Numerics and Error Analysis

CS 205A:
Mathematical Methods for Robotics, Vision, and Graphics

Justin Solomon

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.";
```

Take-Away

Mathematically correct

$\neq$

Numerically sound

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.";
```

Counting in Binary: Integer

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

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

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

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

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

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 Notation

Store *significant* digits

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

- Base: $\beta \in \mathbb{N}$
- Precision: $p \in \mathbb{N}$
- Range of exponents: $b \in [L, U]$

Properties of Floating Point

- Unevenly spaced
  - Machine precision ε_m: smallest ε_m with $1 + \varepsilon_m \not\cong 1$

Properties of Floating Point

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

Infinite Precision

$$\mathbb{Q} = \{{}^{a}\!/\!{}_{b} : a, b \in \mathbb{Z}\}$$

- Simple rules: ${}^{a}\!/\!{}_{b} + {}^{c}\!/\!{}_{d} = {}^{ad+cb}\!/\!{}_{bd}$
- Exact equality possible again

Infinite Precision

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

- Simple rules: ${}^a/_b + {}^c/_d = {}^{ad+cb}/_{bd}$
- Exact equality possible again
- Redundant: ${}^1/_2 = {}^2/_4$
- Restricted set of operations
  Have to decide ahead of time!

Bracketing

Store range $a \pm \varepsilon$

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

$$(x \pm \varepsilon_1) + (y \pm \varepsilon_2) = (x + y) \pm (\varepsilon_1 + \varepsilon_2 + \mathrm{error}(x + y))$$

- Implementation via operator overloading

Sources of Error

- Truncation
- Discretization
- Modeling
- Empirical constants
- User input

Example

What sources of error might affect a financial 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

$$2 \text{ in} \pm 0.02 \text{ in}$$

$$2 \text{ in} \pm 1\%$$

Relative Error: Difficulty

Problem: True value unknown

Relative Error: Difficulty

Problem: True value unknown

Common fix: Be conservative

Indirect 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 $|f(x_{est})|\ll 1$

Backward Error

Backward Error

The amount a problem statement would have to change to realize a given approximation of its solution

Backward Error

Backward Error

The amount a problem statement would have to change to realize a given approximation of its solution

Example 1: $\sqrt{x}$

Backward Error

Backward Error

The amount a problem statement would have to change to realize a given approximation of its solution

Example 1: $\sqrt{x}$

Example 2: $A\vec{x} = \vec{b}$

Conditioning

Well-conditioned:

Small backward error $\implies$ small forward error

Poorly conditioned:

Otherwise

Example: Root-finding

Condition Number

Condition number

Ratio of forward to backward error

Condition Number

Condition number

Ratio of forward to backward error

Root-finding example:

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

Theme

Extremely careful implementation can be necessary.

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

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

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: $\sum_i x_i$

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

Motivation for Kahan Algorithm

$$((a+b)-a)-b \not\cong 0$$

Store *compensation* value!

Details in course notes

Next