An Overview of Computer Number Representations and Error In Computation

Brett Saiki
University of Washington
Math 336

May 2021

1 Introduction

Mathematicians with little or no experience with computer science concepts may naively assume that computers can do exactly what their name implies: they can accurately compute a numerical result given any formula. Unfortunately, this belief is not true; modern computers excel at fast computation, but they are not necessarily accurate. Just by analyzing a couple of examples, it'll become apparent how common accuracy issues can be.

$$f(x) = \sqrt{x+1} - \sqrt{x} \tag{1}$$

Consider Equation (1). For simple values of x, say $x = 1$, a computer does well; Python reports 0.4142135623730951 — every digit but the last is correct. However, if we choose $x = 10^{16}$, we get 0 instead of the expected result 5×10^{-9}.

$$f(x, y) = 9x^4 - y^4 + 2y^2 \tag{2}$$

Now consider Equation (2). For, $x = 1$ and $y = 1$, Python reports 10.0 as expected. But for $x = 10864.0$ and $y = 18817.0$, we get 2.0 instead of the expected 1.0.

What exactly is going on here? From a mathematicians's point of view, these results might seem bearable; the relative error is miniscule in the first example, although much larger in the second example. But once we understand how numbers are stored on computers, we will understand that there were many bad answers that Python could have given that would have been a considerable improvement in both cases.

To fully explain the causes behind these numerical errors, this paper will cover three concepts: how real numbers are represented on hardware; numerical error due to computer number representations; and techniques and programs that mitigate error when computing numerical results.

2 Representing Numbers on Hardware

There are two main ways of representing real numbers on conventional hardware: *fixed-point* and *floating-point.* Fixed point consists of integer representations and fraction representations Floating point is used for most non-integral valued numerical evaluation.

2.1 Integers

Before diving into how integers are represented on hardware, we will begin in the familiar worlds of decimal numbers. First, consider a range of integers, say 0 to $10^n - 1$. We will call this set $\mathbb{D}_n = \{x \in \mathbb{Z} : 0 \leq x < 10^n\}$.

What can we say about $\mathbb{D}_n$? The set contains all the non-negative integers that can be represented by at most n digits; there are 10^n integers in total. Unlike the set of integers, $\mathbb{D}_n$ is not closed under addition, subtraction, or multiplication; it is easy to come up with examples for each operation. For these operators, we can produce integers outside of $\mathbb{D}_n$. Unsuprisingly, $\mathbb{D}_n$ is also not closed under division.

Our set $\mathbb{D}_n$ has few useful properties; however, we could re-define the arithmetic operators so that $\mathbb{D}_n$ is closed under them. Let $W_n : \mathbb{Z} \to \mathbb{D}_n$ be the function such that $W_n(x) = x \bmod 10^n$. Then, we can define addition, subtraction, and multiplication by composing their usual definitions with W_n,

$$x +^{\mathbb{D}_n} y = W_n(x + y) \tag{3}$$

$$x -^{\mathbb{D}_n} y = W_n(x - y) \tag{4}$$

$$x \times^{\mathbb{D}_n} y = W_n(x - y) \tag{5}$$

These operators can be descibed succinctly. They are modular and obey all the usual rules of modular arithmetic. For $y \neq 0$, we define division to be

$$x \div^{\mathbb{D}_n} y = \lfloor x/y \rfloor, \tag{6}$$

the *truncation* of the expected rational result,

But what if we want to express negative integers as well? Let $\mathbb{D}_n^*$ be the set of integers containing negative integers corresponding to $\mathbb{D}_n$. We simply translate the endpoints of $\mathbb{D}_n$ so that 0 is somewhere near the center. Since there are an even number of integers in $\mathbb{D}_n^*$, we will arbitrarily choose to have one more negative number than positive number. That is, $\mathbb{D}_n^* = \{x \in \mathbb{Z} : -10^n/2 \leq x < 10^n/2\}$. The function $W_n^* : \mathbb{Z} \to \mathbb{D}_n^*$ is defined to be

$$W_n^*(x) = [(x + 10^n/2) \bmod 10^n] - 10^n/2 \tag{7}$$

Geometrically, W_n^* is essentially the same function as W_n, i.e. it wraps the integers around such that the largest integer plus one is the smallest integer. Then we can define the four arithmetic operators for $\mathbb{D}_n^*$ to be similar to the ones for $\mathbb{D}_n$, replacing W_n with W_n^*.

