TDT4127 Programming and Numerics
Week 36

Floating point numbers
Basic concepts in numerical mathematics
Learning goals

• Goals
  – Floating point numbers
  – Refresh mathematical concepts
  – General knowledge of numerics

• Curriculum
  – Exercise 1
  – Exercise 2
Floating point numbers

• Continued from last week’s lecture
• Decimal numbers can be both infinitely *large* and *long*
  – For example, π is infinitely long
    • \( \pi = 3.14159265359… \)
  – We can still use it mathematically:
    • \( A = \pi r^2 \)
  – When computing, we use a *truncated* value with an uncertainty:
    • \( \pi = 3.14 \ (\pm \ 0.005) \)
  – We do this for other infinitely long numbers as well:
    • \( \frac{1}{3} = 0.3333 \ (\pm \ 0.0005) \)
• Our representation of decimal numbers must balance *magnitude* and decimal point *precision*.
Floating point numbers

– Floats are a tradeoff between size range and accuracy
– Based on scientific notation for numbers
  • Avogadro’s number: \(10^{23} \times 6.022140857\)
  • Electron rest mass: \(10^{-31} \times 9.109383561\)
  • Large range of numbers, here using only 12 digits (base 10 numbers).
    • Uncertainty lies in the last digit
– Floating point numbers use the same idea, but in base 2
  • \(a = (-1)^{sg} \times 2^{e-b} \times s\)
    – Sign: \(sg\) is 1 bit representing 0 or 1, allows negative/positive numbers
    – Exponent: \(e\) is a positive integer, adjusts size
    – Bias: \(b\) is a predetermined integer allowing for negative exponents
    – Significand: \(s\) is a number between 1 and 2 of the form
      \[
      s = 1.s_1s_2s_3s_4s_5s_6... = 1 + s_1 \times 2^{-1} + s_2 \times 2^{-2} + s_3 \times 2^{-3} + s_4 \times 2^{-4} + s_5 \times 2^{-5} + s_6 \times 2^{-6} + ...
      \]
    – This is like scientific notation in base 2, with uncertainty in the last digit.
    – More in Exercise 1, after which we will mostly not have to worry about them.
Operations with floating point numbers

• Addition/subtraction requires care due to roundoff error
  – When adding, the smaller number loses significance
  – Example in base 10: 12345.67 + 1.224567 with 7 digit precision:
    
    \[
    \begin{array}{c}
    12345.67 \\
    + 1.224567 \\
    = 12346.894567 \approx 12346.89
    \end{array}
    \]
  – Same effect as adding 1.22 since the last four digits are lost.
  – When adding many small numbers to a larger number, we lose precision unless it is done carefully.
    • Workarounds such as Kahan’s algorithm is an algorithm for doing so. Not curriculum.
Operations with floating point numbers

• Multiplication/division are safe
  – We add/subtract exponents and multiply/divide the significands.

• Checking for equality is very unsafe
  – If $a$ and $b$ are floats, $a = b$ if all their bits are the same.
  – Due to imprecision, numbers that should be equal after some computation, may not be equal.
  – Example: Are $d = (a + b) + c$ and $e = a + (b + c)$ equal?
    $a = 123456.7,\quad b = 123.4567,\quad c = 0.4567891$
    
    $d = 123580.2 + 0.4567891 = 123580.7$
    
    $e = 123456.7 + 123.9135 = 123580.6$

• This concludes the rest of last week’s lecture
The goal of numerics

- To solve «unsolvable» equations

\[
\log(\cos(x^2)) = \frac{e^{x^3}}{1 + \sqrt{x}}
\]

- Saves a lot of time and lets us *do more with maths*!

- Based on **algorithms**
  - Recipes expressed mathematically
  - **Implementation** done by programming

- Three main questions we will discuss for each topic:
  - **What** do the algorithms look like?
  - **When** do the algorithms work?
  - **How well** do they work?
What do the algorithms look like?

