# Demystifying Floating Point

John Farrier, Booz Allen Hamilton

https://github.com/DigitalInBlue/CPPCon2015

## Goals

- Understand the basics of floats
- Understand the language of floats
- Get a feel for when further investigation is required
- Have a few tools ready to help
- Be prepared to keep learning about IEEE floats

https://github.com/DigitalInBlue/CPPCon2015

## You are in good company…

“Nothing brings fear to my heart more than a floating point number.”

— Gerald Jay Sussman, Professor, MIT

https://github.com/DigitalInBlue/CPPCon2015

# You are in good company…

"Modeling error is usually several orders of magnitude greater than floating point error. People who nonchalantly model the real world and then sneer at floating point as just an approximation strain at gnats and swallow camels."

— John D. Cook, Singular Value Consulting

https://github.com/DigitalInBlue/CPPCon2015

## Commonly Committed Floating Point Fallacies

- “It’s floating point error”
    - “All floating point involves magical rounding errors”
- “Linux and Windows handle floats differently”
- “Floating point represents an interval value near the actual value”
- “A double holds 15 decimal places and I only need 3, so I have nothing to worry about”
- “My programming language does better math than your programming language”*

**Ada is a bit ambiguous when it comes to math, but most modern languages…*

## StackOverflow

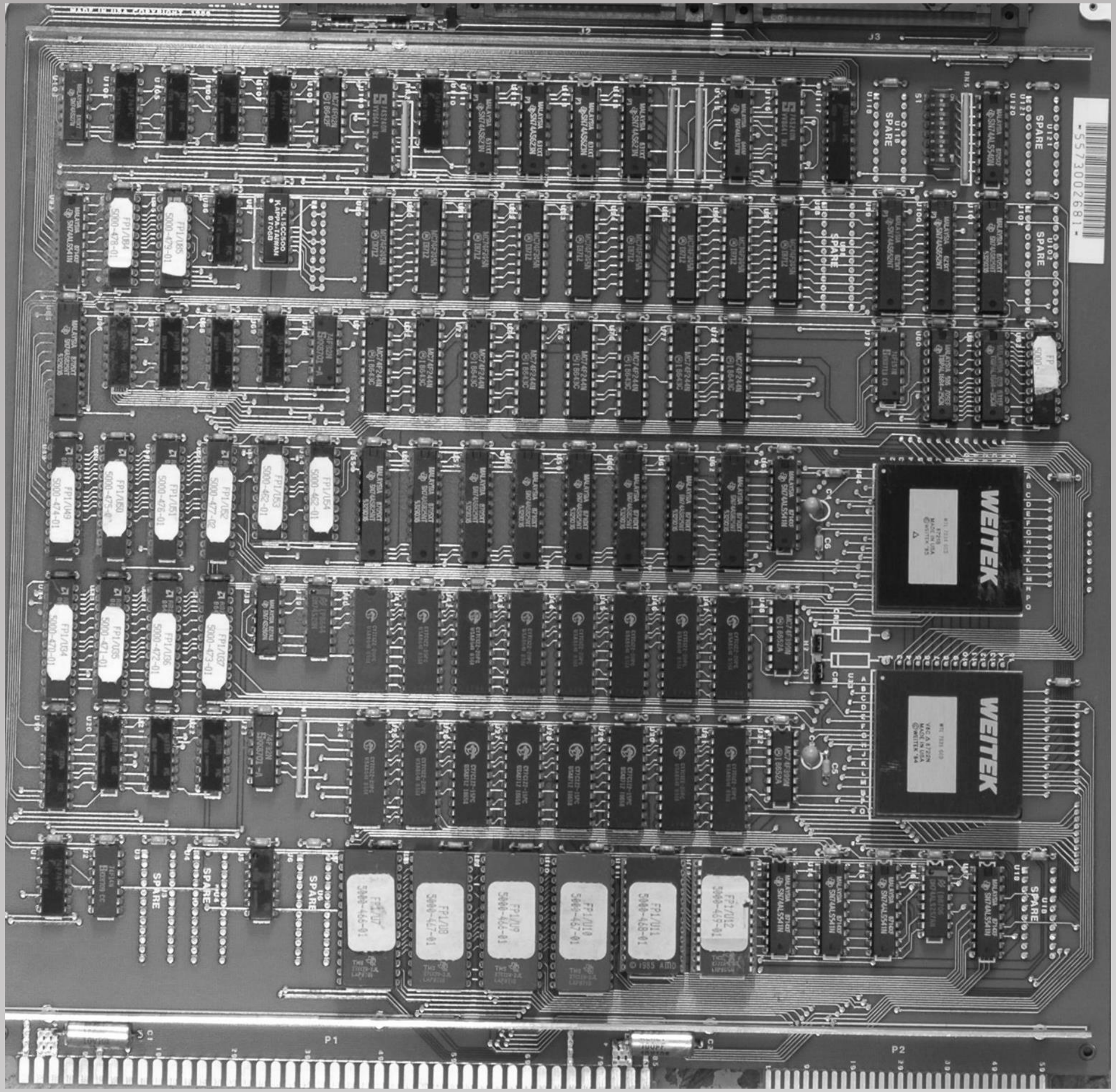
- All roads lead to:

  *What Every Computer Scientist Should Know About Floating-Point Arithmetic*

  David Goldberg, March, 1991 issue of Computing Surveys.