So far, we have been describing the rules of integer arithmetic on computers by considering their decimal counterparts. To understand how computers use integers, we only need to change our base. Integers are stored using base-2 instead of base-10, so we define the non-negative set of integers requiring n binary digits to be $\mathbb{B}_n = \{x \in \mathbb{Z} : 0 \leq x < 2^n\}$ and the corresponding set including negative values to be $\mathbb{B}_n^* = \{x \in \mathbb{Z} : -2^{n-1} \leq x < 2^{n-1}\}$. Every other operator is defined using the bounds used above.

We will close out this section by mentioning the terminology often used to describe these sets and operations. The sets $\mathbb{B}_n$ and $\mathbb{B}_n^*$ are called the *n-bit unsigned integers* and the *n-bit signed integers*, respectively. Modern 64-bit computers define 4 widths of integers: 8-, 16-, 32-, and 64-bit integers, both signed and unsigned. The condition $x \neq W_n^{\mathbb{B}}(x)$ is called *overflow*. While this condition is useful in modular arithmetic, computer scientists try to avoid overflow since it can lead to serious problems such as buffer overflow, where a program will read or write data it shouldn't be touching. Thus, addition, subtraction, and multiplication are only really useful on a subset of the domain $\mathbb{B}_n \times \mathbb{B}_n$ where overflow does not occur.

2.2 Fixed point

While integer representations are technically fixed-point representations, fixed-point generally refers to the fixed-point representations that aren't integer representations. Fixed point representations are not as universal as integers but are important for low accuracy graphics code and obscure bit twiddling tricks.

We define $\mathbb{P}_{s,n} = \{x \in \mathbb{B}_n : x \times 2^s\}$ and $\mathbb{P}_{s,n}^* = \{x \in \mathbb{B}_n^* : x \times 2^s\}$ to be the unsigned and signed *n-bit fixed-point numbers with scale* 2^s where $s \in \mathbb{Z}$. Notice that each $\mathbb{P}_{s,n}$ has the same number of elements $\mathbb{B}_n$ (respectively $\mathbb{P}_{s,n}^*$ and $\mathbb{B}_n^*$), but each element is separated by 2^s. Moreover, $\mathbb{B}_n$ is just $\mathbb{P}_{0,n}$ (respectively $\mathbb{B}_n^*$ is $\mathbb{P}_{0,n}$) which is why integers belong to the fixed-point family. It may be easier to think of the fixed-point numbers as the set of numbers uniquely defined by n digits but whose binary point (eqv. decimal point) is no longer fixed after the last digit.

The fixed-point representations have the same properties as the integers. Addition, subtraction, and multiplication are exact unless overflow occurs, that is, adding numbers in a fixed-point set yield results in the same fixed-point set. Most importantly, we can store n-bit fixed-point numbers as n-bit integers and simply keep track of the scale separately (we say the scale is *implicit*). Then we can implement the arithmetic operators for $\mathbb{P}_{s,n} \times \mathbb{P}_{s,n}$ as follows

$$(x \times 2^s) +^{\mathbb{P}_{s,n}} (y \times 2^s) = (x +^{\mathbb{B}_n} y) \times 2^s \tag{8}$$
$$(x \times 2^s) -^{\mathbb{P}_{s,n}} (y \times 2^s) = (x -^{\mathbb{B}_n} y) \times 2^s \tag{9}$$
$$(x \times 2^s) \times^{\mathbb{P}_{s,n}} (y \times 2^s) = [(x \times^{\mathbb{B}_n} y) \times^{\mathbb{B}_n} 2^s] \times 2^s \tag{10}$$
$$(x \times 2^s) \div^{\mathbb{P}_{s,n}} (y \times 2^s) = [(x \div^{\mathbb{B}_n} y) \times^{\mathbb{B}_n} 2^{-s}] \times 2^s \tag{11}$$

Thus fixed-point operations can be implemented using integer arithmetic with shifting (multiplying by 2^m is the same as shifting binary digits). Working with operands of different scaling factors requires additional scaling but we leave that exercise up to the reader.

This particular description of fixed-point numbers is not standard in computer science. Fixed point numbers are generally described as having *n integer bits and f fraction bits*, but this is equivalent to $\mathbb{P}_{n+f,-f}$. Our description of fixed-point numbers describes the relationship between $\mathbb{P}_{s,n}$ and $\mathbb{B}_n$ and is much more easily understandable. Additionally, the computer science notation of fixed-point numbers is limited since a positive s implies negative fractional bits and $s < -n$ implies negative integer bits, neither of which makes particular sense.

2.3 Floating point

The problem with fixed-point representations is we can store a limited range of values. What if we want to add 10^{40} and 3^{100}? We could of course find a large n-bit integer representation, but generally operations will slow down drastically as we increase n since we need to use special software to model integer representations (remember that modern CPUs can operate on a maximum of 64 bits). The solution is the floating-point representation standardized under the Institute of Electrical and Electronics Engineers (IEEE) Standard for Floating-Point Arithmetic in 1985. This standard is usually referred to as the IEEE-754 standard.

