An Introduction to Floating-Point Arithmetic and Computation

Jeff Arnold

CERN openlab
9 May 2017

Agenda

- Introduction
- Standards
- Properties
- Error-Free Transformations
- Summation Techniques
- Dot Products
- Polynomial Evaluation
- Value Safety
- Pitfalls and Gremlins
- Tools
- References and Bibliography

Why is Floating-Point Arithmetic Important?

- It is ubiquitous in scientific computing
  - Most research in HEP can't be done without it
- Algorithms are needed which
  - Get the best answers
  - Get the best answers all the time
  - "Best" means the right answer for the situation and context
  - There is always a compromise between fast and accurate

Important to Teach About Floating-Point Arithmetic

- A rigorous approach to floating-point arithmetic is seldom taught in programming courses
- Not enough physicists/programmers study numerical analysis
- Many physicists/programmers think floating-point arithmetic is
  - inaccurate and ill-defined
  - filled with unpredictable behaviors and random errors
  - mysterious
- Physicists/programmers need to be able to develop correct, accurate and robust algorithms
  - they need to be able to write good code to implement those algorithms

Reasoning about Floating-Point Arithmetic

Reasoning about floating-point arithmetic is important because

- One can prove algorithms are correct without exhaustive evaluation
  - One can determine when they fail
- One can prove algorithms are portable
- One can estimate the errors in calculations
- Hardware changes have made floating-point calculations appear to be less deterministic
  - SIMD instructions
  - hardware threading

Accurate knowledge about these factors increases confidence in floating-point computations

Classification of real numbers

In mathematics, the set of real numbers $\mathbb{R}$ consists of

- rational numbers $\mathbb{Q}$ $\{p/q : p, q \in \mathbb{Z}, q \neq 0\}$
  - integers $\mathbb{Z}$ $\{p : |p| \in \mathbb{W}\}$
  - whole $\mathbb{W}$ $\{p : p \in \mathbb{N} \cup 0\}$
  - natural $\mathbb{N}$ $\{p : p \in \{1, 2, ...\}\}$
- irrational numbers $\{x : x \in \mathbb{R}\, x \notin \mathbb{Q}\}$
  - algebraic numbers $\mathbb{A}$
  - transcendental numbers

Dyadic rationals: ratio of an integer and 2^b where b is a whole number

Some Properties of Floating-Point Numbers

Floating-point numbers do not behave as do the real numbers encountered in mathematics.

While all floating-point numbers are rational numbers

- The set of floating-point numbers does not form a field under the usual set of arithmetic operations
- Some common rules of arithmetic are not always valid when applied to floating-point operations
- There are only a finite number of floating-point numbers

Floating-Point Numbers are Rational Numbers

What does this imply?

- Since there are only a finite number of floating-point numbers, there are rational numbers which are not floating-point numbers
- The decimal equivalent of any finite floating-point value contains a finite number of non-zero digits
- The values of transcendentals such as π, e and $\sqrt{2}$ cannot be represented exactly by a floating-point value regardless of format or precision

How Many Floating-Point Numbers Are There?

- $\sim 2^{p+1}(2e_{max} + 1)$
- Single-precision: $\sim 4.3 \times 10^{9}$
- Double-precision: $\sim 1.8 \times 10^{19}$
- Number of protons circulating in LHC: $\sim 6.7 \times 10^{14}$

Standards

There have been three major standards affecting floating-point arithmetic:

- IEEE 754-1985 Standard for Binary Floating-Point Arithmetic
- IEEE 854-1987 Standard for Radix-Independent Floating-Point Arithmetic
- IEEE 754-2008 Standard for Floating-Point Arithmetic
  - This is the current standard
  - It is also an ISO standard (ISO/IEC/IEEE 60559:2011)

IEEE 754-2008

- Merged IEEE 754-1985 and IEEE 854-1987
  - Tried not to invalidate hardware which conformed to IEEE 754-1985
- Standardized larger formats
  - For example, quad-precision format
- Standardized new instructions
  - For example, fused multiply-add (FMA)

From now on, we will only talk about IEEE 754-2008

Operations Specified by IEEE 754-2008

All these operations must return the correct finite-precision result using the current rounding mode

- Addition
- Subtraction
- Multiplication
- Division
- Remainder
- Fused multiply add (FMA)
- Square root
- Comparison

Other Operations Specified by IEEE 754-2008

- Conversions between different floating-point formats
- Conversions between floating-point and integer formats
  - Conversion to integer must be correctly rounded
- Conversion between floating-point formats and external representations as character sequences
  - Conversions must be monotonic
  - Under some conditions, binary $\rightarrow$ decimal $\rightarrow$ binary conversions must be exact ("round-trip" conversions)

Special Values

- Zero
  - zero is signed
- Infinity
  - infinity is signed
- Subnormals
- NaN (Not a Number)
  - Quiet NaN
  - Signaling NaN
  - NaNs do not have a sign

Rounding Modes in IEEE 754-2008

The result must be the infinity-precise result rounded to the desired floating-point format.
Possible rounding modes are

- Round to nearest
  - round to nearest even
    - in the case of ties, select the result with a significand which is even
    - required for binary and decimal
    - the default rounding mode for binary
  - round to nearest away
    - required only for decimal
- round toward 0
- round toward $+\infty$
- round toward $-\infty$

Exceptions Specified by IEEE 754-2008

- Underflow
  - Absolute value of a non-zero result is less than the smallest non-zero finite floating-point number
  - Result is 0
- Overflow
  - Absolute value of a result is greater than the largest finite floating-point number
  - Result is $\pm\infty$
- Division by Zero
  - x/y where x is finite and non-zero and $y = 0$
- Inexact
  - The result, after rounding, is different than the infinitely-precise result

Exceptions Specified by IEEE 754-2008

- Invalid
  - An operand is a NaN
  - $\sqrt{x}$ where $x < 0$
    - however, $\sqrt{-0} = -0$
  - $(\pm\infty) \pm (\pm\infty)$
  - $(\pm 0) \times (\pm\infty)$
  - $(\pm 0)/(\pm 0)$
  - $(\pm\infty)/(\pm\infty)$
  - some floating-point→integer or decimal conversions

