MATH7030 Advanced Numerical Methods

Chapter 1 Introduction to Numerical Computation

by Yuliang Wang

北京师范大学香港浸会大学联合国际学院

BEIJING NORMAL UNIVERSITY · HONG KONG BAPTIST UNIVERSITY
UNITED INTERNATIONAL COLLEGE

Email: `yuliangwang@uic.edu.cn`

1 Floating-point numbers and arithmetic

Decimal and binary number system

- Decimal Number System (Base 10)

$$123.45 = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 + 4 \times 10^{-1} + 5 \times 10^{-2}$$

- Binary number system (Base 2)

$$1101_2 = 2^3 + 2^2 + 0 + 2^0 = 13$$

$$1101.11_2 = 2^3 + 2^2 + 0 + 2^0 + 2^{-1} + 2^{-2} = 13.75$$

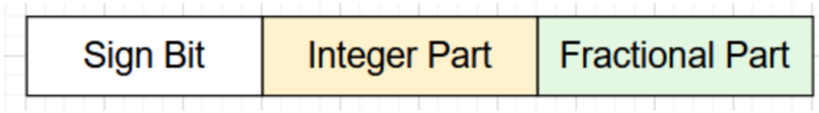
Exercise 1.1. Repeating expansions.

Exercise 1.2. Arithmetic operations on binary numbers.

Exercise 1.3. Convert numbers from base 10 to base 2.

Fixed-point representation

- Fixed-point representation has a fixed number of bits for the integral and fractional parts.
- There are three parts of the fixed-point number representation: **Sign bit**, **Integral part,** and **Fractional part**.



  - Sign bit: the negative number has a sign bit 1, while a positive number has a bit 0.
  - Integral Part: its length depends on the size of the computer word; for an 8-bit word, the integral part is 4 bits.
  - Fractional part: for an 8-bit word, the fractional part is 3 bits.

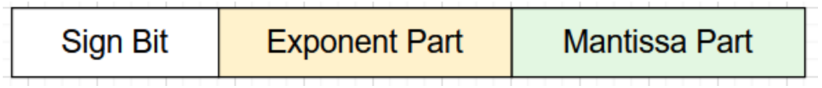
Example 1.1 Write 4.5 in 8-bit fixed-point representation.

Floating-point representation

- A floating-point representation has three parts: **Sign bit**, **Exponent Part**, and **Mantissa**:

$$\pm m \times b^E$$

  - m: **mantissa/significand**
  - b: **base**
  - E: **exponenent**



- A **single-precision** word consists of 32 bits: 1 bit for the sign (0 for $+$, 1 for $-$), 8 bits for the exponent, and 23 bits for the mantissa.
- The mantissa is always **normalized** so that $1 \leqslant m < 2$, e.g. $10 = 1010_2 = 1.010 \times 2^3$. So the first bit of the mantissa is always 1.

Floating-point representation

- A number that can be stored exactly using this scheme is called a **floating-point number**, e.g. $1010_2 = 1.010 \times 2^3$ is a floating-point number:

$$\boxed{0 \,\big|\, E = 3 \,\big|\, 10100\cdots0}$$

The exact format of E is discussed later.

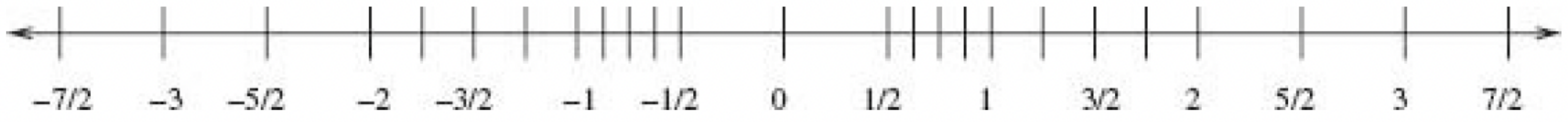
- Since the first bit in m is always 1, one can omit to obtain the **hidden-bit representation**:

$$\boxed{0 \,\big|\, E = 3 \,\big|\, 0100\cdots0}$$

- $0.1 = 0.000\overline{1100}_2 = 1.100\overline{1100}_2 \times 2^{-4}$ is not a floating-point number since it must be rounded in order to be stored. In single-precision, $1.\,10011001100110011001100\backslash 1100\overline{1100}$ is approximated by either $\boxed{0 \,\big|\, E = -4 \,\big|\, 10011001100110011001100}$ or $\boxed{0 \,\big|\, E = -4 \,\big|\, 10011001100110011001101}$