Before defining floating-point numbers, we must consider two important fixed-point sets. The set $\mathbb{E}_n = \mathbb{B}_n^* \setminus \{-2^{n-1}, -2^{n-1} - 1\}$ is the set of *n-bit exponent values.* Informally, we remove the two lowest values from the set of n-bit signed integers to construct $\mathbb{E}_n$ (this will be explained later). We call $P_{n,-n}$ the set of *n-bit mantissa values.* This set is simply the rational values on $[0, 1)$ of the form $m \times 2^{-n}$ where $m = 0, 1, 2, \ldots$ (they equally partition the unit interval).

Now, we can start defining floating-point numbers. We will start by defining the *normal floating-point values with n-bit mantissa and e-bit exponent* to be

$$\mathbb{F}_n^* = \{m \in \mathbb{P}_{n,-n}, s \in \mathbb{E}_e^*, b \in \{0, 1\} : (-1)^b (1 + m) \times 2^s\}$$

The value b serves to define both positive and negative values. Unlike fixed-point representations, this set is symmetric around 0. Additionally floating-point uses a variable scale of 2^s and a mutliplier of equally-spaced rationals on $[1, 2)$. You can think of floating-point numbers as numbers in scientific notation in binary represented by $n+1$ (implicit one to left of binary point) binary digits.

To define the full set of floating-point values for any m and e, we must analyze a special set of values. Let $\mathbb{F}_n'$ be the set of *denormalized floating-point values with n-bit mantissa and e-bit exponent* to be

$$\mathbb{F}_{e,n}' = \{m \in \mathbb{P}_{n,-n} \setminus \{0\}, b \in \{0, 1\} : (-1)^b m \times 2^{2^{e-1}-1}\}$$

The muliplier $2^{e-1} - 1$ is the smallest scale found in the set of normalized floating-point values. Since our mantissa is in the interval $(0, 1)$, we can conclude that denormalized values are smaller in magnitude than the normal values of a given floating-point representation.

Now, we can finally define the set of floating-point numbers. The *floating-point numbers with n-bit mantissa and e-bit exponent* is defined as

$$\mathbb{F}_{e,n} = \mathbb{F}_{e,n}^* \cup \mathbb{F}_{e,n}' \cup \{-\infty, 0, \infty\}$$

Generally, we can store a floating-point number from $\mathbb{F}_{e,n}$ in an integer with $n+e+1$ bits. We store the sign in the highest binary digit, then the exponent, and finally the mantissa. Recall that the set of exponent values $\mathbb{E}_n$ has two fewer values than the set of integers $\mathbb{B}_n^*$. This is necessary to encode two special states. The first of the two is used for zero when $m = 0$ and denormalized numbers when $m \neq 0$. The second encodes $\pm\infty$ when $m = 0$ and the non-numerical value "NaN" when $m \neq 0$ which is used to represent an undefined or complex result, e.g. $\sqrt{-1} = \text{NaN}$. See Figure A.1 for a table of floating-point numbers in $\mathbb{F}_{2,2}$ for a more concrete example.

Like fixed-point arithmetic, $\mathbb{F}_{e,n}$ is not closed under the arithmetic operations. In fact, for most floating-point inputs, the arithmetic operators produce non-floating-point results. An easy analogy is adding 1.01×10^5 and 2.22×10^2 with only three significant figures; the result is 1.01222×10^5 which clearly requires more than three significant digits. Therefore, we must define a function $\text{fl} : \mathbb{R} \to \mathbb{F}_{e,m}$ that rounds some real number to the nearest floating-point value.

There are five possible rounding schemes: towards zero; towards ∞; towards $-\infty$; towards nearest with ties towards value with even least significant digit; and towards nearest with ties away from zero. The first three are also called truncation, ceiling, and floor, respectively. The fourth is generally preferred to the fifth since it does not introduce bias when rounding a large data set. For the rest of this paper, *floating-point rounding* will refer to either the fourth or fifth strategy.

The function fl has two special cases: the condition $|x| > \sup\{\mathbb{F}_{e,m} \setminus \infty\}$ and the condition $x < \inf\{x \in \mathbb{F}_{e,m} \setminus \{0\} : |x|\}$ is called *underflow.* For overflow, $\text{fl}(x) = \pm\infty$ and for underflow, $\text{fl}(x) = 0$.

