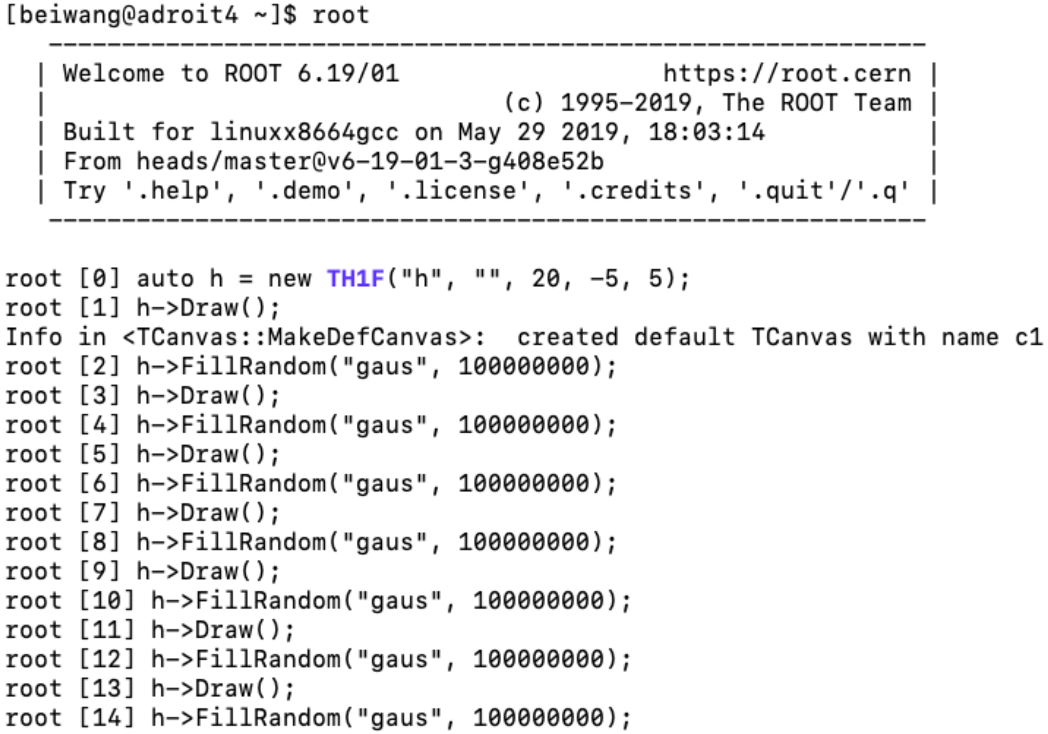
FLOATING POINT ARITHMETIC IS NOT REAL

Bei Wang
Princeton University

Fourth Computational and Data Science school for HEP (CoDaS-HEP)

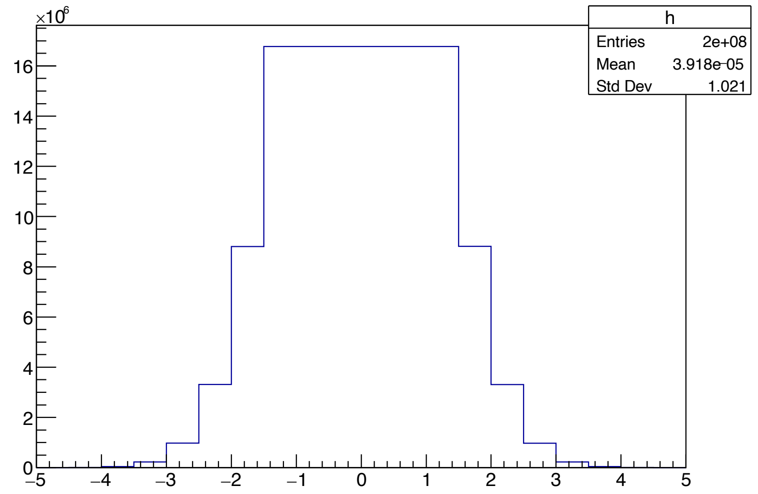
Aug 2, 2022

1D Histogram in ROOT



```
[beiwang@adroit4 ~]$ root
   ------------------------------------------------------------
  | Welcome to ROOT 6.19/01                  https://root.cern |
  |                               (c) 1995-2019, The ROOT Team |
  | Built for linuxx8664gcc on May 29 2019, 18:03:14           |
  | From heads/master@v6-19-01-3-g408e52b                      |
  | Try '.help', '.demo', '.license', '.credits', '.quit'/'.q' |
   ------------------------------------------------------------

root [0] auto h = new TH1F("h", "", 20, -5, 5);
root [1] h->Draw();
Info in <TCanvas::MakeDefCanvas>:  created default TCanvas with name c1
root [2] h->FillRandom("gaus", 100000000);
root [3] h->Draw();
root [4] h->FillRandom("gaus", 100000000);
root [5] h->Draw();
root [6] h->FillRandom("gaus", 100000000);
root [7] h->Draw();
root [8] h->FillRandom("gaus", 100000000);
root [9] h->Draw();
root [10] h->FillRandom("gaus", 100000000);
root [11] h->Draw();
root [12] h->FillRandom("gaus", 100000000);
root [13] h->Draw();
root [14] h->FillRandom("gaus", 100000000);
```

- The 1D histogram of a Gaussian distribution with 100M samples



- What happens after the second or the third fill? The center part of the Gaussian starts to fatten out

Thanks Jim Pivarski to provide this great exercise!

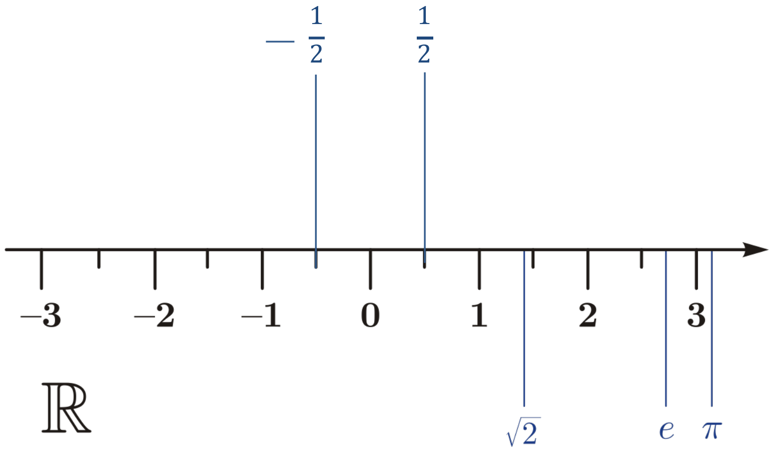
Outlines

- Basics
  - Real numbers
  - Representation of real numbers
  - Computer representation of real numbers
- IEEE Floating Point Representation
  - Formats
  - Roundings
  - Exceptions
- Approximate Math
- Goals
  - Basic understanding of computer representation of numbers
  - Basic understanding of floating point arithmetic
  - Consequences of floating point arithmetic for scientific computing
  - Basic understanding about fast math

BASICS

The Real Numbers

- The real numbers can be represented by a line



- ***Integers*** are the numbers, e.g., 0, 1, -1, 2, -2, …
- ***Rational*** numbers are those that consist of a ratio of two integers, e.g., 1/2, 2/3, 6/3; some of these are integers
- ***Irrational*** numbers are the real numbers that are not rational, e.g., $\sqrt{2}$, π, e.

Decimal Numbers

- Nature notation:

3902.7349

- Disadvantages:
  - Small number like 0.000000000082 has lot of zeros before anything interesting shows up. Similarly for large numbers
  - It's hard to estimate the magnitude of a large number, e.g., 4221302112

Representation a Real Number in Scientific Notation

- In scientific notation, every *real number* can be represented by

$$x = (-1)^{s}\left(\sum_{i=0}^{\infty} d_i\, B^{-i}\right)B^{e}$$

where $s \in \{0, 1\}, B \geq 2, i \in \{0, 1, 2, \ldots\}, d_i \in \{0, \ldots, B-1\}$ and $d_0 > 0$ when $x \neq 0$. B and e are integers.

