E7441: Scientific computing

Vlad Popovici, Ph.D.

RECETOX

Outline

There is nothing more practical than a good theory.
Kurt Lewin (1890–1947)

Outline

Bibliography:

- KONG Q., SIAUW T., BAYEN A. (2020). Python programming and numerical methods. Academic Press. ISBN: 9780128195499
- HEATH M.T. (2002). Scientific Computing. An introductory survey. McGraw-Hill, 2nd edition. ISBN: 0-07-239910-4 Good accompanying materials at https://heath.cs.illinois.edu/scicomp/notes/index.html, including slides and demos! Used as basis for the first part of the course.
- KEPNER J. (2009). Parallel Matlab for Multicore and Multinode Computers. SIAM Publishing. ISBN: 978-0-898716-73-3
- GENTLE J.E. (2005). Elements of Computational Statistics. Springer. ISBN:978-0387954899

Computing environments for the course:

- PYTHON 3, https://www.python.org - with NUMPY and SCIPY packages
- recommended: JUPYTERLAB for exercises
- suggestion: install PYTHON and related packages using a distribution like ANACONDA or MAMBA for easier integration of dependencies
- R, http://www.r-project.org - "environment for statistical computing and graphics"

WARNING: Some pieces of code shown during the course may not represent the optimal implementation in the given language. They are merely a device for demonstrating some principles.

Scientific computing

Wikipedia:

"Computational science (also scientific computing or scientific computation) is concerned with constructing mathematical models and quantitative analysis techniques and using computers to analyze and solve scientific problems."

Scientific computing

Wikipedia:

"Computational science (also scientific computing or scientific computation) is concerned with constructing mathematical models and quantitative analysis techniques and using computers to analyze and solve scientific problems."

Basically: find numerical solutions to mathematically-formulated problems.

(J. Hadamard) A problem is well posed if its solution

- exists
- is unique
- has a behavior that changes continuously with the initial conditions;

otherwise, it is ill posed.
Inverse problems are often ill posed.
Example: 3D to 2D projection.

- continuous domain $\rightarrow$ discrete domain
- well-posed but ill-conditioned problems: small errors in input lead to large variations in the solution
- improve conditioning by regularization

General computational approach

- continuous domain → discrete domain
- infinite → finite
- differential → algebraic
- nonlinear → (combination of) linear
- accept approximate solutions, but control for the error

Approximations

- Modeling approximations:
  - "model" = approximation of the nature
  - data - inexact measurements or previous results
- Implementation/computational approximations:
  - discretization of the continuous domain; truncation
  - rounding
- errors in input data
- errors propagated by the algorithm
- accuracy of the final result

Example: area of the Earth

- model: sphere
- $A = 4\pi r^2$
- $r = ?$
- $\pi = 3.14159\ldots$
- rounded arithmetic

Errors

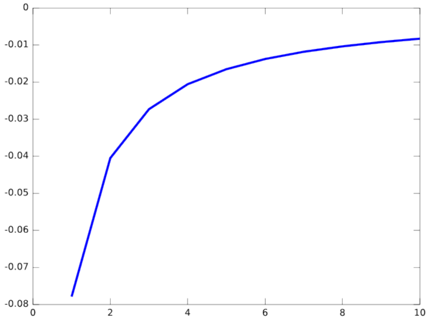
- Absolute error: approximate value ($\hat{x}$) - true value (x)
- Relative error:
$$\frac{\text{absolute error}}{\text{true value}}$$
- $\rightarrow$ approximate value $=$ (1 $+$ relative error) $\times$ (true value)
- if the relative error is $\sim 10^{-d}$, it means that $\hat{x}$ has about d exact digits: there exists $\tau = \pm(0.0\dots0n_{d+1}n_{d+2}\dots)$ such that $\hat{x} = x + \tau$
- true value is usually not known $\rightarrow$ use estimates or bounds on the error
- relative error can be taken relative to the approximate value

Example/exercise - Homework!