  *93 pages*

  “Floating-point arithmetic is considered an esoteric subject by many people.“



# Anatomy of IEEE Floats

https://github.com/DigitalInBlue/CPPCon2015

## IEEE Float Specification

- IEEE 754-1985, IEEE 854-1987, IEEE 754-2008
- Provide for portable, provably consistent math
- Ensure some significant mathematical identities hold true:
  - $x + y == y + x$
  - $x + 0 == x$
  - $if\,(x == y), then\ x\ - y == 0$
  - $\frac{x}{\sqrt{(x^2+y^2)}} \leq 1$

## IEEE Float Specification

- Ensure every floating point number is unique
  - *Zero* is a special case because of this
- Ensure every floating point number has an opposite
- Specifies algorithms for addition, subtraction, multiplication, division, and sqrt

# IEEE Float Specification - Layout

- An approximation using scientific notation
  - $x = -1^{s} \times 2^{e} \times 1.m$
  - $x = -1^{signBit} \times base2^{exponent} \times 1.mantissa$
  - $x = -1^{[0]} \times base2^{[10000000]} \times 1.[10010010000111111011011]$
  - $x = [0][10000000][10010010000111111011011]$
- 32-bits = 1 sign bit + 8 exponent bits + 23 mantissa bits
- 64-bits = 1 sign bit + 12 exponent bits + 52 mantissa bits

*Note: Finite real numbers may not have a perfect IEEE Float representation.*

# IEEE Float Specification - Special Floats

- Divide by Zero
  - $1/0$
- Not a Number (NaN)
  - $\frac{0}{0}, 0 \times \infty$
- Signed Infinity
  - Overflow protection
- Signed Zero
  - Underflow protection, preserves sign
  - $+0 = -0$

```
Software Failure.   Press left mouse button to continue.
Guru Meditation #00000004.0000AAC0
```

## Now that we are experts…

https://github.com/DigitalInBlue/CPPCon2015

# Simple Example

```
auto zeroPointOne = 0.1f;
auto zeroPointTwo = 0.2f;
auto zeroPointThree = 0.3f;
auto sum = zeroPointOne + zeroPointTwo;

cppCon() << "zeroPointOne == " << zeroPointOne << "\n";
cppCon() << "zeroPointTwo == " << zeroPointTwo << "\n";
cppCon() << "zeroPointThree == " << zeroPointThree << "\n";
cppCon() << "sum == " << sum << "\n";
```

## Simple Example

```
zeroPointOne   == 0.100000001490116119384765625000
zeroPointTwo   == 0.200000002980232238769531250000
zeroPointThree == 0.300000011920928955078125000000
sum            == 0.300000011920928955078125000000
```

## Simple Example

```
zeroPointOne   == 0.100000001490116119384765625000
zeroPointTwo   == 0.200000002980232238769531250000
zeroPointThree == 0.300000011920928955078125000000
sum            == 0.300000011920928955078125000000
```

```
zeroPointOne   == 0.100000000000000005551115123126
zeroPointTwo   == 0.200000000000000011102230246252
zeroPointThree == 0.299999999999999988897769753748
sum            == 0.300000000000000044408920985006
```

## Storage of 1.0

```
1.0000000000000000
```

$$-1^{signBit} \times base2^{exponent} \times 1.mantissa$$

$$-1^{0} \times 2^{0} \times 1.0$$

# Storage of 1.0

```
1.0000000000000000
```

$$-1^{signBit} \times base2^{exponent} \times 1.mantissa$$

$$-1^{0} \times 2^{0} \times 1.0$$

```
[0][00000000][00000000000000000000000]
```

## Storage of 1.0

```
1.0000000000000000
```

$$-1^{signBit} \times base2^{exponent} \times 1.mantissa$$

$$-1^{0} \times 2^{0} \times 1.0$$

```
[0][01111111][00000000000000000000000]
```

## Storage of 1.0

```
1.0000000000000000
```

$$-1^{signBit} \times base2^{exponent} \times 1.mantissa$$
$$-1^{0} \times 2^{0} \times 1.0$$

```
[0][01111111][00000000000000000000000]
```

- The exponent is *shift-127* encoded
- `0xeeeeeeee - 127`

## Storage of 1.0

```
1.0000000000000000
[0][01111111][00000000000000000000000]
```

## Storage of 1.0

```
      1.0000000000000000
[0][01111111][00000000000000000000000]
[0][01111111][00000000000000000000001]
      1.0000001192092896
```

