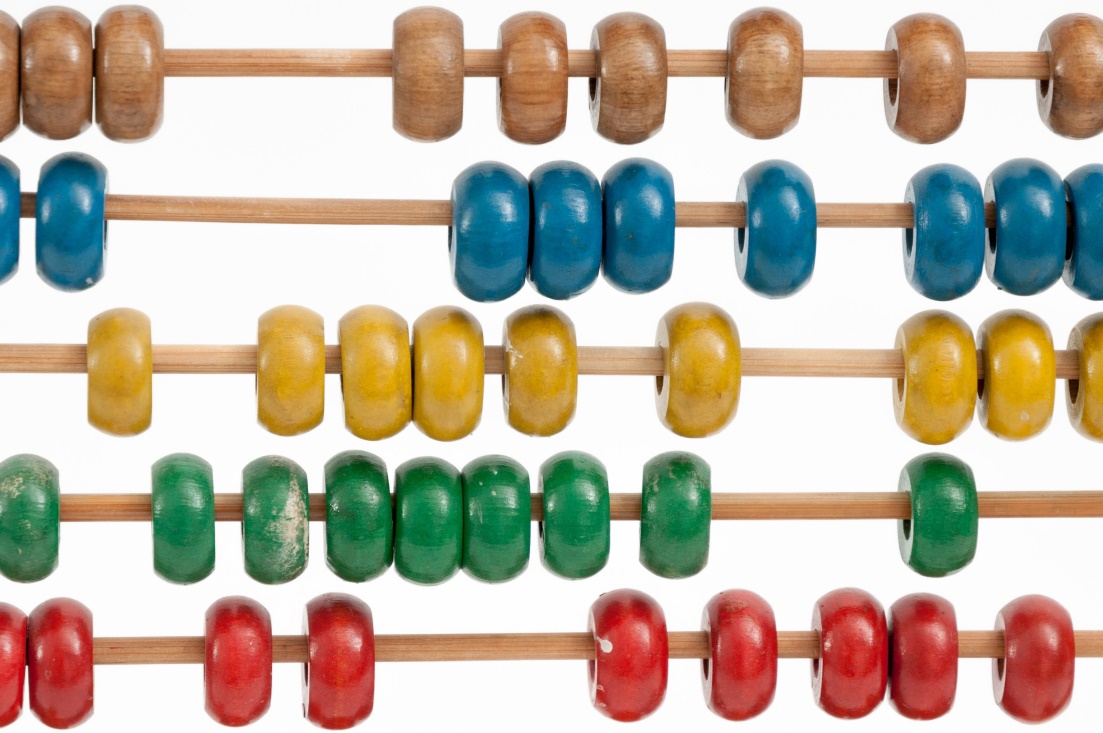
# It Just Doesn’t Add Up!



Modern distributed computing has allowed us to exceed the accuracy of the basic arithmetic operations provided by the CPU. This is a particularly common problem for addition of long sequences of floating point numbers. That is, given a sufficiently large sequence of finite precision numbers, when added with finite precision math, will yield an exceeding inaccurate result. *This can be really bad!*

Much of modern analysis depends on being able to add long sequences of numbers correctly. For example, sums of powers of ***X*** are frequently computed in many statistical models. Many “weighting” strategies used in Finance and Data Science require correctly computed sums of long sequences. Even computing monetary totals can be at risk. A large retailer could incorrectly total sales amounts for a sufficiently large sales volume.

We will explore the background of the problem, offer mitigations for the problem, and showcase concrete examples of the addition problem in a few numerical and statistical processing systems.

