Lecture 4,5,6: Floating Point Arithmetic

BT 2020 – Numerical Methods for Biology

Karthik Raman
Department of Biotechnology, Bhupat & Jyoti Mehta School of Biosciences
Initiative for Biological Systems Engineering (IBSE)
Robert Bosch Centre for Data Science and Artificial Intelligence (RBC-DSAI)

Indian Institute of Technology Madras

https://home.iitm.ac.in/kraman/lab/
https://web.iitm.ac.in/ibse/
https://rbcdsai.iitm.ac.in/

"There are 10 types of people in this world — those who understand binary and those who don't!"

— Anonymous

Binary Numbers

- Positional notation (Decimal):

$$18.015 = 1 \times 10^1 + 8 \times 10^0 + 0 \times 10^{-1} + 1 \times 10^{-2} + 5 \times 10^{-3}$$

- Binary is similar, except the base is now 2

$$(11.011)_2 = 1 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} = 3.375_{10}$$

- Fractions in binary only terminate if the denominator has 2 as the only prime factor!
- $(0.\overline{1})_2 = ?$

Representing Integers on a Computer

- How many integers can you represent in n bits?
- (unsigned) 2^n: $0, 1, 2, \ldots, 2^n - 1$
- (signed) 2^n: $-2^{n-1}, -2^{n-1} + 1, \ldots, 0, 1, \ldots, 2^{n-1} - 1$
- In 32 bits, largest unsigned integer is $2^{32} - 1 = 4,294,967,295$; what happens if you add 1 to it?

- How *many* floating point numbers can you represent in n bits?
- (un)surprisingly, the answer is again 2^n
- Which 2^n numbers from $\mathbb{R}$ should we represent?
- Trade-off between **precision** and **range**

How do you represent floating point numbers on a computer?

Floating-Point Notation

- Similar to scientific notation, e.g. 2.018×10^3, 2.9979×10^8, $6.62607004 \times 10^{-34}$
- The point *floats*, unlike in a *fixed-point* notation, where the number decimals before and after the point are fixed — this is limiting — why?
- Formally, any floating-point number system $\mathbb{F}$ is characterised by four integers
  - β, the base
  - p, the precision
  - $[L, U]$, the exponent range
- Any floating-point number $x \in \mathbb{F}$ has the form

$$x = \pm \left(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \ldots + \frac{d_{p-1}}{\beta^{p-1}} \right) \beta^E; \quad L \leq E \leq U$$

Floating-Point Notation

$$x = \pm \left(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \ldots + \frac{d_{p-1}}{\beta^{p-1}} \right) \beta^E; \quad L \leq E \leq U$$

- Typically, $\beta = 2$ on most computers
- Therefore, the digits $d_i \in 0, 1$
- The string of p digits $d_0 d_1 \ldots d_{p-1}$ is called *mantissa/significand*; $d_1 d_2 \ldots d_{p-1}$ is *fraction*, E is *exponent*
- For IEEE single-precision, $p = 24, L = -126, U = 127$
- For IEEE double-precision, $p = 53, L = -1,022, U = 1,023$

Floating-Point Notation

Normalisation

- A floating-point system is normalised if the leading digit d_0 is always non-zero, unless the number being represented is zero
- Normalisation is advantageous because
  - No digits are wasted on leading zeroes, maximising precision
  - Representation of each number is unique
  - In a binary system, leading bit is anyway 1! — helps gain an extra bit for precision

Floating-Point Notation

Properties

- What is the number of normalised floating-point numbers?
  - 2: 2 possibilities for sign
  - 1: 1 choice for d_0
  - 2^{p-1}: 2 choices for $d_1, d_2, \ldots, d_{p-1}$
  - $U - L + 1$: possible choices for exponent
- For double precision, this value is $2 \times 1 \times 2^{52} \times 2046 + 1$ (for zero)
- Smallest possible number is $1.\underbrace{00\ldots00}_{p-1 \text{ digits}} \times 2^L = 2^{-1022}$

Implications of Floating-Point Representation

Suppose that we had a hypothetical base-10 computer with a 5-digit word size. Assume that one digit is used for the sign, two for the exponent, and two for the mantissa. For simplicity, assume that one of the exponent digits is used for its sign, leaving a single digit for its magnitude. A general representation of the number following normalisation would be

$$s_1 d_1 . d_2 \times 10^{s_0 d_0}$$

- What does the number line look like? Can you spot
- Where does overflow occur?
- Underflow?
- Hole around zero?