Formats Specified in IEEE 754-2008

Formats

- Basic Formats:
  - Binary with sizes of 32, 64 and 128 bits
  - Decimal with sizes of 64 and 128 bits
- Other formats:
  - Binary with a size of 16 bits
  - Decimal with a size of 32 bits

Transcendental and Algebraic Functions

The standard *recommends* the following functions be correctly rounded:

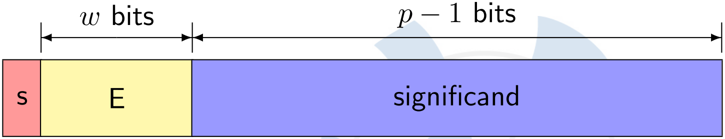
- $e^x, e^x - 1, 2^x, 2^x - 1, 10^x, 10^x - 1$
- $log_\alpha(\Phi)$ for $\alpha = e, 2, 10$ and $\Phi = x, 1 + x$
- $\sqrt{x^2 + y^2}, 1/\sqrt{x}, (1 + x)^n, x^n, x^{1/n}$
- $\sin(x), \cos(x), \tan(x), \sinh(x), \cosh(x), \tanh(x)$ and their inverse functions
- $\sin(\pi x), \cos(\pi x)$
- And more ...

We're Not Going to Consider Everything...

The rest of this talk will be limited to the following aspects of IEEE 754-2008:

- Binary32, Binary64 and Binary128 formats
    - The radix in these cases is always 2: $\beta = 2$
    - This includes the formats handled by the SSE and AVX instruction sets on the x86 architecture
    - We will not consider any aspects of decimal arithmetic or the decimal formats
    - We *will not* consider "double extended" format
        - Also known as the "IA32 x87" format
- The rounding mode is assumed to be round-to-nearest-even

Storage Format of a Binary Floating-Point Number



IEEE Name	Format	Size	w	p	e_{min}	e_{max}
Binary32	Single	32	8	24	-126	+127
Binary64	Double	64	11	53	-1022	+1023
Binary128	Quad	128	15	113	-16382	+16383

Notes:

- $E = e - e_{min} + 1$
- $e_{max} = -e_{min} + 1$
- $p - 1$ will be addressed later

The Value of a Floating-Point Number

The *format* of a floating-point number is determined by the quantities:

- *radix* β
  - sometimes called the "base"
- *sign* $s \in \{0, 1\}$
- *exponent* e
  - an integer such that $e_{min} \le e \le e_{max}$
- *precision* p
  - the number of "digits" in the number

The Value of a Floating-Point Number

The *value* of a floating-point number is determined by

- the format of the number
- the digits in the number: x_i, $0 \le i < p$, where $0 \le x_i < \beta$.

The value of a floating-point number can be expressed as

$$x = (-)^s \beta^e \sum_{i=0}^{p-1} x_i \beta^{-i}$$

where the *significand* is

$$m = \sum_{i=0}^{p-1} x_i \beta^{-i}$$

with

$$0 \le m < \beta$$

The Value of a Floating-Point Number

The value of a floating-point number can also be written

$$x = (-)^s \beta^{e-p+1} \sum_{i=0}^{p-1} x_i \beta^{p-i-1}$$

where the *integral significand* is

$$M = \sum_{i=0}^{p-1} x_i \beta^{p-i-1}$$

and M is an integer such that

$$0 \leq M < \beta^p$$

The Value of a Floating-Point Number

The value of a floating-point number can also be written as

$$x = \begin{cases} (-)^s \dfrac{M}{\beta^{-(e-p+1)}} & \text{if } e - p + 1 < 0 \\ (-)^s \beta^{e-p+1} M & \text{if } e - p + 1 \geq 0 \end{cases}$$

where M is the integral significand.

This demonstrates explicitly that a floating-point number is a rational dyadic number.

Requiring Uniqueness

$$x = (-)^s \beta^e \sum_{i=0}^{p-1} x_i \beta^{-i}$$

To make the combination of e and $\{x_i\}$ unique, x_0 must be non-zero *if possible*.

Otherwise, using binary radix ($\beta = 2$), 0.5 could be written as

- $2^{-1} \times 1 \cdot 2^0$ ($e = -1$, $x_0 = 1$)
- $2^0 \times 1 \cdot 2^{-1}$ ($e = 0, x_0 = 0, x_1 = 1$)
- $2^1 \times 1 \cdot 2^{-2}$ ($e = 1, x_0 = x_1 = 0, x_2 = 1$)
- ...

Requiring Uniqueness

This requirement to make $x_0 \neq 0$ if possible has the effect of minimizing the exponent in the representation of the number.

However, the exponent is constrained to be in the range $e_{min} \leq e \leq e_{max}$.

Thus, if minimizing the exponent would result in $e < e_{min}$, then x_0 must be 0.

A non-zero floating-point number with $x_0 = 0$ is called a subnormal number. The term "denormal" is also used.

Subnormal Floating-Point Numbers

$$m = \sum_{i=0}^{p-1} x_i \beta^{-i}$$

- If $m = 0$, then $x_0 = x_1 = \cdots = x_{p-1} = 0$ and the value of the number is ± 0
- If $m \neq 0$ and $x_0 \neq 0$, the number is a normal number with $1 \leq m < \beta$
- If $m \neq 0$ but $x_0 = 0$, the number is subnormal with $0 < m < 1$
  - The exponent of the value is e_{min}

Why have Subnormal Floating-Point Numbers?

- Subnormals allow for "gradual" rather than "abrupt" underflow
- With subnormals, $a = b \Leftrightarrow a - b = 0$

However, processing of subnormals can be difficult to implement in hardware

- Software intervention may be required
- May impact performance

Why $p - 1$?

