Mathematics Methods for Computer Science

Instructor: Xubo Yang

SJTU-SE DALAB

Reference book

Reference book: Solomon, Justin. Numerical Algorithms. Published by AK Peters/CRC Press, 2015.

Two Roles

From discrete mathematics to continuous mathematics.
From exact solutions to numerical approximations.
Focus on numerical analysis and processing of real-valued data.

Two Roles:

- Client of numerical methods
- Designer of numerical methods

Applications:

- computer graphics,
- computer vision,
- big data,
- machine learning,
- ...

Typical Linear Algebra

$$\begin{aligned}
\|A\vec{x}-\vec{b}\|_2^2 &= (A\vec{x}-\vec{b})\cdot(A\vec{x}-\vec{b}) \\
&= (A\vec{x}-\vec{b})^\top(A\vec{x}-\vec{b}) \\
&= \left(\vec{x}^\top A^\top - \vec{b}^\top\right)(A\vec{x}-\vec{b}) \\
&= \vec{x}^\top A^\top A\vec{x} - \vec{x}^\top A^\top\vec{b} - \vec{b}^\top A\vec{x} + \vec{b}^\top\vec{b} \\
&= \|A\vec{x}\|_2^2 - 2\left(A^\top\vec{b}\right)\cdot\vec{x} + \|\vec{b}\|_2^2
\end{aligned}$$

Example: Matrix Vector Multiplication

function MULTIPLY$(A, \vec{x})$
  ▷ Returns $\vec{b} = A\vec{x}$, where
  ▷ $A \in \mathbb{R}^{m \times n}$ and $\vec{x} \in \mathbb{R}^n$
  $\vec{b} \leftarrow \vec{0}$
  for $i \leftarrow 1, 2, \ldots, m$
    for $j \leftarrow 1, 2, \ldots, n$
      $b_i \leftarrow b_i + a_{ij}x_j$
  return $\vec{b}$

(a)

function MULTIPLY$(A, \vec{x})$
  ▷ Returns $\vec{b} = A\vec{x}$, where
  ▷ $A \in \mathbb{R}^{m \times n}$ and $\vec{x} \in \mathbb{R}^n$
  $\vec{b} \leftarrow \vec{0}$
  for $j \leftarrow 1, 2, \ldots, n$
    for $i \leftarrow 1, 2, \ldots, m$
      $b_i \leftarrow b_i + a_{ij}x_j$
  return $\vec{b}$

(b)

Example: Matrix Vector Multiplication

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

(a)

1	2	3	4	5	6

(b) Row-major

1	3	5	2	4	6

(c) Column-major

Topics I

1. Numeric
   - Stability and error analysis
   - Floating-point representation
2. Linear algebra
   - Guassian elemination and LU
   - Column space and QR
   - Eigenproblems
   - Applications
3. Root-finding and optimization
   - Single variable
   - Multivariable
   - Constrained optimization
   - Iterative linear solvers; Conjugate gradients

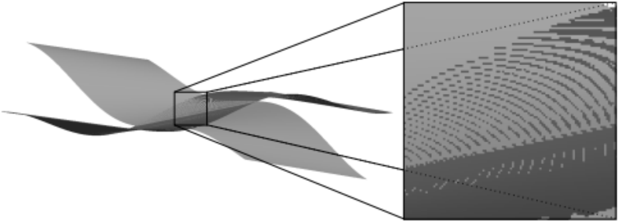
Topics II

4. Interpolation and quadrature
   - Interpolation
   - Approximating integrals (optional)
   - Approximating derivatives (optional)
5. Differential equations (optional)
   - ODEs: time-stepping, discretization
   - PDEs: Poisson equation, heat equation, waves
   - Techniques: Differencing, finite elements (time-permitting)

Lecture

Numerics And Error Analysis

Example: Z-fighting



Prototypical Example

```
double x = 1.0;
double y = x / 3.0;
if (x == y*3.0) cout << "They are equal!";
else cout << "They are NOT equal.";
```

Using Tolerances

```
double x = 1.0;
double y = x / 3.0;
if (fabs(x-y*3.0) <
    numeric_limits<double>::epsilon)
    cout << "They are equal!";
else cout << "They are NOT equal.";
```

A Crucial Point

Mathematically correct

$\neq$

Numerically sound

Rarely if ever should the operator $==$ and its equivalents be used on fractional values. Instead, some tolerance should be used to check if they are equal.

