Brief Intro to Numerical Analysis

Szymon Rusinkiewicz
COS 302, Fall 2020
Numerical Analysis

• Algorithms for solving numerical problems
  – Calculus, algebra, data analysis, etc.
  – Used even if answer is not simple/elegant: “math in the real world”

• Analyze/design algorithms based on:
  – Running time, memory usage (both asymptotic and constant factors)
  – Applicability, stability, and accuracy
Why Is This Hard / Interesting?

- Problems might not have an ideal solution (independent of algorithm)
- Algorithms might give wrong answer (even with perfect real numbers)
  - Iterative, randomized, approximate
- “Numbers” in computers ≠ numbers in math
  - Limited precision and range
- Tradeoffs in accuracy, stability, and running time
Catalog of Errors

• **Inherent error** in data or model
  – “Garbage in, garbage out”
  – Problem is ill-posed or ill-conditioned

• **Approximation errors** in algorithm
  – Discretization error – e.g., too-big steps for derivative
  – Truncation error – e.g., too few terms of Taylor series
  – Convergence error – stopping iteration too early
  – Statistical error – too few random samples

• **Roundoff error** due to floating-point “numbers”
Catalog of Errors

• **Inherent error** in data or model
  – “Garbage in, garbage out”
  – Problem is ill-posed or ill-conditioned

• **Approximation errors** in algorithm
  – Discretization error – e.g., too-big steps for derivative
  – Truncation error – e.g., too few terms of Taylor series
  – Convergence error – stopping iteration too early
  – Statistical error – too few random samples

• **Roundoff error** due to floating-point “numbers”
Well-Posedness and Sensitivity

• Problem is **well-posed** if solution
  – exists
  – is unique
  – depends continuously on problem data

Otherwise, problem is **ill-posed**

• Solution may still be sensitive to input data
  – **Ill-conditioned**: relative change in solution much larger than that in input data
Sensitivity & Conditioning

- Some problems propagate error in bad ways
  - e.g., $y = \tan(x)$ sensitive to small changes in $x$ near $\pi/2$
- Small error in input $\rightarrow$ huge error in solution: ill-conditioned
- Well-conditioned problems may have ill-conditioned inverses, and vice versa
  - e.g., $y = \arctan(x)$
Catalog of Errors

• Inherent error in data or model
  – “Garbage in, garbage out”
  – Problem is ill-posed or ill-conditioned

• Approximation errors in algorithm
  – Discretization error – e.g., too-big steps for derivative
  – Truncation error – e.g., too few terms of Taylor series
  – Convergence error – stopping iteration too early
  – Statistical error – too few random samples

• Roundoff error due to floating-point “numbers”
Catalog of Errors

• Inherent error in data or model
  – “Garbage in, garbage out”
  – Problem is ill-posed or ill-conditioned

• Approximation errors in algorithm
  – Discretization error – e.g., too-big steps for derivative
  – Truncation error – e.g., too few terms of Taylor series
  – Convergence error – stopping iteration too early
  – Statistical error – too few random samples

• Roundoff error due to floating-point “numbers”
Numbers in Computers

• “Integers”
  – Mostly sane, except for limited range

• Floating point
  – Most common approximation to real numbers (alternatives: fixed point, rational)
  – Much larger range
    (e.g. $-2^{31} ... 2^{31}$ for 32-bit integers, vs. $-2^{127} ... 2^{127}$ for 32-bit floating point)
  – Lower precision (e.g. 7 digits vs. 9)
  – *Relative* precision: actual accuracy depends on size
Floating Point Numbers

- Like scientific notation: e.g., $c$ is $2.99792458 \times 10^8$ m/s

- This has the form $(\text{multiplier}) \times (\text{base})^{\text{power}}$

- In the computer,
  - Multiplier is called mantissa
  - Base is almost always 2
  - Power is called exponent
IEEE Floating Point Representation (ISO/IEEE 754 Standard)

• Using 32 bits
  – Type `float` in C / Java,
  
  `np.single` or `np.float32` in NumPy
  – 1 bit: `sign`
    (0 ⇒ positive, 1 ⇒ negative)
  – 8 bits: `exponent` + 127
  – 23 bits: `binary fraction` of the form
    `1.bbbbbbbbbbbbbbbbbbbbbbb`

