Floating-point Numbers Aren't Mathematical Real Numbers

SHARCNET General Interest Webinar Series

Paul Preney, OCT, M.Sc, B.Ed., B.Sc.
Email: preney@sharcnet.ca

University of Windsor, Office of Research and Integrity Services (ORIS)
SHARCNET / Compute Ontario / Digital Research Alliance of Canada

Support Email: support@tech.alliancecan.ca

Jan. 14, 2026

Land Acknowledgement

I am located at the University of Windsor.

The University of Windsor sits on the traditional territory of the **Three Fires Confederacy** of First Nations, which includes the **Ojibwa**, the **Odawa**, and the **Potawatomi**. We respect the longstanding relationships with First Nations people in this 100-mile Windsor-Essex peninsula region and the straits —les détroits— of Detroit.

Overview

Commonly in math there are:

- natural, $\mathbb{N}$, and integer, $\mathbb{Z}$, numbers,
- rational, $\mathbb{Q}$, numbers,
- real, $\mathbb{R}$, numbers, and,
- complex numbers, $\mathbb{C}$.

Overview (cont.)

Natural, $\mathbb{N}$, and integer, $\mathbb{Z}$, numbers are all discrete, exact values and since computers are discrete and treat integers exactly natural and integer numbers typically do not have issues of representation.

Sometimes there are issues to be aware of and dealt with:

- mixing signed and unsigned *bounded* values in the same expression
- relying on "integer wrap-around" with signed *bounded* values with **overflows**
- Typically computer integer types have **no representation** of **not-a-number** or **infinity** so programs needing such must deal with such.

Overview (cont.)

Rational, $\mathbb{Q}$, numbers are ratios of two integers, e.g., p/q.

- On computers rational numbers are typically stored exactly as the two integers that comprise the rational number.
- Rational numbers on computers represented using integers are exact.
- Rational numbers are typically reduced to lowest terms with each calculation.
- Typically computer rational types have **no representation** of **not-a-number** or **infinity** so programs needing such must deal with such explicitly.
- As with integers, issues can arise especially when using bounded integers
  - e.g., values can easily become very large so how integer overflow is dealt with can become an issue

Overview (cont.)

Real numbers, $\mathbb{R}$, are:

- numbers that can be used to measure continuous one-dimensional quantity
- continuous means that pairs of real numbers can have arbitrarily small differences
- many real numbers are infinite in length, e.g., irrational numbers

Complex numbers, $\mathbb{R}$, are:

- a number system that extends $\mathbb{R}$ to have two components: "real" and "imaginary"
- i.e., complex numbers rely on $\mathbb{R}$ numbers and are often represented using real numbers on computers (this presentation assumes real numbers are being used)

Overview (cont.)

Concerning representing real numbers on a computer:

- Computers are discrete and finite so unless the computer is being used to explicitly represent a $\mathbb{R}$ or $\mathbb{C}$ number exactly in symbolic form, the number must be approximated on a computer.
  - e.g., an infinite number cannot be represented on a computer at all
- On a computer, typically programs **approximately** represent $\mathbb{R}$ and $\mathbb{C}$ numbers using a programming language's single and/or double precision formats, e.g.,
  - In C, `float`, `double`, `float _Complex`, `double _Complex`
  - In C++, `float`, `double`, `std::complex<float>`, `std::complex<double>`
  - etc.

Overview (cont.)

Because $\mathbb{R}$ numbers are not represented exactly on a computer, most computer programming languages do **not** refer to types used to represent such as *real* values.

Instead such types on a computer are typically **floating-point** values which are "clever" **finite approximations** to some **limited in numeric precision** rational number.

- Often these approximations can work —but don't expect them to always work!
- The internal base that is used to store and operate on such numeric values can also matter.

Floating-Point Numbers

Floating-point numbers on computers typically use the IEEE representation and designed to have any of the following forms:

- $\pm d.dddddd \times B^{e}$,
  - The number of digits, d, available varies with the actual representation *and* the base used to represent numbers.
  - B is typically the base used to store the number.
    - This is typically 2 or 10. Assume it is 2.
    - C23 introduced these decimal floating-point types: `_Decimal32`, `_Decimal64`, `_Decimal128`.
- $\pm\infty$, or,
- not-a-number (NaN)

Floating-Point Numbers (cont.)