Stirling's approximation for factorials:

$$S_n = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \approx n!, \qquad n = 1, 2, \ldots$$

where $e = \exp(1)$.
Relative error $(S_n - n!)/n!$:



Errors: data and computational

- compute $f(x)$ for $f : \mathbb{R} \to \mathbb{R}$
  - $x \in \mathbb{R}$ is the true value
  - $f(x)$ true/desired result
  - $\hat{x}$ approximate input
  - $\hat{f}$ approximate result
- total error:

$$\begin{aligned}\hat{f}(\hat{x}) - f(x) &= (\hat{f}(\hat{x}) - f(\hat{x})) + (f(\hat{x}) - f(x)) \\ &= \text{computational error} + \text{propagated data error}\end{aligned}$$

- *the algorithm has no effect on propagated error*

Computational error

is sum of:

- truncation error = (true result) - (result of the algorithm using exact arithmetic)
  Example: considering only the first terms of an infinite Taylor series; stopping before convergence
- rounding error = (result of the algorithm using exact arithmetic) - (result of the algorithm using limited precision arithmetic)
  Example: $\pi \approx 3.14$ or $\pi \approx 3.141593$

Finite difference approximation

$$f'(x) = \lim_{h\to 0} \frac{f(x+h)-f(x)}{h} \approx \frac{f(x+h)-f(x)}{h}, \text{for some small } h > 0$$

- truncation error: $f'(x) - \frac{f(x+h)-f(x)}{h} \leq Mh/2$ where $|f''(t)| \leq M$ for t in a small neighborhood of x (HOMEWORK)
- rounding error: $2\epsilon/h$, for ϵ being the precision
- total error is minimized for $h \approx 2\sqrt{\epsilon/M}$

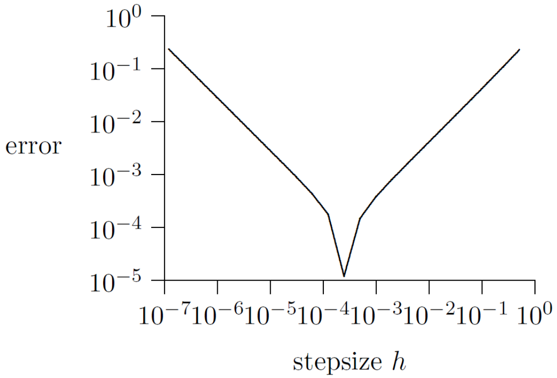


Figure: Total computational error as a tradeoff between truncation and rounding error (from *Heath - Scientific computing*)

Error analysis

For $y = f(x)$, for $f : \mathbb{R} \to \mathbb{R}$ an approximate $\hat{y}$ result is obtained.

- forward error: $\Delta y = \hat{y} - y$
- backward error: $\Delta x = \hat{x} - x$, for $f(\hat{x}) = \hat{y}$

$$\begin{tikzcd}
x \arrow[r, "f"] \arrow[d, <->, "\text{backward error}"'] \arrow[dr, "\hat{f}"] & y \arrow[d, <->, "\text{forward error}"] & = f(x) \\
\hat{x} \arrow[r, "f"] & \hat{y} & = \hat{f}(x) = f(\hat{x})
\end{tikzcd}$$

Compute $f(x) = e^x$ for $x = 1$. Use the first 4 terms from Taylor expansion:

$$\hat{f}(x) = 1 + x + \frac{x^2}{2} + \frac{x^3}{6}$$

- take "true" value: $f(x) = 2.716262$ and compute $\hat{f}(x) = 2.666667$, then
- forward error: $|\Delta y| = 0.051615$, or a relative f. error of about 2%
- backward error: $\hat{x} = \ln \hat{f}(x) = 0.989829 \Rightarrow |\Delta x| = 0.019171$, or a relative b. error of 2%
- these are two perspectives on assessing the accuracy

Exercise

Consider the general Taylor series with limit e:

$$\sum_{n=0}^{\infty} \frac{1}{n!} = e$$

How many terms are needed for an approximation of e to three decimal places?

