and contained in the file `plot.py`. The current environment needs the libraries `numpy` and `matplotlib` to run the code

```
python plot.py
```

References

[1] Gallivan, K. A. *Set 2: Representation, Conditioning and Error*, Lecture Notes. Foundations of Computational Math 1, Florida State University.

[2] Gallivan, K. A. *Set 3: Finite Precision Arithmetic and Numerical Stability*, Lecture Notes. Foundations of Computational Math 1, Florida State University.

[3] Gallivan, K. A. *Set 4: Numerical Stability Examples*, Lecture Notes. Foundations of Computational Math 1, Florida State University.

[4] J. D. Hunter, "Matplotlib: A 2D Graphics Environment", Computing in Science & Engineering, vol. 9, no. 3, pp. 90-95, 2007.

[5] Kincaid, D. & Cheney, W. (2002) *Numerical Analysis: Mathematics of Scientific Computing*. Brooks/Cole.

[6] Quarteroni, A., Sacco, R., & Saleri, R. (2000). *Numerical Mathematics*. Springer.