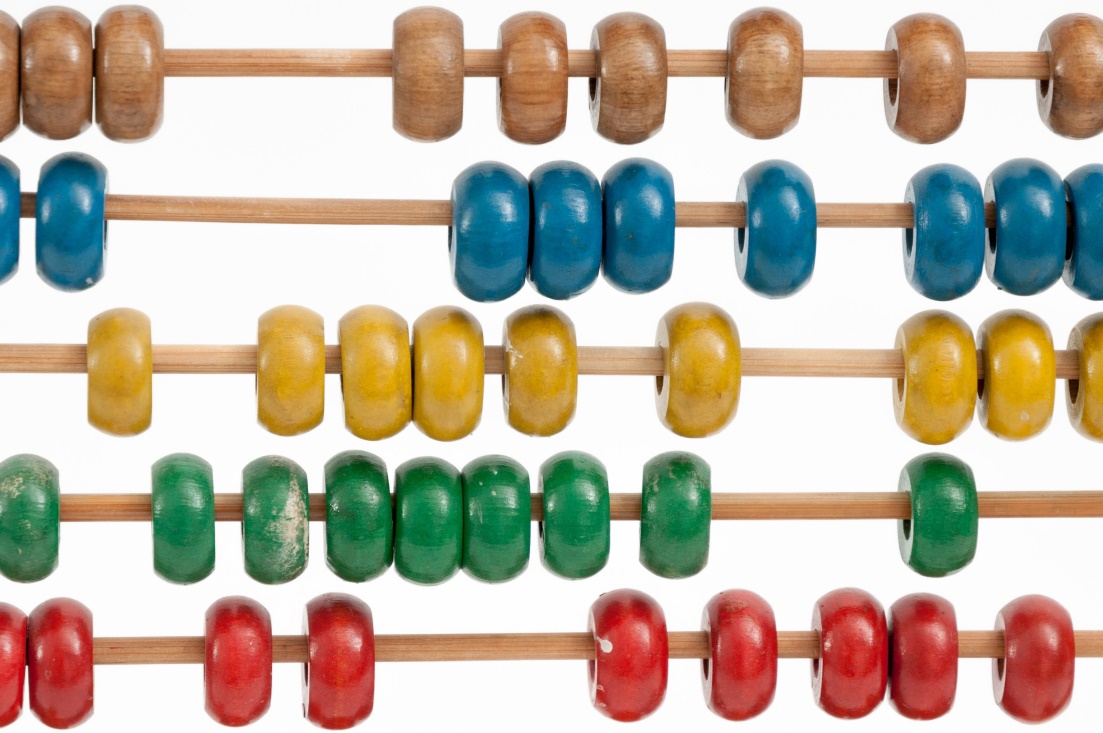
# It Just Doesn’t Add Up! – Part 1



Modern distributed computing, often called “Big Data”, has allowed us to exceed the accuracy of the basic arithmetic operations provided by the computer fixed-precision arithmetic. 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 added with finite precision arithmetic, will yield an exceeding inaccurate result. *This can be really bad!*

Much of modern analysis depends on summing sequences of numbers correctly. For example, sums of powers of ***X*** are frequently computed in many statistical predictive models. Here are more examples:

* Many “weighting” strategies used in Finance and Data Science require summation.
* Computing monetary totals, sales volume for example, can be at risk if an incorrect representation is chosen for monetary amounts.
* Common data engineering tasks that compare the output of two “Big Data” outputs will need to sum a large set of numbers (e.g., differences between old and new output.)

This paper is of interest to data scientists and data engineers using Hive and Spark, and to anyone processing large numerical data sets but not using a good machine learning library that employs corrections for large summations. Examples of distributed machine learning libraries are Spark ***MLib*** and Hadoop ***Mahout*** (see <https://www.quora.com/What-are-the-differences-between-Apache-Mahout-and-Spark-MLlib>.)

## Theoretical Background

We will explore the background of the problem, offer mitigations for the problem, and showcase concrete examples of the summation problem using standard programming techniques. All programs, run-logs, and documentation are found in my ***DemoDev*** GitHub repository at <https://github.com/DonaldET/DemoDev/tree/master/dev-topics-algorithms/dev-topics-badaddr>. The resources section below offers some numerical analysis articles to help understand the theory. The key point about uncorrected summations is that *the relative error for addition is unbounded*! The relative error of addition depends on the size of the data set, and not on the precision of the addition operation.