Suffix your hard-coded floating-point constants in your code, e.g.,

Literal	Std	Type	Description
`f`, `F`		`float`	
`L`, `L`		`long double`	
`bf16`, `BF16`	C++23	`std::bfloat16_t`	"brain float"
`f16`, `F16`	C++23	`std::float16_t`	half precision
`f32`, `F32`	C++23	`std::float32_t`	single precision
`f64`, `F64`	C++23	`std::float64_t`	double precision
`f128`, `F128`	C++23	`std::float128_t`	quad precision

Floating-Point Numbers (cont.)

And remember accuracy varies, e.g., with a binary, base-2, representation:

Type	**Precision Bits**	**Exponent Bits**	**Max Exponent**
`std::bfloat16_t`	8	8	127
`std::float16_t`	11	5	15
`std::float32_t`	24	8	127
`std::float64_t`	53	11	1023
`std::float128_t`	113	15	16383

Problems To Consider

When using floating-point numbers:

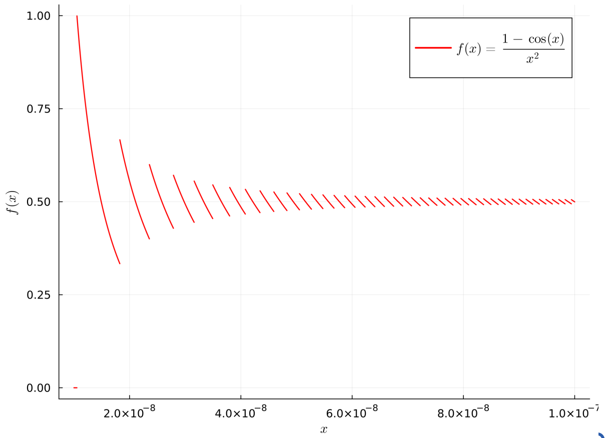
- While it is nice to assume they are real numbers, remember they are not!
- Floating-point numbers are commutative. (This is good.)
- Floating-point numbers are **not associative**. (Real numbers are. This is bad.)
- Floating-point numbers are **not exact**. Don't compare using ==.

Problems To Consider (cont.)

Does $f(x) = \frac{1-\cos x}{x^2}$ (where x is close but not 0) look continuous to you in the next slide's graph?

Explanation: The discontinuities you see in the graph in the next slide are there due to floating-point representation issues. Essentially, as x approaches 0, $\cos x$ will get closer to 1 and the subtraction $1 - \cos x$ will result in a small value (rounded to the nearest representable value). When divided by x^2, which is twice as small as x, the rounded value $1 - \cos x$ gets amplified showing the visible patterned jumps you see in the graph.

Problems To Consider (cont.)



Problems To Consider (cont.)

How about calculating pi?

- e.g., see `https://en.wikipedia.org/wiki/Floating-point_arithmetic%23Minimizing_the_effect_of_accuracy_problems`
    - Should you go to that URL, scroll down to see t_0, the first form, and the second form.
- $t_0 = 1/\sqrt{3}$
- First form: $t_{i+1} = (\sqrt{t_i^2 + 1} - 1)/t_i$
    - NOTE: This version is *numerically unstable*.
- Second form: $t_{i+1} = t_i/(\sqrt{t^2 + 1} + 1)$
- $\pi \approx 6 \times 2^i \times t_i$, which will converge as $i \to \infty$

Problems To Consider (cont.)

Writing code to implement these two forms is straight-forward, e.g., C++ some code for the first form is:

first form C++ code

```
template <std::floating_point T>
std::generator<T> first_form()
{
  using std::sqrt;

  // compute t0...
  T value{ T(1) / sqrt(T(3)) };
  co_yield value;

  // compute tN...
  for (;;)
  {
    value = (sqrt(value * value + T(1)) - T(1)) / value;
    co_yield value;
  }
}
```

Problems To Consider (cont.)

some code for the second form is:

second form C++ code

```
template <std::floating_point T>
std::generator<T> second_form()
{
  using std::sqrt;

  // compute t0...
  T value{ T(1) / sqrt(T(3)) };
  co_yield value;

  // compute tN...
  for (;;)
  {
    value = value / (sqrt(value*value+T(1)) + T(1));
    co_yield value;
  }
}
```

Problems To Consider (cont.)

simple code to transform each form result to π is:

transformation to pi code

```
template <std::integral I, std::floating_point T>
inline T form_to_pi(I i, T t)
{
  using std::pow;
  return t * pow(T(2),i) * T(6);
}
```

Problems To Consider (cont.)

and some example code to compute and produce output:

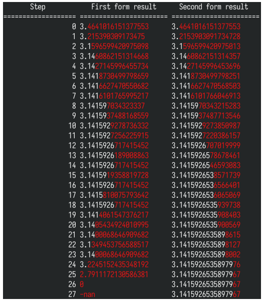
simple output
```
using REAL = double;

auto f1{ first_form<REAL>() }; auto f1it{ f1.begin() };
auto f2{ second_form<REAL>() }; auto f2it{ f2.begin() };
for (size_t i = 0; i != 28; (void)++i, (void)++f1it, (void)++f2it)
  println("{} {} {}", i, form_to_pi(i,*f1it), form_to_pi(i,*f2it));
```

If the output code is enhanced to output using colour (for matching and non-matching digits of π), then the output could appear like the output on the next slide...

Problems To Consider (cont.)

In the following screen capture, white digits match π's digits, red digits do not match, and i is "Step":



```
       Step           First form result      Second form result
================== ====================== ======================
                 0 3.4641016151377553     3.4641016151377553
                 1 3.215390309173475      3.2153903091734728
                 2 3.1596599420975098     3.1596599420975013
                 3 3.1460862151314668     3.1460862151314357
                 4 3.1427145996455734     3.1427145996453696
                 5 3.1418730499798659     3.1418730499798251
                 6 3.1416627470550682     3.1416627470568503
                 7 3.1416101765995217     3.1416101766046913
                 8 3.141597034323337      3.1415970343215283
                 9 3.1415937488168559     3.1415937487713546
                10 3.1415929278736332     3.1415929273850987
                11 3.1415927256225915     3.1415927220386157
                12 3.1415926717415452     3.1415926707019999
                13 3.1415926189008863     3.1415926578678461
                14 3.1415926717415452     3.1415926546593083
                15 3.1415919358819728     3.1415926538571739
                16 3.1415926717415452     3.1415926536566401
                17 3.1415810075793642     3.1415926536065069
                18 3.1415926717415452     3.1415926535939738
                19 3.1414061547376217     3.1415926535908403
                20 3.1405434924010995     3.1415926535900569
                21 3.1400068646909682     3.1415926535898615
                22 3.1349453756588517     3.1415926535898127
                23 3.1400068646909682     3.1415926535898002
                24 3.2245152435348192     3.1415926535897976
                25 2.7911172130586381     3.1415926535897967
                26 0                      3.1415926535897967
                27 -nan                   3.1415926535897967
```

Problems To Consider (cont.)

This example shows:

- One clearly wants to use numerically stable routines like the second form.
- The math involved might be "fine" if done by hand but when implemented on a computer such might yield the same results at all.
- This example benefits since we know the ultimate value we are computing and so the result can be compared.
  - In many situations, one does not know the result being computed.
  - Sometimes, but not always, the error of the computation can be used to know how many digits of accuracy will result —but know there are times when that is not known or cannot be used.

Problems To Consider (cont.)

Something to Remember

When you do math by hand, you tend to figure everything out exactly (and symbolically) until the last step where you numerically evaluate everything that remains. This minimizes issues with errors associated numerically evaluating to a certain (finite) precision.

When you do math on a computer using floating-point numbers, virtually all of those numbers are approximate and therefore each has some associated error with it as the numbers are approximate. This "approximating" is the case for everything with every computational step —unlike doing math by hand—. It is not surprising then that some code winds up being numerically unstable because ultimately too much error can become introduced into the values being computed.

Some Things To Note and Look Out For

If you are writing code using floating-point arithmetic, there are some things that you should note with any mathematical expression being used:

- subtraction of values that result in values very close 0 (zero)
  - This can result in "catastrophic cancellation" and shouldn't be ignored.

Some Things To Note and Look Out For (cont.)

- division by zero or close-to-zero numbers
  - Values close to zero can underflow/round to zero.
  - Dividing by values close to zero can result in very large numbers. If these numbers are too large the resulting computed value might be an ∞ value. (If dividing by zero, the result could also be a NaN value not only ∞.)
  - Floating-point numbers might also use special "denormal" values when values very close to zero. While this can yield additional accuracy, if such values are divided by, then the resulting computed value can become extremely large —and if too large for the floating-point representation being used an ∞ value will be computed.

Some Things To Note and Look Out For (cont.)

- prefer multiplication and division over addition and subtraction (when possible)
  - e.g., if an intermediate result would involve adding a very large positive value and a very large negative value

Some Things To Note and Look Out For (cont.)

