Fundamentals of Floating-Point Arithmetic

Jeff Arnold

5 May 2014

Agenda

- Part I – Fundamentals
  - Motivation
  - Some properties of floating-point numbers
  - Standards
  - More about floating-point numbers
  - A trip through the floating-point numbers
- Part II – Techniques
  - Error-Free Transformations
  - Techniques for Summation
  - Techniques for Multiplication
  - Techniques for Dot Product
  - Techniques for Polynomial Evaluation

Motivation

- Why is floating-point arithmetic important?
- Reasoning about floating-point arithmetic
- Why do standards matter?
- Techniques which improve floating-point
    - Accuracy
    - Versatility
    - Performance

Why is Floating-point Arithmetic Important?

- It is ubiquitous in scientific computing
  - Most research in HEP can't be done without it
- We need to implement algorithms which
  - Get the best answers
  - Get the best answers quickly
  - Get the best answers all the time
  - Where "best" means the right answer for the situation and context
- A rigorous approach to floating-point arithmetic is seldom taught in programming courses
- Too many think floating-point is
  - approximate in some random, ill-defined sense
  - mysterious
  - can often be wrong

Reasoning about Floating-Point Arithmetic

Being able to reason about floating-point arithmetic is important because

- One can prove algorithms are correct
  - Or one can determine the conditions under which they fail
- One can prove algorithms are portable
- One can estimate the round-off error 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 results

Some Properties of Floating-Point Numbers

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

- The set of floating-point numbers does not form a field under the usual set of floating-point operations
- Some common rules of arithmetic are not always valid when applied to floating-point numbers
- Floating-point numbers are all rational numbers
  - but they are only a subset of the rationals
  - thus not all rational numbers are floating-point numbers
- There are only a finite number of floating-point numbers

Floating-Point Numbers are Rationals

Implications:

- The decimal equivalent of any finite floating-point value contains a finite number of non-zero digits
- The values of π, e, $\sqrt{2}$ etc cannot be represented exactly by a floating-point value