Counting in Binary: Integer

$$
\begin{aligned}
463 &= 256 + 128 + 64 + 8 + 4 + 2 + 1 \\
&= 2^8 + 2^7 + 2^6 + 2^3 + 2^2 + 2^1 + 2^0
\end{aligned}
$$

$$\downarrow$$

1	1	1	0	0	1	1	1	1
2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Counting in Binary: Fractional

$$463.25 = 256 + 128 + 64 + 8 + 4 + 2 + 1 + 1/4$$
$$= 2^8 + 2^7 + 2^6 + 2^3 + 2^2 + 2^1 + 2^0 + 2^{-2}$$

$$\downarrow$$

1	1	1	0	0	1	1	1	1	0	1
2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}

Familiar Problem

$$\frac{1}{3} = 0.0101010101\ldots_2$$

Finite number of bits

Fixed-Point Arithmetic

1	1	...	0	0	...	1	1
2^{ℓ}	$2^{\ell-1}$	...	2^{0}	2^{-1}	...	2^{-k+1}	2^{-k}

- Parameters: $k, \ell \in Z$
- $k + \ell + 1$ digits total
- Can reuse integer arithmetic (fast; GPU possibility):

$$a + b = (a \cdot 2^k + b \cdot 2^k) \cdot 2^{-k}$$

Two-Digit Example

$$0.1_2 \times 0.1_2 = 0.01_2 \cong 0.0_2$$

Multiplication and division easily change order of magnitude!

Demand of Scientific Applications

$$9.11 \times 10^{-31} \to 6.022 \times 10^{23}$$

Desired: graceful transition

Observations

- Compactness matters:

$$6.022 \times 10^{23} =$$

602,200,000,000,000,000,000,000

Observations

- Compactness matters:

$$6.022 \times 10^{23} =$$

602,200,000,000,000,000,000,000

- Some operations are unlikely:

$$6.022 \times 10^{23} + 9.11 \times 10^{-31}$$

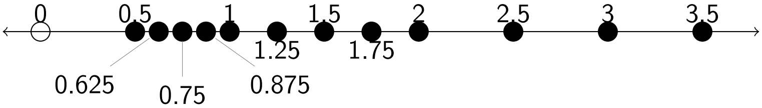
Scientific Notations

Store Significant digits

$$\underbrace{\pm}_{\text{sign}}\underbrace{(d_0 + d_1 \cdot b^{-1} + d_2 \cdot b^{-2} + \cdots + d_{p-1} \cdot b^{1-p}))}_{\text{significand}} \times \underbrace{b^e}_{\text{exponent}}$$

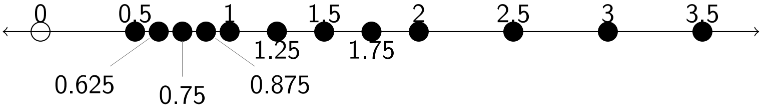
- Base: $b \in N$
- Precision: $p \in N$
- Range of exponents: $e \in [L, U]$

Properties of Floating Point



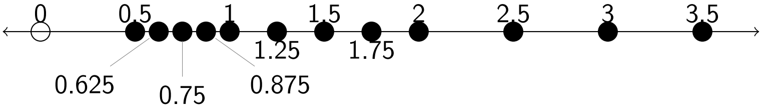
- Unevenly spaced
  - Machine precision ϵ_m: smallest ϵ_m with $1 + \epsilon_m \ncong 1$

Properties of Floating Point



- Unevenly spaced
  - Machine precision ϵ_m: smallest ϵ_m with $1 + \epsilon_m \not\approx 1$
- Needs rounding rule (e.g. "round to nearest, ties to even")

Properties of Floating Point



- Unevenly spaced
  - Machine precision ϵ_m: smallest ϵ_m with $1 + \epsilon_m \not\approx 1$
- Needs rounding rule (e.g. "round to nearest, ties to even")
- Can remove leading 1

Infinite Precision

$$Q = \{a/b : a, b \in Z\}$$

- Simple rules: $a/b + c/d = (ad + cb)/bd$
- Redundant: $1/2 = 2/4$
- Blowup:

$$\frac{1}{100} + \frac{1}{101} + \frac{1}{102} + \frac{1}{103} + \frac{1}{104} + \frac{1}{105} = \frac{188463347}{3218688200}$$

- Restricted operations: $2 \mapsto \sqrt{2}$

Bracketing

Store range $a \pm \epsilon$

- Keeps track of certainty and rounding decisions
- Easy bounds:

$$(x \pm \epsilon_1) + (y \pm \epsilon_2) = (x + y) \pm (\epsilon_1 + \epsilon_2 + error(x + y))$$

- Implementation via operator overloading

Sources of Error

- Rounding (or truncation) error (e.g. PI)
- Discretization error (e.g. derivative: divided differences)
- Modeling error (e.g. butterfly for weather, g)
- Input error (e.g. approximated parameters, typos)

Example

What sources of error might affect planets simulation?

Absolute vs. Relative Error

Absolute Error

The difference between the approximate value and the underlying true value.

Absolute vs. Relative Error

Absolute Error

The difference between the approximate value and the underlying true value.

Relative Error

Absolute error divided by the true value.

Absolute vs. Relative Error

Absolute Error

The difference between the approximate value and the underlying true value.

