ACCURACY OF SIMPLE FLOATING POINT ACCUMULATIONS FOR MONETARY AMOUNTS

W. ROSS MORROW

1. Introduction

Here we analyze the impact of datatypes on basic operations, accumulation with addition and subtraction, of data we intend to represent monetary quantities. We execute calculations with `int`s, `long`s, `float`s, and `double`s that represent what might be reflective of totaling up transactions and payments, and study how the number of amounts to aggregate impacts accuracy. We don't consider operations, like taking taxes or interest using percents, that would either *require* floating point or perfect rational number math. If we got deep enough into taking percents we would have to think about proper rounding of the amounts themselves in real number math anyway, and when that rounding takes place in accumulating amounts. For now, we keep it simple and limited to adding amounts up, and ask mainly about "how many" such amounts to avoid errors in the cents place.

Specifically our goal is to investigate if and when floating point numbers can accurately reflect the true results of such calculations, in cents. Everything we consider can be done exactly with `long`s, which are our ground truth. We consider `int`s too, because there's a general question about the space required to store the results of the calculations we do. We don't care whether the floating point representations themselves, in all their precision, match a true value in cents. (Spoiler 1: they won't.) Rather we only care that those floating point values, *when rounded to cents*, match the true value. This is possible with `double`s, but not `float`s, although depending on the size of the accumulations we might need to use specific summation methods for `double`s (that aren't at all hard to program). If that holds, then we have some assurance that `double` datatypes can serve as an effective intermediary, at least: if we can persist `long int` data, manipulate it with `double`s (so long as this is convenient), and store it back as `long int` *correctly* then we basically never lose precision *in the units we care about* (here presumed to be cents).

Our numerical experiments suggest `double`s are fine in this lens for sums of hundreds of thousands or even millions of actually pretty large amounts. These experiments are, obviously and inevitably, dependent on the assumptions we make in formulating them particularly concerning the distribution of amounts. So we also undertake a theoretical analysis that provides *guarantees* that `double`s are fine, under our assumptions for drawing amounts, for at least aggregations of tens of thousands of amounts. If we're careful about how we sum amounts up, guaranteed accuracy extends to the millions of amounts.

A super important point to note is that, these days, "double is often the default". In particular, python `float`s are `double`s (unless you say otherwise), and I've even heard that

"under the hood" single-precision `float` calculations are done in 64B (double) hardware (and I would love either a correction or a reference). We have to explicitly store `double` data in our databases to keep the precision, though this is really more about storing 8 bytes rather than 4. Point is that while amounts are being used it's probably not an awful mistake to just presume some automagic has put `double` precision to work for you. And definitely *never* use single precision, at least not without careful analysis in its favor.

We also review some basics about floating point representation, including spooky edge cases, but mainly to mediate those with results guaranteeing precision in our particular case. More or less, the takeaway here is use 8 bytes (8B) in either `long int` or `double` form and you're good. But nothing's ever so simple as that kind of soundbite, and I hope this article also displays some of the exciting complexities involved here.

2. Initial Experiments

We use a C code, `draft.c`, to run some tests quickly and be sure of the bytewidth of different datatypes. draft accepts three arguments in this order: an accumulation size (N), a(n optional) number of trials (T), and an (optional) filename for the results. The call below ignores the third argument, and runs $10,000$ trials of $100,000$ element accumulations with addition, subtraction, or both. For each trial, draft draws N random amounts x_n following two models: Our "uniform" model is in the range

$$x_n \sim U[\ 0\ ,\ 2147483647\ ] \mod 10000000$$

This is hacky, but a reasonably uniform distribution of values up to about $100,000. But the "typical" values here are unrealistically large. For better "typical" values, we also use a simple 3-bin model where

$$x_n \sim U[0,1000] \text{ w.p. } 0.5\ ,\ U[1000,10000] \text{ w.p. } 0.35\ ,\ U[10000,100000] \text{ w.p. } 0.15$$

where w.p. stands for "with probability".

Each amount roughly represents a "transaction" value. In the uniform model, we choose $100,000 just to explore extremes as this is where roundoff error will show it's ugly face faster. Put another way, we need sample amounts with very different orders of magnitude to have any hope of revealing issues with a loss of significant digits. Using the 3-bin model, we're working largely with smaller dollar amounts we less interesting things for the computations impatience will allow us to undertake.

With each such random amount, (w.l.o.g.) drawn as a 32-bit (4B) `int` in cents, we also store

- the same value cast "up" to a `long` (8B)
- the same value cast to a 32-bit (4B) `float` in *both* dollars and cents
- the same value cast to a 64-bit (8B) `double` in *both* dollars and cents

For repeated addition and subtraction operations, the `long int` version gives us the ground truth. The maximum value (9,223,372,036,854,775,807) is so large we would need to run an accumulation of over 922 *trillion* values to overflow. This is larger even than the largest (4B) `int`, which is our input type for N, so we completely ignore this. The code wouldn't run anyway; storing an "amount" in our sense costs 36B, and 900T amounts would take

something like 31,000GB of RAM. To run in under 1GB of memory we need about $N \leq 29,826,162$ which still represents a rather large (daily) amount accumulation. For a bit of reference, in 2019 worldwide, Visa and Mastercard together had on average 401,369,863 transactions a day worldwide, and Amex had about 21,917,808 transactions a day. And, interpreting `long`s as values in cents, we can store well over 900k Bezos' (1 Bezos $\sim$ 100 billion dollars) in monetary value.

With a set of N amounts stored in these different types and precisions (dollars and cents for the `float`s and `double`s), we do the following three things:

- add them all up into one total number
- subtract them all into one total number
- add or subtract each number from a total, based on a fair coin flip (50% probability of either)

For each of these operations we do them all with each type and precision. Again, the `long` (in cents) will be exact. The `int` versions will very likely overflow for seemingly reasonable numbers of amounts. The `float` and `double` ones are up in the air.

The code writes a csv file, named `results.csv` by default, with a line for each trial using distinct random amounts. There are columns for each non-exact value – `int` cents, `float` cents, `float` dollars, `double` cents, and `double` dollars – with a 0 if the result is wrong (relative to `long`) and 1 if it is correct. With a `results.csv` (or whatever) file, we can read it in as a `DataFrame` and summarize the fractions of trials in which each type/precision resulted in a correct calculation. This is done below, where the column names amount to first-letter abbreviations (f_c is float, cents). The mean and std just aggregate the 1s and 0s in the columns for a quick look. Actually, our trials are a Bernoulli process so the mean of the outcomes is the probability p of being "correct", and std should be $\sqrt{p(1-p)}$ (which seems about right in the data I've seen).

```
cols = ['trial','op','f_c','f_d','d_c','d_d']
df   = pd.read_csv( 'results.csv' , header=None , names=cols )
gdf  = df[cols[1:]].groupby('op').agg( [ 'mean' , 'std' ] )
```

Results for amounts drawn from the uniform model are shown in Table 1. Basically, using `int`s or `float`s is bad news. `float`s are almost never correct, even for very small sums of $N = 100$ or $N = 1,000$ amounts. `int`s are right for mixed-signed sums until $N = 100,000$ and $N = 100$, but otherwise not usually correct. Conversely, `double`s in cents are *always* correct, and `double`s in dollars are correct until $N = 1,000,000$.