For any real-valued function f, the corresponding floating-point operation f^* is ideally the composition of fl and f. This way $\mathbb{F}_{e,n}$ is closed under $f*$. We say an implementation f^*, i.e. the actual algorithm of a function f, is *correctly rounded* if for every x in the floating domain of f^*, the floating-point result of $f^*(x)$ is $(\text{fl} \circ f)(x)$.

In practice, this is difficult because of the so-called Table-maker's dilemma (see Appendix A.1). For transcendental functions such as x^y, we cannot quickly compute ahead of time the (internal) precision needed for every floating-point arguments in the domain to give a correctly rounded result. Thus, we may relax our definition of correctly rounded and call an implementation f^* correctly rounded if $f(x) \in [y_0, y_1]$ where y_0 and y_1 are neighboring floating-point values and $f^*(x)$ returns either y_0 and y_1. Informally, it is (acceptably) correctly rounded if the rounded result is the nearest or second-nearest floating-point value.

It would have been quite possible to describe floating-point numbers without mentioning integers or fixed-point numbers. However, the previous subsections were arranged so that we could build off of our knowledge of integers. In the next section, we will see how the finiteness of floating-point numbers cause numerical errors. From now on, we will use the symbol $\mathbb{F}$ to refer to the set $\mathbb{F}_{e,n}$ for any arbitrary positive integers e and n.

3 Floating Point Error

We will define two main types of floating-point error: *relative error* and *ordinal error.* Relative error is not unique to floating-point so it is easy to understand. Ordinal error is less intuitive and requires us to analyze how variable scaling affects relative error between any two floating point values.

3.1 Relative Error

We can define two measures of relative error. For any $x \in \mathbb{R}$, the error relative to x is

$$E(x) = \frac{|x - \text{fl}(x)|}{|x|} \qquad x \neq 0 \tag{12}$$

To analyze relative error, we define the *unit roundoff* or *machine epsilon* of $\mathbb{F}_{e,n}$ to be $\epsilon = 2^{-n}$.

Of the four main arithmetic operations, addition and subtraction are far more error prone than multiplication and division. The following theorem demonstrates this tendency [1].

Theorem 3.1. *Let $\mathbb{F}$ be an arbitrary floating point representation, and let $x, y \in \mathbb{F}$. If $x - y$ rounds to a finite number, then the relative error of the result is bounded above by 1.*

More generally, this theorem holds when adding numbers of opposite signs or subtracting numbers of the same sign. Interestingly, adding numbers of the same sign (or subtracting numbers of opposite signs) ensures a tight error bound.

Theorem 3.2. *If $x \geq 0$ and $y \geq 0$, then the relative error of computing $x + y$ is at most 2ϵ.*

Thus, taking the difference of numbers that have varying magnitudes is far more error-prone than adding the same numbers. An excellent example of this is the equation $x^2 - y^2$ which is much more problematic than $(x + y)(x - y)$ since the subtraction in the rewritten expression is between numbers with more similar magnitudes.

3.2 Ordinal Error

Relative error is sensible for real numbers but floating point values are not uniformly distributed. For every integer s, we have an equal number of floating-point numbers on the interval $[2^s, 2^{s+1})$, assuming we have no overflow or underflow. Thus, floating-point values tend to cluster around 0 and spread out as we move towards $\pm\infty$. In fact, there are more floating-point numbers on $[0, 1)$ than on $[1, \infty)$ for any representation $\mathbb{F}_{e,n}$ when $e \geq 2$ and $n \geq 2$.

Our solution is to analyze any floating-point result on a set that evenly distributes floating-point values. Let $x, y \in \mathbb{F}$ be two floating-point values such that y is the nearest floating-point value above x. Then we can construct a function $O : \mathbb{F} \to \mathbb{Z}$ such that $O(0) = 0$ and $O(x) - O(y) = 1$. We say that O maps floating-point numbers to their corresponding *ordinal* value. See Figure A.1 for a more concrete example of ordinal values.

Thus we have an alternative definition of error. We define *ordinal error* to be the distance between two floating-point numbers when mapped through O. Formally, for any real-valued function $f : \mathbb{R}^n \to \mathbb{R}$ and its corresponding implementation f^*, the ordinal error for any input in the domain of f^* is

$$E_O(x) = |O(f^*(x)) - O((\text{fl} \circ f)(x))|, \qquad x \in \mathbb{F}^n$$

We reach an interesting corollary, or rather a redefinition of a previously defined concept: an implementation $f^* : \mathbb{F}^n \to \mathbb{F}$ is said to be correctly rounded if $O(f^*(x), (\text{fl} \circ f)(x)) = 0$ for all $x \in \mathbb{F}^n$. Moreover, if a value is correctly rounded, it is within ϵ of the real value. Thus, we can also define correctly rounded based on the machine epsilon.