Backward error analysis

- idea: approximate result is the exact solution of a modified problem
- how far from the original problem is the modified version?
- how much error in the input data would explain all the error in the result?
- an approximate solution is good if it is an exact solution for a nearby problem
- backward analysis is usually easier

Sensitivity and conditioning

- insensitive (well-conditioned) problem: relative changes in input data causes similar relative change in the result
- large changes in solution for small changes in input data indicate a sensitive (ill-conditioned) problem;
- condition number:

$$\text{cond} = \frac{\text{absolute relative change in solution}}{\text{absolute relative change in input}} = \frac{|\Delta y/y|}{|\Delta x/x|}$$

- if $\text{cond} >> 1$ the problem is sensitive

- condition number is a scale factor for the error:

$$\text{relative forward err} = \text{cond} \times \text{relative backward err}$$

- usually, only upper bounds of the cond. number can be estimated, $\text{cond} \leq C$, hence

$$\text{relative forward err} \leq C \times \text{relative backward err}$$

- $\hat{x} = x + \Delta x$
- forward error: $f(x + \Delta x) - f(x) \approx f'(x)\Delta x$, for small enough Δx
- relative forward error: $\approx \frac{f'(x)\Delta x}{f(x)}$
- $\Rightarrow$ cond $\approx \left|\frac{xf'(x)}{f(x)}\right|$

- $\hat{x} = x + \Delta x$
- forward error: $f(x + \Delta x) - f(x) \approx f'(x)\Delta x$, for small enough Δx
- relative forward error: $\approx \frac{f'(x)\Delta x}{f(x)}$
- $\Rightarrow$ cond $\approx \left|\frac{xf'(x)}{f(x)}\right|$

Example: tangent function is sensitive in neighborhood of $\pi/2$

- $\tan(1.57079) \approx 1.58058 \times 10^5$; $\tan(1.57078) \approx 6.12490 \times 10^4$
- for $x = 1.57079$, cond $\approx 2.48275 \times 10^5$

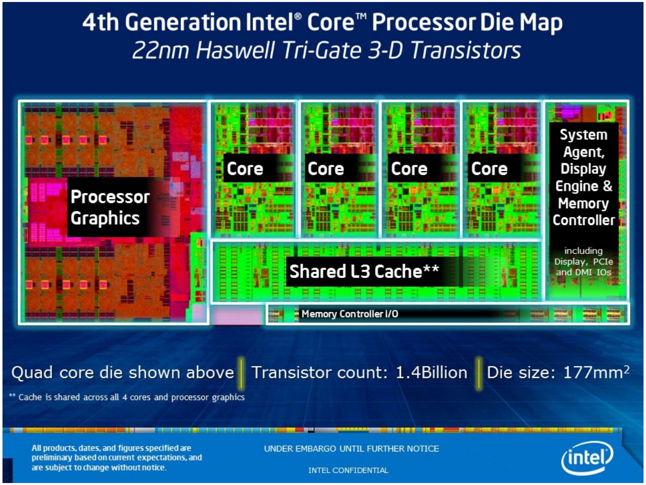
Stability

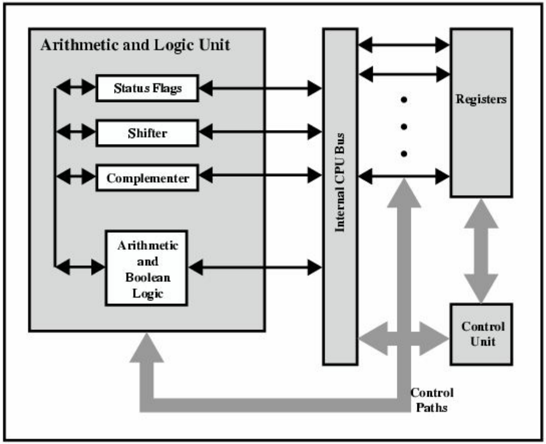
- an algorithm is stable if is relatively insensitive to perturbations during computation
- stability of algorithms is analogous to conditioning of problems
- backward analysis: an algorithm is stable if the result produced is the exact solution of a nearby problem
- stable algorithm: the effect of computational error is no worse than the effect of small error in input data

