FCM Project 1

Jonathan Engle

October, 4th 2023

1 Executive Summary

In this report, we observe the floating point system as well as the accumulation of true error throughout this project. Analytic implementation further verifies that the error brought by our rounding methods can be bounded. Further analysis is done to observe the nature of translating rounded numbers to floating point numbers as well as floating point arithmetic. Results are shown via graphs denoting where our errors are with respect to our bounds for correctness and accumulation. Results are obtained by generating random vectors where the smallest normalized floating point number is $10^{-8+4-1} = 10^{-5}$ and the largest floating point number is $10^{0+4-1} = 10^3$(i.e $9,999$.)

2 Statement of the Problem

Analyzing machine precision and error analysis is important when dealing with numerical methods and machines. This analysis allows us to understand how machines store numbers with some small perturbations to our actual real numbers which we wish to represent. This project is designed to take the theoretical results found in class and verify them using numerical methods. The theoretical results which we wish to verify are that:

$$|\frac{fl(x) - x}{x}| = |\delta| \leq u_n = \frac{(\beta^{1-t})}{2} = \frac{10^{1-4}}{2}$$

Where $fl(x)$ is our floating point approximation (with bounds applied) of a randomly generated real number. Please note, that our floating point approximation contains less accuracy than the standard machine precision. An additional property that we verify is the correctness for floating point addition. Where, $\forall f_1, f_2 \in \mathcal{F}, f_1 \boxplus f_2 = (f_1 + f_2)(1 + \delta), |\delta| < u_n$. With this we identify that our we must have the relative error bound. The final result we look to show is that our theoretical result of a relative condition number bounds our relative error between two exact sums of our floating point vector and our exact vector. We analyze these properties with a large sampling of random real numbers which are then implemented through our algorithms listed bellow.

3 Description of the Algorithms and Implementation

The rounding of real numbers and translation to floating point approximations are done directly in **C++**, on where there are three major algorithms. The first algorithm we impose is the rounding algorithm. This is a standard rounding method that takes our real number and rounds to our desired length (bellow). The driving idea behind this method is to take our randomly generated real number, and if our remainder of that exact real number is greater than or equal to our threshold of 5, then we round up. If the remainder is strictly less than 5 we round down. As one can imagine this creates error as we are representing our desired real number as a rounded number where this particular error can accumulate over time. We address these issues and analyze them in the following algorithms.

The second algorithm translates our rounded real number generated in algorithm 1 to a floating point number which lies with in $\mathcal{F}f(4, 10, e_{min}, emax)$. Floating point numbers are how machines store and represent our numbers we wish to work with. Floating point numbers take the form of:

s	e	m

Where s is denoted as our sign of the number we wish to represent (i.e positive or negative), e is denoted as our largest exponent (relative to base β) on which the floating point number can take and finally, m represents the Mantissa which contains the physical digits of our floating point number. A special case on which we must be careful of is the trivial case of 0. 0 lies in all floating point systems and is stored as 0 rather than a large block of zeros or taking e to be a large negative number.

The third algorithm takes two of these floating point numbers that have passed through algorithm 1 and 2 respectively, and performs an operation of our choosing (addition) and yields another floating point number which also lies within $\mathcal{F}$. For our example we take an $x, y \in \mathcal{F}$ and add them together to create another floating point number call it m where $m \in \mathcal{F}$. To perform this algorithm we must be cautious, as we must satisfy that $x + y \in \mathcal{F}$, to do we reduce our range for our randomly generated numbers to be $\frac{\beta^{e_{max}+t-1}}{2}$ for our specific example and specifications we obtain $\frac{10^{0+4-1}}{2} = \frac{10^3}{2}$ for our new range of addition.

Finally, our fourth algorithm computes the floating point version of $s_n = \sum_{k=1}^{n} \xi_k, \forall \xi_i \in \mathcal{F}$ by computing an $\hat{s}_n = s_n + \sum_{k=2}^{n} s_k \delta_k + O(u^2)$. We will further discuss this method and its results in the bellow sections. Please note, that for algorithm 4 we have to tighten our bounds to ensure that each set of randomly generated ξ_i still lie within our $\mathcal{F}$. For algorithms 1 through 4 consider a $\beta = 10, t = 4, e_{\min} = -8,$ and $e_{\max} = 0$. Further analysis with regards to the Experimental Design, Implementation and Results will be discussed in the following section.