## “Epsilon”

- The difference between 1.0 and the next available floating point number
- Useful for programmatic “almost equal” computations
- `std::numeric_limits<T>::epsilon`

# Significant Digits

- Significant digits measure precision
  - Remember the "exponent" field
  - They are not a magnitude
- How close is close enough?
  - Define as significant digits, not absolute error

| | Sign | Exponent | Mantissa | Total | Exponent bias | Bits precision | Significant Digits |
|---|---|---|---|---|---|---|---|
| **Half (IEEE 754-2008)** | 1 | 5 | 10 | 16 | 15 | 11 | 3-4 |
| **Single** | 1 | 8 | 23 | 32 | 127 | 24 | 6-9 |
| **Double** | 1 | 11 | 52 | 64 | 1023 | 53 | 15-17 |
| **x86 extended precision** | 1 | 15 | 64 | 80 | 16383 | 64 | 18-21 |
| **Quad** | 1 | 15 | 112 | 128 | 16383 | 113 | 33-36 |

http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF

## Storage of the Very Small

- For 32-bit floats, the minimum base 10 exponent is -36.
- How is $1.0e^{-37}$ represented?

```
1.0e-37
[0][00000000][11011001110001111101110]
0.09999999e-37
```

## “Denormalized Number”

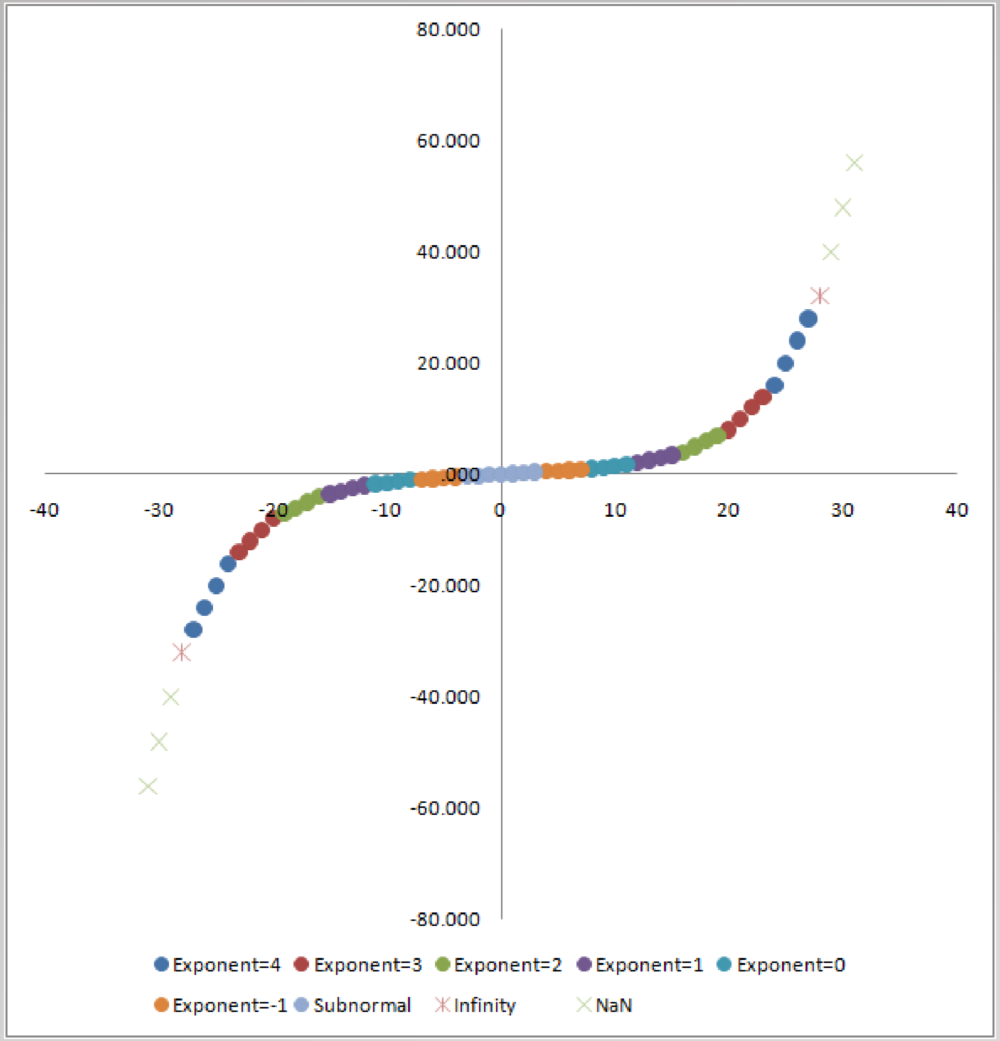
- Numbers that have a zero exponent
- Required when the exponent is below the minimum exponent
- Helps prevent underflow

```
1.0e-37
[0][00000000][11011001110001111101110]
0.09999999e-37
```