Results for amounts drawn from the 3-bin model are shown in Table 2 and paint a different picture. `int`s are correct until $N = 10,000,000$, suggesting (unsurprisingly) that it is the average size of the amounts that make `int`s poor performing. `float`s in cents are correct until $N = 1,000$ and in dollars for $N = 100$, but otherwise generally incorrect. This is better performance, but still to poor to be usable. `double`s in dollars or cents are now never wrong.

TABLE 1. Accuracy of sequential accumulation of N amounts for `int` data in cents, and `float` and `double` data in dollars and cents. Amounts are the mean (μ) and standard deviation (σ) of an indicator variable for correctness over 10,000 independent random trials, expressed as percents. Amounts accumulated are drawn according to the uniform model.

	`int` (c)		`float` (c)		`float` ($)		`double` (c)		`double` ($)	
op	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
$N = 100$										
+	100.00	0.00	0.62	7.85	0.42	6.47	100.00	0.00	100.00	0.00
-	100.00	0.00	0.62	7.85	0.42	6.47	100.00	0.00	100.00	0.00
±	100.00	0.00	4.82	21.42	5.32	22.44	100.00	0.00	100.00	0.00
$N = 1,000$										
+	0.00	0.00	0.20	1.41	0.01	1.00	100.00	0.00	100.00	0.00
-	0.00	0.00	0.20	1.41	0.01	1.00	100.00	0.00	100.00	0.00
±	100.00	0.00	0.56	7.46	0.53	7.26	100.00	0.00	100.00	0.00
$N = 10,000$										
+	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	100.00	0.00
-	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	100.00	0.00
±	100.00	0.00	0.05	2.24	0.02	1.41	100.00	0.00	100.00	0.00
$N = 100,000$										
+	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	100.00	0.00
-	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	100.00	0.00
±	75.77	42.85	0.01	1.00	0.00	0.00	100.00	0.00	100.00	0.00
$N = 1,000,000$										
+	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	99.93	2.65
-	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	99.93	2.65
±	29.08	45.42	0.00	0.00	0.00	0.00	100.00	0.00	100.00	0.00
$N = 10,000,000$										
+	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	10.87	31.13
-	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	10.87	31.13
±	9.83	29.77	0.00	0.00	0.00	0.00	100.00	0.00	100.00	0.00

TABLE 2. Accuracy of sequential accumulation of N amounts for `int` data in cents, and `float` and `double` data in dollars and cents. Amounts are the mean (μ) and standard deviation (σ) of an indicator variable for correctness over 10,000 independent random trials, expressed as percents. Amounts accumulated are drawn according to the simple 3 bin model.

	`int` (c)		`float` (c)		`float` ($)		`double` (c)		`double` ($)	
op	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
$N = 100$										
+	100.00	0.00	100.0	0.00	99.33	8.16	100.00	0.00	100.00	0.00
-	100.00	0.00	100.0	0.00	99.33	8.16	100.00	0.00	100.00	0.00
±	100.00	0.00	100.0	0.00	100.0	0.00	100.00	0.00	100.00	0.00
$N = 1,000$										
+	100.00	0.00	100.0	0.00	8.43	27.79	100.00	0.00	100.00	0.00
-	100.00	0.00	100.0	0.00	8.43	27.79	100.00	0.00	100.00	0.00
±	100.00	0.00	100.0	0.00	82.05	38.38	100.00	0.00	100.00	0.00
$N = 10,000$										
+	100.00	0.00	0.29	5.38	0.23	4.79	100.00	0.00	100.00	0.00
-	100.00	0.00	0.29	5.38	0.23	4.79	100.00	0.00	100.00	0.00
±	100.00	0.00	100.00	0.00	14.02	34.72	100.00	0.00	100.00	0.00
$N = 100,000$										
+	100.00	0.00	0.00	0.00	0.02	1.41	100.00	0.00	100.00	0.00
-	100.00	0.00	0.00	0.00	0.02	1.41	100.00	0.00	100.00	0.00
±	100.00	0.00	95.30	21.16	1.28	11.24	100.00	0.00	100.00	0.00
$N = 1,000,000$										
+	100.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	100.00	0.00
-	100.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	100.00	0.00
±	100.00	0.00	11.01	31.30	0.13	3.60	100.00	0.00	100.00	0.00
$N = 10,000,000$										
+	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	100.00	0.00
-	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	100.00	0.00
±	100.00	0.00	0.00	0.00	0.02	1.41	100.00	0.00	100.00	0.00

3. What are Floating-Point Numbers Anyway?

We aren't really going to give a comprehensive explanation of these results, but we will provide hints and analysis. And to do that we need to review floating point math.

TABLE 3. Rough floating point sketch of a few numbers, basically just factoring out a sign and power of two and representing the decimal in base-10 instead of binary.

`fl`(1)	$s = 0$	$e = 1023$	$b = 0$
`fl`(-1)	$s = 1$	$e = 1023$	$b = 0$
`fl`(10)	$s = 0$	$e = 1026$	$b = 0.25$
`fl`(100)	$s = 0$	$e = 1029$	$b = 0.5625$

3.1. **Floating Point Numbers.** IEEE 754 `double`s store real numbers x essentially as tuples

$$(s(x), b(x), e(x)) \in \{0,1\} \times \{0,1\}^{52} \times [0, 2047]$$

where

$$\texttt{double}(x) = (-1)^{s(x)} \left(1 + \sum_{k=1}^{52} \frac{b_{52-k}(x)}{2^k}\right) 2^{e(x)-1023}$$

The "first" bit is a sign bit, the next 11 are a base-2 exponent (read as an unsigned int), and the last 52 are base-2 decimals (the "fractional part" or "significand"). Table 3 has some rough examples, and Table 4 has some *exact* examples. The exact representations come from the code `plotbits.c`, which you can read and run to get a better sense of how floating point numbers are represented (at least in the Intel chips used by Mac OSX and Apple clang version 11.0.3). You can change values in main to print the representation of different numbers, like in the call

```
print_double_bits( M_PI );
```

which prints `math.h`'s defined π value as

```
(d) 3.1415926535897931:
    sign... 0
    exp.... 1024 (00000100|00000000)
    bits... 10010010|00011111|10110101|01000100|01000010|11010001
    full... 0,1000000|0000,1001|00100001|11111011|01010100|01000100|00101101|00011000
```

Here the "|" characters separate bytes (B) in multi-byte datatypes. In the `full` line of the print we also use "," to separate *logical* fields of the `double` representation. At the very least, reviewing this may give an appreciation of the startling complexity (and ingenuity!) of effectively representing real numbers in computers.