Ordinal error gives a sense of distance to the “best possible” floating point result and removes any ambiguity when reporting absolute error. As a side note, ordinal error is loosely related to accuracy to the n digit by a logarithmic relation, so being inaccurate at the 5th digit results in an ordinal error that is orders of magnitude larger than being inaccurate at the 10th digit.

3.3 Error, In Depth

Now that we have a better understanding of floating-point numbers and floating-point error, we will return to the two examples presented in the introduction. For this section, we will use double-precision floating-point values, i.e. $\mathbb{F}_{11,52}$, one of the standard floating-point representations on modern computers.

Recall that Equation (1) is the difference of square roots: $f(x) = \sqrt{x+1} - \sqrt{x}$. Assuming that square root is implemented to be correctly rounded and that $x \geq 0$, we can compute a rough relative error bound using Theorem 3.1 and Theorem 3.2. That is,

$$E(f(x)) \approx E(+) + 2E(\sqrt{\ }) + E(-) = 1 + 4\epsilon$$

Thus, for any $x \in \mathbb{F}$, the floating-point result $f^*(x)$ can vary by slightly more than the magnitude of the result. In the introduction, we identified $x = 10^{16}$ as a problematic input. In fact, $f(x) = 5 \times 10^{-9}$ and $f^*(x) = 0$. We see that the relative error for this input is 1 which is within our estimated error bound. For the same value of x, the ordinal error is

$$E_O(x) = |O(5 \times 10^{-9}) - O(0.0)| = 4.85 \times 10^{18}$$

This number is not particularly useful or intuitive. Although to put that number in perspective, there are approximately 1.84×10^{19} double-precision floating-point values. That is, nearly a quarter of double-precision floating-point values are between 0 and 5×10^{-9}, and any of them would have been a better result.

But what specifically is going wrong here? Relative error is fairly useless whenever an expression contains a subtraction (or addition of opposite signs), and ordinal error is far too confusing to use. Instead, we will analyze each operation individually.

We know that for our input value $x = 10^{16}$ is guaranteed to be correctly rounded at least within ϵ of its true value. However, we can make a stronger assumption: $\mathrm{fl}(x)$ is exactly x. The proof behind this will not be stated, but the next floating-point value, i.e. $y > x$ and $|O(x) - O(y)| = 1$ is $y = 10^{16} + 16$. Thus, $10^{16} + 1$ is definitely not representable, so we must round down to the nearest value. Therefore, $\mathrm{fl}(x + 1) = x$.

The problem with the subsequent operations is now obvious. For this particular input value, the expression reduces to

$$\sqrt{x+1} - \sqrt{x} \quad \rightsquigarrow \quad \sqrt{x} - \sqrt{x} = 0$$

We conclude this section, by describing some interesting features of floating-point error. Mainly, floating-point error is not compositional, not locally or distantly predictable, and is silent. This first two are more relevant to mathemeticians. For any operation, the accuracy of the inputs give little indication of the accuracy of the output. In our difference of square roots example, $\sqrt{x+1}$ was still relatively accurate but the final result is quite inaccurate.

Moreover, error can accumulate slowly but grow large over many operations, or it can be eliminated entirely by strange cancellation. Thus its difficult to establish the root cause of error in large mathematical programs. The last feature of floating-point error is more relavant when developing numerical code. For the most part, inaccurate expressions will still produce numerical results, and will not give any indication how or where computation went wrong which can prove to be quite frustrating for any computer scientist.

4 Mitigating Error

Floating-point error is unpredictable and difficult to deal with. As a result, computer scientists try extensively to avoid the issue. There are a few techniques that are commonly used to eliminate floating-point error in numerical code.

4.1 Increasing Precision

An obvious solution is to increase n for our representation $\mathbb{F}_{e,n}$ to allow additional binary digits in the significand. Think about the decimal analogue, increasing the number of digits when using scientific notation obviously increases accuracy of the end result.

Recall Equation (2), our two-variable polynomial, $9x^4 - y^4 + 2y^2$, from the introduction and recall that for $x = 10864.0$ and $y = 18817.0$, the expression evaluates to 2.0 in double precision rather 1.0. The error is completely eliminated if we increase $n = 52$ to $n = 64$. Thus when evaluating the expression using values in $\mathbb{F}_{15,64}$, also known as extended precision, we get the correct result of 1.0.

