Introduction to floating point internals and limitations

Dmitrii Milideev, 2020-02-13

Why does it matter?

- We develop our algorithms for real numbers
- Computers can store only a finite subset of reals
- We use float and double types as black boxes, hoping for the best.
- Sometimes we have problems because of that, sometimes not.
  - https://web.ma.utexas.edu/users/arbogast/misc/disasters.html
  - Patriot missile crash
  - The short flight of the Ariane 5.
  - Parliamentary elections in Schleswig-Holstein.

What is floating point number

Every real number (except zero) can be represented in the form

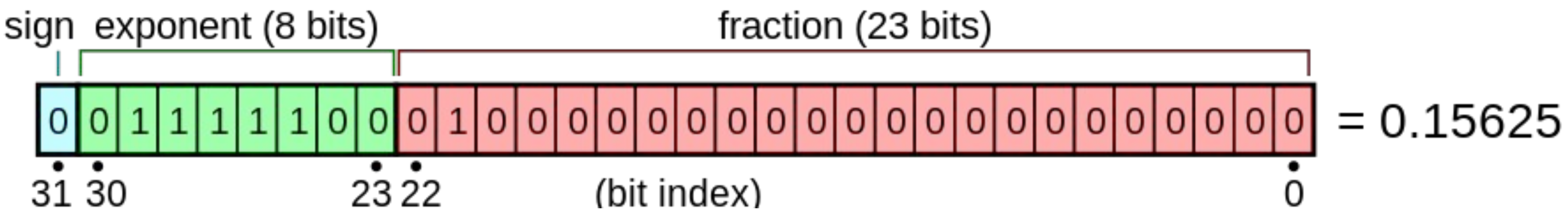
$$\pm 1.\mathtt{ddd...} \times 2^e$$

Floating point numbers have limited number of digits

$$\pm 1.\mathtt{dd...d} \times 2^e$$

IEEE754 representation

Picture taken from https://en.wikipedia.org/wiki/Single-precision_floating-point_format



- Exponent is stored as unsigned integer equal to `<real_exponent> + 127`
  - This format known as biased representation
  - 127 is constant known exponent bias - every IEEE754 number format has it’s own bias.
- Exponent range is `-126` to `+127`. Corresponding biased values are `0x01` and `0xFE`.

Special biased exponent value: 0x00

- Representation of zero - all fraction bits are zero
  - Zero can be signed.
- Denormal numbers

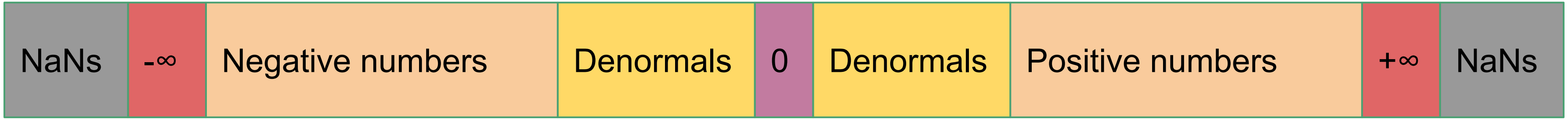
$$\pm 0.\mathrm{dd...d} \times 2^{-127}$$

  - Form a fixed-point subset around zero
  - Super-slow, don't use them.

Special biased exponent value: 0xFF

- All fraction bits are zero - infinity
  - Positive infinity.
  - Negative infinity.
- Non-zero fraction part - NaN

Overview of available values



Ulps and Ufps

- Let's say we store p fraction bits (p = 23 for floats and 52 for doubles)

$$\pm 1.d_1 d_2 \ldots d_p \times 2^e$$

- Actual value of the number

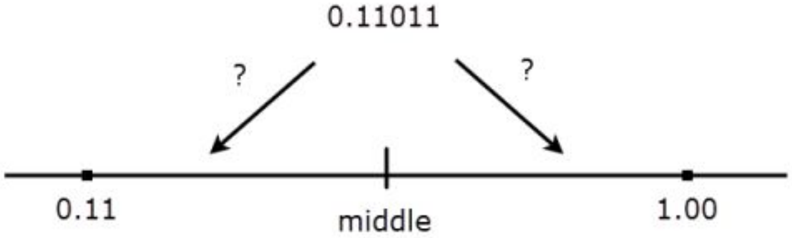
$$x = \pm(2^e \cdot 1 + 2^{e-1} \cdot d_1 + 2^{e-2} \cdot d_2 + \ldots + 2^{e-p} \cdot d_p)$$

- $\mathrm{ufp}(x) = 2^e$
  - can be defined for real numbers as $2^{\lfloor \log |x| \rfloor}$ (according to S.Rump)
- $\mathrm{ulp}(x) = \mathrm{ufp}(x) \cdot 2^{-p}$