Floating-point representation

- The gap between 1 and the next larger floating-point number is called the **machine precision** and is often denoted by ϵ.
- In single precision, the next floating-point number after 1 is $1+2^{-23}$. Thus, for single precision, we have $\epsilon = 2^{-23} \approx 1.2 \times 10^{-7}$.
- The default precision in MATLAB is **double precision**, where a word consists of 64 bits: 1 for the sign, 11 for the exponent, and 52 for the significand. Hence in double precision the next larger floating-point number after 1 is $1+2^{-52}$, so that the machine precision is $2^{-52} \approx 2.2 \times 10^{-16}$.
- Consider a toy system in which only significands of the form $1.b_1b_2$ can be represented and only exponents 0, 1 , and -1 can be stored. What are the numbers that can be represented in this system?



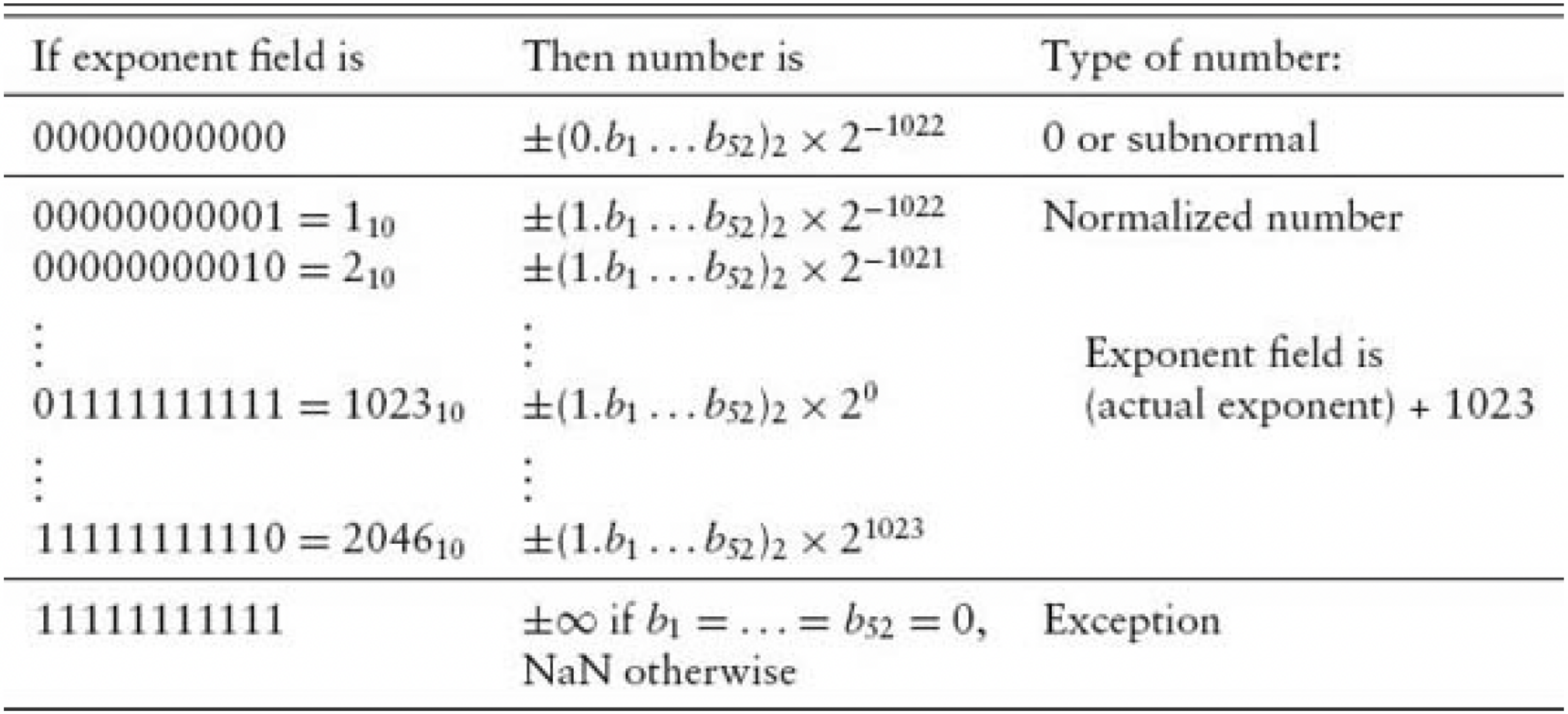
Floating-point representation

- the machine precision ϵ is 0.25.
- In general, the gaps between representable numbers become larger as we move away from the origin. This is acceptable since the relative gaps remains of reasonable size.
- However, the gap between 0 and the smallest positive number is much larger than the gap between the smallest and next smallest positive number.
- This is the case with single-and double-precision floating-point numbers as well.
- This gap can be filled in using **subnormal** numbers.

IEEE standard for double precision

- A special representation is needed for 0,
- and also for $\pm\infty$ (the result of dividing a nonzero number by 0),
- and also for NaN (Not a Number; e.g., $0/0$).
- These are done with special bits in the exponent field.
- Special bits in the exponent field are also used to signal subnormal numbers.

IEEE standard for double precision



If exponent field is	Then number is	Type of number:
00000000000	$\pm(0.b_1\dots b_{52})_2 \times 2^{-1022}$	0 or subnormal
$00000000001 = 1_{10}$	$\pm(1.b_1\dots b_{52})_2 \times 2^{-1022}$	Normalized number