Let's look carefully at one of these in particular, 100. The formula translates to

$$\begin{aligned} 100 &= 1.5625 \times 64 = (1 + 0.5 + 0.0625)\, 64 \\ &= \left(1 + \frac{1}{2} + \frac{1}{16}\right) 64 = (-1)^0 \left(1 + \frac{1}{2} + \frac{1}{2^4}\right) 2^6 \end{aligned}$$

While the formulas and the data look insane, this isn't so bad, right?

TABLE 4. Actual floating point representations (in `double`s) for a few select numbers. $b_{51:0}$ denotes that the bitstrings are actually printed in reverse relative to a left-right reading, with 51 first on the left and 0 last on the right. From runs of `plotbits.c`.

$\texttt{fl}(1)$	$s=0$	$e=1023$	$b_{51:0}=0000000000000000000000000000000000000000000000000000$
$\texttt{fl}(-1)$	$s=1$	$e=1023$	$b_{51:0}=0000000000000000000000000000000000000000000000000000$
$\texttt{fl}(0.5)$	$s=0$	$e=1022$	$b_{51:0}=0000000000000000000000000000000000000000000000000000$
$\texttt{fl}(10)$	$s=0$	$e=1026$	$b_{51:0}=0100000000000000000000000000000000000000000000000000$
$\texttt{fl}(100)$	$s=0$	$e=1029$	$b_{51:0}=1001000000000000000000000000000000000000000000000000$
$\texttt{fl}(0.01)$	$s=0$	$e=1016$	$b_{51:0}=0100011110101110000101000111101011100000101000111$
$\texttt{fl}(\pi)$	$s=0$	$e=1024$	$b_{51:0}=1001001000011111101101010100010001000010110100001$

3.2. **Doubles in Cents.** Let's make sure that a 32-bit integer i, in cents, can be represented exactly *up to cents* in a double. (Recall we presume we can store any given amount, in cents, using a standard `int`, but need long `int`s to store reasonable accumulations of such amounts.) This way we at least know that we can store exact values without *relevant* precision loss in a double. If we can't do that, we should probably give up. (Spoiler 2: we can.)

It is actually quite trivial to show that an `int` has an exact `double` representation. Obviously our integer i has a unique base-2 representation:

$$\texttt{int}(i) = (-1)^{s(i)}\left(\sum_{k=1}^{31} b_k 2^{k-1}\right) = (-1)^{s(i)}\left(\sum_{k=1}^{31} b_k 2^{k-31}\right)2^{30} = (-1)^{s(i)}\left(\sum_{k=0}^{30}\frac{b_{31-k}}{2^k}\right)2^{30}$$

More or less this is what is stored as the `int` value. Now if $b_{31}=1$, we have

$$\texttt{int}(i) = (-1)^{s(i)}\left(1+\sum_{k=1}^{30}\frac{b_{31-k}}{2^k}\right)2^{30}$$

which is in `double` representation $(s(i), b', 1053)$ with fractional part bits b' satisfying

$$b'_{52-k} = b_{31-k}\ ,\ \ k=1,\ldots,30 \quad \text{and} \quad b'_{52-k}=0\ ,\ \ k=31,\ldots,52$$

Otherwise, let $b_{31}=\cdots=b_{31-k_0-1}=0$, $b_{31-k_0}=1$; then

$$\begin{aligned}\texttt{int}(i) &= (-1)^{s(i)}\left(\sum_{k=k_0}^{30}\frac{b_{31-k}}{2^k}\right)2^{30} = (-1)^{s(i)}\left(\sum_{k=k_0}^{30}\frac{b_{31-k}}{2^{k-k_0}}\right)2^{30-k_0}\\ &= (-1)^{s(i)}\left(1+\sum_{k=k_0+1}^{30}\frac{b_{31-k}}{2^{k-k_0}}\right)2^{30-k_0} = (-1)^{s(i)}\left(1+\sum_{k=1}^{30-k_0}\frac{b_{31-k+k_0}}{2^k}\right)2^{30-k_0}\end{aligned}$$

which is in `double` representation $(s(i), b', 1053 - k_0)$ with fractional part bits b' satisfying

$$b'_{52-k} = b_{31-k+k_0} \quad , \quad k = 1, \ldots, 30 - k_0 \quad \text{and} \quad b'_{52-k} = 0 \quad , \quad k = 30 - k_0 + 1, \ldots, 52$$

So, roughly speaking, by absorbing the number of "leading" zero bits into the exponent and (possibly) reording the indexing of the `int` bits into the double's fractional part, we have an exact representation of the `int` as a double. (Note that single-precision `float`s only have fractional parts with 23 bits, meaning we would have to truncate the bit sum in the above leading to casting error when the integers have nonzero leading bits and thus are large.)

If we keep our `double` values in cents, that's it for one direction: taking an `int` and putting it in a double. If we keep our `double` values in dollars, we have to also divide by 100.0 which will introduce some extra error. We deal with that later.

Now we can think about the second part: taking the `double` back to an `int`. If we are using dollars, we have to multiply by 100.0 to get cents, then convert. But let's first consider `double`s storing cents. To convert a `double` to an `int` we have to think a bit about *how* we want to do this. A naive typecast `(int)double` conversion will *truncate* the `double`, disregarding anything past the decimal (the fractional part) in a base-10 representation; a cast `(int)round(double)` conversion will round up or down based on the fractional part, obtaining a pure integer `double` representation, and then cast that.

The naive typecast is where we can possibly get into trouble and where we have canonical (but edge case) examples like

```
(int)(0.9999999999999999) = 0
```

Rounding isn't exactly "safe" either because

```
(int)round(0.5) = 1
(int)round(0.4999999999999999) = 0
```

While those examples sound scary, let's think it through in light of the casting derivation above. Admitting $k_0 = 0$ we have the general formula

$$\texttt{double}(\texttt{int}(i)) = (-1)^s \left(1 + \sum_{k=1}^{30-k_0} \frac{b_{31-k+k_0}}{2^k}\right) 2^{30-k_0}$$

What is this number's decimal part? If we literally multiply by the power of two, we have

$$\texttt{double}(\texttt{int}(i)) = (-1)^s \left(2^{30-k_0} + \sum_{k=1}^{30-k_0} \frac{b_{31-k+k_0}}{2^k} 2^{30-k_0}\right) = (-1)^s \left(2^{30-k_0} + \sum_{k=1}^{30-k_0} \frac{b_{31-k+k_0}}{2^{k+k_0-30}}\right)$$

We can then ask: when are the exponents in the denominator positive? For any decimal part can't come from *non-positive* (negative or zero) powers of two in the denominator, those are integers in the sum. But the denominator powers are positive if, and only if, $k > 30 - k_0$ and our sum only goes up to $30 - k_0$; so *there are no positive exponents*. Thus the sum *has no fractional part* in real number terms, being a sum of integers, and *both* truncation and rounding will give exact answers. In other very mathy words:

$$\texttt{int}(\texttt{double}(\texttt{int}(i))) = \texttt{int}(i)$$

or `int` (either truncation or rounding) is a left-inverse of `double` *on the range of* `int` in the (8B) floating-point field.