• Using 64 bits
  – Type `double` in C / Java,
    `float` in plain Python,
    `np.double` or `np.float64` in NumPy
  – 1 bit: `sign`
    (0 ⇒ positive, 1 ⇒ negative)
  – 11 bits: `exponent` + 1023
  – 52 bits: `binary fraction` of the form
    `1.bbbbbbbbbbbbbbbbbbbbbbbbbbb
    bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb`
Floating Point Example

- **Sign (1 bit):**
  - \(1\) ⇒ negative

- **Exponent (8 bits):**
  - \(10000011_B = 131\)
  - \(131 - 127 = 4\)

- **Mantissa (23 bits):**
  - \(1.10110110000000000000000_B\)
  - \(1 + (1*2^{-1}) + (0*2^{-2}) + (1*2^{-3}) + (1*2^{-4}) + (0*2^{-5}) + (1*2^{-6}) + (1*2^{-7}) = 1.7109375\)

- **Number:**
  - \(-1.7109375 \times 2^4 = -27.375\)
Floating Point Consequences

- "Machine epsilon": smallest positive number you can add to 1.0 and get something other than 1.0

  - For 32-bit: $\varepsilon \approx 10^{-7}$
    - No such number as $1.000000001$
    - Rule of thumb: “almost 7 digits of precision”

  - For double: $\varepsilon \approx 2 \times 10^{-16}$
    - Rule of thumb: “not quite 16 digits of precision”

- These are all relative numbers
Floating Point Consequences, cont.

• Just as decimal number system can represent only certain rational numbers with finite digit count…
  – Example: 1/3 cannot be represented

• Binary number system can represent only certain rational numbers with finite digit count
  – Example: 1/5 cannot be represented

• Beware of roundoff error
  – Error resulting from inexact representation
  – Can accumulate

<table>
<thead>
<tr>
<th>Decimal Approx</th>
<th>Rational Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>.3</td>
<td>3/10</td>
</tr>
<tr>
<td>.33</td>
<td>33/100</td>
</tr>
<tr>
<td>.333</td>
<td>333/1000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binary Approx</th>
<th>Rational Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0/2</td>
</tr>
<tr>
<td>0.01</td>
<td>1/4</td>
</tr>
<tr>
<td>0.010</td>
<td>2/8</td>
</tr>
<tr>
<td>0.0011</td>
<td>3/16</td>
</tr>
<tr>
<td>0.00110</td>
<td>6/32</td>
</tr>
<tr>
<td>0.001101</td>
<td>13/64</td>
</tr>
<tr>
<td>0.0011010</td>
<td>26/128</td>
</tr>
<tr>
<td>0.00110011</td>
<td>51/256</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
So What?

• Simple example: add $\frac{1}{10}$ to itself 10 times

```python
sum = 0.0
for i in range(10):
    sum += 0.1
if sum == 1.0:
    print("All is well")
else:
    print("Yikes!")
```
So What?

• Simple example: add $\frac{1}{10}$ to itself 10 times

```python
sum = 0.0
for i in range(10):
    sum += 0.1
if sum == 1.0:
    print("All is well")
else:
    print("Yikes!")
```

Yikes!
• Result: \( \frac{1}{10} + \frac{1}{10} + \ldots \neq 1 \)

• Reason: 0.1 can’t be represented exactly in binary floating point
  – Like \( \frac{1}{3} \) in decimal

• **Rule of thumb**: comparing floating point numbers for equality is “always” wrong
More Subtle Problem

• Using quadratic formula

\[ x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \]

to solve \( x^2 - 9999x + 1 = 0 \)

  – Only 4 digits: single precision should be OK, right?

• Correct answers: 0.0001… and 9998.999…

• Actual answers in single precision: 0 and 9999

  – First answer is 100% off!

  – Total cancellation in numerator because \( b^2 \gg -4ac \)
Catalog of Errors

- **Inherent error** in data or model
  - “Garbage in, garbage out”
  - Problem is ill-posed or ill-conditioned

- **Approximation errors** in algorithm
  - Discretization error – e.g., too-big steps for derivative
  - Truncation error – e.g., too few terms of Taylor series
  - Convergence error – stopping iteration too early
  - Statistical error – too few random samples

- **Roundoff error** due to floating-point “numbers”
Error Tradeoff Example – Computing Derivative

\[ f'(x) \approx \frac{f(x + h) - f(x)}{h} \]

- total error
- discretization error
- roundoff error