- For normal numbers, x_0 is always 1
- For subnormal numbers and zero, x_0 is always 0
- There are many more normal numbers than subnormal numbers

An efficient storage format:

- Don't store x_0 in memory; assume it is 1
- Use a special exponent value to signal a subnormal or zero; $e = e_{min} - 1$ seems useful
  - thus $E = 0$ for both a value of 0 and for subnormals

A Walk Through the Doubles

`0x0000000000000000`	plus 0
`0x0000000000000001`	smallest subnormal
...	
`0x000fffffffffffff`	largest subnormal
`0x0010000000000000`	smallest normal
...	
`0x001fffffffffffff`	
`0x0020000000000000`	$2\times$ smallest normal
...	
`0x7fefffffffffffff`	largest normal
`0x7ff0000000000000`	$+\infty$

A Walk Through the Doubles

`0x7fefffffffffffff`	largest normal
`0x7ff0000000000000`	$+\infty$
`0x7ff0000000000001`	NaN
...	
`0x7fffffffffffffff`	NaN
`0x8000000000000000`	-0

A Walk Through the Doubles

0x8000000000000000	minus 0
0x8000000000000001	smallest -subnormal
...	
0x800fffffffffffff	largest -subnormal
0x8010000000000000	smallest -normal
...	
0x801fffffffffffff	
...	
0xffefffffffffffff	largest -normal
0xfff0000000000000	$-\infty$

A Walk Through the Doubles

0xffefffffffffffff	largest -normal
0xfff0000000000000	$-\infty$
0xfff0000000000001	NaN
...	
0xffffffffffffffff	NaN
0x0000000000000000	Back to the beginning!

Notation

- Floating-point operations are written
  - $\oplus$ for addition
  - $\ominus$ for subtraction
  - $\otimes$ for multiplication
  - $\oslash$ for division
- $a \oplus b$ represents the floating-point addition of a and b
  - a and b are floating-point numbers
  - the result is a floating-point number
  - in general, $a \oplus b \neq a + b$
  - similarly for $\ominus$, $\otimes$ and $\oslash$
- $fl(x)$ denotes the result of a floating-point operation using the current rounding mode
  - E.g., $fl(a + b) = a \oplus b$

Some Inconvenient Properties of Floating-Point Numbers

Let a, b and c be floating-point numbers. Then

- $a + b$ may not be a floating-point number
  - $a + b$ may not always equal $a \oplus b$
  - Similarly for the operations $-$, $\times$ and $/$
  - Recall that floating-point numbers do not form a field
- $(a \oplus b) \oplus c$ may not be equal to $a \oplus (b \oplus c)$
  - Similarly for the operations $\ominus$, $\otimes$ and $\oslash$
- $a \otimes (b \oplus c)$ may not be equal to $(a \otimes b) \oplus (a \oplus c)$
- $(1 \oslash a) \otimes a$ may not be equal to a

The Fused Multiply-Add Instruction (FMA)

- Computes $(a \times b) + c$ in a single instruction
- There is only one rounding
  - There are two roundings with sequential multiply and add instructions
- May allow for faster and more accurate calculation of
  - matrix multiplication
  - dot product
  - polynomial evaluation
- Standardized in IEEE 754-2008
- Execution time similar to an add or multiply but latency is greater.

The Fused Multiply-Add Instruction (FMA)

However... Use of `FMA` may change floating-point results

- $fl(a \times b + c)$ is not always the same as $(a \otimes b) \oplus c$
- The compiler may be allowed to evaluate an expression as though it were a single operation
- Consider

```
double a, b, c;
c = a >= b ? std::sqrt(a * a - b * b) : 0;
```

There are values of `a` and `b` for which the computed value of `a * a - b * b` is negative even though `a>=b`

The Fused Multiply-Add Instruction (FMA)

Consider the following example:

```
double x = 0x1.3333333333333p+0;
double x1 = x*x;
double x2 = fma(x,x,0);
double x3 - fma(x, x, -x*x));
```

```
x                   =  0x1.3333333333333p+0;
x1 = x*x            =  0x1.70a3d70a3d70ap+0
x2 = fma(x,x,0)     =  0x1.70a3d70a3d70ap+0
x3 = fma(x,x,-x*x)  = -0x1.eb851eb851eb8p-55
```

x3 is the difference between the exact value of `x*x` and its value converted to double precision. The relative error is ≈ 0.24 ulp

The Fused Multiply-Add Instruction (FMA)

Floating-point contractions

- Evaluate an expression as though it were a single operation

```
double a, b, c, d;
// Single expression; maybe replaced
// by a = FMA(b, c, d)
a = b * c + d;
```

- Combine multiple expression into a single operation

```
double a, b, c, d;
// Multiple expressions; maybe replaced
// by a = FMA(b, c, d)
a = b; a *=c; a +=d;
```

The Fused Multiply-Add Instruction (FMA)

Contractions are controlled by compiler switch(es) and `#pragmas`

- `-fpp-contract=on|off|fast`
- `#pragma STDC FP_CONTRACT ON|OFF`

IMPORTANT: Understand how your particular compiler implements these features

- `gcc` behavor has changed over time and may change in the future
- `clang` behaves differently than `gcc`

Forward and Backward Errors

The problem we wish to solve is

$$f(x) \to y$$

but the problem we are actually solving is

$$f(\hat{x}) \to \hat{y}$$

Our hope is that

$$\hat{x} = x + \Delta x \approx x$$

and

$$f(\hat{x}) = f(x + \Delta x) = \hat{y} \approx y = f(x)$$

Forward and Backward Errors

For example, if

$$f(x) = sin(x) \text{ and } x = \pi$$

then

$$y = 0$$

However, if

$$\hat{x} = \texttt{M\_PI}$$

then

$$\hat{x} \neq x \text{ and } f(\hat{x}) \neq f(x)$$

Note we are assuming that if $\hat{x} \equiv x$ then `std::sin(x̂)` $\equiv sin(x)$