## Simplified Fixed Precision Addition Example

Let’s begin with a simplified example of addition error in computer finite-precision arithmetic. Suppose we have a three significant-digit calculator (e.g., similar to a Slide Rule of the ancients). We need to sum a sequence that would require six significant digits in standard arithmetic. The tableau below illustrates the summation process, summing from large to small values and small to large values.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | **Large to Small Addition** | | | | | | |  | **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 Addition** | | | | | | |  | **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 three digit finite precision calculator has computed 149.0 as the estimated sum when adding 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 large-to-small 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 summation in this direction is only 0.2%.

Imagine that we have *many* more numbers to add up than illustrated in this small example; but all the remaining numbers are less than one. The sum will never increase beyond 149. As a result, we can make the absolute error very large. We can make the relative error -100% in this example (see note ***Sample Relative Error Proof*** below.)

While the 0.9% error may not seem so bad, imagine we are dealing accounting for a valuable substance like Diamonds, and that the example numbers represent carat weights of diamonds. Diamonds average about $1000 a carat for small ones. A Jewelry company would be upset with a $1,306 of under counted diamonds. Clearing we would need more digits to achieve satisfactory accounting.

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

* Representational error, and
* Operational error.

Our floating point numbers are approximations of the true quantities as 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 computer floating point arithmetic. When the quantity 1/10 is added 10 times, the sum not exactly equal one. We would expect the sum to be exactly one when normal exact arithmetic.

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

There is an answer to this problem. Here is a metaphor for mitigation we will explore, expressed in the form of an old joke:

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

We explore a more detailed view of the problem and how it manifests itself in many software systems.

## The Addition Problem in Standard Programming

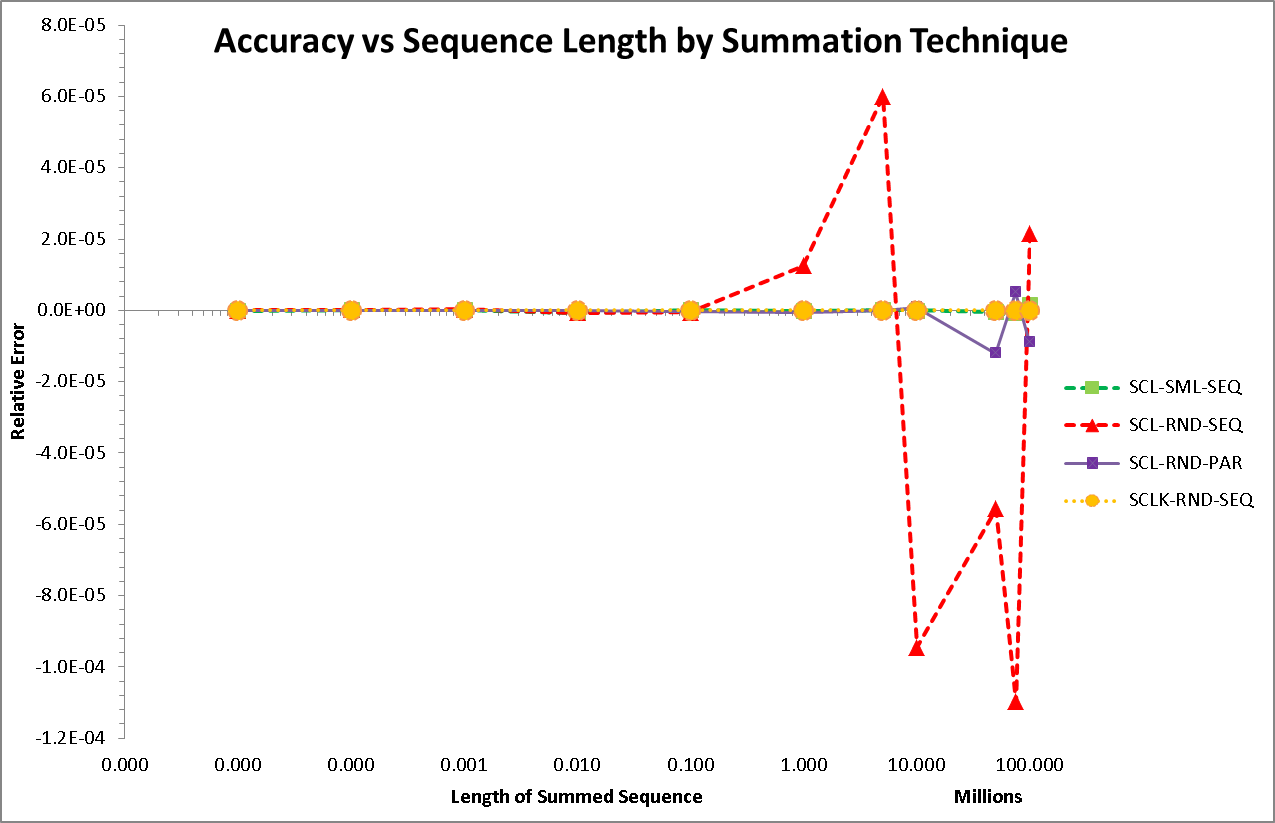
The first step is to assess the impact of the problem. Modern computers use 16 significant digits in floating point calculations, and problems will not show up in small summations. To estimate problem boundaries, we will perform an experiment using a sequence of numbers with a known sum, introduce representational error for sequence members, and then perform addition of the sequence in various manners. All the code for this experiment is found in ***DemoDev***, and the mathematical background is in document ***SummationBackground.docx*** (see reference 2.)