Relative Error

Absolute error divided by the true value.

$$2\ cm \pm 0.02\ cm$$

$$2\ cm \pm 1\%$$

Example: Catastrophic cancellation

$d \equiv 1 - 0.99 = 0.01$
± 0.004
$d = 0.01 \pm 0.008$

Absolute error $= 0.008$

Relative error $=$?

Relative Error: Difficulty

Problem: Generally not computable

Relative Error: Difficulty

Problem: Generally not computable

Common fix: Be conservative

Computable Measures of Success

Root-finding problem

For $f : \mathbb{R} \to \mathbb{R}$, find x^* such that $f(x^*) = 0$

Actual output: x_{est} with $|f(x_{est})| \ll 1$

May not be able to evaluate $|x_{est} - x_0|$

Can compute $|f(x_{est}) - f(x_0)| \equiv f(x_{est})$ (a calculable proxy)

Forward Error

Forward Error

The difference between the approximated and actual solution.

Backward Error

Backward Error

The amount the problem statement would have to change to make the approximate solution exact

Backward Error

Backward Error

The amount the problem statement would have to change to make the approximate solution exact

Example 1: $\sqrt{x}$ (e.g. x=2)

Backward Error

Backward Error

The amount the problem statement would have to change to make the approximate solution exact

Example 1: $\sqrt{x}$ (e.g. x=2)
Example 2: $A\vec{x} = \vec{b}$

Conditioning

What if backward error is small but nonzero?
Does this condition necessarily imply small forward error?

Conditioning

What if backward error is small but nonzero?
Does this condition necessarily imply small forward error?

Well-conditioned (or insensitive):
Small backward error $\Longrightarrow$ small forward error

Conditioning

What if backward error is small but nonzero?
Does this condition necessarily imply small forward error?

Well-conditioned (or insensitive):
Small backward error $\Longrightarrow$ small forward error

Poorly conditioned (or sensitive/stiff):
Otherwise

Example: Root-finding: $ax = b \rightarrow x_0 \equiv b/a$
Hint: calculate forward and backward errors, check $|a| \ll 1, or |a| \gg 1$

Condition Number

Condition number

Ratio of forward to backward error

Condition Number

Condition number

Ratio of forward to backward error

Root-finding example: f(x) = 0

$$c = \frac{1}{|f'(x^*)|}$$

Common Cause of Bugs in Numerical Software

Beware of operations that transition between orders of magnitude, like division by small values and subtraction of similar quantities.

E.g. $AX = b$

Theme

Extremely careful implementation can be necessary.

Example: Vector Norms $\|\vec{x}\|_2$

```
double normSquared = 0;
for (int i = 0; i < n; i++)
normSquared += x[i]*x[i];
return sqrt(normSquared);
```

Overflow issue

Improved $\|\vec{x}\|_2$

```
double maxElement = epsilon;

for (int i = 0; i < n; i++)
maxElement = max(maxElement, fabs(x[i]));
for (int i = 0; i < n; i++) {
double scaled = x[i] / maxElement;
normSquared += scaled*scaled;
}
return sqrt(normSquared) * maxElement;
```

More Involved Example: Large Scale Summation $\Sigma_i x_i$

```
double sum = 0;
for (int i = 0; i < n; i++)
sum += x[i];
```

Simple Sum and Kahan Sum

function SIMPLE-SUM($\vec{x}$)
    $s \leftarrow 0$ ▷ Current total
    for $i \leftarrow 1, 2, \ldots, n : s \leftarrow s + x_i$
    return s

(a)

Simple Sum and Kahan Sum

function SIMPLE-SUM($\vec{x}$)
  $s \leftarrow 0$ ▷ Current total
  for $i \leftarrow 1, 2, \ldots, n : s \leftarrow s + x_i$
  return s

(a)

$$((a + b) - a) - b \stackrel{?}{=} 0$$

Store compensation value !

Simple Sum and Kahan Sum

function SIMPLE-SUM($\vec{x}$)
  $s \leftarrow 0$ ▷ Current total
  for $i \leftarrow 1, 2, \ldots, n : s \leftarrow s + x_i$
  return s

(a)

$$((a+b)-a)-b \stackrel{?}{=} 0$$

Store compensation value !

function KAHAN-SUM($\vec{x}$)
  $s, c \leftarrow 0$ ▷ Current total and compensation
  for $i \leftarrow 1, 2, \ldots, n$
    $v \leftarrow x_i + c$ ▷ Try to add x_i and compensation c to the sum
    $s_{\text{next}} \leftarrow s + v$ ▷ Compute the summation result of this iteration
    $c \leftarrow v - (s_{\text{next}} - s)$ ▷ Compute compensation using the Kahan error estimate
    $s \leftarrow s_{\text{next}}$ ▷ Update sum
  return s

(b)