Example

$$(71)_{10} = (7x10^{0} + 1x10^{-1})x10^{1}$$

$$(71)_{2} = (1x2^{0} + 0x2^{-1} + 0x2^{-2} + 0x2^{-3} + 1x2^{-4} + 1x2^{-5} + 1x2^{-6})x2^{6}$$

$$-\left(\frac{1}{10}\right)_{10} = (-1)^{1}(1x10^{0})x10^{-1}$$

$$-\left(\frac{1}{10}\right)_{2} = (-1)^{1}(0.0001100110011\ldots)_{2} = (-1)^{1}\,(1x2^{0} + 1x2^{-1} + 0x2^{-2} + 0x2^{-3} + 1x2^{-4} + \ldots)x2^{-4}$$

$$(\sqrt{2})_{10} = (1.414213\ldots)_{10}$$

$$(\pi)_{10} = (3.141592\ldots)_{10}$$

Computer Representation of Numbers

- A computer has finite number of bits, thus can only represent a finite subset of the real numbers
- They are called *floating point numbers* and can be represented as

$$x = (-1)^s \left(\sum_{i=0}^{p-1} d_i \, B^{-i} \right) B^e$$

finite

where s $\in \{0, 1\}, B \geq 2, d_i \in \{0, \dots, B-1\}$ with $d_0 > 0$ when $x \neq 0. i \in \{0, \dots, p-1\}, e \in \{e_{min}, \dots, e_{max}\}$

- B is called the *base*
- $\sum_{i=0}^{p-1} d_i \, B^{-i}$ is called the *significand* (or *mantissa*)
- p is called the *precision*
- e is the *exponent*
- The representation is *normalized* when $d_0 > 0$ (scientific notation)

Quiz: Binary Representation

- When B=2

$$(x)_2 = (-1)^s \left(1.b_1 b_2 \ldots b_{p-1}\right) 2^e$$

  - $b_0 = 1$ is a *hidden bit* (in a normalized binary system)
  - $b_1 b_2 \ldots b_{p-1}$ is called the *fractional* part of the significand
  - $e \in \{e_{min}, \ldots, e_{max}\}$
  - The gap between 1 and the next larger floating point number is called *machine epsilon, ε*

- **Questions:** In this binary system
  - What is the largest number?

$$(x_{max})_2 = (1 - 2^{-p}) 2^{e_{max}+1}$$

  - What is the smallest positive normalized number?

$$(x_{min})_2 = 2^{e_{min}}$$

  - What is the ε ?

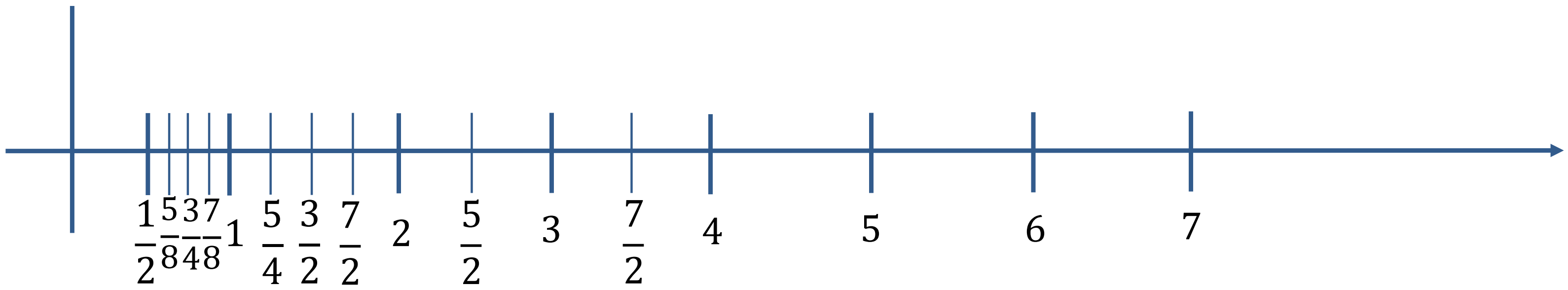
$$\varepsilon_2 = (1.00..1)\times 2^0 - (1.00..0)\times 2^0 = (0.00 \ldots 1)_2 \times 2^0 = 2^{-(p-1)}$$

A Toy Floating Point Number System

- For p=3, $e_{min} = -1, e_{max}$=2, the binary representation is

$$(x)_2 = (-1)^s(1.b_1b_2)2^e$$

- Look at the positive numbers with e=0, 1, 2, -1



- The largest number is 7 and the smallest positive normalized number is $^1/_2$
- The spacing between 1 and $^5/_4$ is $^1/_4$ (epsilon = $^1/_4$)

IEEE FLOATING POINT REPRESENTATION (IEEE 754 Standard)

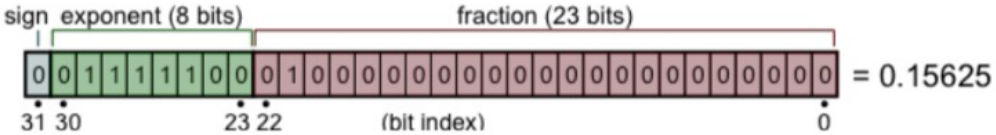
IEEE Floating Point Representation

- IEEE floating point numbers (in binary) can all be expressed in the form

$$(x)_2 = (-1)^s(b_0.b_1b_2...b_{p-1})2^{e-e_{bias}}$$

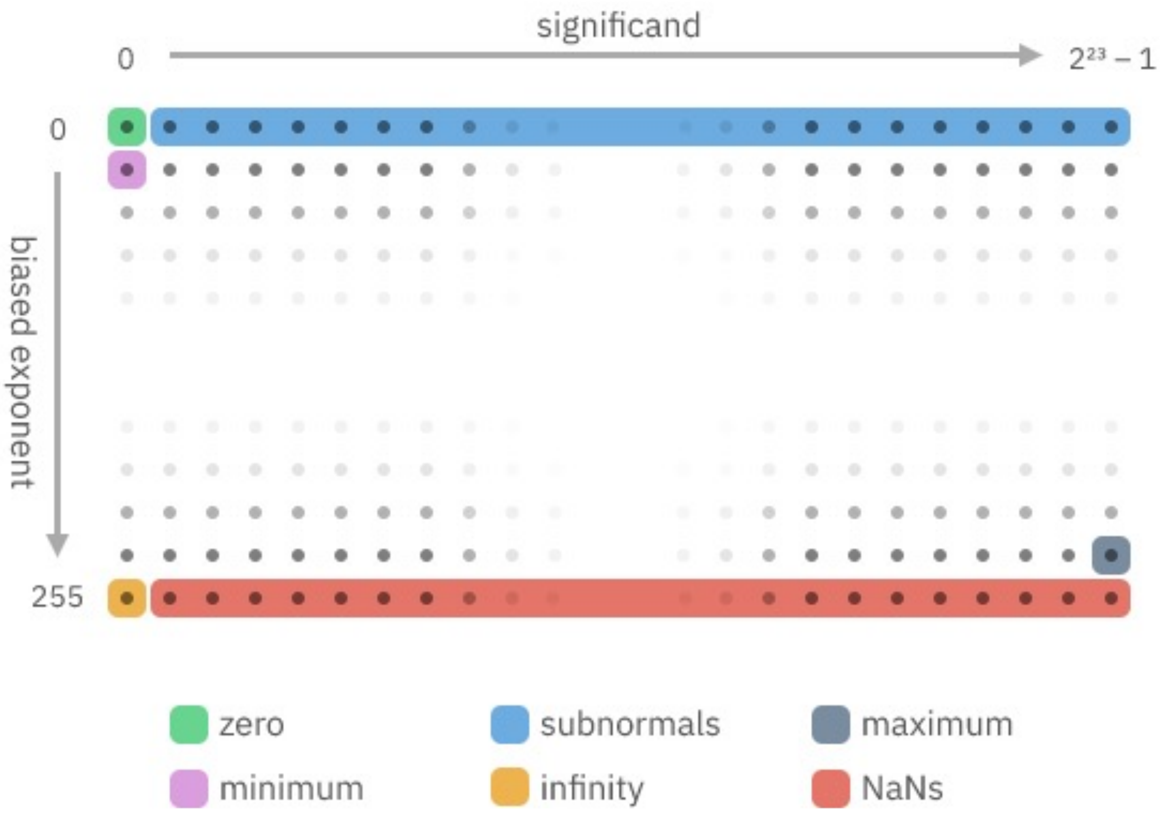
where p is the precision. The exponent is stored biased as an unsigned integer.