What exactly are we saying here? We just said, with our edge cases, that "`double` → `int` is noisy", but then right away we said "`double` → `int` is exact". This is not a contradiction. What we've said in our two steps is that if we convert an `int` to `double` we get an exact `int` in the double's language and can convert it right back to an `int`; we *never* get the funky edge case expansions with weird decimal place expansions that cause trouble when trying to write back to an `int`. (That's what left inverse *on the range of* means.) If we read an `int` into a `double`, we can read it right back out. That's the first thing we wanted to show, that we can at least represent amounts in cents using `double`s without *any* error.

So are those truncation and rounding rules never relevant? Of course not. Those become relevant as soon as we start doing floating point math with a `double` converted from an `int`. In math, we *cannot* be sure the following holds:

$$\texttt{int}(\ \texttt{f}(\ \texttt{double}(\texttt{int}(i))\ )\ ) = \texttt{f}(\texttt{int}(i))$$

where `f` is some program that operates on `double`s, but should map integers to integers.

3.3. **Doubles in Dollars.** Let's think of `double`s in dollars, instead of cents. In this frame we can think about some of the basic operations that might deleteriously affect precision. Given an integer i representing an amount in cents, we've shown $\texttt{double}(\texttt{int}(i))$ is exact. Now what happens when we convert to dollars by dividing by $\texttt{double}(100.0)$? How do we get that in floating point terms? What about multiplying by $\texttt{double}(100.0)$, which we'll have to do to get cents back from a value in dollars?

All the literal operations here are complicated and messy. The code `compare_conversions.c` explores when

$$\texttt{fl}\Big(100.0 \times \texttt{fl}\big(\texttt{double}(i)/100.0\big)\Big) \quad \text{is not equal to} \quad \texttt{double}(i)$$

That is, when converting into and out of dollars from cents is *byte* exact rather than just *cents* exact. (We'll address cent exactness later.) Specifically, `compare_conversions.c` examines all the positive integers up to `INT_MAX = 2,147,483,647`. For 100% of the non-negative integers representable in 32 (signed) bits, we can *round* to the correct value when converting to and from dollars; but in only 86% of the conversions $\texttt{fl}(100.0 \times \texttt{fl}(i/100.0))$ result in the same *bytes* in the resulting `double` as the (exact) `int` originally cast into a double.

One easy example where there is a deviation is 7: The literal `int` conversion is

```
(d) 7.0000000000000000:
    sign... 0
    exp.... 1025 (00000100|00000001)
    bits... 11000000|00000000|00000000|00000000|00000000|00000000
    full... 0,1000000|0001,1100|00000000|00000000|00000000|00000000|00000000|00000000
```

but $\texttt{fl}(100.0\texttt{fl}(7.0/100.0))$ is

```
(d) 7.0000000000000009:
    sign... 0
    exp.... 1025 (00000100|00000001)
    bits... 11000000|00000000|00000000|00000000|00000000|00000000
    full... 0,1000000|0001,1100|00000000|00000000|00000000|00000000|00000000|00000001
```

Note the literal bit mismatch in the least significant bit. This is inconsequential from the perspective of a valid cents place value using rounding or truncation, but shows that even unit conversions can introduce *some* error.

4. Real Analysis (ha ha)

The examples above aren't real analysis, in the sense that they are only examples. Can we do any analysis?

4.1. **Two Bounds.**

There's a great, but very dense, book on floating point errors and numerous mathematical methods called Accuracy and Stability of Numerical Algorithms by Nick Higham. Chapter 2 presents a model of relative errors with floating point arithmetic:

$$\texttt{fl}(x \texttt{ op } y) = (x \texttt{ op } y)(1+\delta) \qquad |\delta| \leq u \approx 10^{-16} \qquad \texttt{op} = +, -, \times, /$$

Here u is the *unit roundoff* ("the gap between numbers" from 30,000 feet). For IEEE floating point `double`s, $u \approx 10^{-16}$ (see Eqn. 4.4 on page 82 or `float.h`). This model holds for IEEE 754 math, and is very useful. Chapter 4 covers summation, and for naive "sequential" summation (as implemented in `draft.c`, but somewhat oddly called "recursive" summation in Higham's book) presents a weak error bound

$$\left| \sum_n x_n - \texttt{fl}\left(\sum_n x_n \right) \right| = |E_N| \leq (N-1)u \sum_n |x_n| + O(u^2)$$

where `fl` is the floating-point version of the summation, E_N is the error from summing N numbers x_n and u is again the *unit roundoff*. Note that because it references only the *signs* of the numbers summed, this bound applies to addition, subtraction, or any mix of the two for the same summand magnitudes.

Oh: and if you haven't heard the term "weak" before in reference to a bound, it doesn't mean "sometimes applies" but rather means "probably larger above the actual value across the relevant cases than we hope we could get." A "weak" bound is a bound, just one that mathy folks might expect improvement on or be generally dissatisfied with. Indeed, this bound can be improved with more specific summation methods, like the *pairwise summation* we consider later, but we'll still take a look at what it says about computing accuracy.

First, though, let's actually address storing `double`s as dollars, where we can use the first model and bound.

4.2. **Doubles in Dollars, again.** So let's look at that first bound from above and conversions to and from dollars. We start with multiplication by 100. It's good to practice here anyway. Literally, the bound says

$$\texttt{fl}(100x) = 100x(1+\delta) \qquad |\delta| \leq u \approx 10^{-16}$$

Let's rearrange to get the magnitude of the literal error on one side:

$$\begin{aligned} \texttt{fl}(100x) &= 100x(1+\delta) = 100x + 100x\delta \\ \texttt{fl}(100x) - 100x &= 100x\delta \\ |\texttt{fl}(100x) - 100x| &= 100|x||\delta| \leq |x|10^{-14} \end{aligned}$$

We can do the same thing for division and obtain:

$$|\texttt{fl}(x/100) - x/100| = |x||\delta|/100 \leq |x|10^{-18}$$

Now, these are both *relative* bounds, in the sense that $|x|$ itself appears in the bound.

However we can scope out what sort of values are reasonable here. Let's say we're multiplying a `double` that represents dollars by 100 to get it's cents value. To get appreciable errors (in cents) we would have to have a bound of order $O(1)$, which means $x \sim 10^{14}$! We're never going to work with financial numbers so large on a daily basis. For an amount like 1,000,000,000 ($10,000,000) we still have about 5 digits of accuracy in the sense that

$$|\texttt{fl}(100x) - 100x| \leq 10^{-5}$$

While we still have to exercise some care with rounding vs truncation, the `double` cents value of a `double` in dollars is going to be exceptionally precise even for very large amounts.