Forward and Backward Errors

Absolute forward error: $|\hat{y} - y| = |\Delta y|$

Relative forward error: $\dfrac{|\hat{y} - y|}{|y|} = \dfrac{|\Delta y|}{|y|}$

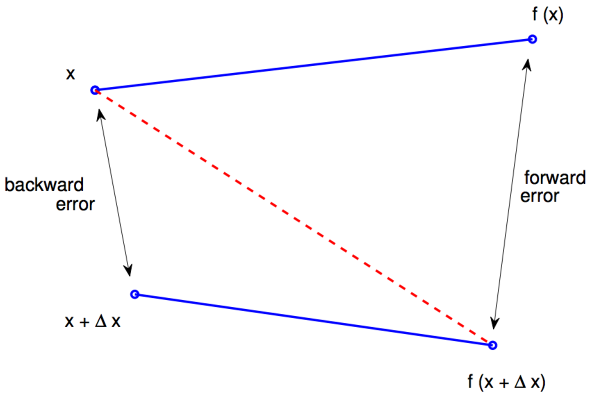
This requires knowing the exact value of y and that $y \neq 0$

Absolute backward error: $|\hat{x} - x| = |\Delta x|$

Relative backward error: $\dfrac{|\hat{x} - x|}{|x|} = \dfrac{|\Delta x|}{|x|}$

This requires knowing the exact value of x and that $x \neq 0$

Forward and Backward Errors



By J.G. Nagy, Emory University. From *Brief Notes on Conditioning, Stability and Finite Precision Arithmetic*

Condition Number

- Well conditioned: small Δx produces small Δy
- Ill conditioned: small Δx produces large Δy

$$\begin{aligned} \text{condition number} &= \frac{\text{relative change in } y}{\text{relative change in } x} \\ &= \frac{\left|\frac{\Delta y}{y}\right|}{\left|\frac{\Delta x}{x}\right|} \\ &\approx \left|\frac{x f'(x)}{f(x)}\right| \end{aligned}$$

Condition Number

- $ln\, x$ for $x \approx 1$

$$\text{Condition number } \approx \left| \frac{x f'(x)}{f(x)} \right| = \left| \frac{1}{ln\, x} \right| \rightarrow \infty$$

- $sin\, x$ for $x \approx \pi$

$$\text{Condition number } \approx \left| \frac{x}{sin\, x} \right| \rightarrow \infty$$

Error Measures

ulp: $ulp(x)$ is the place value of the least bit of the significand of x

If $x \neq 0$ and $|x| = \beta^e \sum_{i=0}^{p-1} x_i \beta^{-i}$, then $ulp(x) = \beta^{e-p+1}$

IEEE 754 and ulps

IEEE 754 requires that all results be correctly rounded from the infinitely-precise result.

If x is the infinitely-precise result and $\hat{x}$ is the "round-to-even" result, then

$$|x - \hat{x}| \leq 0.5ulp(\hat{x})$$

Approximation Error

```
const double a = 0.1;
const double b = 0.01;
```

- Both 0.1 and 0.01 are rational numbers but neither is a floating-point number
- The value of a is greater than 0.1 by $\sim 5.6 \times 10^{-18}$ or ~ 0.4 ulps
- The value of b is greater than 0.01 by $\sim 2.1 \times 10^{-19}$ or ~ 0.1 ulps

Approximation Error

```
const double a = 0.1;
const double b = 0.01;
double c = a * a;
```

- c is greater than b by 1 ulp or $\sim 1.7 \times 10^{-18}$
- c is greater than 0.01 by $\sim 1.9 \times 10^{-18} > 1$ ulp

Approximating π

```
#include <cmath>
const float a = M_PI;
const double b = M_PI;
```

- The value of a is greater than π by $\sim 8.7 \times 10^{-8}$
- The value of b is less than π by $\sim 1.2 \times 10^{-16}$

This explains why `sin(M_PI)` is not zero: the argument is not exactly π

Associativity

```
const double a = +1.0E+300;
const double b = -1.0E+300;
const double c = 1.0;
double x = ( a + b ) + c; //  x is 1.0
double y = a + ( b + c ); //  y is 0.0
```

- The order of operations matters!
- The compiler and the compilation options used matter as well
  - Some compilation options allow the compiler to re-arrange expressions
  - Some compilers re-arrange expressions by default

Distributivity

```
const double a = 10.0/3.0;
const double b = 0.1;
const double c = 0.2;
double x = a * (b + c);
//  x is 0x1.0000000000001p+0
double y = (a * b) + (a * c);
//  y is 0x1.0000000000000p+0
```

- Again, the order of operations, the compiler and the compilation options used all matter

The “Pigeonhole” Principle

- You have $n+1$ pigeons (i.e., discrete objects)
- You put them into n pigeonholes (i.e., boxes)
- At least one pigeonhole contains more than one pigeon.

The "Pigeonhole" Principle

An example of using the "Pigeonhole" Principle:

- The number of IEEE Binary64 numbers in $[1, 2)$ is $N = 2^{52}$
- The number of IEEE Binary64 numbers in $[1, 4)$ is $2N$
- Each value in $[1, 4)$ has its square root in $(1, 2]$
- Since there are more values in $[1, 4)$ than in $[1, 2)$, there must be at least two distinct floating-point numbers in $[1, 4)$ which have the same square root

Catastrophic Cancellation

Catastrophic cancellation occurs when two nearly equal floating-point numbers are subtracted.

If $x \approx y$, their signifcands are nearly identical. When they are subtracted, only a few low-order digits remain. I.e., the result has very few significant digits left.

Sterbenz's Lemma

Lemma

Let a and b be floating-point numbers with

$$b/2 \leq a \leq 2b$$

. If subnormal numbers are available, $a \ominus b = a - b$.

Thus there is no rounding error associated with $a \ominus b$ when a and b satisfy the criteria.

However, there may be lost of significance.

Error-Free Transformations

An error-free transformation (EFT) is an algorithm which transforms a (small) set of floating-point numbers into another (small) set of floating-point numbers of the same precision *without any loss of information*.

$$f(x, y) \longmapsto (s, t)$$

Error-Free Transformations

EFTs are most useful when they can be implemented using only the precision of the floating-point numbers involved.

EFTs exist for

- Addition: $a + b = s + t$ where $s = a \oplus b$
- Multiplication: $a \times b = s + t$ where $s = a \otimes b$
- Splitting: $a = s + t$

Additional EFTs can be derived by composition. For example, an EFT for dot products makes use of those for addition and multiplication.

