Numerics and Error Analysis


Doug James
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 ≠ Numerically sound
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

\[463 = 256 + 128 + 64 + 8 + 4 + 2 + 1\]
\[= 2^8 + 2^7 + 2^6 + 2^3 + 2^2 + 2^1 + 2^0\]

\[\downarrow\]

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^n$</td>
<td>$2^8$</td>
<td>$2^7$</td>
<td>$2^6$</td>
<td>$2^5$</td>
<td>$2^4$</td>
<td>$2^3$</td>
<td>$2^2$</td>
<td>$2^1$</td>
</tr>
</tbody>
</table>

CS 205A: Mathematical Methods
Numerics and Error Analysis 5 / 1
Counting in Binary: Fractional

\[ 463.25 = 256 + 128 + 64 + 8 + 4 + 2 + 1 + \frac{1}{4} \]
\[ = 2^8 + 2^7 + 2^6 + 2^3 + 2^2 + 2^1 + 2^0 + 2^{-2} \]

\[
\begin{array}{cccccccccccc}
1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & . & 0 & 1 \\
2^8 & 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 & 2^{-1} & 2^{-2}
\end{array}
\]
Familiar Problem

\[
\frac{1}{3} = 0.010101010101\ldots_2
\]

*Finite* number of bits
Fixed-Point Arithmetic

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>...</td>
<td>0.</td>
<td>0</td>
</tr>
<tr>
<td>$2^\ell$</td>
<td>$2^{\ell-1}$</td>
<td>...</td>
<td>$2^0$</td>
<td>$2^{-1}$</td>
</tr>
</tbody>
</table>

- 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

$$0.12 \times 0.12 = 0.012 \cong 0.02$$

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

\[
\pm \left( d_0 + d_1 \cdot b^{-1} + d_2 \cdot b^{-2} + \cdots + d_{p-1} \cdot b^{1-p} \right) \times b^e
\]

- **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 $\varepsilon_m$: smallest $\varepsilon_m$ with $1 + \varepsilon_m \neq 1$
Properties of Floating Point

- Unevenly spaced
  - Machine precision $\varepsilon_m$: smallest $\varepsilon_m$ with $1 + \varepsilon_m \neq 1$

- Needs rounding rule
  (e.g. “round to nearest, ties to even”)
Properties of Floating Point

- Unevenly spaced
  - Machine precision $\varepsilon_m$: smallest $\varepsilon_m$ with $1 + \varepsilon_m \neq 1$

- Needs rounding rule
  (e.g. “round to nearest, ties to even”)

- Can remove leading 1
Infinite Precision

\[ \mathbb{Q} = \{ \frac{a}{b} : a, b \in \mathbb{Z} \} \]

- Simple rules: \( \frac{a}{b} + \frac{c}{d} = \frac{ad+cb}{bd} \)
- Redundant: \( \frac{1}{2} = \frac{2}{4} \)
- 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 $a \pm \varepsilon$

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

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

- Implementation via operator overloading
Sources of Error

- Rounding
- Discretization
- Modeling
- Input
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

\[
\begin{align*}
2 \text{ in } &\pm 0.02 \text{ in} \\
2 \text{ in } &\pm 1\% 
\end{align*}
\]
Problem: Generally not computable
Relative Error: Difficulty

**Problem:** Generally not computable

**Common fix:** Be conservative
Root-finding problem

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

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

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

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

Example 1: $\sqrt{x}$
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 $\Rightarrow$ small forward error

Poorly conditioned:
Otherwise

Example: Root-finding
Condition number

Ratio of forward to backward error
Condition Number

Condition number

Ratio of forward to backward error

Root-finding example:

\[
\frac{1}{|f'(x^*)|}
\]
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