Is this all we need to eliminate error? At first glance, this solution may seem sufficient: increase n to get more accurate results. However, increasing precision has limits. It increases the size of the domain where the error is minimal, but only marginally. For our difference of square roots, the upper bound where the expression becomes 0 is somewhere between 10^{19} and 10^{20} for $n = 64$ (compared to 10^{16} for $n = 52$).

More importantly, the time spent computing common math functions such as e^x and $\sqrt{x}$ increase drastically as we increase n. In fact, for certain implementations like fused-multiply-add (fma) which computes $x * y + z$ within ϵ of the real result, increasing n from 52 to 64 causes the run time to increase by 10 times. Thus increasing n beyond 100 will result in marginal gains in accuracy but an untenable increase in run time.

Tools like MPFR, Multiple Precision Floating-Point Reliable Library, compute correctly rounded results for any $\mathbb{F}_{e,n}$, but the algorithms in the library are generic and are orders of magnitude slower than finely-tuned math libraries that are sufficiently accurate, rather than correctly rounded.

4.2 Interval Analysis

When working with error, it is often more convenient to express a result using some error bound. Interval analysis produces an interval as an output that is *guaranteed* to contain the real result.

We will being by redefining our common floating-point functions. First, the function fl will be defined as follows:

$$\mathrm{fl}(x) = [x_l, x_h], \qquad x \in \mathbb{R} \tag{13}$$

where $x_l, x_h \in \mathbb{F}$ are the nearest floating-point values below and above x respectively. In the special case that x is *exact*, that is, $x \in \mathbb{F}$, $\mathrm{fl}(x) = [x, x]$. In either case, it is trivial to see that $x \in \mathrm{fl}(x)$.

Addition and multiplication are also easy to construct. Let $\mathbf{x} = [x_l, x_h]$ and $\mathbf{y} = [y_l, y_h]$, then

$$\mathbf{x} + \mathbf{y} = [\inf \mathrm{fl}(x_l + y_l), \sup \mathrm{fl}(x_h + y_h)] \tag{14}$$

That is, the endpoints of $\mathbf{x} + \mathbf{y}$ are the lowest rounded value and highest rounded value possibly attainable by the operation. Similarly, multiplication is defined by

$$\mathbf{xy} = [\inf \mathrm{fl}(x_l y_l), \sup \mathrm{fl}(x_h y_h)] \tag{15}$$

Subtraction and division are defined slightly differently. For subtraction, we subtract the opposite endpoints.

$$\mathbf{x} - \mathbf{y} = [\inf \mathrm{fl}(x_l - y_h), \sup \mathrm{fl}(x_h - y_l)] \tag{16}$$

And division is defined by

$$\mathbf{x}/\mathbf{y} = [\inf \mathrm{fl}(x_l / y_h), \sup \mathrm{fl}(x_h / y_l)] \tag{17}$$

However, there is a slight problem with division if $0 \in \mathbf{y}$. In that case, it is quite possible our end result is undefined if divison by zero occurs. For the most part, this issue is left up to the developer; they may choose to signal if the resulting interval is problematic.

Interval analysis for monotonic functions like $\sqrt{x}$ and e^x is easy enough to imagine while functions like x^y or $\sin x$ are much more difficult to implement, so we leave that exercise up to the reader. For any function though, if we begin with the assumption that our arguments contain the real value, it's not hard to see that the result will also contain the real value.

Thus, for any computation, we are guaranteed an interval that contains the real result. Recall the difference of square roots example again. For $x = 10^{16}$, the expression evaluates to $[0, 1.4901 \times 10^{-8}]$. This interval indeed contains the real result 5×10^{-9}.

Futhermore, we see that interval analysis implicitly encodes the quality of a result. Informally, a floating-point result has high error if and only if the associated interval computed using interval analysis is wide on the ordinals. For example, if we compute the expression again with $x = 1$, we get $[0.41421356237309492, 0.41421356237309515]$, a fairly narrow interval. This implies that computing the expression normally will give an accuate result.

4.3 Rewriting

In Section 3, we made the observation that certain operations are more problematic than others. Moreover, we know that floating-point operations are not associative or distributive, so switching the order of operations can affect the result. Therefore, we reach the conclusion that evaluating algebraically equivalent expressions can possibly produce more accurate expressions. We call this technique *rewriting* and it the most common technique used to improve numerical code.

Once again, recall the difference of square roots example. We know that the subtraction operation introduces a large amount of error. However, we can rewrite the expression to remove that issue:

$$\sqrt{x+1}-\sqrt{x} \quad \rightsquigarrow \quad \left(\sqrt{x+1}-\sqrt{x}\right)\cdot\frac{\sqrt{x+1}+\sqrt{x}}{\sqrt{x+1}+\sqrt{x}} \quad \rightsquigarrow \quad \frac{1}{\sqrt{x+1}+\sqrt{x}}$$

This new expression on the right is nearly correctly rounded for all $x \in \mathbb{F} \geq 0$. For our problematic input value $x = 10^{16}$, we get the correct value of 5×10^{-9}. Using interval analysis, we get $[4.9999999999999985\times10^{-9},\ 5.0000000000000001\times10^{-9}]$ which has an ordinal width of 2.