We create a sequence of 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 in the three orders.

Finally, we use a numerical analysis technique to reduce errors in summation (see <https://en.wikipedia.org/wiki/Kahan_summation_algorithm>.) Using the Kahan technique, we repeat the sums and error computations. It is important to note that the commonly used method of *provisional means* is a mechanism to avoid *overflow* in calculating distribution moments, and is not an accuracy enhancement (see <http://www.pmean.com/04/ProvisionalMeans.html>.)

A summary of experimental results are shown in the following 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:

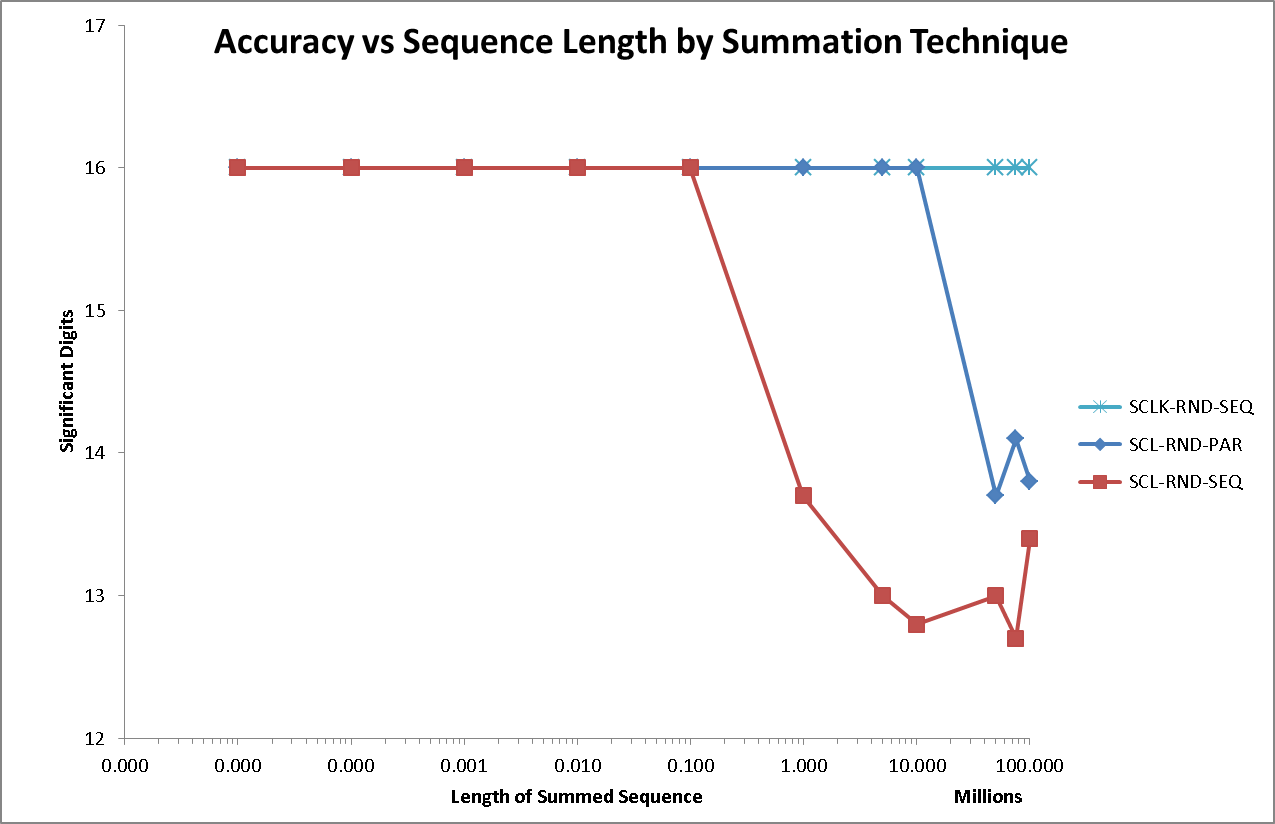
|  |  |  |  |
| --- | --- | --- | --- |
| **Sequence** | **Add Order** | **Distribution** | **Label** |
| Scaled | Smallest first | Sequential | SCL-SML-SEQ |
| Scaled | Largest first | Sequential | SCL-LRG-SEQ |
| Scaled | Random | Sequential | SCL-RND-SEQ |
| Scaled | Random | Parallel | SCL-RND-PAR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