An EFT for Addition

Require: $|a| \geq |b|$

1: $s \leftarrow a \oplus b$

2: $t \leftarrow b \ominus (s \ominus a)$

3: **return** (s, t)

Ensure: $a + b = s + t$ where $s = a \oplus b$ and t are floating-point numbers

A possible implementation

```
void
FastSum(const double a, const double b,
        double * const s, double * const t) {
    // No unsafe optimizations!
    *s = a+b;
    *t = b-(*s-a);
    return;
}
```

Another EFT for Addition: TwoSum

$s \leftarrow a \oplus b$

$z \leftarrow s \ominus a$

$t \leftarrow (a \ominus (s \ominus z) \oplus (b \ominus z)$

return (s, t)

Ensure: $a + b = s + t$ where $s = a \oplus b$ and t are floating-point numbers

A possible implementation

```
void
TwoSum(const double a, const double b,
       double * const s, double * const t) {
    // No unsafe optimizations!
    *s = a+b;
    double z = *s-a;
    *t = (a-(*s-z))+(b-a);
    return;
```

Comparing `FastSum` and `TwoSum`

- A realistic implementation of `FastSum` requires a branch and 3 floating-point opertions
- `TwoSum` takes 6 floating-point operations but requires no branches
- `TwoSum` is usually faster on modern pipelined processors
- The algorithm used in `TwoSum` is valid in radix 2 even if underflow occurs but fails with overflow

Precise Splitting Algorithm

- Given a base-2 floating-point number x, determine the floating-point numbers x_h and x_l such that $x = x_h + x_l$
- For $0 < \delta < p$, where p is the precision and δ is a parameter,
  - The signficand of x_h fits in $p - \delta$ bits
  - The signficand of x_l fits in $\delta - 1$ bits
  - All other bits are 0
  - δ is typically chosed to be $\lceil p/2 \rceil$
- No information is lost in the transformation
  - Aside: how do we end up only needing $(p - \delta) + (\delta - 1) = p - 1$ bits?
- This scheme is known as Veltkamp's algorithm

Precise Splitting EFT

Require: $C = 2^s + 1$; $C \otimes x$ does not overflow

$a \leftarrow C \otimes x$

$b \leftarrow x \ominus a$

$x_h \leftarrow a \oplus b$

$x_l \leftarrow x \ominus x_h$

return (x_h, x_l)

Ensure: $x = x_h + x_l$

Precise Splitting EFT

Possible implementation

```
void
Split(const double x, const int delta,
      double * const x_h, double * const x_l) {
    // No unsafe optimizations!
    double c = (double)((1UL << delta) + 1);
    *x_h = (c * x) + (x - (c * x));
    *x_l = x - *x_h;
    return;
}
```

Precise Multiplication

- Given floating-point numbers x and y, determine floating-point numbers s and t such that $a \times b = s + t$ where $s = a \otimes b$ and

$$t = ((((x_h \otimes y_h) \ominus s) \oplus (x_h \otimes y_l)) \oplus (x_l \otimes y_h)) \oplus (x_l \otimes y_l).$$

- Known as Dekker's algorithm

Precise Multiplication EFT

The algorithm is much simpler using FMA

1: $s \leftarrow x \otimes y$
2: $t \leftarrow \boldsymbol{FMA}(x, y, -s)$
3: **return** (s, t)

Ensure: $x * y = s + t$ where $s = x \otimes y$ and t are floating-point numbers

Possible implementation

```
void
Prod(const double a, const double b,
     double * const s, double * const t) {
  // No unsafe optimizations!
  *s = a * b;
  *t = FMA(a, b, -*s);
  return;
}
```

Summation Techniques

- Traditional
- Sorting and Insertion
- Compensated
- Reference: Higham: *Accuracy and Stability of Numerical Algorithms*

Summation Techniques

Condition number:

$$C_{sum} = \frac{\sum_i |x_i|}{|\sum_i x_i|}$$

- If C_{sum} is not too large, the problem is not ill-conditioned and traditional methods may be sufficient
- If C_{sum} is too large, we need to have results appropriate to a higher precision *without actually using a higher precision*
- Obviously, if higher precision is readily available, use it

Traditional Summation

$$s = \sum_{i=0}^{n-1} x_i$$

```
double
Sum(const double* x, const unsigned int n)
{   // No unsafe optimizations!
  double sum = x[0]
  for(unsigned int i = 1; i < n; i++) {
    sum += x[i];
  }
  return;
}
```

Sorting and Insertion

- Reorder the operands
  - By value or magnitude
  - Increasing or decreasing
- Insertion
  - First sort by magnitude
  - Remove x_1 and x_2 and compute their sum
  - Insert that value into the list keeping the list sorted
  - Repeat until only one element is in the list
- Many Variations
  - If lots of cancellations, sorting by decreasing magnitude may be better but not always

Compensated Summation

- Based on `FastTwoSum` and `TwoSum` techniques
- Knowledge of the exact rounding error in a floating-point addition is used to correct the summation
- Developed by William Kahan

Compensated (Kahan) Summation

```
Function Kahan (x,n)
    Input: n > 0
    s ← x_0
    t ← 0
    for i = 1 to n − 1 do
        y ← x_i − t // Apply correction
        z ← s + y // New sum
        t ← (z − s) − y // New correction ≈ low part of y
        s ← z // Update sum
    end
    return s
```

Compensated (Kahan) Summation

```
double
Kahan(const double* x, const unsigned int n)
{   // No unsafe optimizations!
  double s = x[0];
  double t = 0.0;
  for( int i = 1; i < n_values; i++ ) {
    double y = x[i] - t;
    double z = s + y;
    t = ( z - s ) - y;
    s = z;
  }
  return s;
}
```

Compensated Summation

Many variations known. Consult the literature for papers with these authors:

- William M Kahan
- Donald Knuth
- Douglas Priest
- S M Rump, T Ogita and S Oishi
- Jonathan Shewchuk
- AriC project (CNRS/ENS Lyon/INRIA)

Choice of Summation Technique

- Performance
- Error Bound
  - Is it (weakly) dependent on n?
- Condition Number
  - Is it known?
  - Is it difficult to determine?
  - Some algorithms allow it to be determined simultaneously with an estimate of the sum
  - Permits easy evaluation of the suitability of the result
- No one technique fits all situations all the time

Dot Product

$$
\begin{aligned}
S &= \boldsymbol{x}^{\mathsf{T}}\boldsymbol{y} \\
&= \sum_{i=0}^{n-1} x_i \cdot y_i
\end{aligned}
$$

where $\boldsymbol{x}$ and $\boldsymbol{y}$ are vectors of length n.