$00000000010 = 2_{10}$	$\pm(1.b_1\dots b_{52})_2 \times 2^{-1021}$	
$\vdots$	$\vdots$	Exponent field is
$01111111111 = 1023_{10}$	$\pm(1.b_1\dots b_{52})_2 \times 2^{0}$	(actual exponent) + 1023
$\vdots$	$\vdots$	
$11111111110 = 2046_{10}$	$\pm(1.b_1\dots b_{52})_2 \times 2^{1023}$	
11111111111	$\pm\infty$ if $b_1 = \dots = b_{52} = 0$, NaN otherwise	Exception

IEEE 754 standard for double precision

- The smallest positive normalized floatingpoint number that can be stored is $1.0_2 \times 2^{-1022} \approx 2.2 \times 10^{-308}$, while the largest is $1.1\ldots1_2 \times 2^{1023} \approx 1.8 \times 10^{308}$.
- The exponent field for normalized floatingpoint numbers represents the actual exponent plus 1023. We can represent exponents between -1022 and $+1023$.
- The two special exponent field bit patterns are all 0 s and all 1 s .
- An exponent field consisting of all 0s signals either 0 or a subnormal number.
- The smallest positive subnormal number that can be represented is $2^{-52} \times 2^{-1022} = 2^{-1074}$.
- The number 0 is represented by an exponent field consisting of all 0s and a significand field of all 0s.
- An exponent field consisting of all 1s signals an exception. If all bits in the significand are 0 , then it is $\pm\infty$. Otherwise it represents NaN.

Rounding

If x is a real number that cannot be stored exactly, then it is replaced by a nearby floating-point number according to one of the following rules:

1. **Round down**: $\text{round}(x)$ is the largest floating-point number that is less than or equal to x.
2. **Round up**: $\text{round}(x)$ is the smallest floating-point number that is greater than or equal to x.
3. **Round towards 0**: $\text{round}(x)$ is either the round down or round up of x, whichever lies between 0 and x.
4. **Round to nearest**: $\text{round}(x)$ is either the round down or round up of x, whichever is closer. In case of a tie (x falls in the middle), it is the one whose least significant (rightmost) bit is 0 (ties to even).