## Floating Point Precision

- Representation is not uniform between numbers
- Most precision lies between 0.0 and 0.1
- Precision falls away
- `std::nextafter`

## Floating Point Precision

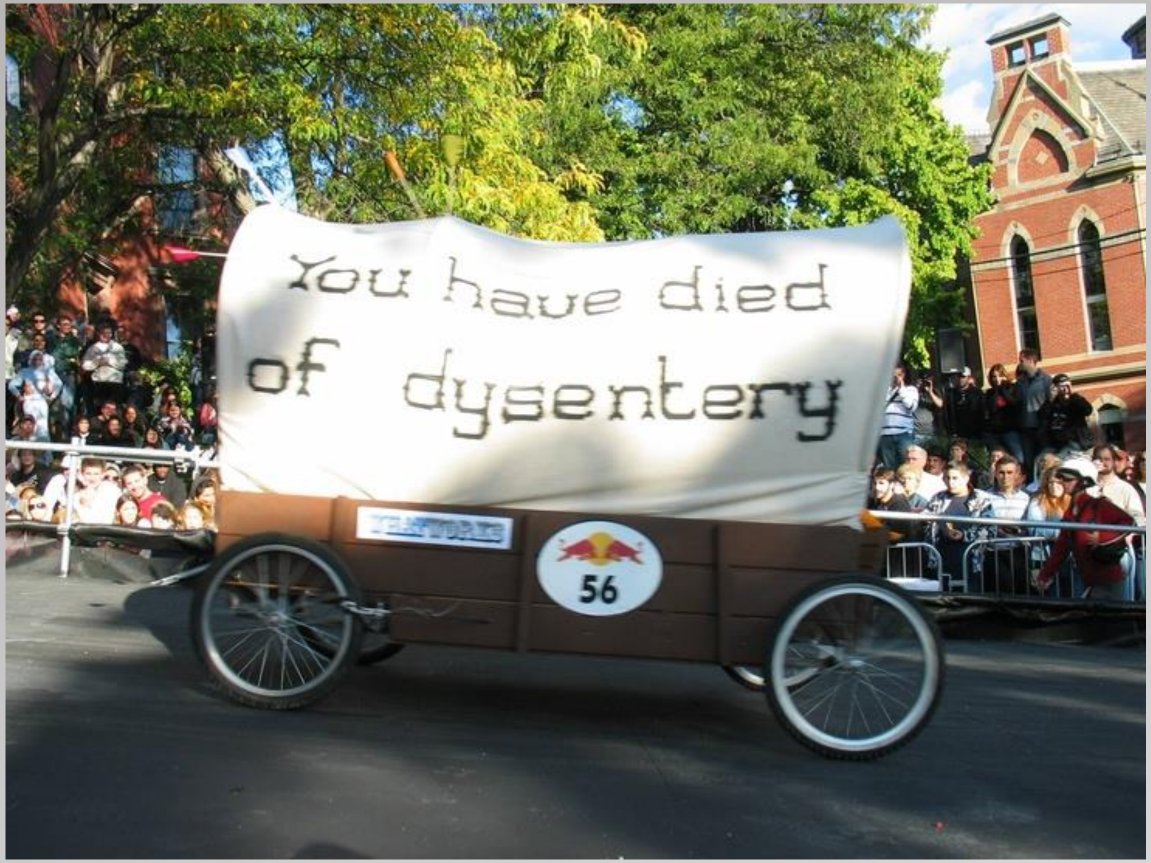


http://blogs.msdn.com/b/dwayneneed/archive/2010/05/07/fun-with-floating-point.aspx

# Floating Point Precision

The number of floats from 0.0

- …to 0.1 = `1,036,831,949`
- …to 0.2 = `8,388,608`
- … to 0.4 = `8,388,608`
- … to 0.8 = `8,388,608`
- … to 1.6 = `8,388,608`
- … to 3.2 = `8,388,608`



## Errors in Floating Point

https://github.com/DigitalInBlue/CPPCon2015

## Storage of π

```
π = 3.14159265
πf= 3.14159274
Δ = 0.00000009
```

# Measuring Error: “Ulps”

- Units in Last Place
  - “Harrison” and “Goldberg” definitions
  - (6-8 definitions floating around)
- “The gap between the two floating-point numbers nearest to x, even if x is one of them.” – W. Kahan
  - https://www.cs.berkeley.edu/~wkahan/LOG10HAF.TXT
- IEEE 754 requires that that elementary arithmetic operations are correctly rounded to within 0.5 ulps
- Transcendental functions are generally rounded to between 0.5 and 1.0 ulps

```
π = 3.14159265
πf= 3.14159274
Δ = 0.00000009
           9 ulps
```