Note that this technique generally increases the complexity of the expression (and thus makes it marginally slower); however, compared to the previous techniques, rewriting is quite reasonable to use. The problem is finding rewritten expressions with improved error. Manually finding better expressions is an arduous task since floating-point error is not predictable. Thus we generally leave such tasks to computer scientists with considerable experience with numerical analysis or automated tools that can quickly search through many possible implementations [6].

4.4 Specialized Techniques: Kahan's Summation

The previous three techniques are general methods for minimizing floating point, but for certain scenarios, we may prefer more targeted improvements. The following subsection describes Kahan's summation, a technique for accurately computing the sum of N floating-point numbers.

The motivation for reducing error for summations is simple: summations are common. Adding $\sum_1^N x_j$ causes error to accumulate with each addition operation, so we should extend the precision of the accumulator so the error of the result is minimized. Let s be the accumulator, c be the first-order error term, and let arr be the list of numbers we are summing. Assume all variables store values in the same representation.

Algorithm 1: Kahan's summation

```
s ← 0
c ← 0
for i in arr do
    y ← i - c
    t ← s + y
    c ← (t - s) - y
    s ← t
end
return s
```

The algorithm goes as follows: we first subtract the ith number by the current first-order error term to absorb as much error as possible. Note that if $|i| \gg |c|$, then $\mathrm{fl}(i - c) = i$. Then t is the ith partial sum plus any error absorbed by y. Subtracting our ith partial sum by the previous partial sum and the ith number gives us the new error term of t. For each number, we repeat this process and finally return our result s.

But how much better is Kahan's algorithm? For any sum $\sum_1^N x_j$, we can conclude that following. For the naive summation method, the error is bounded above by $n\epsilon \sum |x_j|$ while for Kahan's summation algorithm, the error is bounded above by 2ϵ. Using Big-Oh notation, we say that the error of the naive solution is $O(n)$ while the error of Kahan's algorithm is $O(1)$; that is, the error of Kahan's algorithm does not change no matter our choice of N.

In fact as we send $N \to \infty$, Kahan's summation will always produce accurate results as long as for all $n \geq N$, $\mathrm{fl}\left(\sum_1^n x_j\right) \neq \pm\infty$. In other words, the partial sums must never overflow. Meanwhile, there is no guarantee that the naive solution is anywhere near the correct result.

The important technique in Kahan's algorithm is using an accumulator to track error which is used often to increase the accuracy of numerical algorithms. We can extend Kahan's techniques to common operations like computing the determinant of a 2-by-2 matrix [3]. Let P be a 2-by-2 matrix

$$P = \begin{pmatrix} a & b \\ c & d \end{pmatrix},$$

and assume that the function FMA (fused multiply-add) can compute $xy + z$ within ϵ of the real result. Then, we can compute the determinant of P within 2ϵ of the real result.

Algorithm 2: Kahan's method for computing $ad - bc$ using FMA

```
w ← bc
e ← -FMA(b, c, -w)       // e = w - bc
f ← FMA(a, d, -w)
x ← f + e
return x
```

Matrix operations are important since many computer programs operate on matrices. Thus it is important for that the error of functions such as computing determinants is minimized.

4.5 Computer Algebra Systems

We present computer algebra systems as the final technique for minimizing numerical error mostly because such tools dodge the issue of error by performing no numerical computation at all. The most well-known example of such a tool is Wolfram Mathematica.

Little is known about how Mathemtica works since the software is proprietary, but it probably uses interval analysis and high-precision computing for numerical computation. More importantly, however, it may perform symbolic manipulation to minimize rounding error. Additionally, Mathematica performs exact computation when giving rational inputs and expressions with non-trancendental operators.

For the most part, Mathematica is the state-of-the-art for mathematical computational and should be regarded as the go-to tool for symbolic and exact computation. However, inexact computation can still give incorrect answers. Try the difference of square roots example with inexact inputs!

5 Conclusion

This paper briefly describes computer number representations, causes of numerical error, and techniques to mitigate error. As described in this paper, numerical error is difficult to analyze. However, it is important for anyone using a computer for numerical computation to understand these issues at a high level. Users should keep in mind the limitations of representations and algorithms and be able to diagnose possible causes of error. As we have seen throughout this paper, it is quite easy for a seemingly simple operation to produce nonsensical numerical result. For general numerical computation, we recommend using well-tested software that has been developed to perform accurate computations to minimize the possibility of error.