- when writing floating-point values to a text file/stream *and* those values need to be read in later as in order to have *exactly the same values* that were written earlier:
  - When writing the values out, be sure to write all digits of the floating-point representation being used.
  - Consider using the "hexfloat" format as this format always outputs/reads the exact number. (C calls this "hexadecimal exponent notation" and uses $\%a$ with `printf()` and `scanf()`.) This format outputs the number exactly by outputting the the mantissa in hexadecimal form followed by the exponent in decimal form.

A Kahan Sum Example: Overview

The **Kahan sum** is a **compensated summation** algorithm that significantly reduces the numerical error in the total obtained by adding a sequence of floating-point values relative to the naive way of adding numbers.

Naively summing numbers has a **worst-case** error that grows **proportional to** n. Using the **Kahan sum** (with sufficient precision) the **worst-case** error bound is effectively **independent of** n.

- Effectively this means adding a large number of values can be summed with an error that only depends on the floating-point precision of the numbers used.

A Kahan Sum Example: Overview (cont.)

If code is already computing a sum then performing a Kahan sum in the same manner instead will compute that sum with less error.

- While floating-point (summation) is not associative, code is often written assuming it is and the results are often acceptable. When using a Kahan sum, the summation result will typically result in the same sum so, in effect, this also usually allows non-commutative sums to be computed.

A Kahan Sum Example: Overview (cont.)

NOTE:

Many, but not all, modern programming languages' **reduce** and **prefix sum** operations (sequential and parallel versions) are non-commutative (and non-associative). Such operations will typically execute more quickly than performing the same operations in a way that preserves commutativity and associativity especially since the latter may require more sequential execution.

A Kahan Sum Example: Overview (cont.)

What are the costs of using one of the variants of the Kahan sum?

As explored for this presentation, the costs of doing a compensatated sum on modern computers (sequentially) are *approximately:*

Algorithm	**Relative Time**	**Number of Variables**
Normal sum	1	1
Kahan sum (original)	6	2
Kahan-Babushka-Neumaier sum	3	2
Kahan-Babushka-Klein sum	18	3

A Kahan Sum Example: Overview (cont.)

NOTE:

Be aware that the particulars of the function(s) being computed and the capabilities of the hardware being used will also affect the amount of time required and the relative times mentioned here are only approximate, however, your own results should be similar.

A Kahan Sum Example: Overview (cont.)

The names of the Kahan sums used in these slides are based on those in this Wikipedia article:

- `https://en.wikipedia.org/wiki/Kahan_summation_algorithm`

These slides' code is meant to illustrate computing the various sums although, in practice, one would need/want more functions, move operations defined, etc. for efficiency reasons as not all floating-point types are efficiently copied and moved by value. Implementating such is beyond the scope of this presentation.

Sum

Normally to compute a sum using modern libraries/programming languages one needs the ability to do two things:

1. add a value with another value to continue computing a sum, and,
2. add an existing sum to another sum in order to continue computing a sum.

The second item is required in order to use many reduction and prefix sum libraries.

Sum (cont.)

This can be achieved similarly to the following code:

sum

```
template <std::floating_point T>
class normal_sum {
private:
  T sum_;

public:
  normal_sum() : sum_(0) { }
  // ...
  void add(T const& other) { sum_ += other; }
  void add(normal_sum const& other) { sum_ += other.sum_; }
  T sum() const { return sum_; }
};
```

Kahan Sum

A traditional Kahan sum could be implemented similarly to the following code:

Kahan sum

```
template <std::floating_point T>
class kahan_sum {
private:
  T sum_;
  T compensation_;

public:
  kahan_sum() : sum_(0), compensation_(0) { }
  // ...
  void add(T value) {
    T y{ value - compensation_ };
    T volatile t{ sum_ + y }; // keep this volatile
    T volatile z{ t - sum_ }; // keep this volatile
    compensation_ = z - y;
    sum_ = t;
  }
```

Kahan Sum (cont.)

```
  void add(kahan_sum const& other) {
    add(other.compensation_); // add other.compensation_ amount to this sum
    add(other.sum_); // add other.sum_ amount to this sum
  }

  T sum() const { return sum_; }
};
```

Kahan Sum (cont.)

Concerning the Kahan sum:

- The `volatile` variables need to be `volatile` in order to prevent the compiler from optimizing away the operations they depend on as those operations must be done.
  - The `volatile` used here is what the C and C++ languages define and use. The `volatile` keyword in Java is *not* the same. Other programming languages may or may not have an equivalent construct. It is *extremely* important that the computation of the compensated sum are not optimized away.
- Although literature discusses performing a Kahan sum, such does not usually discuss how to combine those sums. The latter is often also necessary to use modern libraries / programming languages so such appears here. (In these slides the compensation and then sum are added to the current sum to combine two Kahan sums.)