What about dividing? It should be obvious from the bound that we have even less to worry about here. We will be dividing `double` values in cents by 100 to get `double` values in dollars. We can even convert a cent-Bezos ($x \sim O(10^{13})$) to dollars without an appreciable error (the bound is still 10^{-5}). For $10,000,000 we have about 11 digits of accuracy:

$$|\texttt{fl}(x/100) - x/100| \leq 10^{-11}$$

So these bounds are for the conversions alone, not the actual process of conversion, accumulation of error through repeated operations, and re-conversion. We might model this as

$$x \mapsto x/100 \mapsto F(x/100) \mapsto 100F(x/100)$$

The full scope is complicated enough that it's not worth a detailed analysis, but rather exploration through examples. We can analyze conversion into dollars and re-conversion to cents:

$$\begin{aligned} \texttt{fl}(100 \times \texttt{fl}(x/100)) &= \texttt{fl}(100 \times (x(1+\delta_1)/100)) = (100x/100)(1+\delta_1)(1+\delta_2) \\ &= x(1+\delta_1+\delta_2+\delta_1\delta_2) \approx x(1+2\delta) \end{aligned}$$

where $|\delta| \leq u$, of course. So we get more error, but not appreciably more error because the $\delta = O(u)$ factor is simply doubled.

We could *assume* a relative bound on the error of the programmatic implementation of a mapping F and get a bit further. For example, suppose that $F(x/100(1+\delta)) \approx F(x/100)(1+\delta)$ (relative errors on the order of the unit roundoff in the inputs to F transfer to the same order relative errors in it's values), and that $\texttt{F}(x) = F(x)(1+\eta)$ for some relative error η (which we won't bound yet), where `F` is the prgrammed version of F. Then

$$\begin{aligned}
\texttt{fl}\Big(100 \times \texttt{F}\big(\texttt{fl}(x/100)\big)\Big) &= \texttt{fl}\Big(100 \times \texttt{F}\big(x/100(1+\delta_1)\big)\Big) \\
&= \texttt{fl}\Big(100 \times F\big(x/100(1+\delta_1)\big)(1+\eta)\Big) \\
&= \texttt{fl}\Big(100 \times F\big(x/100\big)(1+\delta_1)(1+\eta)\Big) \\
&= 100F\big(x/100\big)(1+\delta_2)(1+\eta)(1+\delta_1) \\
&\approx 100F\big(x/100\big)(1+2\delta)(1+\eta) \\
&\approx 100F\big(x/100\big)(1+O(|\eta|))
\end{aligned}$$

Note we've reduced this to a leading order bound in the magnitude of η. Very many repeated, possibly noisy operations on the input could greatly increase the error in the result. But it has relatively nothing to do with the conversions into or out of "dollars"; it's all related to the noisy operations in between.

Even that isn't a complete model though. Really we are considering

$$x_1, \ldots, x_N \mapsto x_1/100, \ldots, x_N/100 \mapsto F(x_1/100, \ldots, x_N/100) \mapsto 100F(x_1/100, \ldots, x_N/100)$$

where it might not make sense to try to describe the relative error bounds in a programmatic implementation `F` of F. Here, as shown below for sums, the accumulation of tiny errors in each of the converted inputs can be significant in greatly increasing errors past u, but maybe not so great as to mean anything for out units of concern.

4.3. **Sums: A Worst-Case, Sufficient Analysis.** Ok, now we can consider the second bound for sums of floating point numbers. Our `double` calculations with sequential summation are *guaranteed* to be accurate *to the cents place* if

$$|E_N| < 0.5 = 5 \times 10^{-1} \text{ in cents} \qquad |E_N| < 0.005 = 5 \times 10^{-3} \text{ in dollars}$$

where "in cents" or "in dollars" refers to the units used in storing the amounts in `double`s.

Let's pause and make sure this is correct. We know

$$\texttt{fl}\left(\sum_n x_n\right) = \sum_n x_n + E_N$$

(This is just the definition of E_N.) We presume every x_n (and thus $\sum_n x_n$) is an integer, and will just round the left value when dealing with `double`s in cent units. Any decimal part of $\texttt{fl}\left(\sum_n x_n\right)$ comes from the error E_N. We also expect these errors to be small, just

decimals (or we're in *really* bad shape). If $|E_N| < 0.5$, then

$$\sum_n x_n - 0.5 < \sum_n x_n + E_N < \sum_n x_n + 0.5$$

and thus

$$\texttt{round}\left(\texttt{fl}\left(\sum_n x_n\right)\right) = \texttt{round}\left(\sum_n x_n + E_N\right) = \sum_n x_n$$

by the normal rounding rule "up if the decimal is ≥ 0.5".

If we have dollar-`double`s, we actually round

$$100 \times \texttt{fl}\left(\sum_n \left(\frac{x_n}{100}\right)\right) = 100\sum_n \left(\frac{x_n}{100}\right) + 100E_N = \sum_n x_n + 100E_N$$

So if $100E_N < 0.5$, that is $E_N < 5 \times 10^{-3}$, then again

$$\sum_n x_n - 0.5 < \sum_n x_n + 100E_N < \sum_n x_n + 0.5$$

and the rounded value is, again, correct.

Returning to our analysis, *sufficient* (but probably not even close to necessary) conditions are

$$(N-1)u\sum_n |x_n| < 5 \times 10^{-1} \text{ (cents)} \qquad (N-1)u\sum_n |x_n| < 5 \times 10^{-3} \text{ (dollars)}$$

(dropping the $O(u^2)$ "remainder", as it won't be relevant). We've chosen numbers x_n such that, in dollars (because we want the `double` decimals accurate to cents), satisfy $|x_n| \leq 10^5$. Then, compounding our very sufficient condition further with other loose bounds, we have

$$\begin{aligned} \text{cents:} \quad & (N-1)u\sum_n |x_n| \leq 10^7(N-1)Nu \leq 10^{-9}N^2 \\ \text{dollars:} \quad & (N-1)u\sum_n |x_n| \leq 10^5(N-1)Nu \leq 10^{-11}N^2 \end{aligned}$$

Obviously if the right-hand sides are small enough, we have a bound on the error guaranteeing correctness:

$$\begin{aligned} \text{cents:} \quad & 10^{-9}N^2 < 5 \times 10^{-1} \iff N < \sqrt{5} \times 10^4 \approx 22360 \\ \text{dollars:} \quad & 10^{-11}N^2 < 5 \times 10^{-3} \iff N < \sqrt{5} \times 10^4 \approx 22360 \end{aligned}$$

At least in this analysis, our factors of 100 cancel and we get the same result.