Accuracy

- accuracy: closeness of the result to the true solution of the problem
- depends on the conditioning of the problem AND on the stability of the algorithm
- stable algorithm + well-conditioned problem = accurate results

CPUs


4th Generation Intel® Core™ Processor Die Map
22nm Haswell Tri-Gate 3-D Transistors
Processor Graphics
Core
Core
Core
Core
System Agent, Display Engine & Memory Controller
including Display, PCIe and DMI IOs
Shared L3 Cache**
Memory Controller I/O
Quad core die shown above
Transistor count: 1.4Billion
Die size: 177mm²
** Cache is shared across all 4 cores and processor graphics
All products, dates, and figures specified are preliminary based on current expectations, and are subject to change without notice.
UNDER EMBARGO UNTIL FURTHER NOTICE
INTEL CONFIDENTIAL
intel


Arithmetic and Logic Unit
Status Flags
Shifter
Complementer
Arithmetic and Boolean Logic
Internal CPU Bus
Registers
Control Unit
Control Paths

Number representation

- internally, all data are represented in binary format (each digit can be either 0 or 1, e.g. 1011001...)
- bit, nybble, byte
- word $\rightarrow$ specific to architecture: 1, 2, 4, or 8 bytes
- integers:
  - unsigned (≥ 0): on n bits: $0, \ldots, 2^n - 1$. The stored representation (for 1 byte) is $b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$ for a value $x = \sum_{i=0}^{7} b_i 2^i$.
  - signed: 1 bit for sign, rest for the absolute value; $-2^{n-1}, \ldots, 0, \ldots, 2^{n-1} - 1$. The stored representation (for 1 byte) is $b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$ for a value $x = b_7(-2^7) + \sum_{i=0}^{6} b_i 2^i$.

Floating-point numbers

- like in scientific notation: mantissa $\times$ radix$^{\text{exponent}}$, e.g. 2.35×10^3
- formally
$$x = \pm\left(b_0 + \frac{b_1}{\beta} + \frac{b_2}{\beta^2} + \cdots + \frac{b_{p-1}}{\beta^{p-1}}\right) \times \beta^E$$
  where
  β is the radix (or base)
  p is the precision
  $L \leq E \leq U$ are the limits of the exponent
  $0 \leq b_k \leq \beta$
- mantissa: $m = b_0 b_1 \dots b_{p-1}$; fraction: $b_1 b_2 \dots b_{p-1}$
- the sign, mantissa and exponent are stored in fixed-sized fields (the radix is implicit for a given system, $\beta = 2$ usually)

Normalization:

- $b_0 \neq 0$ for all $x \neq 0$
- mantissa m satisfies $1 \leq m < \beta$
- ensures unique representation, optimal use of available bits

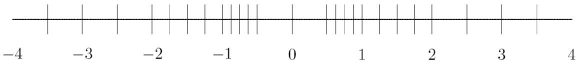
Internal representation (on 64 bits - "double precision", binary representation):

$$x = \boxed{\text{sign} \mid \text{exponent} \mid \text{fraction}} = \boxed{b_{63}}\boxed{b_{62} \ldots b_{52}}\boxed{b_{51} \ldots b_0}$$

Properties:

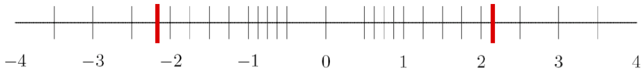
- only a finite number of discrete values can be represented
- total number of floating point numbers representable in normalized format is
$$2(\beta - 1)\beta^{p-1}(U - L + 1) + 1$$
- undeflow level (smallest number): $UFL = \beta^L$
- overflow level (largest number): $OFL = \beta^{U+1}(1 - \beta^{-p})$
- not all real numbers can be represented exactly:
  - *machine numbers*
  - rounding → rounding error