• Example: Bisection method
  – Simple algorithm (root finder) for finding zeroes of functions: \( f(x) = 0 \)
  – Root finders can also be used to solve equations:
    \[
    f(x) = g(x) \iff f(x) - g(x) = h(x) = 0
    \]
• Start with two points \( a \) and \( b \) such that \( f(a) < 0, f(b) > 0 \)
  • Then, there is a point \( z \) between \( a \) and \( b \) where \( f(z) = 0 \)
  • This point is also called a root of \( f \)
• Let \( c = (a+b)/2 \), check the value of \( f(c) \)
  – If \( f(c) < 0 \), swap \( a \) for \( c \) and repeat
  – If \( f(c) > 0 \), swap \( b \) for \( c \) and repeat
  – If \( f(c) = 0 \), we have a solution!
• Start again with new \( a \) and \( b \).
  – A single step like this is called an iteration.
  – Repeated iterations makes a smaller and smaller interval around \( z \).
Iteration 1

$f(a) > 0$, $f(b) < 0$, $f(c) < 0$

$\rightarrow$ Swap $b$ for $c$
Iteration 2
f(a) > 0, f(b) < 0, f(c) > 0

-> Swap a for c
Iteration 3

\[ f(a) > 0, \ f(b) < 0, \ f(c) < 0 \]

\[ \rightarrow \text{Swap b for c} \]
When do the algorithms work?

• Algorithms work based on requirements, and it is important to meet them
  – Otherwise, absurd results can occur

• Example: Bisection method
  – Correct initialization: Need to start with two point $a$ and $b$ such that $f(a)$ and $f(b)$ have different signs.
    • Otherwise, we don’t know if there is a zero in the interval
  – Properties of the function $f$
    • We require that $f$ is continuous
    • A continuous function does not make jumps
    • Otherwise, our intuition that there is a point $z$ between $a$ and $b$ where $f(z) = 0$ does not hold!
A discontinuous function: 
\[ f(x) = \begin{cases} 
-1, & x < 0.3 \\
1, & x \geq 0.3 
\end{cases} \]

- No zeroes, but the starting interval \( a = 0, b = 1 \) is still OK!
- If we run the algorithm, it will try to find a non-existant root. Absurd!
How well do the algorithms work?

- How fast is the *convergence*?
- Example: Bisection vs Newton’s method to find root of 
  \[ f(x) = (x-0.3)(x-3) \]
  
  - Newton’s method is taught in week 39

<table>
<thead>
<tr>
<th>Iteration no.</th>
<th>(a,b), Bisection</th>
<th>x, Newton</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(0,1)</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>(0,0.5)</td>
<td>-0.0769230</td>
</tr>
<tr>
<td>2</td>
<td>(0.25,0.5)</td>
<td>0.25886586</td>
</tr>
<tr>
<td>3</td>
<td>(0.25,0.375)</td>
<td>0.29939185</td>
</tr>
<tr>
<td>4</td>
<td>(0.25,0.3125)</td>
<td>0.29999986</td>
</tr>
<tr>
<td>5</td>
<td>(0.28125,0.3125)</td>
<td>0.30000000</td>
</tr>
</tbody>
</table>

- Newton’s method is extremely fast! 5 iterations to get 8 digits of accuracy.
The «user manual» for algorithms

• Several issues to keep in mind, all of which we will go through for every algorithm:
  – Convergence speed
  – Accuracy of solution
  – Error estimates
  – Conditions for use
## Timeline

<table>
<thead>
<tr>
<th>Week</th>
<th>Numerikk</th>
<th>Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>Introduction</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>Programming, floating points</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>Refresh maths, floating points</td>
<td>Bisection method</td>
</tr>
<tr>
<td>37</td>
<td>Numerical integration</td>
<td>Trapezoidal rule, Simpson’s rule</td>
</tr>
<tr>
<td>38</td>
<td>Numerical equation solvers in 1D</td>
<td>Newton’s method</td>
</tr>
<tr>
<td>39</td>
<td>Numerical equation solvers in 1D</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>Solving linear systems</td>
<td>Gaussian elimination</td>
</tr>
<tr>
<td>41</td>
<td>Solving linear systems</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>Numerical equation solvers in nD</td>
<td>Newton’s method for systems</td>
</tr>
<tr>
<td>43</td>
<td>Numerical solution of differential equations</td>
<td>Euler’s method, Heun’s method, Runge-Kutta methods</td>
</tr>
<tr>
<td>44</td>
<td>Numerical solution of differential equations</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>Numerical integration with adaptive Simpson’s rule</td>
<td>Adaptive Simpson’s rule</td>
</tr>
<tr>
<td>46</td>
<td>Repetition</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>Repetition</td>
<td></td>
</tr>
</tbody>
</table>
Summary

• Floating point numbers are used for real (decimal) numbers and are **inexact**
• Addition of small and large numbers can cause problems
• Do not make code that relies on checking whether two floats are equal
  – Integers, on the other hand, are okay!
• Numerics solve practical mathematical problems
  – Algorithms behave differently
    • Some are faster than others
    • Some put stricter requirements on the problem
  – We will learn algorithms for numerical integration, equation solving and differential equation solving.
Questions?