## Background

Let’s begin with a motivating example. Suppose we have a three significant-digit calculator (e.g., similar to a Slide Rule from the ancients). We need to sum a sequence with six significant digits. The tables below illustrate the summation process, executed from large to small values and small to large values.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | **Large to Small Addin** | | | | | | |  | **mantissa** | | | | **exp** | |  |  |  |  |  |  |  |  |  | 0 | 0 | 0 |  | | 1 | 2 | 3 | . |  |  |  |  | . | 1 | 2 | 3 | 10\*\*3 | |  | 2 | 3 | . | 4 |  |  |  | . | 1 | 4 | 6 | 10\*\*3 | |  |  | 3 | . | 4 | 5 |  |  | . | 1 | 4 | 9 | 10\*\*3 | |  |  |  | . | 4 | 5 | 6 |  | . | 1 | 4 | 9 | 10\*\*3 | |  |  |  |  |  |  |  |  |  |  |  |  |  | | ***1*** | ***5*** | ***0*** | ***.*** | ***3*** | ***0*** | ***6*** |  |  | **1** | **4** | **9** | ***<-Result*** | | |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | **Small to Large Addin** | | | | | | |  | **mantissa** | | | | **exp** | |  |  |  |  |  |  |  |  |  | 0 | 0 | 0 |  | |  |  |  | . | 4 | 5 | 6 |  | . | 4 | 5 | 6 | 10\*\*0 | |  |  | 3 | . | 4 | 5 |  |  | . | 3 | 9 | 0 | 10\*\*1 | |  | 2 | 3 | . | 4 |  |  |  | . | 2 | 7 | 3 | 10\*\*2 | | 1 | 2 | 3 | . |  |  |  |  | . | 1 | 4 | 9 | 10\*\*3 | |  |  |  |  |  |  |  |  |  |  |  |  |  | | ***1*** | ***5*** | ***0*** | ***.*** | ***3*** | ***0*** | ***6*** |  |  | 1 | 5 | 0 | ***<-Result*** | |

The true sum of these four numbers is 150.306, but our finite precision calculator has computed 149.0 as the estimated sum, when added from large to small values. This example resulted in an absolute error of -1.306 (149.0 – 150.3), with an associated relative error of or -0.00869 (-1.306 / 150.3). The sum is 0.9% low. We do much better summing from small to large, with a relative error of -.0.00200 (-0.3 / 150.3.) The sum is only 0.2% low in this direction.

Now imagine that we have *many* more numbers to add up than illustrated in this example, but the remaining numbers are all less than one. The sum will never increase beyond 149. We can make the absolute error very large and worse yet, we can make the relative error -100% (see note ***33*** below.)

While this problem may not seem so bad, imagine we are dealing with Diamonds, and these numbers represent carat weights of diamonds at a major Jeweler. Diamonds average about $1000 a carat for small ones. A Jewelry company would be upset with a $1,306 of under counted diamonds.

We suffer from two sources of error when we operate on quantities using Finite Precision Arithmetic:

* Representational and
* Operational errors.

Our floating point numbers are approximations of the true quantities when represented in finite precision arithmetic. This is the cause of representational error.

Here is a classic example of the *representation* problem shown using Python evaluation:

|  |  |
| --- | --- |
| **Code** | **Output** |
| **x = float(1.0) / float(10.0)**  **print('Add x = {:.18f} 10 times'.format(x))**  **wrong\_sum = float(0.0)**  **for i in range(10):**  **wrong\_sum += x**  **print('Sum is: {:.18f}'.format(wrong\_sum))** | Add x = 0.100000000000000006 10 times  Sum is: 0.999999999999999889 |

We see that 0.1 is not exactly represented in floating point. When the quantities are added 10 times, they do not exactly equal one, as one would expect using exact arithmetic.

Operational error, as shown in the first 3 three digit addition example, is a property of the computing system used to operate with finite precision representation. A good Finite Precision Math Tutorial is found at <https://www.cs.purdue.edu/homes/skeel/CS515/2.pdf>.

Now that you have gotten this far, here is a metaphor for the problem mitigation we will explore, expressed in the form of an old joke:

A man goes to a doctor, holding his hand behind his back, and says: “Doc, it hurts when I do this”. The doctor replies: “well then, don’t do that!”