### Experiment Summary:

We see from the graphs 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 of element of the experiment 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 (ccccc.)

## Lessons Learned

This experiment suggests loss of accuracy occurs with as few as 100,000 summations, and becomes pronounced at 100,000,000 summations. A sequence of entries with greater variability than our test sequence 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, we should try to:

* Limit the length of sequences by using multiple smaller subsequences.
* Order addition from smallest to largest where possible.
* Employ mathematical corrections in the summation process.

The first mitigation is accomplished by distribution of summation in Spark and Hive, and parallelization is direct Java and Python programming. This is a basic divide-and-conquer technique. A sample Kahan Java implementation is available in the ***DemoDev*** GitHub repo at <https://github.com/DonaldET/DemoDev/tree/master/dev-topics-algorithms/dev-topics-badaddr/analysis>.

## Pervasiveness

The summation problem is not limited to Java. We find the problem in Python and R languages as well. Python and R examples are found in the ***DemoDev*** GitHub repo at <https://github.com/DonaldET/DemoDev/tree/master/dev-topics-algorithms/dev-topics-badaddr/analysis>.

Most statistical packages incorporate corrections for summation. Big Data tools like Hive and Spark *do not* have such corrections in place. One must exercise care when creating Big Data sums with these powerful tools. A follow-up article will survey some popular Python and R libraries to assess summation accuracy.

Ω

## Reference Materials

1. Representation error problems in Excel: <https://www.microsoft.com/en-us/microsoft-365/blog/2008/04/10/understanding-floating-point-precision-aka-why-does-excel-give-me-seemingly-wrong-answers/>.
2. Academic articles defining the problem: <https://github.com/DonaldET/DemoDev/blob/master/dev-topics-algorithms/dev-topics-badaddr/documentation/SummationBackground.docx>.
3. Detailed **bad-addr** algorithm description: <https://github.com/DonaldET/DemoDev/blob/master/dev-topics-algorithms/dev-topics-badaddr/documentation/Addition_Checker_Description.docx>.

Note: ***Sample Relative Error Proof***

Let **RE** be the relative error for adding n additional values of size α, where n >> 1 and 0 < α < 0.001. Furthermore, let F represent the sum of the first four terms in the motivating example above, demonstrated in the **Background** section. Then:

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