The default is round to the nearest.

Rounding

Exercise 1.4. Using double precision, find the floating-point representation of $0.1 = 1.1001\overline{1100}_2 \times 2^{-4}$ under the four rounding modes.

- The absolute rounding error associated with a number x is defined as $|\text{round}(x) - x|$.
- If $x = \pm m \times 2^E$ (excluding subnormal numbers), then the absolute rounding error is always $< \epsilon \times 2^E$ for any rounding mode, and $\leqslant \frac{\epsilon}{2} \times 2^E$ for round to nearest, where ϵ is the machine precision.
- The relative rounding error associated with a number x is defined as $|\text{round}(x) - x| / |x|$, which is always $< \epsilon$ for any rounding mode, and $\leqslant \frac{\epsilon}{2}$ for round to nearest. In other words, we can write

  $$\text{round}(x) = x(1+\delta), \qquad \text{where } |\delta| < \epsilon \qquad (\text{or } \leq \tfrac{\epsilon}{2} \text{ for round to nearest}).$$

Rounding

The IEEE standard requires that the result of an operation (addition, subtraction, multiplication, or division) on two floating-point numbers must be the **correctly rounded** value of the exact result.

It means that if a and b are floatingpoint numbers and $\oplus$, $\ominus$, $\otimes$, and $\oslash$ represent floating-point addition, subtraction, multiplication, and division, then we will have

$$\begin{aligned}
a \oplus b &= \mathrm{round}(a+b) = (a+b)(1+\delta_1),\\
a \ominus b &= \mathrm{round}(a-b) = (a-b)(1+\delta_2),\\
a \otimes b &= \mathrm{round}(ab) = (ab)(1+\delta_3),\\
a \oslash b &= \mathrm{round}(a/b) = (a/b)(1+\delta_4),
\end{aligned}$$

where $|\delta_i| < \epsilon$ (or $\leq \epsilon/2$ for round to nearest), $i = 1, \ldots, 4$.

Exceptions

- **Overflow** occurs when the true result of an operation is greater than the largest floating-point number $(1.1\ldots1_2 \times 2^{1023} \approx 1.8 \times 10^{308}$ for double precision).
    - Using round up or round to nearest, the result is set to ∞;
    - using round down or round towards 0 , it is set to the largest floating-point number.
- **Underflow** occurs when the true result is less than the smallest floating-point number. The result is stored as a subnormal number if it is in the range of the subnormal numbers, and otherwise it is set to 0.
- Other operations that produce ∞: for any $x > 0$ or $x = \infty$, $\frac{x}{0}, x + \infty, x \times \infty$ all produce ∞. Similar rules apply to $-\infty$.
- Some operations to produce NaN (incomplete list):

$$\infty - \infty, \quad -\infty + \infty, \quad 0 \times \infty, \quad 0 \div 0, \quad \infty \div \infty, \quad \sqrt{-1}$$

2 Conditioning of problems

Types of errors

Types of errors in scientific computing:

1. physical model $\longrightarrow$ mathematical model (simplifications etc.)
2. mathematical model $\longrightarrow$ numerical model (discretization, truncation etc.)
3. numerical model $\longrightarrow$ numerical algorithm, implemented on a computer (rounding errors etc.)
4. errors in input data (measurement, inverse problems)

Given the true value y and the computed value $\hat{y}$,

- $|\hat{y} - y|$ is called the **absolute error**
- $\frac{|\hat{y} - y|}{|y|}$ is called the **relative error**

Conditioning of problems

The **conditioning** of a problem measures how sensitive the output is to small changes in the input.

Let $f\colon \mathbb{R} \to \mathbb{R}$. Consider the problem: given $x \in \mathbb{R}$, find $y = f(x)$.

If $\hat{x} \neq x$ (but close to x), and $\hat{y} = f(\hat{x})$. How close is $\hat{y}$ to y?

If

$$|\hat{y} - y| \approx C(x)|\hat{x} - x|,$$

then we might call $C(x)$ the **absolute condition number** of the function f at x.

