CS 450 – Numerical Analysis

Chapter 1: Scientific Computing †

Prof. Michael T. Heath

Department of Computer Science
University of Illinois at Urbana-Champaign
heath@illinois.edu

January 28, 2019

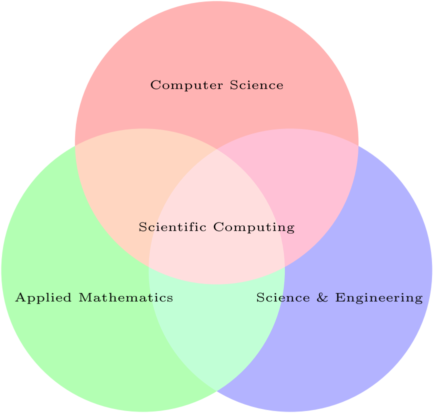
---

†Lecture slides based on the textbook *Scientific Computing: An Introductory Survey* by Michael T. Heath, copyright © 2018 by the Society for Industrial and Applied Mathematics. http://www.siam.org/books/cl80

Scientific Computing

What Is Scientific Computing?

- Design and analysis of algorithms for solving mathematical problems arising in science and engineering numerically



- Also called *numerical analysis* or *computational mathematics*

Scientific Computing, continued

- Distinguishing features of *scientific* computing
  - Deals with *continuous* quantities (e.g., time, distance, velocity, temperature, density, pressure) typically measured by real numbers
  - Considers effects of *approximations*
- Why *scientific computing*?
  - Predictive simulation of natural phenomena
  - Virtual prototyping of engineering designs
  - Analyzing data

Numerical Analysis → Scientific Computing

- Pre-computer era (before ∼1940)
  - Foundations and basic methods established by Newton, Euler, Lagrange, Gauss, and many other mathematicians, scientists, and engineers
- Pre-integrated circuit era (∼1940-1970): *Numerical Analysis*
  - Programming languages developed for scientific applications
  - Numerical methods formalized in computer algorithms and software
  - Floating-point arithmetic developed
- Integrated circuit era (since ∼1970): *Scientific Computing*
  - Application problem sizes explode as computing capacity grows exponentially
  - Computation becomes an essential component of modern scientific research and engineering practice, along with theory and experiment

Mathematical Problems

- Given mathematical relationship $y = f(x)$, typical problems include
  - Evaluate a function: compute output y for given input x
  - Solve an equation: find input x that produces given output y
  - Optimize: find x that yields extreme value of y over given domain
- Specific type of problem and best approach to solving it depend on whether variables and function involved are
  - discrete or continuous
  - linear or nonlinear
  - finite or infinite dimensional
  - purely algebraic or involve derivatives or integrals

General Problem-Solving Strategy

- Replace difficult problem by easier one having same or closely related solution
  - infinite dimensional $\rightarrow$ finite dimensional
  - differential $\rightarrow$ algebraic
  - nonlinear $\rightarrow$ linear
  - complicated $\rightarrow$ simple
- Solution obtained may only *approximate* that of original problem
- Our goal is to estimate accuracy and ensure that it suffices

Approximations

Approximations

I've learned that, in the description of Nature, one has to tolerate approximations, and that work with approximations can be interesting and can sometimes be beautiful.

— P. A. M. Dirac

Sources of Approximation

- *Before* computation
  - modeling
  - empirical measurements
  - previous computations
- *During* computation
  - truncation or discretization (mathematical approximations)
  - rounding (arithmetic approximations)
- Accuracy of final result reflects all of these
- Uncertainty in input may be amplified by problem
- Perturbations during computation may be amplified by algorithm

Example: Approximations

- Computing surface area of Earth using formula $A = 4\pi r^2$ involves several approximations
  - Earth is modeled as a sphere, idealizing its true shape
  - Value for radius is based on empirical measurements and previous computations
  - Value for π requires truncating infinite process
  - Values for input data and results of arithmetic operations are rounded by calculator or computer