Example: let $\beta = 2, p = 3, L = -1, U = 1$, there are 25 distinct numbers that can be represented:



- $UFL = 0.5_{10}$; $OFL = 3.5_{10}$
- note the non-uniform coverage
- $\forall x \in \mathbb{R}$, $fl(x)$ is the floating point representation; $x - fl(x)$ is the rounding error

Rounding rules



- *chop* = round toward zero: truncate the base–β representation after $p-1$st digit
- *round to nearest*: $fl(x)$ is the closest machine number to x

Machine precision

- machine precision, ϵ_{mach}
    - with chopping: $\epsilon_{\text{mach}} = \beta^{1-p}$
    - with rounding to nearest: $\epsilon_{\text{mach}} = \frac{1}{2}\beta^{1-p}$
- called also *unit roundoff*: the smallest number ϵ such that $fl(1+\epsilon) > 1$
- maximum relative error of representation

$$\left|\frac{fl(x) - x}{x}\right| \leq \epsilon_{\text{mach}}$$

- usually $0 < UFL < \epsilon_{\text{mach}} < OFL$

Machine precision - example

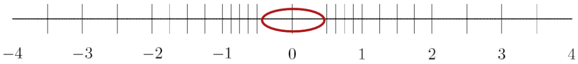
For $\beta = 2, p = 3, L = -1, U = 1$,

- $\epsilon_{\text{mach}} = (0.01)_2 = (0.25)_{10}$ with chopping
- $\epsilon_{\text{mach}} = (0.001)_2 = (0.125)_{10}$ with rounding to nearest

The usual case (IEEE fp systems):

- $\epsilon_{\text{mach}} = 2^{-24} \approx 10^{-7}$ in single precision
- $\epsilon_{\text{mach}} = 2^{-53} \approx 10^{-16}$ in double precision
- $\rightarrow$ about 7 and 16 decimals of precision, respectively
- (in R: p-value $< 2.2e - 16$)

Gradual underflow



- to improve representation of numbers around 0 - use subnormal (or denormalized) numbers
- when exponent is at minimum, alow leading digits to be 0
- subnormals are less precise
- → gradual underflow

Special values

IEEE standard:

- Inf: infinity; the result of $1/0$
- NaN: the result of $0/0$ or Inf/Inf
- special representation of the exponent field

Floating-point arithmetic

- *addition/subtraction*: denormalization might be required: $3.52 \times 10^3 + 1.97 \times 10^5 = 0.0352 \times 10^5 + 1.97 \times 10^5 = 2.0052 \times 10^5$ $\rightarrow$ might cause loss of some digits
- *multiplication/division*: the result may not be representable
- overflow is more serious than underflow: how to approximate large numbers?
- for underflow, the result may be approximated by 0
- in FP arithm. addition and multiplication are commutative but *not* associative: if ϵ is slightly smaller than ϵ_{mach}, then $(1+\epsilon)+\epsilon = 1$, but $1 + (\epsilon + \epsilon) > 1$
- ideally, $x \text{ flop } y = fl(x \ op \ y)$; IEEE standard ensures this for within range results

Example: divergent series

$$\sum_{n=1}^{\infty} \frac{1}{n}$$

- in FP arithm, the sum of the series is finite;
- depending on the system, this is because:
  - after a while, the sum overflows
  - $1/n$ underflows
  - for all n such that
$$\frac{1}{n} < \epsilon_{\text{mach}} \sum_{k=1}^{n-1} \frac{1}{k}$$
    the sum does not change anymore

Cancellation

- subtracting 2 numbers of the same magnitude usually cancels the *most significant* digits:
  $1.92403 \times 10^2 - 1.92275 \times 10^2 = 1.28000 \times 10^{-1} \rightarrow$ only 3 significant digits
- let $\epsilon > 0$ be slightly smaller than ϵ_{mach}, then $(1 + \epsilon) - (1 - \epsilon)$ yields 0 in FP arithmetic, instead of 2ϵ.