# Measuring Error: “Relative Error”

- The difference between the “real” number and the approximated number, divided by the “real” number.

$$\frac{\pi\ - (\pi f)}{\pi} = 2.864789e^{-8}$$

```
π = 3.14159265
πf= 3.14159274
Δ = 0.00000009
           9 ulps
  2.864789e-8 β
```

## Rounding Error

- Induced by approximating an infinite range of numbers into a finite number of bits
- Math is done exactly, then rounded*
  - Towards the nearest
  - Towards Zero
  - Towards positive infinity (round up)
  - Towards negative infinity (round down)

*Look up “exactly rounded” as well

## Rounding Error

- What about rounding the half-way case? (i.e. 0.5)
  - Round Up vs. Round Even

- Correct Rounding:
  - Basic operations (add, subtract, multiply, divide, sqrt) should return the number nearest the mathematical result.
  - If there is a tie, round to the number with an even mantissa

## “Guard Bit”, “Round Bit”, “Sticky Bit”

- Only used while doing calculations
  - Not stored in the float itself
  - The mantissa is shifted in calculations to align radix
- The guard bits and round bits are extra precision
- The sticky bit is an OR of anything that shifts through it

```
[0][00000000][00000000000000000000000][G][R][S...]
```

# “Guard Bit”, “Round Bit”, “Sticky Bit”

`[G][R][S]`

`[0][-][-]` - Round Down (do nothing)

`[1][0][0]` - Round Up if the mantissa LSB is 1

`[1][0][1]` - Round Up

`[1][1][0]` - Round Up

`[1][1][1]` - Round Up

```
[0][00000000][00000000000000000000000][G][R][S...]
```

## Significance Error

- Compute the Area of a Triangle
  - Heron's Formula:
    - $A = \sqrt{\left(\frac{x+y+Z}{2}\right)\left(\left(\frac{x+y+Z}{2}\right) - x\right)\left(\left(\frac{x+y+Z}{2}\right) - y\right)\left(\left(\frac{x+y+Z}{2}\right) - z\right)}$
  - Kahan's Algorithm:
    - Sort x, y, z such that $x \geq y \geq z$
    - If $z < x - y$, then no such triangle exists.
    - Else $A = \sqrt{\frac{(x+(y+z))\times(z-(x-y))\times(z+(x-y))\times(x+(y-z))}{4}}$

## Significance Error

```
/// Area of a triangle
/// From http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html
/// From https://www.cs.berkeley.edu/~wkahan/Triangle.pdf
TEST(CPPCon2015, AreaOfATriangleFloat)
{
const auto a = 100000.0f;
const auto b = 99999.99979f;
const auto c = 0.00029f;

ASSERT_TRUE(a >= b);
ASSERT_TRUE(b >= c);

auto heronsFormula = HeronsFormula(a, b, c);
auto kahansFormula = KahansFormula(a, b, c);
EXPECT_NE(kahansFormula, heronsFormula);
cppCon() << “Kahan: " << kahansFormula;
cppCon() << “Heron: " << heronsFormula;
cppCon() << “delta: " << kahansFormula - heronsFormula << "\n";
}
```

## Significance Error

```
Heron:  0.000000000000000000000000000000
Kahan: 14.500000000000000000000000000000
delta: 14.500000000000000000000000000000
```

## Significance Error

```
Heron:  0.000000000000000000000000000000
Kahan: 14.500000000000000000000000000000
delta: 14.500000000000000000000000000000
```

Hint: The Answer is 10.0

## Significance Error

```
Heron:  0.000000000000000000000000000000
Kahan: 14.500000000000000000000000000000
delta: 14.500000000000000000000000000000
```

```
Heron:  9.999999809638328700000000000000
Kahan: 10.000000077021038000000000000000
delta:  0.000000267382709751018410000000
```

# Significance Error - Use Stable Algorithms

- Loss of Significance
  - Keep big numbers with big numbers, little numbers with little numbers.
- Parentheses can help
- Analysis of Algorithms is Critical
- The compiler won't re-arrange your math if it cannot prove it would yield the same result, even if the computation would be faster
  - $x = b - \left((a + b) - a\right)$ should not be replaced with $x = 0$
  - See the "Kahan's Algorithm" example

## Significance Error – Simulation Time

- Time is used to compute distance, velocity, acceleration
  - High frequency phenomena
  - Sensors: Doppler shift, pulse compression, PRF
- These computed values feed into other computed values which may or may not require a time component
- Thousands of computations per frame of simulation
  - Thousands of little compounding errors per frame
- Combine with poor or nonexistent testing

# Significance Error – Simulation Time

- Accumulate Time (works for arbitrary time steps)

```
auto totalFrames = size_t(0);
auto frameLength = 0.01f;
auto simTime = 0.0f;

// Run 120 Frames
auto simTime120Frames = 1.20f;
for(; totalFrames < 120; totalFrames++)
{
  simTime += frameLength;
}
```