The result is that, even in this loosest of worst case analyses with the simplest summation strategy, the (IEEE) `double` calculation is *guaranteed* to be cent-accurate for sums of less than $22,360$ items of the magnitude we assume. We assume pretty large magnitude

amounts, on average; reducing this size to a more reasonable level like \$100 or \$1,000 would increase N. Specifically, if the largest x_n is like $M = 10^m$ in dollars,

$$\begin{aligned} \text{cents:} \quad & (N-1)u\sum_n |x_n| \le 10^{-(14-m)}N^2 \\ \text{dollars:} \quad & (N-1)u\sum_n |x_n| \le 10^{-(16-m)}N^2 \end{aligned}$$

and our sufficient condition is

$$\begin{aligned} \text{cents:} \quad & 10^{-(14-m)}N^2 < 5\times 10^{-1} \quad \iff \quad N < \sqrt{5}\times 10^{6.5-m/2} \\ \text{dollars:} \quad & 10^{-(16-m)}N^2 < 5\times 10^{-3} \quad \iff \quad N < \sqrt{5}\times 10^{6.5-m/2} \end{aligned}$$

For $m = 3$ and amounts below \$1,000, these bounds are about $N < 223,606$; for $m = 2$, $N < 2,236,067$.

While these bounds are useful guarantees, they are at least an order of magnitude below the level of correctness we see in our actual experiments. Table 1 shows that in the uniform model our sums with doubles in cents were correct even with 10M amounts, whereas with doubles in dollars correct up to 100k amounts (and surely higher, though not 1M). For the 3-bin model, $m = 3$ and we don't know how suboptimal the bound of $N < 223,606$ here is, as doubles in both units are correct through 10M amounts (Table 2).

4.4. **A Technicality.** Technically we haven't included the errors in converting to dollars. If we want to handwave around this, we can make an argument perhaps like the following: The error we care about is most like

$$\begin{aligned} E_N &= \left| \sum_n \frac{c_n}{100} - \texttt{fl}\left(\sum_n \texttt{fl}\left(\frac{c_n}{100}\right)\right) \right| \\ &= \left| \sum_n \frac{c_n}{100} - \texttt{fl}\left(\sum_n \frac{c_n}{100}(1+\delta_n)\right) \right| \end{aligned}$$

(We should probably also add in the `float` conversions make to cents.) This is not what is bounded in the sum above, but we can turn it into something like that as follows:

$$\begin{aligned}
\left| |E_N| - \left| \sum_n \frac{c_n}{100}\delta_n \right| \right| &\le \left| E_N - \left(-\sum_n \frac{c_n}{100}\delta_n \right) \right| = \left| E_N + \sum_n \frac{c_n}{100}\delta_n \right| \\
&= \left| \sum_n \frac{c_n}{100}(1+\delta_n) - \texttt{fl}\left(\sum_n \frac{c_n}{100}(1+\delta_n) \right) \right| \\
&\le (N-1)u \sum_n \left| \frac{c_n}{100} \right| |1+\delta_n| \\
&\le (N-1)u \sum_n \left| \frac{c_n}{100} \right| + (N-1)u \sum_n \left| \frac{c_n}{100} \right| |\delta_n| \\
&= (N-1)u \left(1 + \sum_n w_n |\delta_n| \right) \sum_n \left| \frac{c_n}{100} \right| \qquad w_n = \frac{|c_n/100|}{\sum_m |c_m/100|} \\
&= (N-1)u \sum_n \left| \frac{c_n}{100} \right| + O(u^2) \\
&\approx (N-1)u \sum_n \left| \frac{c_n}{100} \right|
\end{aligned}$$

What this says about the error magnitude $|E_N|$ we care about is

$$|E_N| = \left| \sum_n \frac{c_n}{100}\delta_n \right| + \triangle \qquad |\triangle| \le (N-1)u \sum_n \left| \frac{c_n}{100} \right|$$

Going way out on a limb, let's say

$$|E_N| = 10^{-(18 - \log_{10} N - \log_{10} \mathbb{E}c)} + \triangle \qquad |\triangle| \le 10^{-(18 - \log_{10} \mathbb{E}c - 2\log_{10} N)}$$

The dominant term is still $|\triangle|$, because of the factor $2\log_{10} N$.

For example, in our code $\log_{10} \mathbb{E}c = 6$ (which is actually kind of huge), and

$$|E_N| = 10^{-(12 - \log_{10} N)} + \triangle \qquad |\triangle| \le 10^{-2(6 - \log_{10} N)}$$

Then if $\log_{10} N \sim 5$, we have

$$|E_N| = 10^{-7} + \triangle \qquad |\triangle| \le 10^{-2}$$

4.5. **Improving the Bounds.** Standard sample mean results for i.i.d. variables like the Law of Large Numbers, Central Limit Theorem, Markov's Inequality, and possibly even Large Deviations theory can improve our bounds, but possibly not by much. We can derive some improvements with constant factor gaps, but not improvement in orders of magnitude. We give one example here.

The LLN says that $\bar{X}_N \to \mathbb{E}X$ where X is the random variable from which we draw amounts. Then

$$|E_N| \le N(N-1)u\bar{X}_N \to N(N-1)u\mathbb{E}X \approx N^2 u \mathbb{E}X$$

and our sufficient bounding exercise comes down to

$$N < \frac{\sqrt{5} \times 10^{7.5}}{\sqrt{\mathbb{E}X}} \approx \frac{2.361 \times 10^{7.5}}{\sqrt{\mathbb{E}X}} = 2.361 \times 10^{7.5 - \log_{10} \mathbb{E}X/2}$$

for X in cents. For our uniform model, $\mathbb{E}X = 5 \times 10^6$, $\sqrt{\mathbb{E}X} = \sqrt{5} \times 10^3$, and thus

$$N < 10^{4.5} \approx 31,623$$

suffices for an accuracy guarantee allowing 50% more amounts. Even this, though, is far below what we see in the experiments for doubles in cents. For our 3-bin model, $\mathbb{E}X = 10,425$, $\sqrt{\mathbb{E}X} \approx 102.1$, and

$$N < \frac{\sqrt{5} \times 10^{7.5}}{102.1} \approx 692,563$$

While improved, this bound is again well below the correctness we see in the experiments.

4.6. **Pairwise Summation.** There are other ways to sum a bunch of numbers, one of them being *pairwise summation* (Chapter 4 of Higham again). Basically, sum pairs of numbers, then sum those sums, then sum *those* sums, and so on until the summation is done. Pairwise summation is straightforward to implement with recursion:

```
void pairsums( int N , double * x )
{
    if( N == 1 ) { return; }
    if( N == 2 ) { x[0] += x[1]; return; }
    for( int n = 0 ; n < N/2 ; n++ ) x[n] = x[2*n] + x[2*n+1];
    if( N % 2 == 1 ) { x[N/2-1] += x[N-1]; }
    pairsums(N/2, x);
}
```

This function will overwrite `x` with intermediate results and the top level call returns with the total sum in `x[0]`.