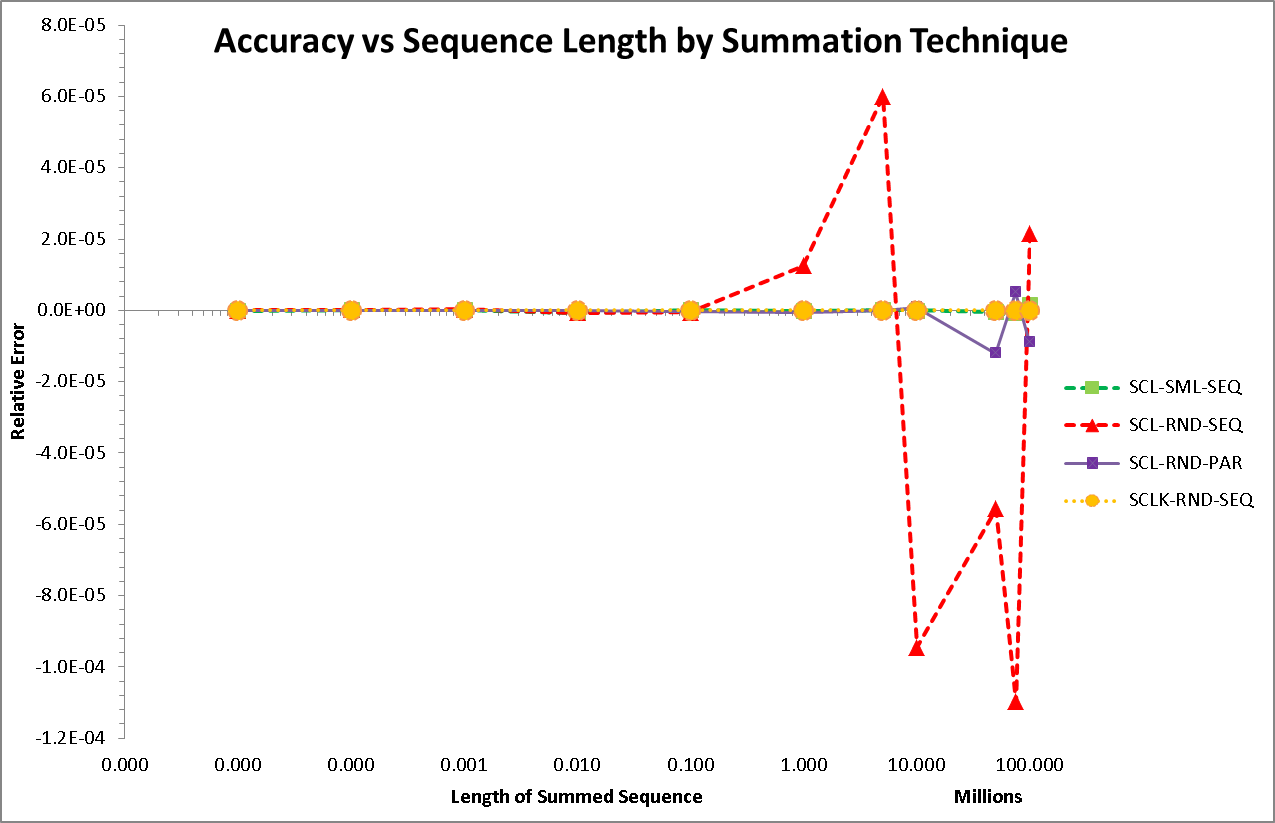
Now let’s get a more detailed view of the problem and how it manifests itself in my software systems.

## The Addition Problem in Detail

We will perform an experiment using a sequence of numbers with a known sum, introduce representational error, and then perform addition of the sequence in various manners. All the code for this experiment is in Java and found in the DemoDev GitHub repository at <https://github.com/DonaldET/DemoDev/tree/master/dev-topics-algorithms/dev-topics-badaddr>. A detailed experiment description is found there at <https://github.com/DonaldET/DemoDev/blob/master/dev-topics-algorithms/dev-topics-badaddr/documentation/Summation.docx>.

We sum up to one hundred million integers, generated in order from 1 to 100,000,000. We compare the computed sum to mathematically correct sum, and then evaluate relative error. We do these sums by adding from smallest to largest, largest to smallest, and in random order.

We then scale the integer sequences by dividing by a large prime number, thereby introducing representation error. We then repeat the sums. Finally, we use a numerical analysis technique to reduce errors in addition and repeat the sums and error computations. The results are shown in these graphs:



We see relative error show up at around summing the sequence of the first 100,000 integers. The series shown in the graph are:

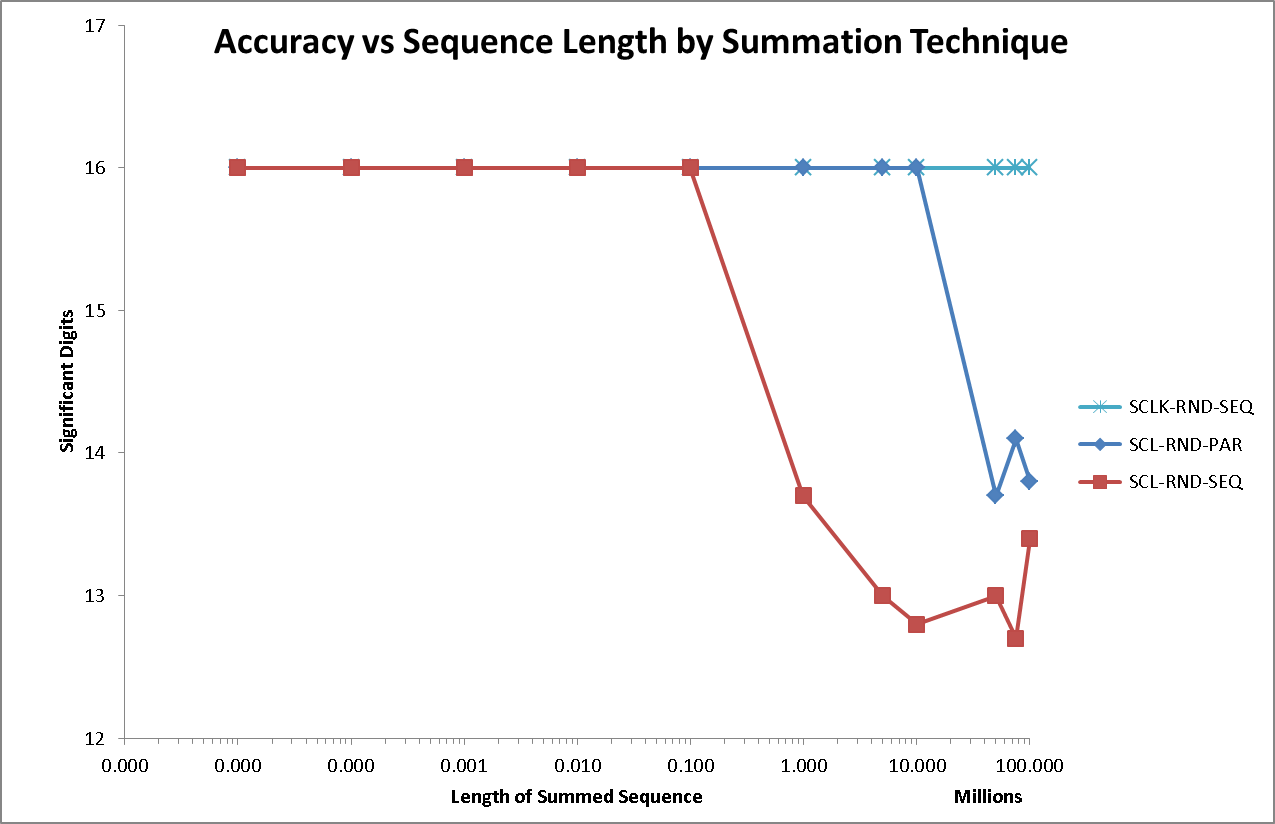
|  |  |  |  |
| --- | --- | --- | --- |
| **Sequence** | **Add Order** | **Distribution** | **Label** |
| Scaled | Smallest first | Sequential | SCL-SML-SEQ |
| Scaled | Largest first | Sequential | SCL-LRG-SEQ |
| Scaled | Random | Sequential | SCL-RND-SEQ |
| Scaled | Random | Parallel | SCL-RND-PAR |
| Scaled-Kahan | Random | Sequential | SCLK-RND-SEQ |

### Experiment Summary:

We see from the graph that:

* The Kahan algorithm used with random summation order (SCLK-RND-SEQ), shown in orange with short dashes and circle marker, maintains almost zero relative error.
* Adding from smallest to largest value (SCL-SML-SEQ), shown in green with a rectangle marker, also maintains zero relative error.
* Adding in parallel (7 threads in this case) lowers the relative error, but does not eliminate it. Shown in purple with solid line and square marker.
* Finally, adding in random order shows the greatest error. Shown in read with short dashes and triangle marker.

The next graph summarizes the overall accuracy in terms of significant digits:

`

This graph shows we lose nearly four significant digits when summing the 100,000,000 scaled integers randomly. Parallelization helps a little because it divides the larger summation into roughly equal smaller summations (7 in this case.) If we employ a mathematical correction like Kahan summation, we are able to limit relative error.

Lessons Learned

This experiment suggests loss of accuracy occurs with as few as 100,000 summations, and becomes significant at 100,000,000 summations. A shorter sequence of entries with greater variability would show this behavior sooner (e.g., sums of squares.) We also see that summation order and addition technique influences the accuracy of the result. In general, should try to:

* Limit the length of sequences (parallelize.)
* Order addition from smallest to largest where possible.
* Employ mathematical corrections in the addition process.

dfdfsdfsdf

Note ***33***:

Let RE be the relative error for adding n values of size α, where n >> 1 and α << 1. Furthermore, let F represent the of the first four terms in the motivating example in the **Background** section. Then:

* RE(n) = - (n \* α) / (F + n \* α) = -1 / (1 + F / (n \* α)) = lim(n -> ∞) { -1 / 1} = -100%