Approximating π

Consider

```
#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 $\sim 1.2 \times 10^{-16}$

How Many Floating-Point Numbers Are There?

- Single-precision: $\sim 4.3 \times 10^9$
- Double-precision: $\sim 1.8 \times 10^{19}$
- Number of protons circulating in LHC: $\sim 3.2 \times 10^{14}$

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 Binary Floating-Point Arithmetic
  - This is the current standard
  - It is also an ISO standard (ISO/IEC/IEEE 60559:2011)

A Bit of History

- IEEE 754-1985
  - Described binary arithmetic
  - Specified single and double precision along with an "extended" format for each
  - Single precision was required
- IEEE 854-1987
  - Described "radix-independent" arithmetic
  - Established constraints on
    - precision and exponent range
    - between various formats

A Bit of History

By 2000, revisions were needed

- New formats were being used
- New instructions were being introduced
- New algorithms were developed for computations which had previously been considered "too difficult" to implement and, thus, not standardized
  - Radix conversion
  - Correctly-rounded elementary functions
- There were ambiguities in the existing standards
- It was difficult to write portable code which met certain requirements of the standard

IEEE 754-2008 – A New Standard

- Merged IEEE 754-1985 and IEEE 854-1987
    - But 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
- Square root

Operations Specified by IEEE 754-2008

- Conversion to/from integer
  - Conversion to integer must be correctly rounded
- Conversion to/from decimal strings
  - Conversions must be monotonic
  - Under some conditions, binary $\rightarrow$ decimal $\rightarrow$ binary conversions must be exact ("round-trip" conversions)

Rounding Modes in IEEE 754-2008

- 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 $+\infty$
  - round toward $-\infty$
  - round toward 0

Special Values

- Zero
  - zero is signed
- Infinity
  - infinity is signed
- NaN (Not a Number)
  - Quiet NaN
  - Signaling NaN
  - NaNs do not have a sign
    - afterall, they aren't a number
- Subnormals

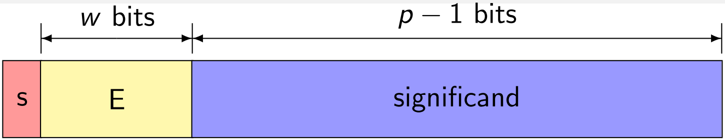
Exceptions Specified by IEEE 754-2008

- Underflow
  - Absolute value of a non-zero result is less than $\beta^{e_{min}}$ (i.e., it is subnormal)
  - Some ambiguity: before or after rounding?
- Overflow
  - Absolute value of a result is greater than the largest finite value $2^{e_{max}} \times (2 - 2^{1-p})$ in binary
  - 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 not the same as 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$\rightarrow$integer or decimal conversions

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:

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

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
    - $p = 11$
    - $e_{min} = -14$, $e_{max} = +15$
  - Decimal with a size of 32 bits

Larger Formats in IEEE 754-2008

- Parameterized based on size k:
  - k must be a multiple of 32
  - $k \geq 128$
  - $p = k - roundnearest(4 \times log_2(k)) + 13$
  - $w = k - p$
  - $e_{max} = 2^{w-1} - 1$
- Example: Binary1024
  - $k = 1024$
  - $p = 1024 - 40 + 13 = 997$
  - $w = 27$
  - $e_{max} = +67108863$

The Value of a Floating-Point Number

The value of a floating-point number is determined by four quantities:

- *radix* β
  - sometimes called the "base"
- *sign* $s \in \{0, 1\}$
- *exponent* e
  - an integer such that $e_{min} \leq e \leq e_{max}$
- *precision* p
  - the digits are x_i, $0 \leq i < p$, where $0 \leq x_i < \beta$

The Value of a Floating-Point Number

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

$$0 \leq m < \beta$$

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

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

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

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

In other words: minimize the exponent

Subnormal Floating-Point Numbers

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

- If $m = 0$, the value of the number is ± 0
- If $m \neq 0$
  - If x_0 is non-zero, the number is a normal number
    - $1 \leq m < \beta$
  - If x_0 is zero, the number is a subnormal number
    - $0 < m < 1$
    - This is what happens if minimizing the exponent would cause it to go below e_{min}

Why have Subnormal Floating-Point Numbers?

– Processing of subnormals can be difficult to implement in hardware
  - Software intervention may be required

+ Subnormals allow for “gradual underflow”

+ With subnormals, $a = b \Leftrightarrow a - b = 0$
  - If a and b are floating-point numbers and $a \neq b$, then $a \ominus b$ is guaranteed to be non-zero if subnormals exist

Why $p - 1$?

- For normal numbers, x_0 is always 1
- For subnormal numbers and zero, x_0 is always 0
- 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

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 . . .

Transcendental Functions

Why correct rounding may be difficult to accomplish in all cases

Consider $2^{0x1.e4596526bf94dp-10}$ in round-to-nearest

- The precise answer is
  `0x1.0053fc2ec2b537ffffffffffffff4...`
- The result must be calculated to at least 114 bits to determine the correctly rounded result because it is very close to the midpoint between two floating-point numbers
  - the midpoint is `0x1.0053fc2ec2b5380...`
- This is known as the "Table Maker's Dilemma"

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 is thus 2: $\beta = 2$
  - This includes the formats handled by the SSE and AVX instruction sets
  - We will not consider any aspects of decimal arithmetic or the decimal formats
  - We *will not* consider "double extended" format
    - Also known as "IA32 x87" format
- We will always assume the rounding mode is round-to-nearest-even

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

Associativity

Consider

```
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!
- So do the compiler and the compilation options used