- For example: IEEE single precision format (32 bits): $(x)_2 = -1^s(b_0.b_1b_2...b_{23})2^{e-127}$, $e_{bias}=2^{8-1}-1$



Exponent	**Fraction Zero**	**Fraction Non-zero**	Numerical value represented
00000000	±0	Subnormal numbers	$(-1)^{sign}$x2^{-126}x0.fraction
00000001,…,11111110	Normalized numbers	Normalized numbers	$(-1)^{sign}$x$2^{exponent-127}$x1.fraction
11111111	±∞	NaN	error pattern

Map of Float



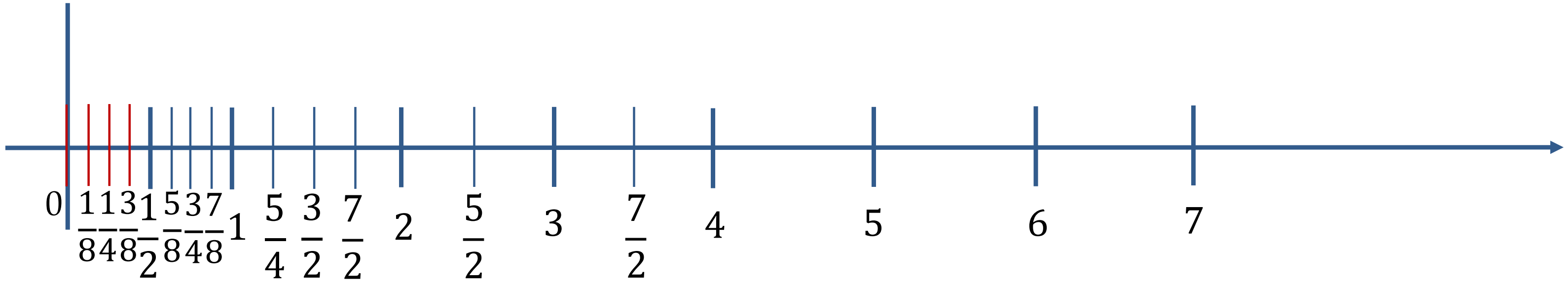
https://ciechanow.ski/exposing-floating-point

Subnormal Numbers

- Subnormal numbers serve two purposes
  - Provide a way to represent numeric value 0
  - Provide gradual underflow where possible floating point numbers are spaced evenly between 0 and x_{min} (the smallest normalized floating point numbers)
- They represent numerical values

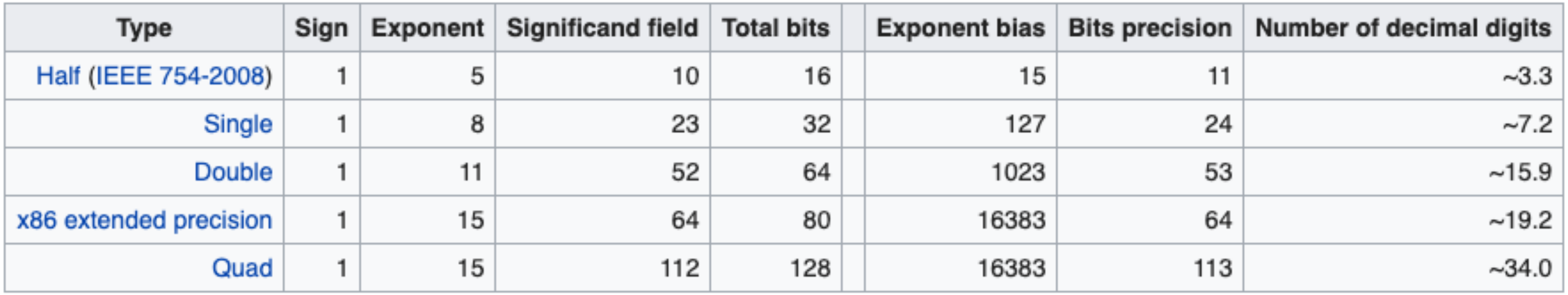
$$-1^{s}(0.\text{fraction})2^{e_{min}}$$

- Example, in the toy floating point system: p=3, e_{min}=-1, the non-negative subnormal numbers are:



IEEE 754 Binary Formats

- IEEE provides five basic binary formats



Type	Sign	Exponent	Significand field	Total bits		Exponent bias	Bits precision	Number of decimal digits
Half (IEEE 754-2008)	1	5	10	16		15	11	~3.3
Single	1	8	23	32		127	24	~7.2
Double	1	11	52	64		1023	53	~15.9
x86 extended precision	1	15	64	80		16383	64	~19.2
Quad	1	15	112	128		16383	113	~34.0

https://en.wikipedia.org/wiki/Floating-point_arithmetic

- C++ *numerical_limits* class template provides a standardized way to query various properties of floating point types

Intrinsic Errors in Floating Point Arithmetic

Rounding, differences in addend exponents, cancellation, near overflow/underflow errors

Rounding

- A a positive REAL number in the *normalized range* ($x_{min} \leq x \leq x_{max}$) can be represented as

$$(x)_2 = (1.b_1b_2\ldots b_{p-1}\ldots)\text{x}2^e,$$

where x_{min}(=$2^{e_{min}}$) and x_{max}(= $(1-2^{-p})2^{e_{max}+1}$) are the smallest and largest normalized floating point numbers. (Subscript 2 for binary representation is omitted since now.)

- The nearest floating point number less than or equal to x is

$$x_{-} = (1.b_1b_2\ldots b_{p-1})\text{x}2^e$$

- The nearest floating point number larger than x is

$$x_{+} = (1.b_1b_2\ldots b_{p-1} + 0.00\ldots 1)\text{x}2^e$$

- The gap between x_{+} and x_{-}, called ***unit in the last place*** (ulp) is

$$2^{-(p-1)}2^e$$

- The **absolute rounding error** is

$$\boxed{abserr(x) = |round(x) - x| < 2^{-(p-1)}2^e = \text{ulp}}$$

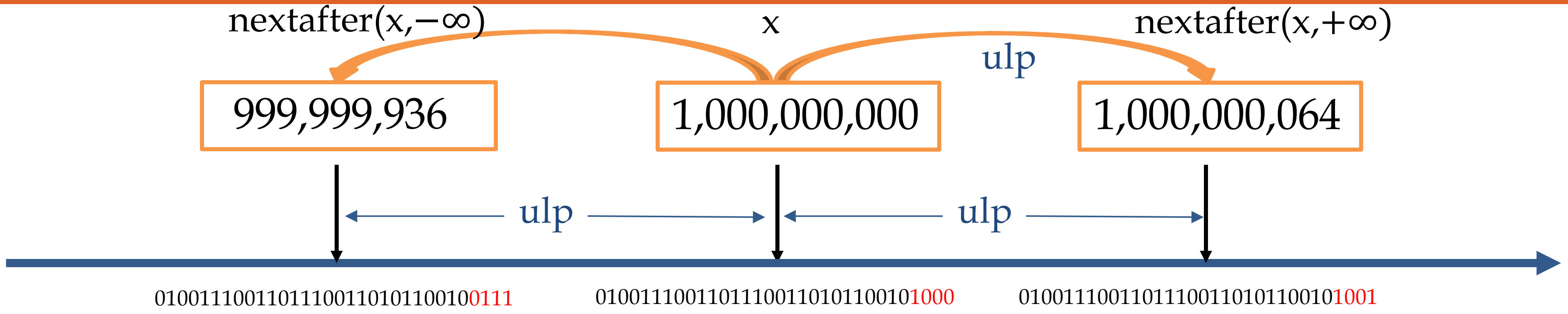
- The **relative rounding error** is

$$\boxed{relerr(x) = \frac{|round(x)-x|}{|x|} < \frac{2^{-(p-1)}2^e}{2^e} = 2^{-(p-1)} = \varepsilon}$$

Rounding Modes

- The IEEE standard defines five rounding modes
  - The first two round to a nearest value; the others are called directed roundings
- Roundings to nearest
  - **Round to nearest, ties to even – rounds to the nearest value;** if the number falls midway it is rounded to the nearest value with an even (zero) least significant bit, which occurs 50% of the time; this is the default algorithm for binary floating-point and the recommended default for decimal
  - Round to nearest, ties away from zero – rounds to the nearest value; if the number falls midway it is rounded to the nearest value above (for positive numbers) or below (for negative numbers)
- Directed roundings
  - Round toward 0 – directed rounding towards zero (also known as truncation).
  - Round toward +∞ – directed rounding towards positive infinity (also known as rounding up or ceiling).
  - Round toward −∞ – directed rounding towards negative infinity (also known as rounding down or floor).

