HW1

Kevin Smith

October, 2023

1 Executive Summary

In this report, we consider how the floating point system may affect the accuracy of our results given a specified machine precision. We examine the conversion from real numbers to the floating point system, the error created by floating-point addition, as well as how the accumulation of errors occurs.

2 Statement of The Problem

In part one and part two we consider the floating point system with β = 10, t = 4, e_{min} = −8 and e_{max} = 0. In doing so we are able to convert any randomly generated real numbers between our bounds of $[-\ 9999,\ 9999]$ into our floating point system. After converting our randomly generated numbers, we can then perform operations on them within our floating point system. In parts two and three we consider our errors more specifically, checking that our algorithmically derived numbers fall within a specific error bound, u_n. In this report, we will consider the error of translation into our floating point system, the error of addition within our floating point system, and our accumulation error. Respectively,

1. $|\frac{fl(x)-x}{x}| = |\delta| \leq u_n$ (2.1)

2. $\left|\frac{(f_1+f_2)-(f_1 \boxplus f_2)}{f_1+f_2}\right| = |\delta| \leq u_n$ (2.2)

3. $|\widehat{s_n} - s_n| \leq \sum_{k=2}^{n} ||\xi_{1:n}||u_n = (n-1)||\xi_{1:n}||u_n$ (2.3)

3 Description of the Algorithms and Implementation

We used four algorithms in total to manipulate real numbers within our floating point system, which were translated into Python. The first algorithm (3.1) took our mantissa, m, t, remainder, as well as our exponent e. This algorithm rounded our floating point numbers generated by the other algorithms by rounding to the nearest number within our t. The second algorithm (3.2) took a randomly generated x in the real number system and converted it to our floating point system using (1), as well as while loops to ensure that our mantissa remains within our t. This algorithm returns our mantissa and our exponent. The third algorithm (3.3) took two numbers within our floating point system, as well as their exponents, and found the sum of these numbers. After finding the sum, the algorithm then converts our sum into our floating point system similar to (2), returning our summed mantissa and our exponent. Our final algorithm (3.4) generated n floating point numbers and found the sum of these numbers using (3.3), returning our sum.

4 Description of the Experimental Design and Results

In the first part of our examination of the floating point system, we simply looked into our algorithms and developed them with the parameters stated above. We began looking more deeply into this system by examining the error of floating point translation. I began by generating 10,000,000 x's within the bounds of our mantissa, m. Saving the original x, I translated it into our floating point system using (3.2). Each time I would find the error, using (2.1) and compare it

to our $u_n = .0005$. In doing so, I was able to find that our floating point conversion error was bounded by our u_n.

I then proceeded to analyze the error derived from our floating point addition. I generated 100,000 x's and y's as well as generated our exponents, e between (-8,0). In order to prevent underflow in this case, I generated the x and y's between (-9999, -1000)∪(1000, 9999). When performing the floating point addition, I implemented an if-statement which would iterate to the next randomly generated set of numbers if the sum was over 9,999 or under 1,000. I then proceeded to implement algorithm (3.3). While iterating, I saved these errors into a list and checked them against (2.2) to verify that our error was bounded by our u_n. Using the list of errors I was able to perform a statistical analysis. I verified that all of our errors were bounded by u_n. I used the list to find the mean of our errors, being $7.5e^5$ with a variance of $5e^9$. Our maximum error was $.00044$, while being very close to our tolerance it is apparent that this was just an outlier considering our mean and the fact that our minimum error was $2.5e^8$. I then generated two figures, the first figure depicts the spread of the frequency of errors. Thus, showing that our floating point system is an accurate way to compute sums, within the most amount of errors occurring below .0001. The second figure depicts a scatter plot of our errors over n, it is apparent that most of our means once again are below .0001, with some outliers occurring above .0003 which is well within our tolerance. By examining these statistical analyses, we are able to decisively say that our algorithms do not lead to an exorbitant amount of error when it comes to the addition of two floating point numbers.

While the floating point addition was accurate, the accumulation of errors had yet to be considered. To begin, we examined the summation of n numbers within our floating point system. I began by generating a single x within (-9999, -1000)∪(1000, 9999). I then proceeded to enter a while loop generating a random y within the previously stated bounds. I also generated corresponding exponents for x and y. I would then perform (3.3) repeatedly, in the same fashion as the previous problem, but after performing each addition I would save the summed numbers into x as well as save the exponent. I performed this operation 10,000,000 times in order to observe the error using (2.3). In doing so I was able to validate that our accumulation error was well within our bound, without a single occurrence of an error greater than u_n.