Remember...

- Between two real numbers, there are an infinite number of real numbers
- Between two floating-point numbers, there are a finite number of floating-point numbers (maybe even zero!)

IEEE 754 Representation — Single Precision

- 32 bits
  - 1 bit for the sign (s)
  - 8 bits for exponent (e) — decides range
  - 23 bits for mantissa (m) — precision

IEEE 754 Representation — Double Precision

- 64 bits
  - 1 bit for the sign (s)
  - 11 bits for exponent (e) — decides range
  - 52 bits for mantissa (m) — precision

$$x = (-1)^s \cdot 1.m \cdot 2^{e_b - 1023}$$

- e_b is stored by 11 bits — can range from 0 to 2047
- or, $-1023 \leq e \leq 1024$
- The extreme values -1023 ($e_b = 0$ — all 0's!) and 1024 (stored as $e_b = 2047$ — all 1's) are *special* — leaving $-1022 = L \leq e \leq U = 1023$
- $m \in 0, 2^{52}$

IEEE 754 Representation — Double Precision

$$x = (-1)^s \cdot 1.m \cdot 2^{e_b - 1023}$$

- Largest number: $s = 0, m = \underbrace{11\ldots11}_{52 \text{ digits}}, e_b = 2046$
- $\text{real}_{\max} = (-1)^0 \cdot (1 + (1 - 2^{-52})) \cdot 2^{1023} \approx 2^{1024}$
- Smallest positive number: $s = 0, m = 0, e_b = 1$
- $\text{real}_{\min} = (-1)^0 \cdot (1 + 0) \cdot 2^{-1022} = 2^{-1022}$

See also

https://en.wikipedia.org/wiki/Double-precision_floating-point_format#Double-precision_examples

IEEE 754 Representation — Double Precision

- m limits the precision of the floating point number
- $0 \leq m < 1$
- The format $2^e \cdot (1 + m)$ provides an implicitly stored 1, so doubles actually have 53 bits of precision

IEEE 754 Representation — Double Precision

Examples

The number 1 is represented as

$$(1)^0 2^0 (1+0)$$

- $s = 0, e = 0, f = 0$ $(e_b = 1023)$

You can use `format hex` in MATLAB to see the bit pattern of the floating point number in hexadecimal. The first three hex digits **(12 bits)** represent the sign bit and the biased exponent, and the remaining 13 hex digits **(52 bits)** represent the mantissa.
For 1, the first 12 bits are 0 011 1111 1111 = 3ff

```
>> format hex
>> 1
   3ff0000000000000
```

Floating-Point Representation

More Examples

- How do you represent the number 12345?
- As an integer, the representation is 11000000111001_2
- As a float, the value will be $1.1000000111001_2 \times 2^{13}$
- To convert to IEEE 754, we need to add a sign bit (0), drop the 1. in the fraction, and convert the rest of the representation to 52 bits by adding trailing zeroes, and change the exponent to $13 + 1023 = 1036 = 10000001100_2$
- The final representation will be $0|100\ 0000\ 1100|1000\ 0001\ 1100\ 1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000 = 40c81c8000000000_{16}$

Special Float-Point Values

The exponents 000_{16} and $7ff_{16}$ have a special meaning:

- $00000000000_2 = 000_{16}$ is used to represent a signed zero (if $f = 0$) and *subnormals* (if $f \neq 0$); and
- $11111111111_2 = 7ff_{16}$ is used to represent ∞ (if $f = 0$) and NaNs (if $f \neq 0$),

$s = 0, e = 0$ $(e_b = 1023), f = 111\ 1111\ 1111$

```
>> Inf
   7ff0000000000000
```

$s = 1, e = 0, f = 111\ 1111\ 1111$

```
>> -Inf
   fff0000000000000
```

$s = 0, e = 0, f \neq 0$

```
>> NaN
   fff8000000000000
```

Special Float-Point Values

$s = 0, e_b = 0, f = 0$

>> 0
0000000000000000

What are the following numbers?