4 Description of the Experimental Design and Results

The following section uses the Algorithms as stated above to verify our theoretical results stated in the statement of the problem section. We will refer to the figures and tables at the end of this document as visual aids to our verification of the theory. It is important to note that our **C++** code generates a CSV file which we then convert to an Excel spreadsheet to aid in the observation of all of the **C++** computations and algorithms.

4.1 Correctness test

Bounding our error is crucial when discussing machine error and numerical methods. Refining our bounds on our error is even better. We first analyze our error of our translate routine (Routine 2). For this experiment/ implementation we round 100 randomly generated numbers within the interval of $[-10^3, 10^3]$. We then go to show can be bounded with respect to u_n, i.e.

$$|\frac{fl(x) - x}{x}| = |\delta| \leq u_n = \frac{(\beta^{1-t})}{2} = \frac{10^{1-4}}{2}$$

Figure one gives us a graphical representation where the orange line represents our upper bound of $u_n = \frac{\beta^{1-t}}{2} = \frac{1}{2*10^3}$ and the blue entries are our respective deltas. After a quick glance we can identify that we have satisfied this condition as expected. For this particular case, some statistical results about the deltas are as follows:

Mean	Median	Variance	Standard Deviation	Max
0.000114986	$9.12169 * 10^{-5}$	$9.12169 * 10^{-5}$	$9.88122 * 10^{-5}$	0.000427616

Noting that the maximum value of $\delta_{max} = 0.000427616 < \frac{1}{2*10^3} = 0.0005$ further validates the fact that our bound holds. A further and more interesting observation that can be made, is when looking at the floating point value that corresponds to our delta, we see that our corresponding remainder is relatively close to 5. With that being said, it is relatively close to our rounding threshold meaning that we would expect that particular value to contain the highest error. This is a fascinating discovery as it explains on why we have the 1/2 portion of our u_n A similar argument can be said for a floating point value with a small remainder, inferring that it is relatively close to 0 or 9. Note if we applied our chopping method instead of our round to nearest, where we "chop" any extraneous digits outside of our range, the strongest bound we would be able to show is:

$$|\frac{fl(x) - x}{x}| \leq \beta^{1-t} = \frac{1}{10^3}$$

This further illustrates that our method for choosing to round instead of chop creates a tighter bound. Simulations can be shown by altering our Algorithm 1 to rounding instead of chopping.

Now we will perform a similar test/ verification for our addition case (Algorithm 3). For this particular case of addition we need to refine our bounds on the randomly generated numbers to ensure that the sum is within our floating point maximum. For this simulation we will condense our bounds to $[-4000, 4000]$ (Note that the maximum magnitude of the bound we can consider for this case is $\pm\frac{\beta^{emax+t-1}}{2}$.)

Mean	Median	Variance	Standard Deviation	Max
0.000131085	0.000109684	$1.7911 * 10^{-8}$	0.000133833	0.000499957

As the theory had suggested we found that the maximum delta for this particular case is less than $u_n = \frac{\beta^{-3}}{2}$. This can visually be seen in Figure 2, by observing our orange u_n line is always greater than our generated deltas. A fascinating observation that we can see in this case is that we have almost generated the worst case scenario where $\delta_{max} = 0.000499957$ which is extremely close to our threshold of $u_n = 0.0005$. As before we can see that the two generated floating point integers that we have had been close to our rounding threshold generating this large delta value. We point this particular case out to as it gives confidence to the method for even the worst cases. Please note that to compute these calculations we use IEEE double precision arithmetic that is the standard machine precision. This does introduce error in real world calculations but is vastly more accurate than our five point floating arithmetic making it a suitable operation for our cases. Now we move on to Accumulation and Condition analysis.