Absolute Error and Relative Error

- *Absolute error*: approximate value − true value
- *Relative error*: $\dfrac{\text{absolute error}}{\text{true value}}$
- Equivalently, approx value = (true value) × (1 + rel error)
- Relative error can also be expressed as percentage

$$\text{per cent error} = \text{relative error} \times 100$$

- True value is usually unknown, so we *estimate* or *bound* error rather than compute it exactly
- Relative error often taken relative to approximate value, rather than (unknown) true value

Data Error and Computational Error

- Typical problem: evaluate function $f : \mathbb{R} \to \mathbb{R}$ for given argument
  - $x =$ true value of input
  - $f(x) =$ corresponding output value for true function
  - $\hat{x} =$ approximate (inexact) input actually used
  - $\hat{f} =$ approximate function actually computed

- Total error: $\hat{f}(\hat{x}) - f(x) =$

$$\hat{f}(\hat{x}) - f(\hat{x}) \quad + \quad f(\hat{x}) - f(x)$$

computational error + propagated data error

- Algorithm has no effect on propagated data error

Example: Data Error and Computational Error

- Suppose we need a "quick and dirty" approximation to $\sin(\pi/8)$ that we can compute without a calculator or computer
- Instead of true input $x = \pi/8$, we use $\hat{x} = 3/8$
- Instead of true function $f(x) = \sin(x)$, we use first term of Taylor series for $\sin(x)$, so that $\hat{f}(x) = x$
- We obtain approximate result $\hat{y} = 3/8 = 0.3750$
- To four digits, true result is $y = \sin(\pi/8) = 0.3827$
- Computational error:
  $\hat{f}(\hat{x}) - f(\hat{x}) = 3/8 - \sin(3/8) \approx 0.3750 - 0.3663 = 0.0087$
- Propagated data error:
  $f(\hat{x}) - f(x) = \sin(3/8) - \sin(\pi/8) \approx 0.3663 - 0.3827 = -0.0164$
- Total error: $\hat{f}(\hat{x}) - f(x) \approx 0.3750 - 0.3827 = -0.0077$

Truncation Error and Rounding Error

- *Truncation error*: difference between true result (for actual input) and result produced by given algorithm using exact arithmetic
  - Due to mathematical approximations such as truncating infinite series, discrete approximation of derivatives or integrals, or terminating iterative sequence before convergence
- *Rounding error*: difference between result produced by given algorithm using exact arithmetic and result produced by same algorithm using limited precision arithmetic
  - Due to inexact representation of real numbers and arithmetic operations upon them
- Computational error is sum of truncation error and rounding error
  - One of these usually dominates

⟨ interactive example ⟩

Example: Finite Difference Approximation

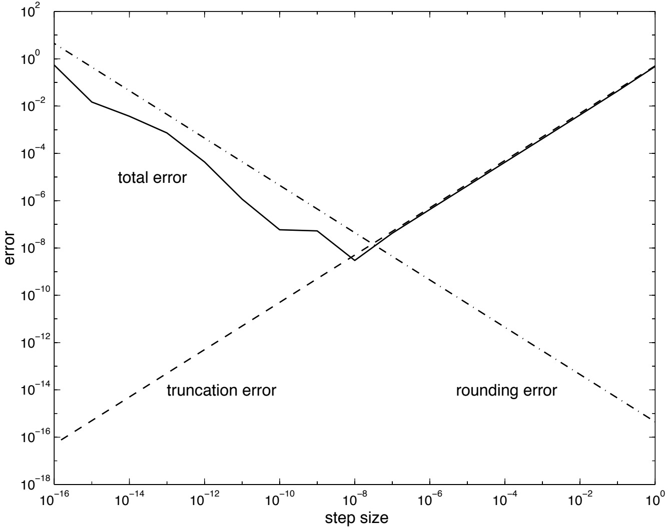
- Error in finite difference approximation