This method has a much tighter error bound:

$$|E_N| \leq \frac{\log_2 N \; u}{1 - \log_2 N \; u} \sum_n |x_n|$$

The $\log_2 N$ factor immediately suggests much smaller errors. But let's suppose $\log_2 N \; u \approx 0$ (which is fair for any reasonable N we could use) and redo our worst-case analysis:

$$\frac{\log_2 N \; u}{1 - \log_2 N \; u} \sum_n |x_n| \approx \log_2 N \; u \sum_n |x_n| \leq 10^5 N \log_2 N \; u = \frac{N \log_2 N}{10^{11}}$$

For what values of N is the RHS guaranteed to be less than 0.005 (focusing on dollars, since it'll work out the same anyway)? Clearly

$$\frac{N \log_2 N}{10^{11}} < 5 \times 10^{-3} \quad \iff \quad N \log_2 N < 5 \times 10^8$$

With some transformations (log transforms and Newton's method), we can estimate

$$N \log_2 N < 5 \times 10^8 \quad \iff \quad N < 20,580,553$$

So if N (the number of amounts to sum) is under about 20 million, we can use pairwise summation and rest assured that the value obtained using `double`s will be correct to the cents place.

The scaling here is much different than for N^2 though. Generalizing to $|x_n| \leq 10^m$ (in dollars),

$$\log_2 N \; u \sum_n |x_n| \leq N \log_2 N 10^{-(16-m)} < 5 \times 10^{-3} \quad \iff \quad N \log_2 N < 5 \times 10^{13-m}$$

which for $m = 3$ is about $N < 1,633,686,624$. That is, for amounts not less than thousands (probably also "typically" thousands) of dollars, N can be as much as a billion and a half and still be *guaranteed* accuracy with pairwise summation.

Let's try it! The code `pairwise.c` has an implementation of the pairwise summation strategy, but is simpler in that we ignore `int`s. Otherwise we can generate and analyze results in the same way. We include `float`s to examine if they perform any better. Results with the uniform model are listed in Table 5, and with the 3-bin model in Table 6. As predicted, pairwise summation has improved the correctness of using `double`s to represent amounts, whether in cent or dollar units. Even with sums of 10M values, `double`s give the correct results when rounded to the cents place. While `float` performance is improved for the 3-bin model (recall Table 2), it is still not good enough to consider usable.

5. A Checkpoint Strategy

Actually, a guarantee of accuracy to cents of accumulations of *any* number of amounts N *may* give us a guarantee for *every* accumulation size, if we appropriately checkpoint our calculations. Technically what that means, for amounts in cents, is the following: Suppose that for some $\bar{N}_X > 0$, where the subscript X denotes dependence on the random amounts, we have a guarantee that

$$\texttt{round}\left(\texttt{fl}\left(\sum_{n=1}^{N} \texttt{double}(x_n)\right)\right) = \sum_{n=1}^{N} \texttt{long}(x_n) \quad \text{for all} \quad N \leq \bar{N}_X$$

Then is there an algorithm "$\texttt{sum}_{N,X}$" computing the sum of *any* N amounts x_n (as `double`s) such that

$$\texttt{round}\left(\texttt{sum}_{N,X}\left(x_1, \ldots, x_N\right)\right) = \sum_{n=1}^{N} \texttt{long}(x_n) \quad \text{for all} \quad N?$$

Technically, we avoid the possibility of overflow here for simplicity, but that would certainly come into play if this idea was taken to it's extremes.

The first step of such an algorithm is easy to envision, and seems like an extended version of pairwise summation. Let $N \in \mathbb{N}$ and $\bar{N}_X$ be an accuracy-assuring bound, obviously with $N > \bar{N}_X$ or algorithm is just summing up the amounts. Let $B = \lfloor N/\bar{N}_X \rfloor$ and $R = N$

TABLE 5. Accuracy of pairwise accumulation of N amounts for `float` and `double` data in dollars and cents. Amounts are the mean (μ) and standard deviation (σ) of an indicator variable for correctness over 10,000 independent random trials, expressed as percents. Amounts accumulated are drawn according to the uniform model.

	`float` (c)		`float` ($)		`double` (c)		`double` ($)	
op	μ	σ	μ	σ	μ	σ	μ	σ
$N = 100$								
+	2.04	14.14	2.00	14.00	100.00	0.00	100.00	0.00
-	2.04	14.14	2.00	14.00	100.00	0.00	100.00	0.00
±	12.03	32.53	9.87	29.83	100.00	0.00	100.00	0.00
$N = 1,000$								
+	0.12	3.46	0.21	4.58	100.00	0.00	100.00	0.00
-	0.12	3.46	0.21	4.58	100.00	0.00	100.00	0.00
±	2.72	16.27	2.63	16.00	100.00	0.00	100.00	0.00
$N = 10,000$								
+	0.04	2.00	0.00	0.00	100.00	0.00	100.00	0.00
-	0.04	2.00	0.00	0.00	100.00	0.00	100.00	0.00
±	0.83	9.07	0.75	8.63	100.00	0.00	100.00	0.00
$N = 100,000$								
+	0.00	0.00	0.00	0.00	100.00	0.00	100.00	0.00
-	0.00	0.00	0.00	0.00	100.00	0.00	100.00	0.00
±	0.24	4.89	0.15	3.87	100.00	0.00	100.00	0.00
$N = 1,000,000$								
+	0.00	0.00	0.00	0.00	100.00	0.00	100.00	0.00
-	0.00	0.00	0.00	0.00	100.00	0.00	100.00	0.00
±	0.06	2.45	0.01	1.00	100.00	0.00	100.00	0.00
$N = 10,000,000$								
+	0.00	0.00	0.00	0.00	100.00	0.00	100.00	0.00
-	0.00	0.00	0.00	0.00	100.00	0.00	100.00	0.00
±	0.02	1.41	0.01	1.00	100.00	0.00	100.00	0.00

mod $\bar{N}_X$. We can break the amounts up into $B + 1$ (or just B, if $R = 0$) "blocks" with

TABLE 6. Accuracy of pairwise accumulation of N amounts for `float` and `double` data in dollars and cents. Amounts are the mean (μ) and standard deviation (σ) of an indicator variable for correctness over 10,000 independent random trials, expressed as percents. Amounts accumulated are drawn according to the simple 3-bin model.