A Appendix

A.1 Table-Maker's Dilemma

The Table-Maker's Dilemma, a phase coined by William Kahan, is an outstanding issue that has plagued developers of computer math libraries for decades. It is the main issue that creates a wide chasm in performance between sufficiently rounded libraries and correctly rounded libraries.

Consider the following example using binary numbers. Assume you compute $f(x)$ for some function f and get 1.00010000 using 8 binary digits (after the binary point), but we want to round to the nearest value with 3 binary digits. Because we are using binary, the value we have is exactly in between 1.000 and 1.001, so we cannot determine the correct rounded result. To do so, we would need additional bits to determine whether to round up or down. For transcendental functions, the real numerical result may be require a non-terminating sequence of digits to represent. Thus, when computing these functions we may require many additional internal bits to resolve rounding correctly. Even for single- ($\mathbb{F}_{8,32}$) and double- ($\mathbb{F}_{11,64}$) precision floating point values, we may require hundreds of extra bits for an unknown number of inputs [4].

For most math library developers, the easiest option is to admit defeat; allow a few incorrectly-rounded results. This gives the added benefit of more consistent and reasonable performance. For other math libraries like MPFR, correctly rounded results are a must, so run times are severely impacted. In recent years, there have been a few techniques offered to avoid the Dilemma altogether including the Minefield Method [2] and polynomial generation via linear systems [5].

$x \in \mathbb{F}_{2,2}$	$O(x) : \mathbb{F} \to \mathbb{Z}$	binary representation	denormal?
$-\infty$	-12	11100	no
-3.5	-11	11011	no
-3	-10	11010	no
-2.5	-9	11001	no
-2	-8	11000	no
-1.75	-7	10111	no
-1.5	-6	10110	no
-1.25	-5	10101	no
-1.0	-4	10100	no
-0.75	-3	10011	yes
-0.5	-2	10010	yes
-0.25	-1	10001	yes
0.0	0	00000	no
0.25	1	00001	yes
0.5	2	00010	yes
0.75	3	00011	yes
1.0	4	00100	no
1.25	5	00101	no
1.5	6	00110	no
1.75	7	00111	no
2	8	01000	no
2.5	9	01001	no
3	10	01010	no
3.5	11	01011	no
∞	12	01100	no
NaN	N/A	any other 5-bit number	no

Figure A.1: The floating point values in $\mathbb{F}_{2,2}$, their respective ordinal value, their binary representations, and whether or not each number is denormal.

A.2 Proofs

Proof of Theorem 3.1:

Assume we are using p binary digits of precision. The maximum relative error of $x - y$ occurs when $x = 1.00...0$ and $y = 0.11...1$. Then the real result is $x - y = 2^{-p}$, but when computing the answer using p digits, the rightmost digit of y is ignored, so the result is 2^{1-p}. Thus the absolute error is $2^{-p} - 2^{1-p} = 2^{-p}$, and the relative error is $2^{-p}/2^{-p} = 1$.

Proof of Theorem 3.2:

Assume we are using p binary digits of precision. Let $x = d \times 2^0$ where $1 \leq d \leq 0$. First, assume there is no carry out when we add x to some number y. Then the digits shifted off of y has a value less than 2^{1-p}, and the sume is at least 1, so the relativer error is less than 2ϵ. If there is a carry out, then the error from shifting is $2^{2-p}/2$. The sum at least 2, so the relative error is

$$\left(2^{1-p} + \frac{1}{2}2^{2-p}\right)/2 = 2^{1-p} \leq 2\epsilon$$

References

[1] D. Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *Computing Surveys*, 23(1), 1991.

[2] J. L. Gustafson. The Minefield Method: A Uniforml Fast Solution to the Table-Maker's Dilemma.

[3] C.-P. Jeannerod, N. Louvet, and J.-M. Muller. Futher Analysis of Kahan's Algorithm for the Accurate Computation of 2 x 2 Determinants. *Mathematics of Computation*, 82(284):2245–2264, 2013.

[4] V. Lefevre and J.-M. Muller. The Table Maker's Dilema: Our Search for Worst Cases.

[5] J. P. Lim and S. Nagarakatte. High Performance Correctly Rounded Math Libraries for 32-bit Floating Point Representations. In *Proceedings of the 42th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, Virtual, 2021. ACM.

[6] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock. Automatically Improving Accuracy for Floating Point Expressions. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, Portland, Oregon, USA, 2015. ACM.