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

  exhibits tradeoff between rounding error and truncation error

- Truncation error bounded by $Mh/2$, where M bounds $|f''(t)|$ for t near x
- Rounding error bounded by $2\epsilon/h$, where error in function values bounded by ϵ
- Total error minimized when $h \approx 2\sqrt{\epsilon/M}$
- Error increases for smaller h because of rounding error and increases for larger h because of truncation error

Example: Finite Difference Approximation



Forward and Backward Error

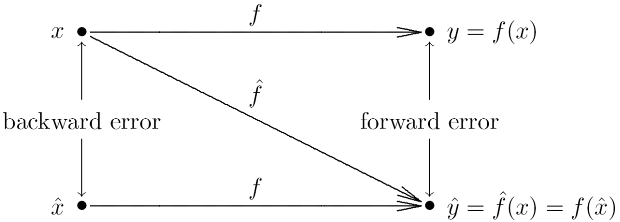
Forward and Backward Error

- Suppose we want to compute $y = f(x)$, where $f : \mathbb{R} \to \mathbb{R}$, but obtain approximate value $\hat{y}$
- *Forward error*: Difference between computed result $\hat{y}$ and true output y,

$$\Delta y = \hat{y} - y$$

- *Backward error*: Difference between actual input x and input $\hat{x}$ for which computed result $\hat{y}$ is exactly correct (i.e., $f(\hat{x}) = \hat{y}$),

$$\Delta x = \hat{x} - x$$



Example: Forward and Backward Error

- As approximation to $y = \sqrt{2}$, $\hat{y} = 1.4$ has absolute forward error

$$|\Delta y| = |\hat{y} - y| = |1.4 - 1.41421\ldots| \approx 0.0142$$

  or relative forward error of about 1 percent

- Since $\sqrt{1.96} = 1.4$, absolute backward error is

$$|\Delta x| = |\hat{x} - x| = |1.96 - 2| = 0.04$$

  or relative backward error of 2 percent

- Ratio of relative forward error to relative backward error is so important we will shortly give it a name

Backward Error Analysis

- Idea: approximate solution is exact solution to modified problem
- How much must original problem change to give result actually obtained?
- How much data error in input would explain *all* error in computed result?
- Approximate solution is good if it is exact solution to *nearby* problem
- If backward error is smaller than uncertainty in input, then approximate solution is as accurate as problem warrants
- Backward error analysis is useful because backward error is often easier to estimate than forward error

Example: Backward Error Analysis

- Approximating cosine function $f(x) = \cos(x)$ by truncating Taylor series after two terms gives

$$\hat{y} = \hat{f}(x) = 1 - x^2/2$$

- Forward error is given by

$$\Delta y = \hat{y} - y = \hat{f}(x) - f(x) = 1 - x^2/2 - \cos(x)$$

- To determine backward error, need value $\hat{x}$ such that $f(\hat{x}) = \hat{f}(x)$
- For cosine function, $\hat{x} = \arccos(\hat{f}(x)) = \arccos(\hat{y})$

Example, continued

- For $x = 1$,

$$\begin{aligned} y &= f(1) = \cos(1) \approx 0.5403 \\ \hat{y} &= \hat{f}(1) = 1 - 1^2/2 = 0.5 \\ \hat{x} &= \arccos(\hat{y}) = \arccos(0.5) \approx 1.0472 \end{aligned}$$

- Forward error: $\Delta y = \hat{y} - y \approx 0.5 - 0.5403 = -0.0403$
- Backward error: $\Delta x = \hat{x} - x \approx 1.0472 - 1 = 0.0472$

Conditioning, Stability, and Accuracy

Well-Posed Problems

- Mathematical problem is *well-posed* if solution
  - exists
  - is unique
  - depends continuously on problem data

  Otherwise, problem is *ill-posed*

- Even if problem is well-posed, solution may still be *sensitive* to perturbations in input data