4.2 Accumulation

Finally, we observe our fourth and final algorithm at work. To test this accumulation we generate a set of floating point numbers $\xi_{1:n} = (\xi_1 \dots \xi_n), \xi_k \in \mathcal{F}$ and define that sum of a particular ξ_k as $s_n = \sum_{k=1}^{n} \xi_k$. To compare for $\boxplus$ we will compute a similar $\hat{s_n}$ to the corresponding s_n as $\hat{s_n} = s_n + \sum_{k=2}^{n} s_k\delta_k + O(u^2)$. Prior to Numerical simulations we must again verify that these sums lie within our floating point arithmetic. To ensure this we refine our bounds to $[-150, 150]$ with $n = 100$ to ensure that this accumulation will stay within $\mathcal{F}$. After implementing our routine with these conditions we are able to identify from the generated Excel spreadsheet that:

$$|\hat{s_n} - s_n| \approx |\sum_{k=2}^{n} s_k\delta_k| \leq (n-1)\|\xi_{1:n}\|_1 u_n$$

$$|-1070.839 - (-1071.33)| = |0.491| \leq |385.05|$$

Where we represent $)\|\xi_{1:n}\|_1$ as the sum of the absolute entries of ξ_i, $n = 100$ and u_n stays consistent as in the previous problems. With these conditions and multiple simulations we are able to conclude that we are able to bound our error accordingly. Further more we also verify that our bound is tight with respect to the actual error. The reason for this "tightness" is due to the magnitude of our large $\|\xi_i\|_1$ which takes the absolute sum of each of the entries which will always be less than or equal to $|\hat{s_n} - s_n|$.

4.3 Conditioning Analysis

Condition analysis allows us to verify that our methods are well or ill conditioned, this is crucial to ensure that we have a well conditioned problem. Condition Analysis discusses the sensitivity of our methods to perturbations where $\xi_i \in \mathcal{F}$ relative to our actual randomly generated real numbers x_i. If we obtain an ill conditioned problem we must revisit our methodology. The following theory and intuition behind our relative condition state is provided bellow where:

$$c_{rel}^{p_{1:n}} = \frac{|\sum_{i=1}^{n} x_i - \sum_{i=1}^{n} \xi(x_i)|}{|\sum_{i=1}^{n} x|} \frac{\|x_{1:n}\|_1}{\|\xi_{1:n}\|_1}$$
$$k_{rel} \approx \max_{p_{1:n}} c_{rel}^{p_{1:n}}$$

Via multiple trials we obtain that $k_{rel} = 0.034409661$ and for our specific case generated via our algorithm (Excel file data) our relative error between the two exact sums of $\sum_{i=1}^{n} \xi_i$ and $\sum_{i=1}^{n} x_i$ is computed as $\frac{|\sum_{i=1}^{n} x_i - \sum_{i=1}^{n} \xi_i|}{|\sum_{i=1}^{n} x_i|}$ which we compute to be $9.523 * 10^{-5}$. This further verifies that the relative error between the two exact sums is consistent with our approximated condition number i.e our problem is well conditioned. Noting that with any small perturbations of $\xi_i \in \mathcal{F}$ our solution will only move slightly.

5 Conclusion

Understanding how machines store a represented real number and operate on these represented numbers is an important study when analyzing numerical methods. Understanding what errors we introduce with these machines prior to implementing our numerical methods is crucial when moving forward through research. Our use of **C++** and Excel allows us to implement the rounding, Translate, Addition and Accumulation routines. While we use IEEE double precision arithmetic which introduces error itself, it is vastly more accurate than our 5-digit decimal arithmetic we analyze. Throughout this project our objective was two show 3 major results and discoveries of numerical methods these results being Correctness, Accumulation and Condition Analysis. We are able to generate theoretical bounds for all of our relative tests to further verify the analysis we introduced. Showing that these methods are consistent and more importantly bounded yields confidence in machine arithmetic and allows us to further proceed in more complex numerical methods by understanding how the basic storing and operations perform.