	`float` (c)		`float` ($)		`double` (c)		`double` ($)	
op	μ	σ	μ	σ	μ	σ	μ	σ
$N = 100$								
+	100.00	0.00	100.00	0.00	100.00	0.00	100.00	0.00
-	100.00	0.00	100.00	0.00	100.00	0.00	100.00	0.00
±	100.00	0.00	100.00	0.00	100.00	0.00	100.00	0.00
$N = 1,000$								
+	100.00	0.00	81.65	38.71	100.00	0.00	100.00	0.00
-	100.00	0.00	81.65	38.71	100.00	0.00	100.00	0.00
±	100.00	0.00	100.00	0.00	100.00	0.00	100.00	0.00
$N = 10,000$								
+	9.65	0.00	9.50	29.32	100.00	0.00	100.00	0.00
-	9.65	0.00	9.50	29.32	100.00	0.00	100.00	0.00
±	100.00	0.00	95.99	19.62	100.00	0.00	100.00	0.00
$N = 100,000$								
+	0.93	9.60	0.81	8.96	100.00	0.00	100.00	0.00
-	0.93	9.60	0.81	8.96	100.00	0.00	100.00	0.00
±	98.29	12.97	45.28	49.78	100.00	0.00	100.00	0.00
$N = 1,000,000$								
+	0.09	2.97	0.09	2.97	100.00	0.00	100.00	0.00
-	0.09	2.97	0.09	2.97	100.00	0.00	100.00	0.00
±	49.14	50.00	13.36	34.02	100.00	0.00	100.00	0.00
$N = 10,000,000$								
+	0.02	1.41	0.04	2.00	100.00	0.00	100.00	0.00
-	0.02	1.41	0.04	2.00	100.00	0.00	100.00	0.00
±	7.98	27.10	3.55	18.50	100.00	0.00	100.00	0.00

sums

$$S_b = \texttt{round}\left(\texttt{fl}\left(\sum_{n=1}^{\bar{N}} \texttt{double}(x_{\bar{N}(b-1)+n})\right)\right) \qquad b = 1, \ldots, B$$

$$S_{B+1} = \begin{cases} \texttt{round}\left(\texttt{fl}\left(\sum_{n=1}^{R} \texttt{double}(x_{\bar{N}B+n})\right)\right) & \text{if } R > 0 \\ 0 & \text{if } R = 0 \end{cases}$$

guaranteed to be accurate to the cents place. Probably obviously, we aim to let these sums define a *new* set of amounts by casting to `long`s and back to `float`s or `double`s without loss as they are exact to the cents place, and repeat. That is: if $B+1 > \bar{N}$ (or $B > \bar{N}$ if $B \mod \bar{N} = 0$), break them up again, and compute a new set of sums exact to the cents place; if $B+1 < \bar{N}$ (or $B < \bar{N}$ if $B \mod \bar{N} = 0$), just sum them up.

The problem is that the set of block sums $S_1, \ldots, S_{B+1}$ does not have the same distribution as X, and thus will not have the same bound $\bar{N}$.

Requiring this sort of procedure with `double`s and pairwise summation is probably pointless. They're just already guaranteed to be accurate for plausible accumulation sizes, using pairwise summation at least.

Recall that the LLN says that $\bar{X}_N \to \mathbb{E}X$ where X is the random variable from which we draw amounts. The sequential summation error bound is more or less

$$|E_N| \leq N_0(N_0-1)u\bar{X}_N \to N_0(N_0-1)u\mathbb{E}X \approx N_0^2 u\mu$$

and our sufficient bounding implies

$$\bar{N}_0 \approx \sqrt{\frac{5\times 10^{15}}{\mu}} = \frac{\sqrt{5}\times 10^{15/2}}{\sqrt{\mu}}$$

We're clearly adding a "0" subscript here to start an iteration.

We can apply the same logic to the $N_1 \in \{B, B+1\}$ first stage block sums, and obtain

$$|E_N| \leq N_1(N_1-1)u\bar{S}_N^{(1)} \to N_1(N_1-1)u\mathbb{E}S^{(1)} \approx N_1^2 u\mathbb{E}S^{(1)}$$

Sufficient bounding implies

$$\bar{N}_1 \approx \sqrt{\frac{5\times 10^{15}}{\mathbb{E}S^{(1)}}} = \frac{\sqrt{5}\times 10^{15/2}}{\sqrt{\mathbb{E}S^{(1)}}}$$

The first stage sums are distributed like sums of $\bar{N}_0$ i.i.d. amounts:

$$S^{(1)} \sim \sum_{n=1}^{\bar{N}_0} X_n \qquad \Longrightarrow \qquad \mathbb{E}S^{(1)} = \sum_{n=1}^{\bar{N}_0} \mathbb{E}X_n = \bar{N}_0\mu$$

Thus

$$\bar{N}_1 \approx \frac{\sqrt{5}\times 10^{15/2}}{\sqrt{\bar{N}_0}\sqrt{\mu}} = \frac{\bar{N}_0}{\sqrt{\bar{N}_0}} = \sqrt{\bar{N}_0} = \left(\frac{5\times 10^{15}}{\mu}\right)^{1/4}$$

In other words, given an original bound $\bar{N}_0$, we should use a bound for the sums of sums of $\sqrt{\bar{N}_0}$.

Let's keep going, as the induction is pretty clear. The second stage sums (sums of sums), if any, are distributed like sums of $\bar{N}_1$ i.i.d. first-stage sums

$$S^{(2)} \sim \sum_{n=1}^{\bar{N}_1} S_n^{(1)} \qquad \Longrightarrow \qquad \mathbb{E}S^{(2)} = \sum_{n=1}^{\bar{N}_1} \mathbb{E}S_n^{(1)} = \bar{N}_1\mathbb{E}S^{(1)}$$

and thus have a sufficient bound

$$\bar{N}_2 \approx \left(\frac{5 \times 10^{15}}{\mathbb{E}S^{(2)}}\right)^{1/2} = \sqrt{\frac{5 \times 10^{15}}{\bar{N}_1 \mathbb{E}S^{(1)}}} = \frac{\bar{N}_1}{\sqrt{\bar{N}_1}} = \sqrt{\bar{N}_1} = \left(\frac{5 \times 10^{15}}{\mu}\right)^{1/8}$$

This is probably enough to perceive the pattern: In a recursion where we compute block sums, then block sums of those, and so on, we should take the (floor of) the square root of the bound passed to a given level as the bound applicable to the next level.

All this is, of course, trusting that we can use the LLN to approximate sample averages by the underlying mean. And not just for the amounts, but for all their blocked sums too.

6. Learnings

What have we learned?

Mainly, I think, we've learned that 4 bytes is not enough. `int`s and `float`s perform rather poorly in accurate sums for amounts following our distribution. If we have to save space, `int`s are better than `float`s but neither has the stability required for decent size accumulations.

However using 8 byte integers (long `int`s) representing cents give us perfect answers for addition and subtraction because of the sheer size of their range relative to the calculations we probably need to perform, particularly sums of (possibly signed) integers representing monetary values (in cents).

`double`s, also 8 bytes but subject to roundoff error, are also probably fine too. We saw no errors in tests using `double`s pegged to cents, but some errors for "very large" sums of (millions of) uniformly-signed amounts pegged to dollars, but not for mixed sign sums. Moreover, if we trust the gurus of numerical analysis in the presence of round-off errors (which we should), there are actual theoretical guarantees of the correctness of using `double`s for tens of thousands of summands with naive summation code and at least tens of millions of summands, if not billions, with smarter code. Both theoretical conclusions are justified by numerical experiments.

Senior Software Engineer @ Bond Financial Technologies, San Francisco, CA
Email address: ross@bond.tech