- *Stablity*: Computational algorithm should not make sensitivity worse

Sensitivity and Conditioning

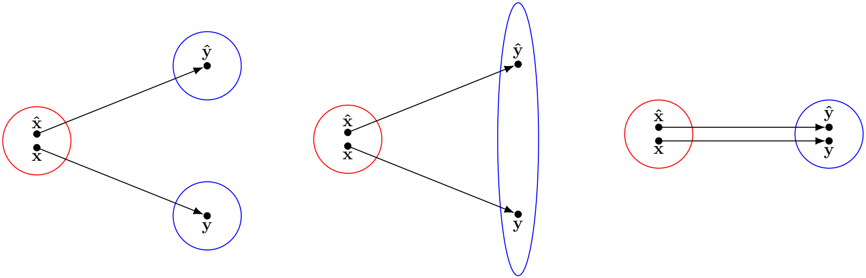
- Problem is *insensitive*, or *well-conditioned*, if relative change in input causes similar relative change in solution
- Problem is *sensitive*, or *ill-conditioned*, if relative change in solution can be much larger than that in input data
- *Condition number*:

$$\text{cond} = \frac{|\text{relative change in solution}|}{|\text{relative change in input data}|}$$

$$= \frac{|[f(\hat{x}) - f(x)]/f(x)|}{|(\hat{x} - x)/x|} = \frac{|\Delta y/y|}{|\Delta x/x|}$$

- Problem is sensitive, or ill-conditioned, if cond $\gg 1$

Sensitivity and Conditioning



Ill-Posed

Ill-Conditioned

Well-Conditioned

Condition Number

- Condition number is *amplification factor* relating relative forward error to relative backward error

$$\left|\begin{array}{c}\text{relative}\\ \text{forward error}\end{array}\right| = \text{cond} \times \left|\begin{array}{c}\text{relative}\\ \text{backward error}\end{array}\right|$$

- Condition number usually is not known exactly and may vary with input, so rough estimate or upper bound is used for cond, yielding

$$\left|\begin{array}{c}\text{relative}\\ \text{forward error}\end{array}\right| \lessapprox \text{cond} \times \left|\begin{array}{c}\text{relative}\\ \text{backward error}\end{array}\right|$$

Example: Evaluating a Function

- Evaluating function f for approximate input $\hat{x} = x + \Delta x$ instead of true input x gives

$$\text{Absolute forward error:} \quad f(x + \Delta x) - f(x) \approx f'(x)\Delta x$$

$$\text{Relative forward error:} \quad \frac{f(x + \Delta x) - f(x)}{f(x)} \approx \frac{f'(x)\Delta x}{f(x)}$$

$$\text{Condition number:} \quad \text{cond} \approx \left| \frac{f'(x)\Delta x / f(x)}{\Delta x / x} \right| = \left| \frac{x\, f'(x)}{f(x)} \right|$$

- Relative error in function value can be much larger or smaller than that in input, depending on particular f and x
- Note that $\text{cond}(f^{-1}) = 1/\text{cond}(f)$

Example: Condition Number

- Consider $f(x) = \sqrt{x}$
- Since $f'(x) = 1/(2\sqrt{x}\,)$,

$$\text{cond} \approx \left| \frac{x\, f'(x)}{f(x)} \right| = \left| \frac{x/(2\sqrt{x}\,)}{\sqrt{x}} \right| = \frac{1}{2}$$

- So forward error is about half backward error, consistent with our previous example with $\sqrt{2}$
- Similarly, for $f(x) = x^2$,

$$\text{cond} \approx \left| \frac{x\, f'(x)}{f(x)} \right| = \left| \frac{x\,(2x)}{x^2} \right| = 2$$

  which is reciprocal of that for square root, as expected
- Square and square root are both relatively well-conditioned

Example: Sensitivity

- Tangent function is sensitive for arguments near $\pi/2$
  - $\tan(1.57079) \approx 1.58058 \times 10^5$
  - $\tan(1.57078) \approx 6.12490 \times 10^4$
- Relative change in output is a quarter million times greater than relative change in input
  - For $x = 1.57079$, cond $\approx 2.48275 \times 10^5$