Distributivity

Consider

```
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

Approximation Error

Consider

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

- The representation of a using round-to-nearest-even is `0x1.999999999999ap-4`
- The value of a is greater than 0.1 by $\sim 5.6 \times 10^{-18}$ or ~ 0.4 ulps
- The representation of b using round-to-nearest-even is `0x1.47ae147ae147bp-7`
- The value of b is greater than 0.01 by $\sim 2.1 \times 10^{-19}$ or ~ 0.1 ulps

Approximation Error

Consider

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

- The value of c is `0x1.47ae147ae147cp-7`
- 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

Walking Through the Floating-Point Numbers

From `0x0000000000000000` to `0xffffffffffffffff`

Walking Through the Floating-Point Numbers

0x0000000000000000	$+0.0$	Positive zero

Walking Through the Floating-Point Numbers

0x0000000000000000	$+0.0$	Positive zero
0x0000000000000001	$+2^{e_{min}} \times 2^{1-p}$ $\sim 4.9 \times 10^{-324}$	Smallest denormal > 0

All floating-point values are a multiple of this quantity

Walking Through the Floating-Point Numbers

0x0000000000000000	$+0.0$	Positive zero
0x0000000000000001	$+2^{e_{min}} \times 2^{1-p}$	Smallest denormal > 0
...		
0x000fffffffffffff	$+2^{e_{min}}(1 - 2^{1-p})$	Largest denormal > 0

Walking Through the Floating-Point Numbers

0x0000000000000000	$+0.0$	Positive zero
0x0000000000000001	$+2^{e_{min}} \times 2^{1-p}$	Smallest denormal > 0
...		
0x000fffffffffffff	$+2^{e_{min}}(1 - 2^{1-p})$	Largest denormal > 0
0x0010000000000000	$+2^{e_{min}}$	Smallest normal > 0
	$\sim 2.2 \times 10^{-308}$	

Walking Through the Floating-Point Numbers

0x0000000000000000	$+0.0$	Positive zero
0x0000000000000001	$+2^{e_{min}} \times 2^{1-p}$	Smallest denormal > 0
...		
0x000fffffffffffff	$+2^{e_{min}}(1 - 2^{1-p})$	Largest denormal > 0
0x0010000000000000	$+2^{e_{min}}$	Smallest normal > 0
...		
0x001fffffffffffff	$+2^{e_{min}}(2 - 2^{1-p})$	

Walking Through the Floating-Point Numbers

0x0000000000000000	$+0.0$	Positive zero
0x0000000000000001	$+2^{e_{min}} \times 2^{1-p}$	Smallest denormal > 0
...		
0x000fffffffffffff	$+2^{e_{min}}(1 - 2^{1-p})$	Largest denormal > 0
0x0010000000000000	$+2^{e_{min}}$	Smallest normal > 0
...		
0x001fffffffffffff	$+2^{e_{min}}(2 - 2^{1-p})$	
0x0020000000000000	$+2^{e_{min}+1}$	

Walking Through the Floating-Point Numbers

0x0000000000000000	$+0.0$	Positive zero
0x0000000000000001	$+2^{e_{min}} \times 2^{1-p}$	Smallest denormal > 0
...		
0x000fffffffffffff	$+2^{e_{min}}(1 - 2^{1-p})$	Largest denormal > 0
0x0010000000000000	$+2^{e_{min}}$	Smallest normal > 0
...		
0x001fffffffffffff	$+2^{e_{min}}(2 - 2^{1-p})$	
0x0020000000000000	$+2^{e_{min}+1}$	
...		
0x7fefffffffffffff	$+2^{e_{max}}(2 - 2^{1-p})$ $\sim 1.8 \times 10^{+308}$	Largest normal > 0

Walking Through the Floating-Point Numbers

0x0000000000000000	$+0.0$	Positive zero
0x0000000000000001	$+2^{e_{min}} \times 2^{1-p}$	Smallest denormal > 0
...		