Quiz: A Toy Rounding Example



- **Example:**

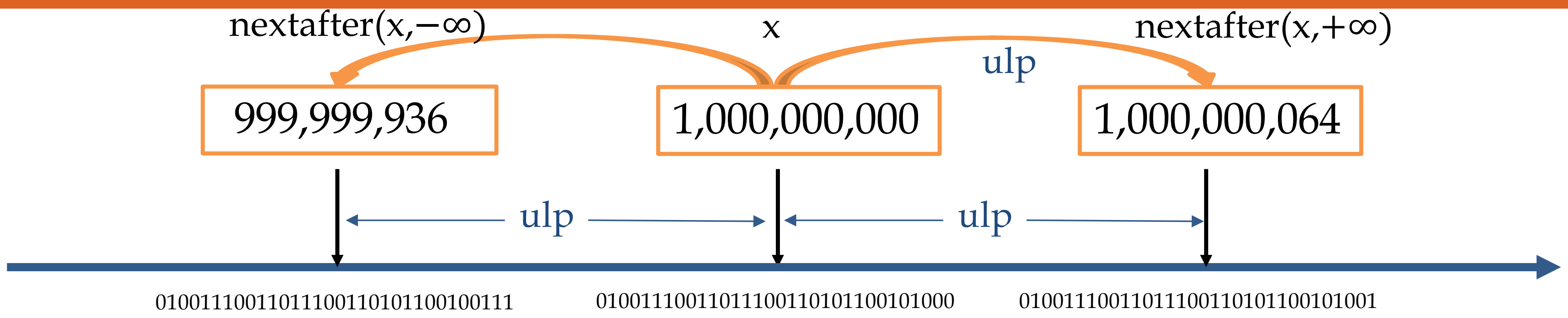
```
float x=1000000000;
float y=1000000032;
float z=1000000033;

std::cout<<std::scientific<<std::setprecision(8)<< x <<' '<< y <<' '<<z<<std::endl;
```

- **Question:** What's the output?
- **Answers:** 1.00000000e+09 1.00000000e+09 1.00000006e+09
- When rounding to the nearest, abserr(x) $\leq \frac{1}{2}$ulp and relerr(x) $\leq \frac{1}{2}\varepsilon$

Precision

- Floating point arithmetic CANNOT precisely represent true arithmetic operations
  - The operands are rounded
    - They exist in a finite number (~2^{32} for single precision)
    - The space between two floating point numbers differs by one ulp
  - Results of operations are rounded
    - $x + \varepsilon - x \neq \varepsilon$
  - Algebra is NOT necessarily associative and distributive
    - $(a + b) + c \neq a + (b + c)$
    - ${}^{a}/_{b} \neq a \, * \, {}^{1}/_{b}$
    - (a + b) * (a – b) ≠ a^2 - b^2
  - Example: what will be the result of 0.1^2?
    - In single precision, 0.1 is rounded and represented as 0.100000001490116119384765625 exactly
    - Squaring it with single-precision floating point hardware (with rounding) gives 0.010000000707805156707763671875
    - It is neither 0.01 nor the representable number closest to 0.01 (the representable number closest to 0.01 is 0.009999999776482582092285156250)

Quiz: Adding a Small and a Large Number



nextafter(x,−∞) x nextafter(x,+∞)

ulp

999,999,936 1,000,000,000 1,000,000,064

ulp ulp

01001110011011100110101100100111 01001110011011100110101100101000 01001110011011100110101100101001

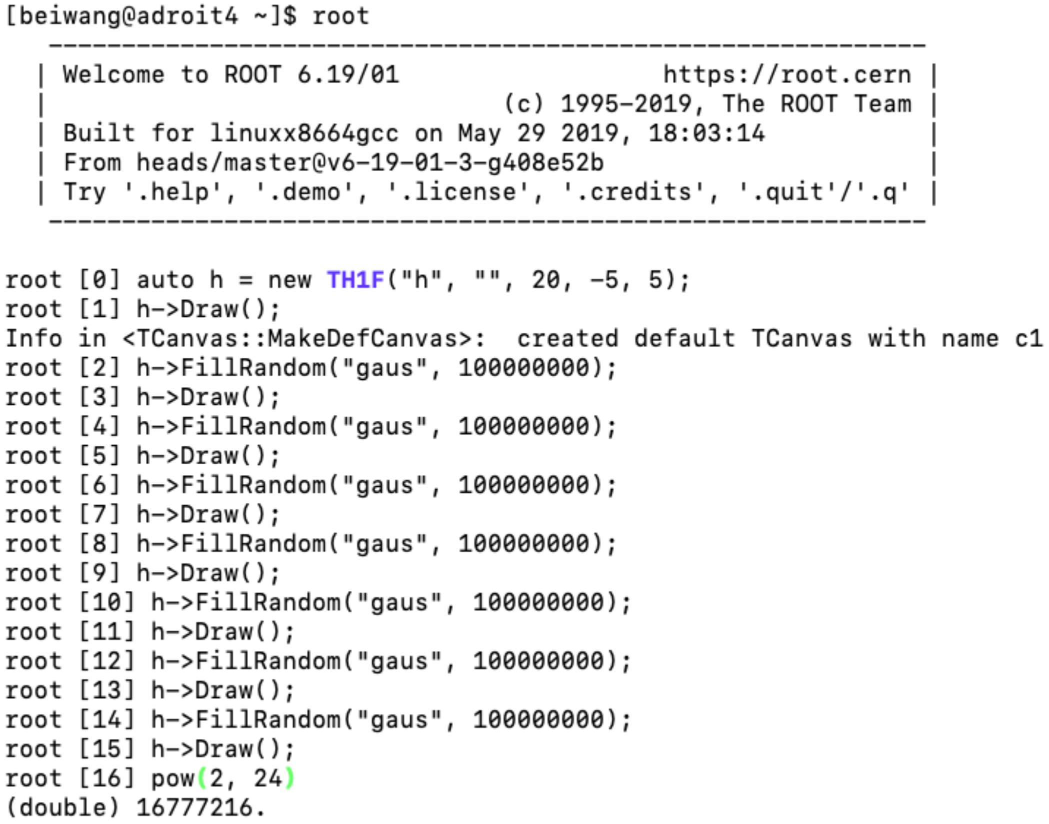
- Example:

```
float x=1000000000;
std::cout<<std::scientific<<std::setprecision(8)
          << x << ‘ ‘<< x+32.f << ‘ ‘<<x+33.f<<std::endl
          << x+32.f-x << ‘ ‘ << x+33.f-x <<std::endl;
```

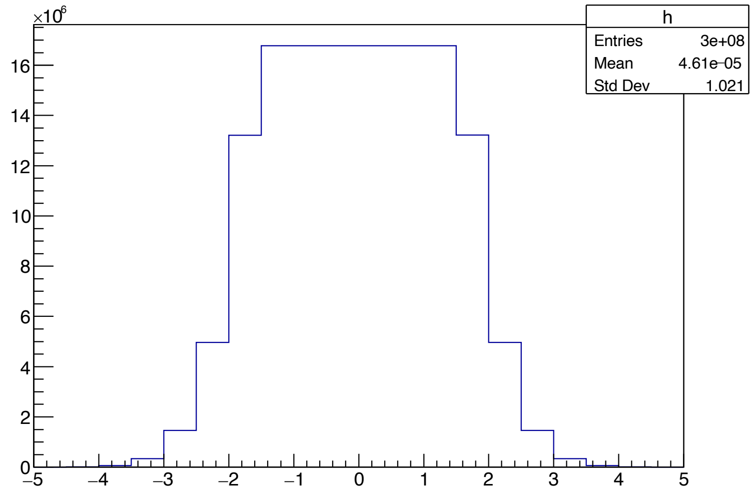
- **Question:** What is the output?
- **Answers:**

```
1.00000000e+09 1.00000000e+09 1.00000006e+09
0.00000000e+00 6.40000000e+01
```