Dot Product

Traditional algorithm

Require: $\boldsymbol{x}$ and $\boldsymbol{y}$ are n-dimensional vectors with $n \geq 0$

1: $s \leftarrow 0$

2: **for** $i = 0$ to $n - 1$ **do**

3: $\quad s \leftarrow s \oplus (x_i \otimes y_i)$

4: **end for**

5: **return** s

Dot Product

The error in the result is proportional to the condition number:

$$C_{dot\ product} = 2 \times \frac{\sum_i |x_i| \cdot |y_i|}{|\sum_i x_i \cdot y_i|}$$

- If C is not too large, a traditional algorithm can be used
- If C is large, more accurate methods are required
  - E.g., lots of cancellation

How to tell? Compute the condition number simultaneously when computing the dot product

Dot Product

FMA can be used in the traditional computation

Require: $\boldsymbol{x}$ and $\boldsymbol{y}$ are n-dimensional vectors with $n \geq 0$

$s \leftarrow 0$
for $i = 0$ to $n - 1$ **do**
    $s \leftarrow FMA(x_i, y_i, s)$
end for
return s

Although there are fewer rounded operations than in the traditional scheme, using FMA does not improve the worst case accuracy.

Dot Product

Recall

- $Sum(x, y)$ computes s and t with $x + y = s + t$ and $s = x \oplus y$
- $Prod(x, y)$ computes s and t with $x + y = s + t$ and $s = x \otimes y$

Since each individual product in the sum for the dot product is transformed using $Prod(x, y)$ into the sum of two floating-point numbers, the dot product of 2 vectors can be reduced to computing the sum of $2N$ floating-point numbers.

To accurately compute that sum, $Sum(x, y)$ is used.

Dot Product

Compensated dot product algorithm

Require: $\boldsymbol{x}$ and $\boldsymbol{y}$ are n-dimensional vectors with $n \geq 0$

1: $(s_h, s_l) \leftarrow (0, 0)$
2: **for** $i = 0$ to $n - 1$ **do**
3: $\quad (p_h, p_l) \leftarrow Prod(x_i, y_i)$
4: $\quad (s_h, a) \leftarrow Sum(s_h, p_h)$
5: $\quad s_l \leftarrow s_l \oplus (p_l \oplus a)$
6: **end for**
7: **return** $s_h \oplus s_l$

The relative accuracy of this algorithm is the same as the traditional algorithm when computed using twice the precision.

Polynomial Evaluation

Evaluate

$$\begin{aligned} p(x) &= \sum_{i=0}^{n} a_i x^i \\ &= a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n \end{aligned}$$

Condition number

$$C(p, x) = \frac{\sum_{i=0}^{n} |a_i| \, |x|^i}{\left|\sum_{i=0}^{n} a_i x^i\right|}$$

Note that $C(p, x) = 1$ for certain combinations of a and x. E.g., if $a_i \geq 0$ for all i and $x \geq 0$.

Horner's Scheme

Nested multiplication is a standard method for evaluating $p(x)$:

$$p(x) = (((a_n x + a_{n-1})x + a_{n-2})x \cdots + a_1)x + a_0$$

This is known as Horner's scheme (although Newton published it in 1711!)

Horner's Scheme

Function *Horner (x,p,n)*

Input: $n \geq 0$

$s_n \leftarrow a_n$

for $i = n - 1$ **downto** 0 **do**

// $s_i \leftarrow (s_{i+1} \times x) + a_i$

$s_i \leftarrow \texttt{FMA}(s_{i+1}, x, a_i)$

end

return s_0

Horner's Scheme

A possible implementation

```
double
Horner(const double x,
       const double * const a,
       const int n) {
    double s = a[ n ];
    for (int i = n - 1; i >= 0; i--} (
        // s = s * x + a[ i ];
        s = FMA(s, x, a[ i ]);
    }
    return s;
}
```

Applying EFTs to Horner's Scheme

Horner's scheme can be improved by applying the EFTs `Sum` and `Prod`

Function *HornerEFT (x,p,n)*

Input: $n \geq 0$

$s_n \leftarrow a_n$

for $i = n-1$ **downto** 0 **do**

$(p_i, \pi_i) \leftarrow \mathbf{Prod}(s_{i+1}, x)$

$(s_i, \sigma_i) \leftarrow \mathbf{Sum}(p_i, a_i)$

end

return s_0, π, σ

The value of s_0 calculated by this algorithm is the same as that using the traditional Horner's scheme.

Applying EFTs to Horner's Scheme

Let π and σ from `HornerEFT` be the coefficients of polynomials of degree $n-1$. Then the quantity

$$s_0 + (\pi(x) + \sigma(x))$$

is an improved approximation to

$$\sum_{i=0}^{n} a_i x^i$$

In fact, the relative error from `HornerEFT` is the same as that obtained using the traditional algorithm with twice the precision.

Simultaneous calculation of a dynamic error bound can also be incorporated into this algorithm.

Second Order Horner's Scheme

Horner's scheme is sequential: each step of the calculation depends on the result of the preceeding step.