- $8000\ 0000\ 0000\ 0000_{16}$
- $0\ 000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2$
- $0\ 000\ 0000\ 0000\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2$
- $4009\ 21fb\ 5444\ 2d18_{16}$
- $3cb0\ 0000\ 0000\ 0000_{16}$

NaNs and Infs

- Invalid operations such as $\sqrt{-1}$ and $\log(-1)$ produce `NaN`
- Any operation involving a `NaN` produces another `NaN`
- All comparisons involving NaN return false
- Operations with `Inf` are as expected:
  - `Inf + finite = Inf`
  - `Inf / Inf = NaN`
  - `finite/Inf = 0`
  - `Inf + Inf = Inf`
  - `Inf - Inf = NaN`

Some Consequences of Floating-Point Arithmetic

```
>> 0.1+0.2-0.3
   5.5511e-17
>> sin(pi/6)^2 + cos(pi/6)^2 - 1
     0
>> 1- sin(pi/6)^2 - cos(pi/6)^2
  -1.1102e-16
>> 1- sin(pi/3)^2 - cos(pi/3)^2
     0
>> c = 0; for k=1:10, c = c + 0.1; end, for k
   =1:10 c = c - 0.1; end, disp(c)
   2.7756e-17
>> c = 0; for k=1:10, c = c + 0.125; end, for
   k=1:10 c = c - 0.125; end, disp(c)
     0
```

Some Consequences of Floating-Point Arithmetic

```
>> 0.1+0.1+0.1
   3fd3333333333334

>> 0.3
   3fd3333333333333

>> 2^53+1-2^53
      0

>> 2^53+2-2^53
      2
```

Floating Point Disasters

- Ariane 5 explosion
- Patriot Missile Failure
- “Rounding error changes Parliament makeup” (Germany, 1992)
- …

Read about these and post on Piazza!

How Not To Use Floating-Point Arithmetic ...

- Floating-Point comparisons! `if f==g`
  - Instead `if (abs(f-g) < tol)`
  - Recall absolute and relative errors
- Testing for convergence `while x_new~=x_old` e.g. $cos(x) = x$
  - Test may never be satisfied, because of oscillatory bit patterns
  - Or may take too long!
- Catastrophic cancellations may occur
  - Be careful with additions, subtractions!
  - e.g. $x^2 + 54.32x + 0.1 = 0$, with four-digit rounding
  - Catastrophic cancellation
    - Effects of round-off usually accumulate slowly, but
    - Subtracting nearly equal numbers leads to severe loss of precision (or adding two numbers of very different magnitude)—error caused by a single operation ("catastrophic")
    - Round-off is inevitable—good algorithms minimise the effect of round-off
  - Know machine precision ε_{mach}!

Floating-Point Arithmetic: Simple Workarounds

- Can the formula be re-arranged?
- e.g.

$$\sqrt{x+1}-1=\frac{x}{\sqrt{x+1}+1}$$

- How to rearrange the quadratic equation formula?

Floating-Point Representation

Other things to read about

- Gradual underflow
- Rounding
- Floating-point operations

Summary

- IEEE 754 has unified floating-point representations across architectures
- Double precision provides excellent precision and range in 64 bits
- All floating-point numbers have a sign bit, some bits for the normalised mantissa, and some bits for the exponent
- Special representations exist for 0, Inf, NaN etc.
- Numerical methods need to be aware of the approximations that arise due to floating-point calculations
- There are known "good practices" that one must observe while working with floating-point numbers