Kahan-Babushka-Neumaier Sum

Kahan-Babushka-Neimaier sum

```cpp
template <std::floating_point T>
class kahan_babushka_neumaier_sum {
private:
  T sum_;
  T compensation_;

public:
  kahan_babushka_neumaier_sum() : sum_(0), compensation_(0) { }
  // ...
  void add(T value) {
    T volatile t{ sum_ + value };
    if (std::abs(value) < std::abs(sum_))
    {
      // sum is larger, low-order digits of value are lost...
      T volatile z{ sum_ - t };
      compensation_ += z + value;
    }
    else
```

Kahan-Babushka-Neumaier Sum (cont.)

```
    {
      // sum is smaller, low-order digits of sum are lost...
      T volatile z{ value - t };
      compensation_ += z + sum_;
    }
    sum_ = t;
  }

  void add(kahan_babushka_neumaier_sum const& other) {
    add(other.compensation_); // add other.compensation_ amount to this sum
    add(other.sum_); // add other.sum_ amount to this sum
  }

  // NOTE: Asking for the sum always adds the compensation amount to sum...
  T sum() const { return sum_ + compensation_; }
};
```

Kahan-Babushka-Klein Sum

The Kahan-Babushka-Klein sum is a double compensation sum, e.g., something like computing a Kahan sum of a Kahan sum.

It requires an additional variable and many more operations to be performed and so it runs more slowly than using a single compensation sum.

Kahan-Babushka-Klein sum

```
template <std::floating_point T>
class kahan_babushka_klein_sum {
private:
  T sum_;
  T cs_;
  T ccs_;

public:
  kahan_babushka_klein_sum() : sum_(0), cs_(0), ccs_(0) { }
  // ...
  void add(T value) {
```

Kahan-Babushka-Klein Sum (cont.)

```
T volatile t{ sum_ + value };
T c;
if (std::abs(value) < std::abs(sum_)) {
  T volatile z{ sum_ - t };
  c = z + value;
} else {
  T volatile z{ value - t };
  c = z + sum_;
}
sum_ = t;
t = cs_ + c;
T cc;
if (std::abs(c) < std::abs(cs_)) {
  T volatile z{ cs_ - t };
  cc = z + c;
} else {
  T volatile z{ c - t };
  cc = z + cs_;
```

Kahan-Babushka-Klein Sum (cont.)

```
      }
      cs_ = t;
      ccs_ = ccs_ + cc;
    }

    void add(kahan_babushka_klein_sum const& other) {
      add(other.ccs_);
      add(other.cs_);
      add(other.sum_);
    }

    T sum() const { return sum_ + cs_ + ccs_; }
  };
```

Some Results

The aforementioned Kahan sums were used to numerically integrate some functions including $f(x) = 4/(1+x^2)$ from 0 to 1 (whose result is π) using `std::transform_reduce()` and `double` floating-point values.

The time result to compute 1,000,000,000 (one billion) intervals using one CPU core was:

Sum Algorithm	**Integration Method**	**Time**
Normal sum	midpoint	1.3s
Kahan sum (original)	midpoint	6.5s
Kahan-Babushka-Neumaier sum	midpoint	2.8s
Kahan-Babushka-Klein sum	midpoint	20.0s

Some Results (cont.)

Sum Algorithm	Integration Method	Time
Normal sum	Simpson's	1.25s
Kahan sum (original)	Simpson's	6.6s
Kahan-Babushka-Neumaier sum	Simpson's	2.8s
Kahan-Babushka-Klein sum	Simpson's	19.8s

Hardware Used: AMD Ryzen Threadripper PRO 3955WX (4.4 GHz max)

Some Results (cont.)

Takeaways:

- Benchmark your code —it may well run fast enough without using parallel constructs.
- These sums only use operations involving the floating-point type used so they all also can be used with GPUs not just CPUs.
  - e.g., using NVIDIA's HPC SDK with `-stdpar`
  - e.g., using CUDA's or ROCm's Thrust libraries
- While the above results were from sequential runs, the same code strategy continues to work when `std::transform_reduce()` is used to run code in parallel.

Some Results (cont.)

- Using the Kahan-Babushka-Neumaier sum is about 3 times slower than not using any compensated sum at all. In many cases such times will be acceptable and will have less summation error. That said, if done in parallel those sums can be done faster.