If

$$\left|\frac{\hat{y} - y}{y}\right| \approx \kappa(x)\left|\frac{\hat{x} - x}{x}\right|,$$

then $\kappa(x)$ might be called the **relative condition number** of f at x.

Conditioning of problems

If $f'(x)$ exists, we can take

$$C(x) = |f'(x)|, \quad \kappa(x) = \left|\frac{xf'(x)}{f(x)}\right|.$$

Example 2.1 Find $C(x)$ and $\kappa(x)$ for $f(x) = 2x$.

Example 2.2 Find $C(x)$ and $\kappa(x)$ for $f(x) = \sqrt{x}$.

Example 2.3 Find the $C(x)$ and $\kappa(x)$ for solving the linear system $Ax = b$, assuming A is nonsingular and fixed.

Stability of algorithms

An algorithm that achieves the level of accuracy defined by the conditioning of the problem is called **stable**, while one that gets unnecessarily inaccurate results called **unstable**.

Example 2.4 (Computing sums) If x and y are two real numbers and they are rounded to floating-point numbers and their sum is computed on a machine with machine precision ϵ.

$$\mathrm{fl}(x+y) \equiv \mathrm{round}(x) \oplus \mathrm{round}(y) = (x(1+\delta_1) + y(1+\delta_2))(1+\delta_3), \quad |\delta_i| \leq \epsilon.$$

where $\mathrm{fl}(\cdot)$ denotes the floating-point result.

Stability of algorithms

Forward error analysis. How much does the computed value differ from the exact solution?

- absolute error: $|\mathrm{fl}(x+y)-(x+y)| \leq (|x|+|y|)(2\epsilon+\epsilon^2)$: small
- relative error: $\left|\frac{\mathrm{fl}(x+y)-(x+y)}{x+y}\right| \leq \frac{(|x|+|y|)(2\epsilon+\epsilon^2)}{|x+y|}$: can be large if $y \approx -x$, even if $\delta_3 = 0$, so the problem is ill-conditioned in this case.

Backward error analysis. Here one tries to show that the computed value is the exact solution to a nearby problem.

$$\mathrm{fl}(x+y) = x(1+\delta_1)(1+\delta_3) + y(1+\delta_2)(1+\delta_3)$$

the computed value is the exact sum of two numbers that differ from x and y by relative amounts no greater than $2\epsilon+\epsilon^2$.

Stability of algorithms

Example 2.5 Compute $\exp(x)$ using the Taylor series expansion

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots.$$

```
function newsum = my_exp(x)
oldsum = 0;
newsum = 1;
term = 1;
n = 0;
while newsum ~= oldsum % Iterate until next term is negligible
    n = n + 1;
    term = term * x/n; % x^n/n! = (x^{n-1}/(n-1)!) * x/n
    oldsum = newsum;
    newsum = newsum + term;
end
end
```

high relative error for large negative values of x

Stability of algorithms

The problem itself is well-conditioned (check).

But the algorithm is unstable.

Exercise 2.1. Modify the code to make it stable for all x.

Numerical differentiation. Forward difference

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(\xi), \quad \xi \in [x, x+b]$$
$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{h}{2}f''(\xi).$$

The term $-\frac{h}{2}f''(\xi) = O(h)$ is referred to as the **truncation error** or discretization error.

Stability of algorithms

But the first term may also contain error in numerical computation.

$$\frac{f(x+h)(1+\delta_1)-f(x)(1+\delta_2)}{h}=\frac{f(x+h)-f(x)}{h}+\frac{\delta_1 f(x+h)-\delta_2 f(x)}{h}, \qquad |\delta_i|<\epsilon$$

$$\left|\frac{\delta_1 f(x+h)-\delta_2 f(x)}{h}\right| \lesssim \frac{2\epsilon|f(x)|}{h}.$$

Example 2.6 Suppose $f(x)=\sin x$. What's the best accuracy by approximating $f'(\pi/4)$ using the forward difference?

Higher order finite difference (centeral difference, etc.)