Rounding

- In order to put real number into finite representation, it needs to be rounded to a representable number.
- In most cases real number has two neighbors. Rounding can be seen as a process of selection between them.
- Special case if real number is located exactly between two floating-point numbers - need to apply tie-breaking rule.

```
We have:                : 1.10110111110000011 11011...
Previous float number : 1.10110111110000011
Next float number     : 1.10110111110000100
Median float number   :
```



Picture copied from https://blog.angularindepth.com/how-to-round-binary-fractions-625c8fa3a1af

Rounding modes

- **Default** - Round to nearest, ties to even
- Round to nearest, ties away from zero
- Directed modes
  - Towards zero
  - Towards $+\infty$
  - Towards $-\infty$

Rounding error

- When rounding to nearest, absolute error is `ulp(x)/2`
- Relative error is bounded by **u** - *unit roundoff* (or *machine epsilon*)
- Constant **u** is dependent only on a number of digits we are storing.
- **u** equals
  - 2^{-24} for float
  - 2^{-53} for double

Rounding happens after every operation

- IEEE754 requires every operation to be exactly rounded
  - Inputs are treated as exact numbers
  - Operations performed with infinite mantissa length
  - Rounded afterwards
- Rounding happens implicitly after *every* arithmetic operation.
  - You get $\circ(a \cdot b)$ instead of $a \cdot b$
- Sometimes you're lucky and result of operation is *exact*
  - Happens when mantissa of the result is short enough to fit your datatype
- Example: for a and b, the rounded multiplication result is somewhere here

$$[\mathbf{(a \cdot b)} - \mathrm{ulp}(a \cdot b)/2 \ ; \ \mathbf{(a \cdot b)} + \mathrm{ulp}(a \cdot b)/2]$$

$$[\mathbf{(a \cdot b)} - \mathbf{u} * x \ ; \ \mathbf{(a \cdot b)} + \mathbf{u} * x]$$

Examples of exact operations

- All divisions and multiplications by 2^x
  - You modify only exponent not mantissa. Nothing to round
- All integers that fit mantissa have exact representation
  - Multiplications, additions and subtractions are all exact
- All fixed-point numbers operations are exact (e.g. only 4 digits after radix point)
  - Essentially, they are integers with shifted exponent value
  - Additions are exact
  - Multiplication requires `n+m` digits, if it fits mantissa - result is exact.

Examples of inexact operations

- All money-related data
  - In binary representation, constant `0.1` is `0.0001100110011(0011)`
  - All dollars-and-cents values cannot be stored exactly. They are always rounded (up or down).
  - You always get slightly imprecise results.
- Obviously, results of most divisions and trig functions.

Fused multiply–add (FMA) operations

- FMA(a, b, c) computes a*b + c with single rounding operation
  - Have implementation in silicon
- You get $\circ(a \cdot b + c)$ instead of $\circ(\circ(a \cdot b) + c)$
- Computations are faster and more accurate with FMA.
  - For example, Eigen always trying to use FMA instructions if possible
- Compilers are pretty conservative with the usage of FMA instructions
  - You have to ask them specifically
- Available since C++11 with `std::fma`

Example: determinant of the 2x2 matrix

We want to compute

$$\mathtt{det(A)} = a_{11} \cdot a_{22} - a_{12} \cdot a_{21}$$

Our data type is 32-bit float from C++.

```
float a11 = 6.0f - 56 * 0x1p-12;
float a12 = 6.0f - 55 * 0x1p-12;
float a21 = 18.0f - 57 * 0x1p-12;
float a22 = 18.0f - 54 * 0x1p-12;
```

Example: determinant of the 2x2 matrix

We want to compute

$$\mathtt{det(A)} = a_{11} \cdot a_{22} - a_{12} \cdot a_{21}$$

Binary representation.

```
float a11 = +1.0111 1111 0010 0000 0000 000 * 2^2
float a12 = +1.0111 1111 0010 0100 0000 000 * 2^2
float a21 = +1.0001 1111 1100 0111 0000 000 * 2^4
float a22 = +1.0001 1111 1100 1010 0000 000 * 2^4
```

Example: determinant of the 2x2 matrix

Let’s compute exact result $a_{11} \cdot a_{22}$

$$a_{11} \cdot a_{22} = +1.1010\ 1110\ 1011\ 0011\ 0010\ 1111\ 01 * 2^{6}$$

Let’s compute rounded result $\circ(a_{11} \cdot a_{22})$

$$\circ(a_{11} \cdot a_{22}) = +1.1010\ 1110\ 1011\ 0011\ 0011\ 000 * 2^{6}$$

Result of multiplication was rounded up.