Histogram Problem in Root



```
[beiwang@adroit4 ~]$ root
   ------------------------------------------------------------
  | Welcome to ROOT 6.19/01                  https://root.cern |
  |                               (c) 1995-2019, The ROOT Team |
  | Built for linuxx8664gcc on May 29 2019, 18:03:14           |
  | From heads/master@v6-19-01-3-g408e52b                      |
  | Try '.help', '.demo', '.license', '.credits', '.quit'/'.q' |
   ------------------------------------------------------------

root [0] auto h = new TH1F("h", "", 20, -5, 5);
root [1] h->Draw();
Info in <TCanvas::MakeDefCanvas>:  created default TCanvas with name c1
root [2] h->FillRandom("gaus", 100000000);
root [3] h->Draw();
root [4] h->FillRandom("gaus", 100000000);
root [5] h->Draw();
root [6] h->FillRandom("gaus", 100000000);
root [7] h->Draw();
root [8] h->FillRandom("gaus", 100000000);
root [9] h->Draw();
root [10] h->FillRandom("gaus", 100000000);
root [11] h->Draw();
root [12] h->FillRandom("gaus", 100000000);
root [13] h->Draw();
root [14] h->FillRandom("gaus", 100000000);
root [15] h->Draw();
root [16] pow(2, 24)
(double) 16777216.
```



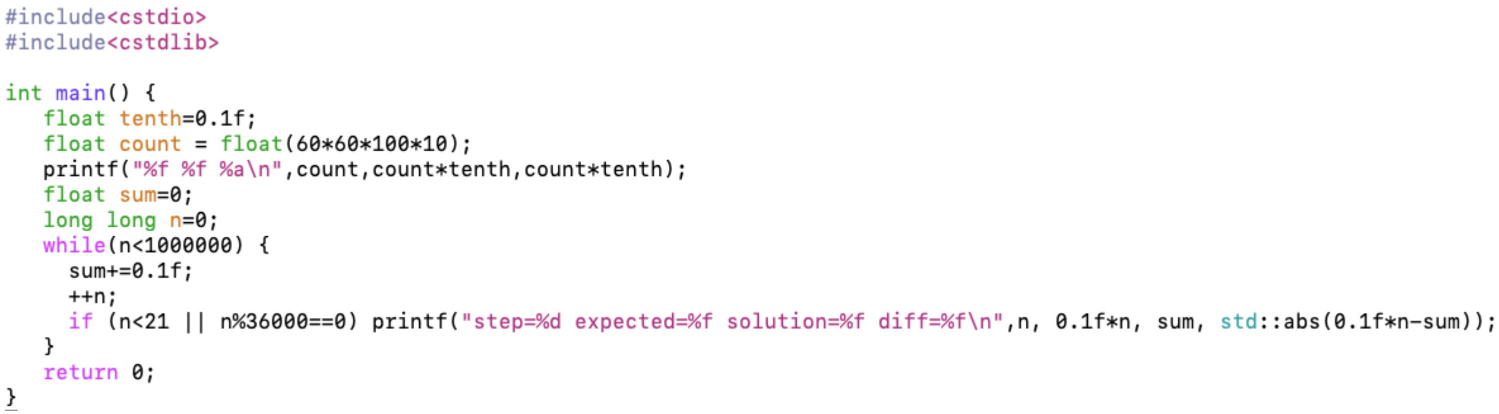
- After the second fill, the middle of the Gaussian starts flattening out
- Since the result of an operation is rounded, when adding a small and a large number, the small number might be dropped (ignored) if it is smaller than the ulp of the large number

Hands-on

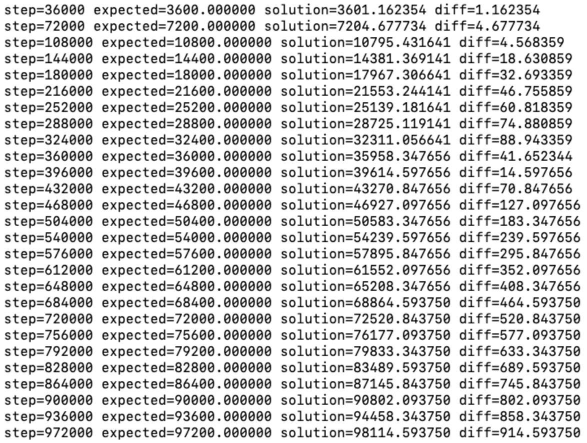
```
git clone https://github.com/beiwang2003/minicourse_fpa.git
```

Hands-on: Summing Many Numbers

- **Questions:** What are the potential arithmetic issues when summing many numbers?
  - Compile the code: *g++ -std=c++11 -Wall -march=native nativeSum.cpp –o nativeSum*
  - Run the code: *./nativeSum*



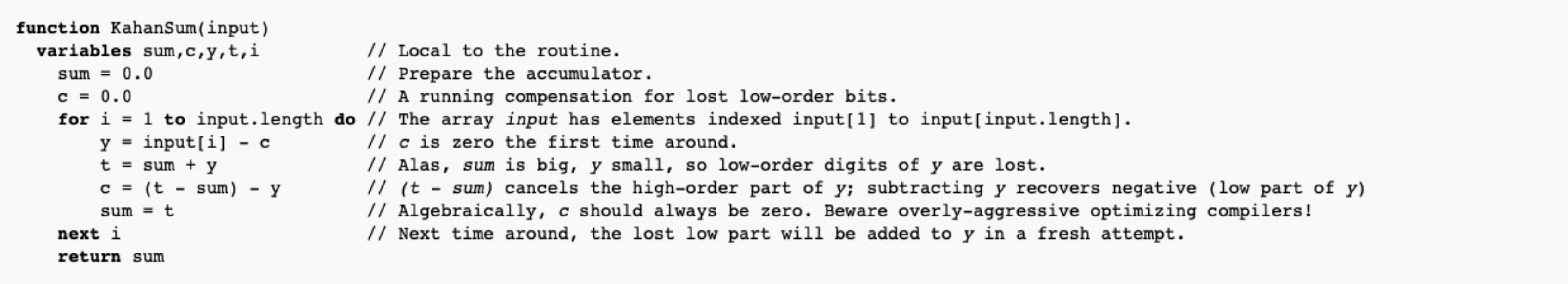
```
#include<cstdio>
#include<cstdlib>

int main() {
   float tenth=0.1f;
   float count = float(60*60*100*10);
   printf("%f %f %a\n",count,count*tenth,count*tenth);
   float sum=0;
   long long n=0;
   while(n<1000000) {
     sum+=0.1f;
     ++n;
     if (n<21 || n%36000==0) printf("step=%d expected=%f solution=%f diff=%f\n",n, 0.1f*n, sum, std::abs(0.1f*n-sum));
   }
   return 0;
}
```



```
step=36000 expected=3600.000000 solution=3601.162354 diff=1.162354
step=72000 expected=7200.000000 solution=7204.677734 diff=4.677734
step=108000 expected=10800.000000 solution=10795.431641 diff=4.568359
step=144000 expected=14400.000000 solution=14381.369141 diff=18.630859
step=180000 expected=18000.000000 solution=17967.306641 diff=32.693359
step=216000 expected=21600.000000 solution=21553.244141 diff=46.755859
step=252000 expected=25200.000000 solution=25139.181641 diff=60.818359
step=288000 expected=28800.000000 solution=28725.119141 diff=74.880859
step=324000 expected=32400.000000 solution=32311.056641 diff=88.943359
step=360000 expected=36000.000000 solution=35958.347656 diff=41.652344
step=396000 expected=39600.000000 solution=39614.597656 diff=14.597656
step=432000 expected=43200.000000 solution=43270.847656 diff=70.847656
step=468000 expected=46800.000000 solution=46927.097656 diff=127.097656
step=504000 expected=50400.000000 solution=50583.347656 diff=183.347656
step=540000 expected=54000.000000 solution=54239.597656 diff=239.597656
step=576000 expected=57600.000000 solution=57895.847656 diff=295.847656
step=612000 expected=61200.000000 solution=61552.097656 diff=352.097656
step=648000 expected=64800.000000 solution=65208.347656 diff=408.347656
step=684000 expected=68400.000000 solution=68864.593750 diff=464.593750
step=720000 expected=72000.000000 solution=72520.843750 diff=520.843750
step=756000 expected=75600.000000 solution=76177.093750 diff=577.093750
step=792000 expected=79200.000000 solution=79833.343750 diff=633.343750
step=828000 expected=82800.000000 solution=83489.593750 diff=689.593750
step=864000 expected=86400.000000 solution=87145.843750 diff=745.843750
step=900000 expected=90000.000000 solution=90802.093750 diff=802.093750
step=936000 expected=93600.000000 solution=94458.343750 diff=858.343750
step=972000 expected=97200.000000 solution=98114.593750 diff=914.593750
```