0x000fffffffffffff	$+2^{e_{min}}(1 - 2^{1-p})$	Largest denormal > 0
0x0010000000000000	$+2^{e_{min}}$	Smallest normal > 0
...		
0x001fffffffffffff	$+2^{e_{min}}(2 - 2^{1-p})$	
0x0020000000000000	$+2^{e_{min}+1}$	
...		
0x7fefffffffffffff	$+2^{e_{max}}(2 - 2^{1-p})$	Largest normal > 0
0x7ff0000000000000	$+\infty$	Positive infinity

Walking Through the Floating-Point Numbers

0x0000000000000000	$+0.0$	Positive zero
0x0000000000000001	$+2^{e_{min}} \times 2^{1-p}$	Smallest denormal > 0
...		
0x000fffffffffffff	$+2^{e_{min}}(1 - 2^{1-p})$	Largest denormal > 0
0x0010000000000000	$+2^{e_{min}}$	Smallest normal > 0
...		
0x001fffffffffffff	$+2^{e_{min}}(2 - 2^{1-p})$	
0x0020000000000000	$+2^{e_{min}+1}$	
...		
0x7fefffffffffffff	$+2^{e_{max}}(2 - 2^{1-p})$	Largest normal > 0
0x7ff0000000000000	$+\infty$	Positive infinity
0x7ff0000000000001		NaN
...		
0x7fffffffffffffff		NaN

Walking Through the Floating-Point Numbers

0x8000000000000000	-0.0	Negative zero

Walking Through the Floating-Point Numbers

`0x8000000000000000`	-0.0	Negative zero
`0x8000000000000001`	$-2^{e_{min}-p+1}$	Smallest denormal < 0
...		
`0x800fffffffffffff`	$-2^{e_{min}}(1-2^{1-p})$	Largest denormal < 0

Walking Through the Floating-Point Numbers

`0x8000000000000000`	-0.0	Negative zero
`0x8000000000000001`	$-2^{e_{min}-p+1}$	Smallest denormal < 0
...		
`0x800fffffffffffff`	$-2^{e_{min}}(1-2^{1-p})$	Largest denormal < 0
`0x8010000000000000`	$-2^{e_{min}}$	Smallest normal < 0
...		
`0x801fffffffffffff`	$-2^{e_{min}}(2-2^{1-p})$	

Walking Through the Floating-Point Numbers

0x8000000000000000	-0.0	Negative zero
0x8000000000000001	$-2^{e_{min}-p+1}$	Smallest denormal < 0
...		
0x800fffffffffffff	$-2^{e_{min}}(1-2^{1-p})$	Largest denormal < 0
0x8010000000000000	$-2^{e_{min}}$	Smallest normal < 0
...		
0x801fffffffffffff	$-2^{e_{min}}(2-2^{1-p})$	
0x8020000000000000	$-2^{e_{min}+1}$	
...		
0xffefffffffffffff	$-2^{e_{max}}(2-2^{1-p})$	Largest normal < 0

Walking Through the Floating-Point Numbers

0x8000000000000000	-0.0	Negative zero
0x8000000000000001	$-2^{e_{min}-p+1}$	Smallest denormal < 0
...		
0x800fffffffffffff	$-2^{e_{min}}(1-2^{1-p})$	Largest denormal < 0
0x8010000000000000	$-2^{e_{min}}$	Smallest normal < 0
...		
0x801fffffffffffff	$-2^{e_{min}}(2-2^{1-p})$	
0x8020000000000000	$-2^{e_{min}+1}$	
...		
0xffefffffffffffff	$-2^{e_{max}}(2-2^{1-p})$	Largest normal < 0
0xfff0000000000000	$-\infty$	Negative infinity

Walking Through the Floating-Point Numbers

`0x8000000000000000`	-0.0	Negative zero
`0x8000000000000001`	$-2^{e_{min}-p+1}$	Smallest denormal < 0
...		
`0x800fffffffffffff`	$-2^{e_{min}}(1-2^{1-p})$	Largest denormal < 0
`0x8010000000000000`	$-2^{e_{min}}$	Smallest normal < 0
...		
`0x801fffffffffffff`	$-2^{e_{min}}(2-2^{1-p})$	
`0x8020000000000000`	$-2^{e_{min}+1}$	
...		
`0xffefffffffffffff`	$-2^{e_{max}}(2-2^{1-p})$	Largest normal < 0
`0xfff0000000000000`	$-\infty$	Negative infinity
`0xfff0000000000001`		NaN
...		
`0xffffffffffffffff`		NaN

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

Questions