Example: determinant of the 2x2 matrix

Let’s compute exact result $a_{12} \cdot a_{21}$

$$a_{12} \cdot a_{21} = +1.1010\ 1110\ 1011\ 0011\ 0011\ 000\textcolor{red}{0\ 1111\ 11} * 2^6$$

Let’s compute rounded result $\circ(a_{12} \cdot a_{21})$

$$\circ(a_{12} \cdot a_{21}) = +1.1010\ 1110\ 1011\ 0011\ 0011\ 000 * 2^6$$

Result of multiplication was rounded down.

Example: determinant of the 2x2 matrix

Let’s compute

$$\circ(a_{11} \cdot a_{22}) - \circ(a_{12} \cdot a_{21})$$

Result is zero.

Correct result is:

```
111 * 0x1p-24
```

```
-6.61612e-06
```

```
-1.1011 1100 0000 0000 0000 000 * 2^-18
```

Problems with error accumulation

- Roundoff error in intermediate results might lead to a pretty big error in final result.
    - Most dangerous part is if you have an error accumulation inside a loop.
- Even if your data does not require large mantissa, your intermediate results usually do.
    - For example, comparing x-coordinates of intersection points of two segments might require 5x mantissa length.

Easy error estimation

- Simplest way to estimate roundoff error is mid-point interval arithmetic or ball arithmetic.
  - See: S.M. Rump. Fast and parallel interval arithmetic, 1999.
- Every number is viewed as a random value from eps-neighborhood of some real value:

$$[x - \varepsilon;\ x + \varepsilon] =: \langle x,\ \varepsilon\rangle$$

- Arithmetic rules are defined like

$$\langle x,\ \varepsilon_1\rangle + \langle y,\ \varepsilon_2\rangle = \langle x+y,\ \varepsilon_1+\varepsilon_2\rangle$$

$$\langle x,\ \varepsilon_1\rangle * \langle y,\ \varepsilon_2\rangle = \langle x*y,\ \varepsilon_1*|y| + \varepsilon_2*|x| + \varepsilon_1*\varepsilon_2\rangle$$

$$\circ(\langle x,\ \varepsilon\rangle) = \langle x,\ \varepsilon + \mathbf{u}*(|x| + \varepsilon)\rangle$$

Easy error estimation: example

$$\mathtt{det(A)} = a_{11} \cdot a_{22} - a_{12} \cdot a_{21}$$

- All elements are bounded by some constant $|a_{ij}| < A$
- Treat inputs as exact: a_{ij} estimation is $\langle A,\ 0 \rangle$
- $a_{ij} \cdot a_{ij}$ estimation is $\langle A \cdot A,\ 0 \rangle$
- $\circ(a_{ij} \cdot a_{ij})$ estimation is $\langle A^2,\ \mathbf{u} \cdot A^2 \rangle$
- $\circ(a_{11} \cdot a_{22}) - \circ(a_{12} \cdot a_{21})$ estimation is $\langle 2 \cdot A^2,\ 2 \cdot \mathbf{u} \cdot A^2 \rangle$
- $\mathtt{det(A)} = \circ(\circ(a_{11} \cdot a_{22}) - \circ(a_{12} \cdot a_{21}))$ estimation is

$$\langle 2 \cdot A^2,\ 3 \cdot \mathbf{u} \cdot A^2 + 2 \cdot \mathbf{u}^2 \cdot A^2 \rangle$$

Easy error estimation: example

$$\langle A^2,\ 3\cdot\mathbf{u}\cdot A^2 + 2\cdot\mathbf{u}^2\cdot A^2\rangle$$

- For float, $\mathbf{u}=2^{-24}$
- Let's say `A=20`
- Error estimation value is

```
7.152557657263969e-05
```

Libraries

- GNU GMP
  - https://gmplib.org/
- boost::multiprecision
  - https://www.boost.org/doc/libs/1_72_0/libs/multiprecision/doc/html/index.html
- Intel C++ Math Library

Thanks!

- Questions?
- Some good references on exactness approach
  - S. Rump is my favorite author, pretty much everything is from him
    http://www.ti3.tu-harburg.de/rump/
    - S.M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation part I: Faithful rounding. http://www.ti3.tu-harburg.de/paper/rump/RuOgOi07I.pdf
    - S.M. Rump. Fast and parallel interval arithmetic
      http://www.ti3.tu-harburg.de/paper/rump/Ru99b.pdf
    - M. Lange and S.M. Rump. Faithfully Rounded Floating-point Computations
      http://www.ti3.tu-harburg.de/paper/rump/LaRu2017b.pdf
    - K. Ozaki, T. Ogita, S. Oishi. A robust algorithm for geometric predicate by error-free determinant transformation https://www.sciencedirect.com/science/article/pii/S0890540112000752
- Big book: J.-M. Muller, Handbook of Floating-Point Arithmetic (not freely available)
  https://www.springer.com/us/book/9783319765259