Consider

$$
\begin{aligned}
p(x) &= a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n \\
&= (a_0 + a_2 x^2 + \cdots) + x(a_1 + a_3 x^2 + \cdots) \\
&= q(x^2) + x r(x^2)
\end{aligned}
$$

- The calculations of $q(x^2)$ and $r(x^2)$ can be done in parallel
- This technique may be applied recursively

Estrin's Method

Isolate subexpressions of the form $(a_k + a_{k+1}x)$ and x^{2n} from $p(x)$:

$$p(x) = (a_o + a_1 x) + (a_2 + a_3 x)x^2 + ((a_4 + a_5 x) + (a_6 + a_7 x)x^2)x^4 + \cdots$$

The subexpressions $(a_k + a_{k+1}x)$ can be evaluated in parallel

Value Safety

"Value Safety" refers to transformations which, although algebraically valid, may affect floating-point results.

Ensuring "Value Safety" requires that no optimizations be done which could change the result of any series of floating-point operations as specified by the programming language.

- Changes to underflow or overflow behavior
- Effects of an operand which is not a finite floating-point number. E.g., $\pm\infty$ or a NaN

Transformations which violate "Value Safety" are not error free transformations.

Value Safety

In "safe" mode, the compiler may not make changes such as

$(x+y)+z \Leftrightarrow x+(y+z)$	Reassociations are not value-safe
$x*(y+z) \Leftrightarrow x*y+x*z$	Distributions are not value-safe
$x*(y*z) \Leftrightarrow (x*y)*z$	May change under-/overflow behavior
$x/x \Leftrightarrow 1.0$	x may be 0, ∞ or a NaN
$x+0 \Leftrightarrow x$	x may be -0 or a NaN
$x*0 \Leftrightarrow 0$	x may be -0, ∞ or a NaN

A Note on Compiler Options

- There are many compiler options which affect floating-point results
- Not all of them are obvious
- Some of them are enabled/disabled by other options
  - `-On`
  - `-march` and others which specify platform characteristics
- Options differ among compilers

Optimizations Affecting Value Safety

- Expression rearrangements
- Flush-to-zero
- Approximate division and square root
- Math library accuracy

Expression Rearrangements

These rearrangements are not value-safe:

- $(a \oplus b) \oplus c \Rightarrow a \oplus (b \oplus c)$
- $a \otimes (b \oplus c) \Rightarrow (a \otimes b) \oplus (a \oplus c)$

To disallow these changes:

gcc Don't use `-ffast-math`

icc Use `-fp-model precise`

- Recall that options such as `-On` are "aggregated" or "composite" options
  - they enable/disable many other options
  - their composition may change with new compiler releases

Disallowing rearrangements may affect performance

Subnormal Numbers and Flush-To-Zero

- Subnormal numbers extend the range of floating-point numbers but with reduced precision and reduced performance
- If you do not require subnormals, disable their generation
- "Flush-To-Zero" means "Replace all generated subnormals with 0"
    - Note that this may affect tests for == 0.0 and != 0.0
- If using SSE or AVX, this replacement is fast since it is done by the hardware

Subnormal Numbers and Flush-To-Zero

- gcc `-ffast-math` enables flush-to-zero
- gcc But `-O3 -ffast-math` disables flush-to-zero
- icc Done by default at -O1 or higher
- icc Use of `-no-ftz` or `fp-model precise` to will prevent this
- icc Use `-fp-model precise -ftz` to get both "precise" behavior and subnormals
  - Options must be applied to the program unit containing `main` as well

Reductions

- Summation is an example of a reduction
- Parallel implementations of reductions are inherently value-unsafe because they may change the order of operations
  - the parallel implementation can be through vectorization or multi-threading or both
  - there are OpenMP and TBB options to make reductions "reproducible"
    - For OpenMP `KMP_DETERMINSTIC_REDUCTION=yes`

icc use of `-fp-model precise` disables automatic vectorization and automatic parallelization via threading

The Hardware Floating-Point Environment

The hardware floating-point environment is controled by several CPU control words

- Rounding mode
- Status flags
- Exception mask
- Control of subnormals

If you change anything affecting the assumed state of the processor with respect to floating-point behavior, you must tell the compiler

- Use `#pragma STDC FENV_ACCESS ON`

icc Use `-fp-model strict`

`#pragma STDC FENV_ACCESS ON` is required if flags are accessed

Precise Exceptions

Precise Exceptions: floating-point exceptions are reported exactly when they occur

To enable precision exceptions

- Use `#pragma float_control(except, on)`

icc Use `-fp-model strict` or `-fp-model except`

Enabling precise exceptions disables speculative execution of floating-point instructions. This will probably affect performance.

Math Library Features – icc

A variety of options to control precision and consistency of results

- `-fimf-precision=<high|medium|low>[:funclist]`
- `-fimf-arch-consistency=<true|false>[:funclist]`
- And several more options
  - `-fimf-absolute-error=<value>[:funclist]`
  - `-fimf-accuracy-bits=<value>[:funclist]`
  - ...

Tools

- `double-double` and `quad-double` data types
  - Implemented in C++
  - Fortran 90 interfaces provided
  - Available from LBL as qd-X.Y.Z.tar.gz
  - "LBNL-BSD" type license

Tools

GMP – The GNU Multiple Precision Arithmetic Library

- a `C` library
- arbitrary precision arithmetic for
  - signed integers
  - rational numbers
  - floating-point numbers
- used by `gcc` and `g++` compilers
- `C++` interfaces
- GNU LGPL license

Tools

MPFR

- a `C` library for multiple-precision floating-point computations
- all results are correctly rounded
- used by gcc and g++
- C++ interface available
- free with a GNU LGPL license

Tools

`CRlibm`

- a C library
- all results are correctly rounded
- C++ interface available
- Python bindings available
- free with a GNU LGPL license

Tools

- `limits`
  - defines characteristics of arithmetic types
  - provides the template for the class `numeric_limits`
  - `#include <limits>`
  - requires `-std=c++11`
  - specializations for each fundamental type
  - compiler and platform specific

Tools

- `cmath`
  - functions to compute common mathematical operations and transformations
  - `#include <cmath>`
  - `frexp`
    - get exponent and significand
  - `ldexp`
    - create value from exponent and significand
  - Note: `frexp` and `ldexp` assume a different "normalization" than usual: $1/2 \leq m < 1$
  - `nextafter`
    - create next representable value
  - `fpclassify`
    - returns one of `FP_INFINITE`, `FP_NAN`, `FP_ZERO`, `F_SUBNORMAL`, `FP_NORMAL`