We then wanted to examine condition numbers within the IEEE double precision floating point system. Using,

$$c_{rel}^{p_{1:n}} = \frac{|\sum_{i=1}^{n} x_i - \sum_{i=1}^{n} (x_i + p_i)|}{|\sum_{i=1}^{n} x_i|} \frac{||x_{1:n}||_1}{||p_{1:n}||_1} \tag{4.1}$$

And

$$K_{rel} \approx max_{p_{1:n}} c_{rel}^{p_{1:n}} \tag{4.2}$$

I randomly generated 100,000 x's, finding the exact sum of these numbers. I then proceeded to generate randomly generated p's, bounded by ($-xu_n$, xu_n). Using these randomly generated p's I was able to approximate our condition number using (4.2). I repeated this step, replacing our x's with the floating point approximations of x, and once again approximated our condition number.

With our two condition numbers I was able to find the relative error between them, as well as find that the approximated condition numbers were quite similar to the relative error between the two exact sums of x's and the floating point translations of the x's.

5 Conclusions

Through our experiments we were able to individually analyze the floating point system as well as the operations within the system. We found that the floating point system creates errors that are consistent with our generated machine tolerance and that there is an efficient way to bound our problem while considering both time efficiency and accuracy. We found that all of our errors are well bounded around our machine precision. This includes the error between the floating point translation of real numbers, the error between the sum of two floating point numbers, as well as the accumulation error of finding the sum between many floating point representations of real numbers.

6 Tables and Figures

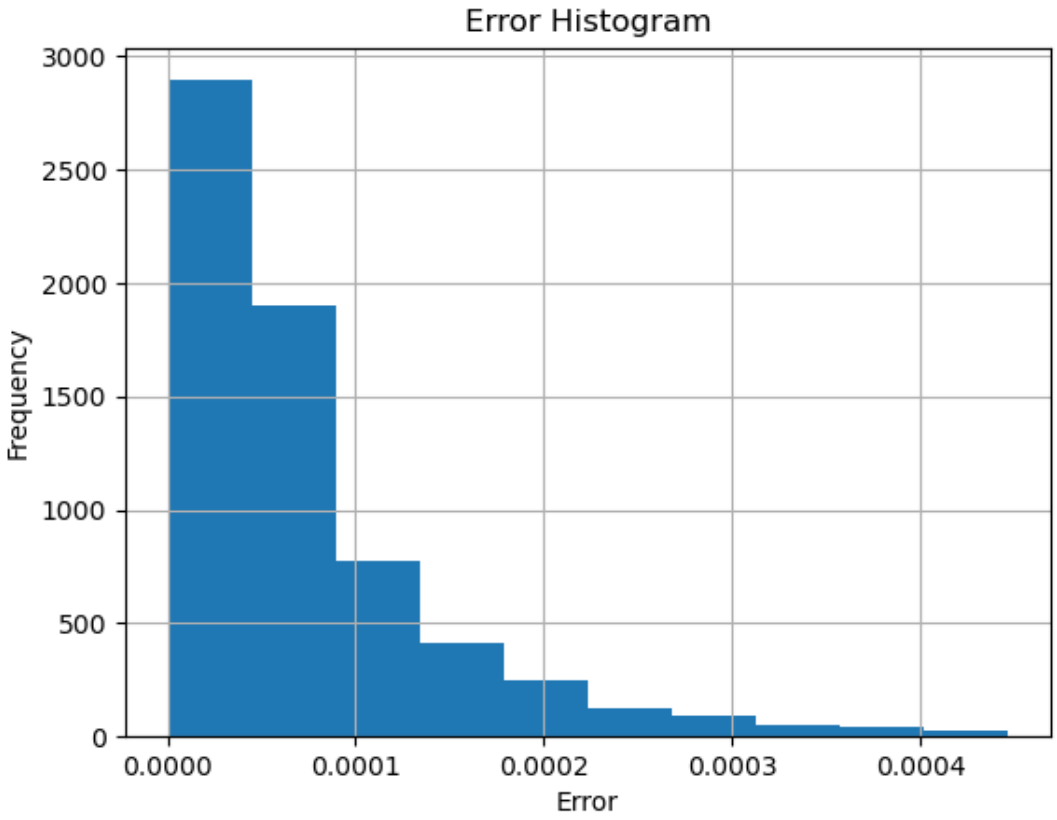

Figure 1: Frequency histogram of the errors as $n \to 100,000$

Figure 2: Scatter plot of the errors as $n \to 100,000$, it is seen that the error does not go above u_n