Stability

- Algorithm is *stable* if result produced is relatively insensitive to perturbations *during* computation

- Stability of algorithms is analogous to conditioning of problems

- From point of view of backward error analysis, algorithm is stable if result produced is exact solution to nearby problem

- For stable algorithm, effect of computational error is no worse than effect of small data error in input

Accuracy

- *Accuracy*: closeness of computed solution to true solution (i.e., relative forward error)
- Stability alone does not guarantee accurate results
- Accuracy depends on conditioning of problem as well as stability of algorithm
- Inaccuracy can result from
  - applying stable algorithm to ill-conditioned problem
  - applying unstable algorithm to well-conditioned problem
  - applying unstable algorithm to ill-conditioned problem (yikes!)
- Applying stable algorithm to well-conditioned problem yields accurate solution

Summary – Error Analysis

- Scientific computing involves various types of approximations that affect accuracy of results
- Conditioning: Does problem amplify uncertainty in input?
- Stability: Does algorithm amplify computational errors?
- Accuracy of computed result depends on both conditioning of problem and stability of algorithm
- Stable algorithm applied to well-conditioned problem yields accurate solition

Floating-Point Numbers

Floating-Point Numbers

- Similar to *scientific notation*
- Floating-point number system characterized by four integers

β	base or radix
p	precision
$[L, U]$	exponent range

- Real number x is represented as

$$x = \pm \left(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \cdots + \frac{d_{p-1}}{\beta^{p-1}} \right) \beta^E$$

where $0 \leq d_i \leq \beta - 1,\ i = 0, \ldots, p-1,$ and $L \leq E \leq U$

Floating-Point Numbers, continued

- Portions of floating-poing number designated as follows
  - *exponent*: E
  - *mantissa*: $d_0 d_1 \cdots d_{p-1}$
  - *fraction*: $d_1 d_2 \cdots d_{p-1}$
- Sign, exponent, and mantissa are stored in separate fixed-width *fields* of each floating-point *word*

Typical Floating-Point Systems

Parameters for typical floating-point systems

system	β	p	L	U
IEEE HP	2	11	-14	15
IEEE SP	2	24	-126	127
IEEE DP	2	53	-1022	1023
IEEE QP	2	113	-16382	16383
Cray-1	2	48	-16383	16384
HP calculator	10	12	-499	499
IBM mainframe	16	6	-64	63

- Modern computers use binary ($\beta = 2$) arithmetic
- IEEE floating-point systems are now almost universal in digital computers

Normalization

- Floating-point system is *normalized* if leading digit d_0 is always nonzero unless number represented is zero
- In normalized system, mantissa m of nonzero floating-point number always satisfies $1 \leq m < \beta$
- Reasons for normalization
  - representation of each number unique
  - no digits wasted on leading zeros
  - leading bit need not be stored (in binary system)

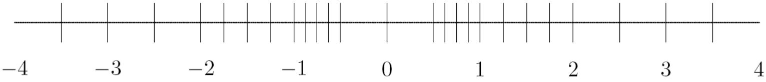
Properties of Floating-Point Systems

- Floating-point number system is finite and discrete
- Total number of normalized floating-point numbers is

$$\boxed{2(\beta - 1)\beta^{p-1}(U - L + 1) + 1}$$

- Smallest positive normalized number: $\boxed{\mathrm{UFL} = \beta^{L}}$
- Largest floating-point number: $\boxed{\mathrm{OFL} = \beta^{U+1}(1 - \beta^{-p})}$
- Floating-point numbers equally spaced only between successive powers of β
- Not all real numbers exactly representable; those that are are called *machine numbers*

Example: Floating-Point System



- Tick marks indicate all 25 numbers in floating-point system having $\beta = 2$, $p = 3$, $L = -1$, and $U = 1$
  - $\mathrm{OFL} = (1.11)_2 \times 2^1 = (3.5)_{10}$
  - $\mathrm{UFL} = (1.00)_2 \times 2^{-1} = (0.5)_{10}$

