Numerics and Error Analysis

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

Doug James (and Justin Solomon)

A Puzzle

What is the value of the following expression?

```
abs(3.*(4./3.-1.)-1.)
```

A Puzzle

What is the value of the following expression?

```
abs(3.*(4./3.-1.)-1.)
```

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

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

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

$$\downarrow$$

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

$$
\begin{aligned}
463.25 &= 256 + 128 + 64 + 8 + 4 + 2 + 1 + {}^1\!/_4 \\
&= 2^8 + 2^7 + 2^6 + 2^3 + 2^2 + 2^1 + 2^0 + 2^{-2}
\end{aligned}
$$

$$\downarrow$$

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

- Parameters: $k, \ell \in \mathbb{Z}$
- $k + \ell + 1$ 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:

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

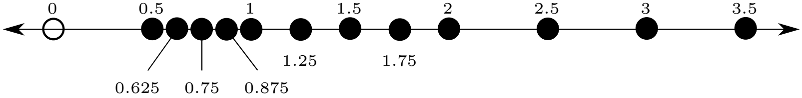
Scientific Notation

Store *significant* digits

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

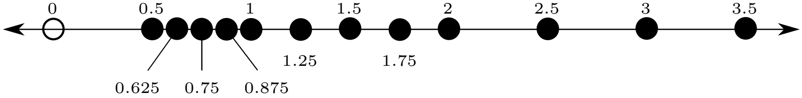
- Base: $b \in \mathbb{N}$
- Precision: $p \in \mathbb{N}$
- Range of exponents: $e \in [L, U]$

Properties of Floating Point



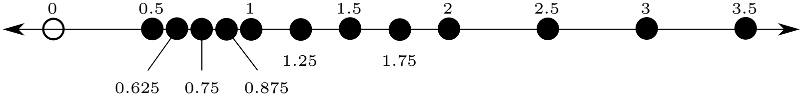
- Unevenly spaced
  - Machine precision ε_m: smallest ε_m with $1 + \varepsilon_m \not\approx 1$

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: ${}^{a}\!/_{b} + {}^{c}\!/_{d} = {}^{(ad+cb)}\!/_{bd}$
- 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 $a \pm \varepsilon$

- Special case of "Interval Arithmetic"
- 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

- Rounding
- Discretization
- Modeling
- Input

Example

What sources of error might affect a financial simulation?

Example

Catastrophic cancellation

Consider the function

$$f(x) = \frac{e^x - 1}{x} - 1$$

near $x = 0$.

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

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: $\sqrt{x}$

Backward Error

Backward Error

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

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

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

Store *compensation* value!

Details in textbook

Next