Inspired by the patriot missile failure problem: http://www-users.math.umn.edu/~arnold/disasters/patriot.html

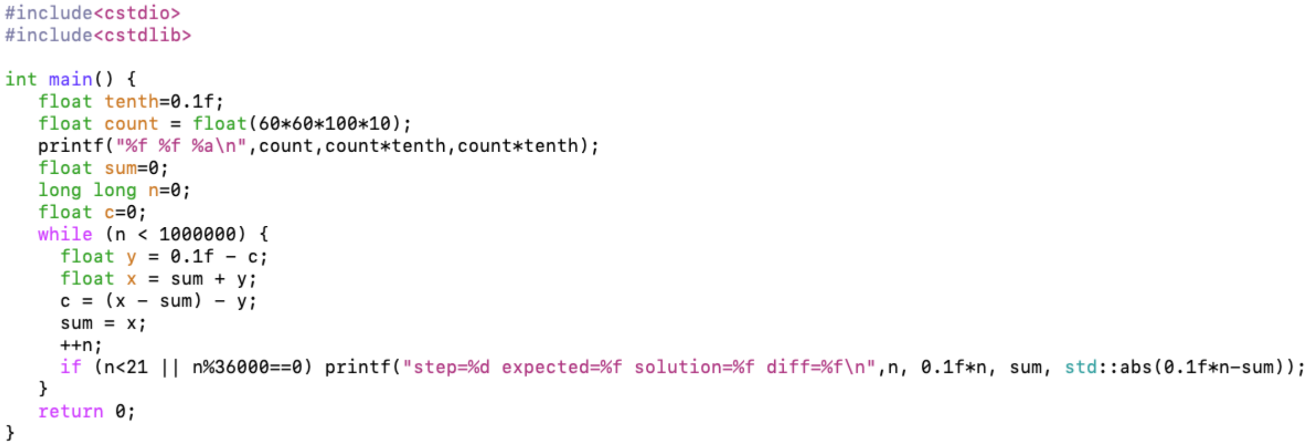
Hands-on: Kahan Summation Algorithm



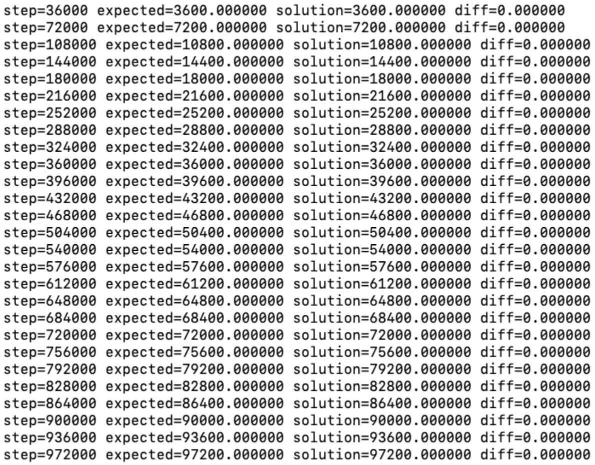
```
function KahanSum(input)
  variables sum,c,y,t,i          // Local to the routine.
    sum = 0.0                    // Prepare the accumulator.
    c = 0.0                      // A running compensation for lost low-order bits.
    for i = 1 to input.length do // The array input has elements indexed input[1] to input[input.length].
        y = input[i] - c         // c is zero the first time around.
        t = sum + y              // Alas, sum is big, y small, so low-order digits of y are lost.
        c = (t - sum) - y        // (t - sum) cancels the high-order part of y; subtracting y recovers negative (low part of y)
        sum = t                  // Algebraically, c should always be zero. Beware overly-aggressive optimizing compilers!
    next i                       // Next time around, the lost low part will be added to y in a fresh attempt.
    return sum
```

See: https://en.wikipedia.org/wiki/Kahan_summation_algorithm

- Compile the code: *g++ -std=c++11 -Wall -march=native kahanSum.cpp –o kahanSum*
- Run the code: *./kahanSum*



```cpp
#include<cstdio>
#include<cstdlib>

int main() {
   float tenth=0.1f;
   float count = float(60*60*100*10);
   printf("%f %f %a\n",count,count*tenth,count*tenth);
   float sum=0;
   long long n=0;
   float c=0;
   while (n < 1000000) {
     float y = 0.1f - c;
     float x = sum + y;
     c = (x - sum) - y;
     sum = x;
     ++n;
     if (n<21 || n%36000==0) printf("step=%d expected=%f solution=%f diff=%f\n",n, 0.1f*n, sum, std::abs(0.1f*n-sum));
   }
   return 0;
}
```



```
step=36000 expected=3600.000000 solution=3600.000000 diff=0.000000
step=72000 expected=7200.000000 solution=7200.000000 diff=0.000000
step=108000 expected=10800.000000 solution=10800.000000 diff=0.000000
step=144000 expected=14400.000000 solution=14400.000000 diff=0.000000
step=180000 expected=18000.000000 solution=18000.000000 diff=0.000000
step=216000 expected=21600.000000 solution=21600.000000 diff=0.000000
step=252000 expected=25200.000000 solution=25200.000000 diff=0.000000
step=288000 expected=28800.000000 solution=28800.000000 diff=0.000000
step=324000 expected=32400.000000 solution=32400.000000 diff=0.000000
step=360000 expected=36000.000000 solution=36000.000000 diff=0.000000
step=396000 expected=39600.000000 solution=39600.000000 diff=0.000000
step=432000 expected=43200.000000 solution=43200.000000 diff=0.000000
step=468000 expected=46800.000000 solution=46800.000000 diff=0.000000
step=504000 expected=50400.000000 solution=50400.000000 diff=0.000000
step=540000 expected=54000.000000 solution=54000.000000 diff=0.000000
step=576000 expected=57600.000000 solution=57600.000000 diff=0.000000
step=612000 expected=61200.000000 solution=61200.000000 diff=0.000000
step=648000 expected=64800.000000 solution=64800.000000 diff=0.000000
step=684000 expected=68400.000000 solution=68400.000000 diff=0.000000
step=720000 expected=72000.000000 solution=72000.000000 diff=0.000000
step=756000 expected=75600.000000 solution=75600.000000 diff=0.000000
step=792000 expected=79200.000000 solution=79200.000000 diff=0.000000
step=828000 expected=82800.000000 solution=82800.000000 diff=0.000000
step=864000 expected=86400.000000 solution=86400.000000 diff=0.000000
step=900000 expected=90000.000000 solution=90000.000000 diff=0.000000
step=936000 expected=93600.000000 solution=93600.000000 diff=0.000000
step=972000 expected=97200.000000 solution=97200.000000 diff=0.000000
```

Algorithm Considerations

- Numerical algorithms often needs to sum up a large number of values
  - e.g., matrix matrix multiplication
- The problem gets even more complicated on parallel computers
- A common technique to maximize floating point arithmetic accuracy is to pre-sort the data. On parallel computer,
  - Divide up the number in groups
  - Sort the data in each group and sum them sequentially by one thread
  - A reduction for the partial sum from each thread