- Delta Time (works for fixed time steps)

```
auto totalFrames = size_t(0);
auto frameLength = 0.01f;
auto simTime = 0.0f;

// Run 120 Frames
auto simTime120Frames = 1.20f;
totalFrames = 120;
simTime = totalFrames * frameLength;
```

# Significance Error – Simulation Time

Accumulate Time (works for arbitrary time steps)

- 1.199999999999999955591079014994 != **1.199999213218688964843750000000** (0.000000786781310990747329014994)
- 12.000000000000000000000000000000 != **12.000179290771484375000000000000** (-0.000179290771484375000000000000)
- 120.000000000000000000000000000000 != **120.007225036621093750000000000000** (-0.007225036621093750000000000000)
- 600.000000000000000000000000000000 != **600.274414062500000000000000000000** (-0.274414062500000000000000000000)
- 3600.000000000000000000000000000000 != **3603.204101562500000000000000000000** (-3.204101562500000000000000000000)

Delta Time (works for fixed time steps)

- 1.200000047683715820312500000000 != **1.199999928474426269531250000000** (0.000000119209289550781250000000)
- 12.000000000000000000000000000000 == **12.000000000000000000000000000000** (0.000000000000000000000000000000)
- 120.000000000000000000000000000000 == **120.000000000000000000000000000000** (0.000000000000000000000000000000)
- 600.000000000000000000000000000000 == **600.000000000000000000000000000000** (0.000000000000000000000000000000)
- - 3600.000000000000000000000000000000 == **3600.000000000000000000000000000000** (0.000000000000000000000000000000)

## Significance Error – Simulation Time

- "Sub-Microsecond Precision"
  - 1.2345e-6 seconds per frame (810044.5 Hz)

# Significance Error – Simulation Time

- “Sub-Microsecond Precision”
  - 1.2345e-6 seconds per frame (810044.5 Hz)

```
0.012345000170171261000000000000 !=
0.014814000576734543 0
(-0.002469000406563282000000000000)

12.3450002670288090000000000000000 !=
11.701118469238281000000000000000
(0.6438817977905273400000000000000)
```

# Significance Error – Don’t use IEEE Floats?

- Use integers
  - Very fast
  - Trade more precision for less range
  - Only input/output may be impacted by floating point conversions
  - Financial applications represent dollars as only cents or tenth’s of cents
- Use a math library
  - Slower
  - Define your own level of accuracy
  - MPFR (w/C++ Wrapper), TTMath, Boost, GMP C++
  - CRlibm (Correctly Rounded Mathematical Library)

(Store `simTime` as `uint64_t` and get microsecond precision for 584555 years.)

# Algebraic Assumption Error

- Mathematical Identities
  - Traditional identities (associative, commutative, distributive) do not hold
- Distributive Rule does not apply: $x \times y - x \times z \neq x\ (y - z)$
- Associative Rule does not apply: $x + (y + z) \neq (x + y) + z$
- Cannot interchange division and multiplication: $\frac{x}{10.0} \neq x\ \times 0.1$

Does a naïve compiler make these assumptions too?

https://msdn.microsoft.com/library/aa289157.aspx

## Algebraic Assumption Error

```
const auto oneRadian = 0.15915494309f;
const auto control = 0.0000000000015915494309f;

const auto oneRadianMultiplied = oneRadian * 1.0e-10f;
const auto oneRadianDivided = oneRadian / 1.0e10f;
```

# Algebraic Assumption Error

```
const auto oneRadian = 0.15915494309f;
const auto control = 0.000000000015915494309f;

const auto oneRadianMultiplied = oneRadian * 1.0e-10f;
const auto oneRadianDivided = oneRadian / 1.0e10f;
```

```
       Control: 0.000000000015915494616658421000
   x*1.0e-10f: 0.000000000015915494616658421000(0.000000000000000000000000000000)
    x/1.0e10f: 0.000000000015915492881934945000(0.000000000000000000173472347597 7)
Relative Error: 0.000000108995891423546710000000
```

## Floating Point Exceptions

- Enable floating point exceptions to be alerted when things go awry.

| IEEE 754 Exception | Result when traps disabled | Argument to trap handler |
|---|---|---|
| overflow | $\pm\infty$ or $\pm\, x_{max}$ | round($x2^{-\alpha}$) |
| underflow | 0, $2^{e_{min}}$ or denormalized | round($x2^{\alpha}$) |
| divide by zero | $\pm\ \infty$ | invalid operation |
| invalid | NaN | invalid operation |
| inexact | round($x$) | round($x$) |