Cancellation - example

For the quadratic equation, $ax^2 + bx + c = 0$, the two solutions are given by

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Problems:

- for very large/small coefficients, the terms b^2 or $4ac$ may over-/underflow $\rightarrow$ rescale coeficients by $\max\{a, b, c\}$.
- cancellation between $-b$ and $\sqrt{\cdot}$ can be avoided by computing one root using $x = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}}$

Exercise: let $x_1 = 2000, x_2 = 0.05$ be the roots of a quadratic equation. Compute the coefficients and then use the above formulas to retrieve the roots. Try `numpy.roots()` function in Python.

Cancellation - another example

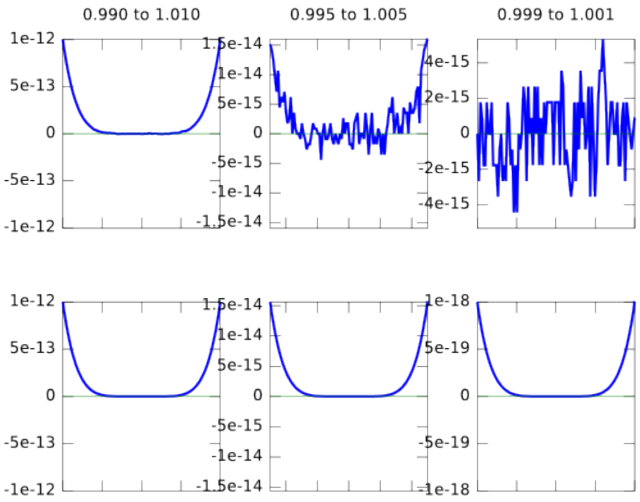
$P(X) = (X-1)^6 = X^6 - 6X^5 + 15X^4 - 20X^3 + 15X^2 - 6X + 1$. What happens around $X = 1$?

```
import matplotlib.pyplot as plt
import numpy as np

epsilon = [0.01, 0.005, 0.001]
for k in range(3):
    x = np.linspace(1 - epsilon[k], 1 + epsilon[k], 100)
    px = x**6 - 6*x**5 + 15*x**4 - 20*x**3 + 15*x**2 - 6*x + 1
    px0 = (x - 1)**6
    plt.subplot(2, 3, k+1)
    plt.plot(x, px, '-b', x, np.zeros(100), '-r')
    plt.axis([1 - epsilon[k], 1 + epsilon[k], -max(abs(px)),
                                  max(abs(px))])
    plt.subplot(2, 3, k+4)
    plt.plot(x, px0, '-b', x, np.zeros(100), '-r')
    plt.axis([1 - epsilon[k], 1 + epsilon[k], -max(abs(px0)),
                                  max(abs(px0))])

plt.show()
```

...mathematically equivalent, but numerically different...



(very small) Project

Study the paper

Moler, C., Morisson, D., *Replacing square roots by Pythagorean sums*. IBM J. Res. Develop. 27(6), 1983

Then, implement the proposed method and compare it with the naive `sqrt()`-based approach.

In Python...

- basic (and not only) numerical functions are in `numpy` package
- ϵ_{mach} is returned by
  - single precision: `np.finfo(np.float32).eps` gives $1.1920929e-07 = 2^{-23}$
  - double precision: `np.finfo(np.float64).eps` gives $2.220446049250313e-16 = 2^{-52}$
- to obtain the smallest or largest single/double precision numbers, use `np.finfo(np.float32).min`, `np.finfo(np.float32).max`, `np.finfo(np.float64).min`, `np.finfo(np.float64).max`
- you have the special constants `np.Inf` and `np.NaN`

Questions?