Cancellation

- Cancellation occurs when we subtract two almost equal numbers
- The consequence is the error could be much larger than the machine epsilon
- For example, consider two numbers

$$x = 3.141592653589793 \text{ (16-digit approximation to } \pi\text{)}$$

$$y = 3.141592653585682 \text{ (12-digit approximation to } \pi\text{)}$$

Their difference is

$$z = x - y = 0.000000000004111 = 4.111\text{x}10^{-12}$$

In a C program, if we store x, y in single precision and display z in single precision, the difference is

0.000000e+00 Complete loss of accuracy

If we store x, y in double precision and display z in double precision, the difference is

4.110933815582030e-12 Partial loss of accuracy

Cancellation: The Solution of Quadratic Equation

- Consider the quadratic equation $ax^2 + bx + c = 0$, the roots are

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Two sources of cancellation

- A better solution will be

$$x_1 = \frac{-b - sign(b)\sqrt{b^2 - 4ac}}{2a}$$

$$x_2 = \frac{2c}{-b - sign(b)\sqrt{b^2 - 4ac}} = \frac{c}{ax_1}$$

- When a=1, b= 1.786737589984535 and c=1.149782767465722x10$^{-8,}$ in double precision, the first formula yields

$$x_1 = {}^{(1.786737601482363+1.786737578486707)}/_{2} = 1.786737589984535$$

$$x_2 = {}^{(1.786737601482363-1.786737578486707)}/_{2} = 0.000000011497828$$

Cancellation

- The second formula yields

$$x_1 = {}^{(1.786737601482363+1.786737578486707)}/_{2} = 1.786737589984535$$

$$x_2 = {}^{2.054360090947453\times10^{-8}}/_{1.786737589984535} = 1.149782767465722\times10^{-8}$$

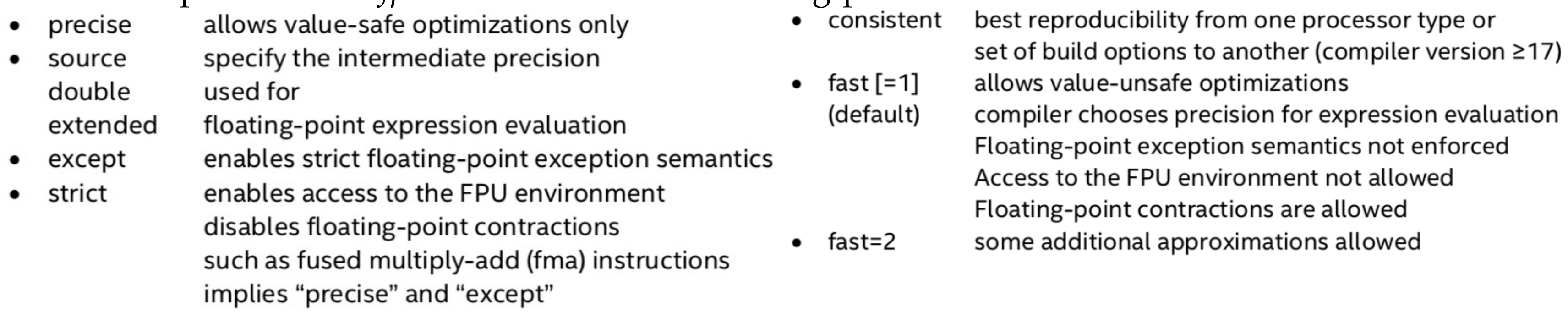
Exceptions

- The IEEE floating point standard defines several exceptions that occur when the result of a floating point operation is unclear or undesirable. Exceptions can be ignored, in which case some default action is taken, such as returning a special value. When trapping is enabled for an exception, an error is signaled whenever that exception occurs. Possible floating point exceptions:
  - **Underflow:** The result of an operation is too small to be represented as a normalized float in its format. If trapping is enabled, the *floating-point-underflow* condition is signaled. Otherwise, the operation results in a denormalized float or zero.
  - **Overflow:** The result of an operation is too large to be represented as a float in its format. If trapping is enabled, the *floating-point-overflow* exception is signaled. Otherwise, the operation results in the appropriate infinity.
  - **Divide-by-zero**: A float is divided by zero. If trapping is enabled, the *divide-by-zero* condition is signaled. Otherwise, the appropriate infinity is returned.
  - **Invalid:** The result of an operation is ill-defined, such as (0.0/0.0). If trapping is enabled, the *floating-point-invalid* condition is signaled. Otherwise, a quiet NaN is returned.
  - **Inexact:** The result of a floating point operation is not exact, i.e. the result was rounded. If trapping is enabled, *the floating-point-inexact* condition is signaled. Otherwise, the rounded result is returned.
- Trapping of these exceptions can be enabled through compiler flags, but be aware that the resulting code will run slower.

Approximate Math

Strict IEEE 754 vs Fast Math

- Compilers can treat FP math either *in "strict IEEE754 mode"* or *"fast math"* using algebra rules for real numbers
- Compiler options allow you to control tradeoffs among accuracy, reproducibility and speed
  - **GCC Compilers**
    - gcc default is "strict IEEE 754 mode"
    - *-O2 –funsafe-math-optimization*: allow arbitrary reassocations and transformations
    - *-O2 –ffast-math*: -funsafe-math-optimization + no exceptions and special quantities handling enforcement
    - *-Ofast*: -O3 (turn on vectorization) + -ffast-math + others
    - See: https://gcc.gnu.org/wiki/FloatingPointMath
  - **Intel Compilers**
    - icc uses compiler switch *–fp-model* to choose the floating-point semantics



precise	allows value-safe optimizations only
source double extended	specify the intermediate precision used for floating-point expression evaluation
except	enables strict floating-point exception semantics
strict	enables access to the FPU environment disables floating-point contractions such as fused multiply-add (fma) instructions implies "precise" and "except"
consistent	best reproducibility from one processor type or set of build options to another (compiler version ≥17)
fast [=1] (default)	allows value-unsafe optimizations compiler chooses precision for expression evaluation Floating-point exception semantics not enforced Access to the FPU environment not allowed Floating-point contractions are allowed
fast=2	some additional approximations allowed

- See: https://software.intel.com/en-us/articles/consistency-of-floating-point-results-using-the-intel-compiler

Speeding Math Up

- Typical cost of operations in modern CPU

operations	instruction	SSE single	SSE double	AVX single	AVX double (FMA)
+, -	ADD, SUB	3	3	3	3 4
*	MUL	5	5	5	5 4
/,sqrt	DIV, SQRT	10-14	10-22	21-29	21-45
1.f/, 1.f/sqrt	RCP, RSQRT	5		7	

- Avoid or factorize-out division and sqrt
  - If possible, compile with "–Ofast" or "-ffast-math"
  - If possible, use hardware-supported reciprocal square root
- Prefer linear algebra to trigonometric functions
- Choose precision to match required accuracy
  - Square and square-root decrease precision
  - Catastrophic precision-loss in the subtraction of almost-equal large numbers

https://agenda.infn.it/event/16941/contributions/34831/attachments/24523/27966/Vincenzo_OptimalFloatingPoint2018.pdf
https://stackoverflow.com/questions/39095993/does-each-floating-point-operation-take-the-same-time

Fused Multiply Add (FMA)

- fma(a,b,c) = round(a*b+c)
  - Opposed to round(round(a*b)+c)
- Single instruction with 4 or 5 cycle latency
  - Opposed to 2 instructions with 5+3 cycle latency
- More precise with one rounding
  - Opposed to two, but the results will be different
- Introduce a “contraction” issue
  - sqrt(a*a – b*b) may be contracted using FMA like fma(a, a, - b*b). If a==b, the result can be nonzero
- Compiler support
  - Intel Compiler: -fp-model = strict (default is ”fast”)
  - GCC (and Clang) flags: -ffp-contract=off (default is “fast”)
  - https://stackoverflow.com/questions/34436233/fused-multiply-add-and-default-rounding-modes