Pitfalls and Gremlins

Catastrophic Cancellation

- $x^2 - y^2$ for $x \approx y$
  - $(x + y)(x - y)$ may be preferable
  - $x - y$ is computed with no round-off error (Sterbenz's Lemma)
  - $x + y$ is computed with relatively small error
- `FMA(x,x,-y*y)` can be very accurate
  - However `FMA(x,x,-x*x) is not usually 0!`
- similarly $1 - x^2$ for $x \approx 1$
  - `-FMA(x,x,-1.0)` is very accurate

Pitfalls and Gremlins

"Gratuitous" Overflow

Consider $\sqrt{x^2+1}$ for large x

- $\sqrt{x^2+1} \to |x|$ as $|x| \to \infty$
- $|x|\sqrt{1+1/x^2}$ may be preferable
    - if x^2 overflows, $1/x \to 0$
    - $|x|\sqrt{1+1/x^2} \to |x|$

Pitfalls and Gremlins

Consider the Newton-Raphson iteration for $1/\sqrt{x}$:

$$y_{n+1} \leftarrow y_n(3 - xy_n^2)/2$$

where $y_n \approx 1/\sqrt{x}$. Since $xy_n^2 \approx 1$, there is at most an alignment shift of 2 when computing $3 - xy_n^2$, and the final operation consists of multiplying y_n by a computed quantity near 1. (The division by 2 is exact.)

If the iteration is rewritten as

$$y_n + y_n(1 - xy_n^2)/2,$$

the final addition involves a large alignment shift between y_n and the correction term $y_n(1 - xy_n^2)/2$ avoiding cancellation.

Pitfalls and Gremlins

This situation can be generalized:

When calculating a quantity from other calculated (i.e., inexact) values, try to formulate the expressions so that the final operation is an addition of a smaller "correction" term to a value which is close to the final result.

Pitfalls and Gremlins

Vectorization and Parallelization

These optimizations affect both results and reproducibility

- Results can change because the order of operations may change
- Vector sizes also affect the order of operations
- Parallalization can change from run to run (e.g., number of threads available). This impacts both results and reproducibily

Pitfalls and Gremlins

And finally... CPU manufacturer can impact results. Not all floating-point instructions execute exactly the same on AMD and Intel processors

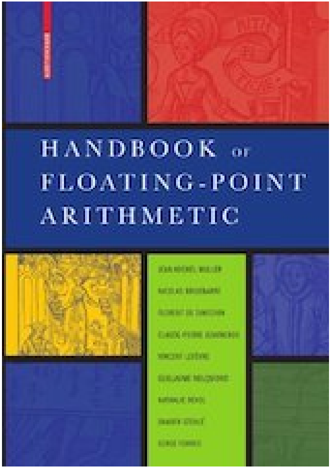
- The `rsqrt` and `rcp` instructions differ
- They are not standardized
- Both implementations meet the specification given by Intel

The exact same non-vectorized, non-parallelized, non-threaded application may give different results on systems with similar processors each vendor.

Pitfalls and Gremlins

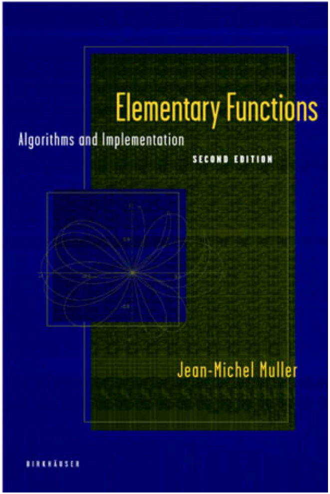
And undoubtedly others, as yet undiscovered.

Bibliography



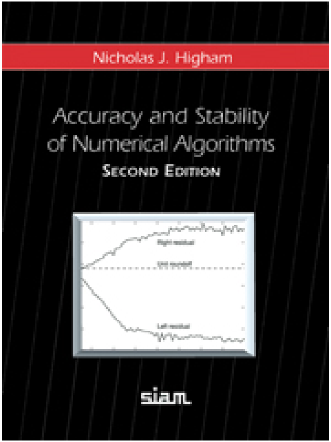
J.-M. Muller et al, *Handbook of Floating-Point Arithmetic*, Birkäuser, Boston, 2010

Bibliography



J.-M. Muller, *Elementary Functions, Algorithms and Implementation* (2^{nd} *Edition)*, Birkäuser, Boston, 2006

Bibliography



N.J. Higham, *Accuracy and Stability of Numerical Algorithms (2^{nd} Edition)*, Siam, Philadelphia, 2002.

Bibliography

- IEEE, *IEEE Standard for Floating-Point Arithmetic*, IEEE Computer Society, August 2008.
- D. Goldberg, *What every computer scientist should know about floating-point arithmetic*, ACM Computing Surverys, 23(1):5-47, March 1991
- Publications from CNRS/ENS Lyon/INRIA/AriC project (J.-M. Muller et al).
- Publications from the PEQUAN project at LIP6, Université Pierre et Marie Curie (Stef Graillat, Christoph Lauter et al).
- Publications from Institut für Zuverlässiges Rechnen (Institute for Reliable Computing), Technische Universität Hamburg-Harburg (Siegfried Rump et al).

Technology, Innovation, Knowledge

Training

- Computing focus
- From beginner to expert level
- Hands-on

Experienced teachers

- Track record in major tech outlets
- Over 1'600 trained

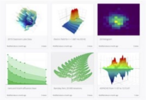
Innovation support

- Tech analysis
- Innovation projects
- Computing strategies

tik.

TIK training topics

- Software/hardware performance and architecture
- Parallelization, vectorization, accelerators
- Compilers, numerical processing
- Python for science, data analysis and visualization



- Full catalog available online here: http://tik.services/knowledge
- Ask your Dept. Training Officers or Technical Training

tik.