$x$ is the exact result of the operation

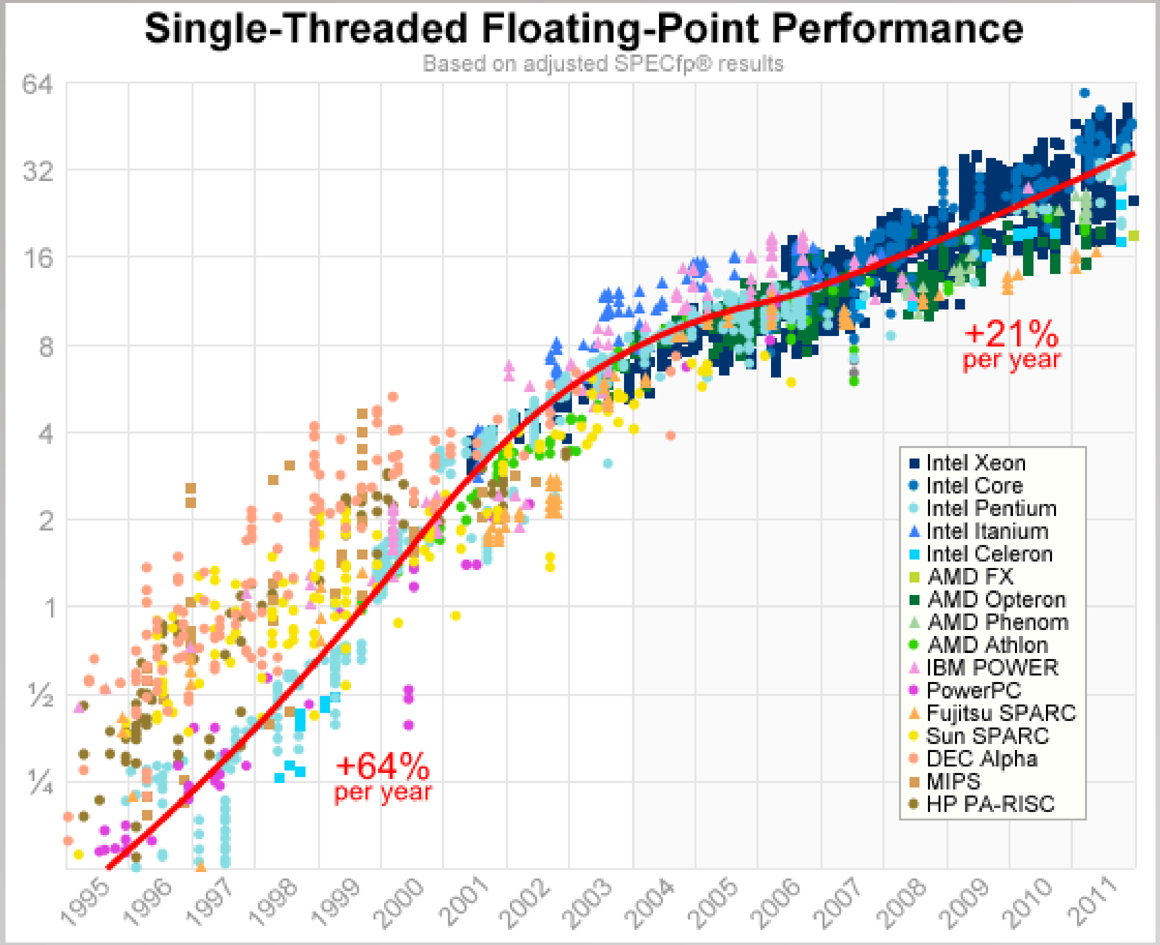
$\alpha = 192$ for single precision, 1536 for double

$x_{max} = 1.111 \ldots 111 \times 2^{e_{min}}$.

See `<cfenv>` - Floating Point Environment

## But Wait!  There's More!

- Binary to Decimal Conversion Error
- Summation Error
- Propagation Error
- Underflow, Overflow
- Type Narrowing/Widening Rules



http://preshing.com/images/float-point-perf.png

# Miscellaneous Notes

https://github.com/DigitalInBlue/CPPCon2015

# Use Your Compiler’s Output

- `warning C4244: ‘initializing’ : conversion from ‘double’ to ‘float’, possible loss of data`
- `warning C4056: overflow in floating point constant arithmetic`
- `warning C4305: 'identifier' : truncation from 'type1' to 'type2'`
- `warning: conversion to 'float' from 'int' may alter its value`
- `warning: floating constant exceeds range of ‘double’`

## Fused Multiply-Add (FMA)

- `a = a + (b * c); a += b * c;`
- Multiplier-accumulator (MAC Unit)
- One rounding
- Compiler Options
  - GCC = `-mfma`
  - VC++ = `#pragma fp_contract (off)`
- Reference: FMA3, FMA4
  - http://en.wikipedia.org/wiki/FMA_instruction_set

# Streaming SIMD Extensions (SSE)

- SSE can provide significant performance gains
    - Supports integer, floating point, logical, conversion, shift, and shuffle operations
- C, C++, Fortran do not natively support SSE, but compiler-specific support exists