Lessons Learned

- Representing real numbers in a computer always involves an approximation and a potential loss of significant digits.
- Testing for the equality of two real numbers is not a realistic way to think when dealing with the numbers in a computer. It is more realistic to test the difference of two numbers with respect to machine epsilon.
- Performing arithmetic on very small or very large numbers can lead to errors that are not possible in abstract mathematics. We can get underflow and overflow, and the order in which we do arithmetic can be important. This is something to be aware of when writing low-level software to to do computations.
- The more bits we use to represent a number, the greater the precision of the representation and the more memory we consume.

https://www.stat.berkeley.edu/~nolan/stat133/Spr04/chapters/representations.pdf

References

- *What Every Computer Scientist Should Know About Floating-Point Arithmetic*. Goldberg. https://docs.oracle.com/cd/E19957-01/806- 3568/ncg_goldberg.html
- *Numerical Computing with IEEE Floating Point Arithmetic.* Overton, SIAM 2001
- Chapter 6, numerical considerations, *Programming Massively Parallel Processors*, A hand-on approach, 3rd edition, David B. Kirk and Wen-wei W. Hwu
- ESC18, "Optimal Floating Point Computation", Vincenzo Innocente, https://agenda.infn.it/event/16941/contributions/34831/attachments/24523/27966/Vincenzo_OptimalFloatingPoint2018.pdf
- CoDas-HEP 2018, "Floating Point is Not Real", Matthieu Lefebvre, https://indico.cern.ch/event/707498/contributions/2916937/attachments/1691892/2724587/codas_fpa.pdf
- https://stackoverflow.com/questions/39095993/does-each-floating-point-operation-take-the-same-time
- *Accumulation of Round-Off Error in Fast Fourier Transforms*, T. Kaneko and B. Liu, Jr. of ACM, Vol. 17, No. 4, October 1970, pp. 637-654.

BACK UP

Floating Point Numbers

- Again, let us consider the floating point representation (assume x≠ 0)

$$x = -1^s \left(\sum_{i=0}^{p-1} d_i \, B^{-i} \right) B^e$$

where s ∈ $\{0, 1\}, B \geq 2, d_i \in \{0, \dots, B-1\}$ with $d_0 > 0, i \in \{0, \dots, p-1\}, e \in \{e_{min}, \dots, e_{max}\}$.

- What is the **largest number** in the system?

$$x_{max} = \left(\sum_{i=0}^{p-1} (B-1)\, B^{-i}\right) B^{e_{max}} = (1 - B^{-p}) B^{e_{max}+1}, (x_{max})_2 = (1 - 2^{-p}) 2^{e_{max}+1}$$

- What is the **smallest positive normalized** in the system

$$x_{min} = B^{e_{min}}, (x_{min})_2 = 2^{e_{min}}$$

- The gap between number 1.0 and the next larger floating point number is called ***machine epsilon.*** What is the machine epsilon in the system?

$$\varepsilon = B^{-(p-1)}, \varepsilon_2 = (1.00..1)_2 - (1.00..0)_2 = (0.00 \dots 1)_2 = 2^{-(p-1)}$$

- The gap between B^E and the next larger floating point number is called ***unit in the last place*** (ulp). What is the upl in the system?

$$ulp(x) = B^{-(p-1)} B^e = \varepsilon \, B^e, ulp(x)_2 = (1.00 \dots 1)_2 2^e - (1.00 \dots 0)_2 2^e = (0.00 \dots 1)_2 2^e = 2^{-(p-1)} 2^e$$

Correctly Rounded Arithmetic

- The IEEE standard requires that the result of addition, subtraction, multiplication and division be **exactly rounded.**
  - Exactly rounded means the results are calculated exactly and then rounded. For example: assuming p =23, x=(1.00..00)$_2$x2^0 and z=(1.00..01)$_2$x2^{-25}, then x-z is

$$\begin{array}{rll}
 & (\ 1.00000000000000000000000| &)_2 \times 2^0 \\
- & (\ 0.00000000000000000000000|0100000000000000000000001 &)_2 \times 2^0 \\
= & (\ 0.11111111111111111111111|1011111111111111111111111 &)_2 \times 2^0 \\
\text{Normalize :} & (\ 1.11111111111111111111111|0111111111111111111111110 &)_2 \times 2^{-1} \\
\text{Round to} & & \\
\text{Nearest :} & (\ 1.11111111111111111111111 &)_2 \times 2^{-1}
\end{array}$$

  - Compute the result exactly is very expensive if the operands differ greatly in size
  - The result of two or more arithmetic operations are NOT exactly rounded
- How is **correctly rounded** arithmetic implemented?
  - Using two additional *guard bits* plus one *sticky* bit guarantees that the result will be the same as computed using exactly rounded [Goldberg 1990]. The above example can be done as

$$\begin{array}{rll}
 & (\ \ 1.00000000000000000000000| &)_2 \times 2^0 \\
- & (\ \ 0.00000000000000000000000|011 &)_2 \times 2^0 \\
= & (\ \ 0.11111111111111111111111|101 &)_2 \times 2^0 \\
\text{Normalize :} & (\ \ 1.11111111111111111111111|01 &)_2 \times 2^{-1} \\
\text{Round to Nearest :} & (\ \ 1.11111111111111111111111 &)_2 \times 2^{-1}
\end{array}$$

Correctly Rounded Arithmetic

- The IEEE standard requires that the result of addition, subtraction, multiplication and division be **exactly rounded.**
  - Exactly rounded means the results are calculated exactly and then rounded. For example: assuming p =23, x=(1.00..00)$_2$x2^0 and z=(1.00..01)$_2$x2^{-25}, then x-z is

$$\begin{array}{rll}
 & (\ 1.00000000000000000000000| &)_2 \times 2^0 \\
- & (\ 0.00000000000000000000000|0100000000000000000000001 &)_2 \times 2^0 \\
= & (\ 0.11111111111111111111111|1011111111111111111111111 &)_2 \times 2^0 \\
\text{Normalize}: & (\ 1.11111111111111111111111|0111111111111111111111110 &)_2 \times 2^{-1} \\
\text{Round to} & & \\
\text{Nearest}: & (\ 1.11111111111111111111111 &)_2 \times 2^{-1}
\end{array}$$

  - Compute the result exactly is very expensive if the operands differ greatly in size
  - The result of two or more arithmetic operations are NOT exactly rounded
- How is **correctly rounded** arithmetic implemented?
  - Using two additional *guard bits* plus one *sticky* bit guarantees that the result will be the same as computed using exactly rounded [Goldberg 1990]. The above example can be done as

$$\begin{array}{rll}
 & (\ \ 1.00000000000000000000000| &)_2 \times 2^0 \\
- & (\ \ 0.00000000000000000000000|011 &)_2 \times 2^0 \\
= & (\ \ 0.11111111111111111111111|101 &)_2 \times 2^0 \\
\text{Normalize}: & (\ \ 1.11111111111111111111111|01 &)_2 \times 2^{-1} \\
\text{Round to Nearest}: & (\ \ 1.11111111111111111111111 &)_2 \times 2^{-1}
\end{array}$$

Cancellation

- Cancellation occurs when we operate numbers that are not in floating point format
- For every $x \epsilon \mathbb{R}$, there exists $|\varepsilon| < \varepsilon_{mach}$ such that

$$round(x) = x\,(\,1 + \,\varepsilon\,).$$

- Thus

$$\begin{aligned} round\big(round(x) - round(y)\big) &= \big(round(x) - round(y)\big)(1 + \varepsilon_3) \\ &= (x(1 + \varepsilon_1) - y(1 + \varepsilon_2))\ (1 + \varepsilon_3) \\ &= (x - y)(1 + \varepsilon_3) + (x\varepsilon_1 \text{-} y\varepsilon_2)(1 + \varepsilon_3), \end{aligned}$$

and if $(x - y) \neq 0$,

$$\left|\frac{(round(round(x) - round(y))}{x - y}\right| = \left|\,\varepsilon_3 + \frac{x\,\varepsilon_1 - y\varepsilon_2}{x - y}(1 + \varepsilon_3)\,\right|$$

when $x\,\varepsilon_1 - y\varepsilon_2 \neq 0$, and x-y is small, the error could be $\gg\ \varepsilon_{mach}$