6 Tables and Figures

Figure 1: Delta relative to our calculated u_n

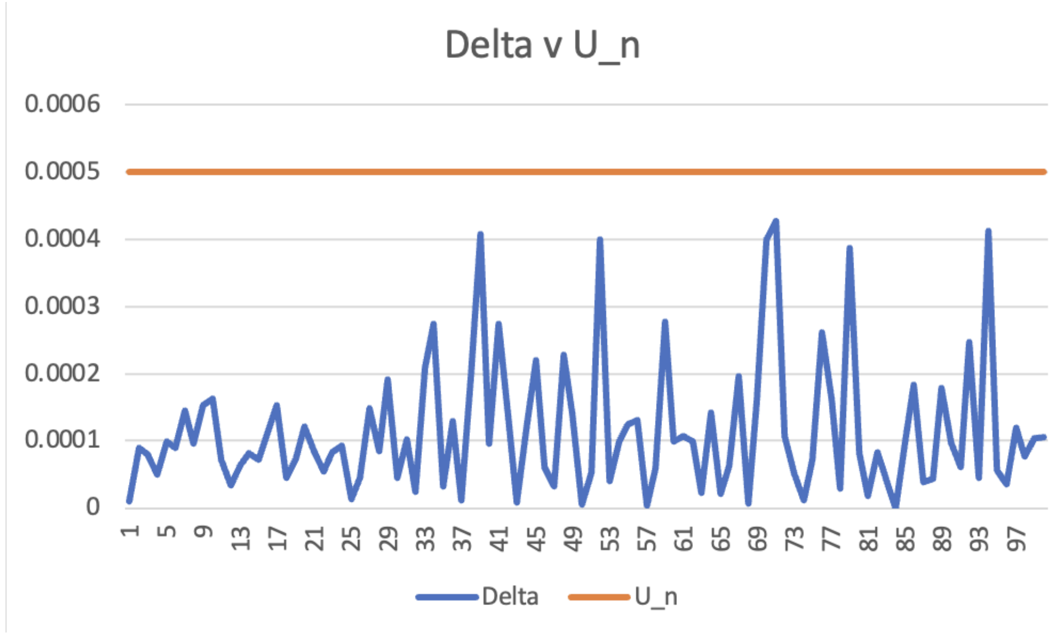
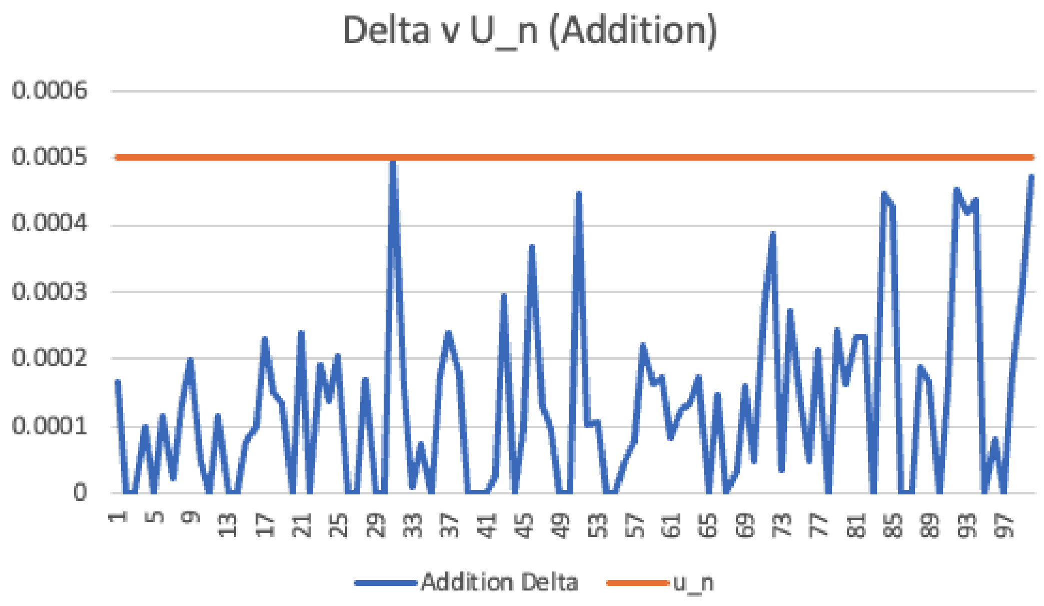


Figure 2: Delta relative to our calculated u_n for addition


Delta v U_n (Addition)
Addition Delta
u_n