## Float Tricks

```
/// Fast reciprocal square root approximation for x > 0.25
/// Quake’s float Q_rsqrt(float number) is much more entertaining.
inline float FastInvSqrt(float x)
{
	int tmp = ((0x3f800000 << 1) + 0x3f800000 - *(long*)&x) >> 1;
	auto y = *(float*)&tmp;
	return y * (1.47f – 0.47f * x * y * y);
}
```

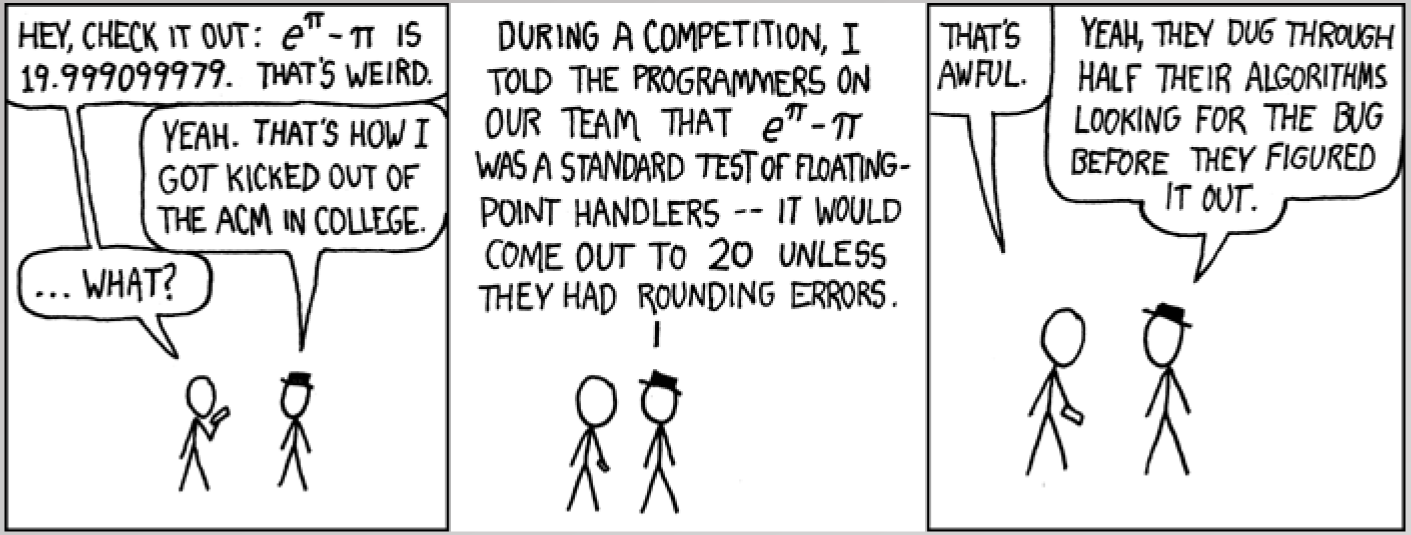
## Testing

- Design for Numerical Stability
- Perform Meaningful Testing
- Document assumptions
- Track sources of approximation
- Quantify goodness
  - Well conditioned algorithms
- Backward error analysis
  - Are the outputs identical for slightly modified inputs?

```cpp
TEST(CPPCon2015, pointOnePlusPointTwo)
{
auto zeroPointOne = 0.1f;
auto zeroPointTwo = 0.2f;
auto zeroPointThree = 0.3f;
auto sum = zeroPointOne + zeroPointTwo;

EXPECT_DOUBLE_EQ(0.3f, zeroPointThree);
EXPECT_EQ(0.3f, zeroPointThree);
EXPECT_EQ(0.3f, sum);

EXPECT_EQ(zeroPointThree, sum);
EXPECT_DOUBLE_EQ(zeroPointThree, sum);
}
```


HEY, CHECK IT OUT: $e^{\pi} - \pi$ IS 19.999099979. THAT'S WEIRD.
YEAH. THAT'S HOW I GOT KICKED OUT OF THE ACM IN COLLEGE.
... WHAT?
DURING A COMPETITION, I TOLD THE PROGRAMMERS ON OUR TEAM THAT $e^{\pi} - \pi$ WAS A STANDARD TEST OF FLOATING-POINT HANDLERS -- IT WOULD COME OUT TO 20 UNLESS THEY HAD ROUNDING ERRORS.
THAT'S AWFUL.
YEAH, THEY DUG THROUGH HALF THEIR ALGORITHMS LOOKING FOR THE BUG BEFORE THEY FIGURED IT OUT.

http://xkcd.com/217/
http://www.explainxkcd.com/wiki/index.php/217:_e_to_the_pi_Minus_pi

https://github.com/DigitalInBlue/CPPCon2015

# Demystifying Floating Point

John Farrier, Booz Allen Hamilton

https://github.com/DigitalInBlue/CPPCon2015