- At sufficiently high magnification, all normalized floating-point systems look grainy and unequally spaced

⟨ interactive example ⟩

Rounding Rules

- If real number x is not exactly representable, then it is approximated by "nearby" floating-point number $\mathrm{fl}(x)$
- This process is called *rounding*, and error introduced is called *rounding error*
- Two commonly used rounding rules
  - *chop*: truncate base-β expansion of x after $(p-1)$st digit; also called *round toward zero*
  - *round to nearest*: $\mathrm{fl}(x)$ is nearest floating-point number to x, using floating-point number whose last stored digit is even in case of tie; also called *round to even*
- Round to nearest is most accurate, and is default rounding rule in IEEE systems

⟨ interactive example ⟩

Machine Precision

- Accuracy of floating-point system characterized by *unit roundoff* (or *machine precision* or *machine epsilon*) denoted by ϵ_{mach}
  - With rounding by chopping, $\epsilon_{\text{mach}} = \beta^{1-p}$
  - With rounding to nearest, $\epsilon_{\text{mach}} = \frac{1}{2}\beta^{1-p}$
- Alternative definition is smallest number ϵ such that $\text{fl}(1+\epsilon) > 1$
- Maximum relative error in representing real number x within range of floating-point system is given by

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

Machine Precision, continued

- For toy system illustrated earlier
  - $\epsilon_{\text{mach}} = (0.01)_2 = (0.25)_{10}$ with rounding by chopping
  - $\epsilon_{\text{mach}} = (0.001)_2 = (0.125)_{10}$ with rounding to nearest
- For IEEE floating-point systems
  - $\epsilon_{\text{mach}} = 2^{-24} \approx 10^{-7}$ in single precision
  - $\epsilon_{\text{mach}} = 2^{-53} \approx 10^{-16}$ in double precision
  - $\epsilon_{\text{mach}} = 2^{-113} \approx 10^{-36}$ in quadruple precision
- So IEEE single, double, and quadruple precision systems have about 7, 16, and 36 decimal digits of precision, respectively

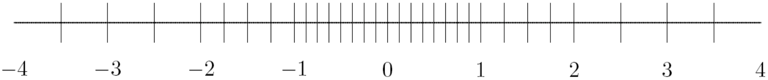
Machine Precision, continued

- Though both are "small," unit roundoff ϵ_{mach} should not be confused with underflow level UFL
  - ϵ_{mach} determined by number of digits in *mantissa*
  - UFL determined by number of digits in *exponent*
- In *practical* floating-point systems,

$$\boxed{0 < \mathrm{UFL} < \epsilon_{\mathrm{mach}} < \mathrm{OFL}}$$

Subnormals and Gradual Underflow

- Normalization causes gap around zero in floating-point system
- If leading digits are allowed to be zero, but only when exponent is at its minimum value, then gap is "filled in" by additional *subnormal* or *denormalized* floating-point numbers



- Subnormals extend range of magnitudes representable, but have less precision than normalized numbers, and unit roundoff is no smaller
- Augmented system exhibits *gradual underflow*

Exceptional Values

- IEEE floating-point standard provides special values to indicate two exceptional situations
  - `Inf`, which stands for "infinity," results from dividing a finite number by zero, such as $1/0$
  - `NaN`, which stands for "not a number," results from undefined or indeterminate operations such as $0/0$, $0 * \texttt{Inf}$, or $\texttt{Inf}/\texttt{Inf}$
- `Inf` and `NaN` are implemented in IEEE arithmetic through special reserved values of exponent field

Floating-Point Arithmetic

Floating-Point Arithmetic

- *Addition or subtraction*: Shifting mantissa to make exponents match may cause loss of some digits of smaller number, possibly all of them
- *Multiplication*: Product of two p-digit mantissas contains up to $2p$ digits, so result may not be representable
- *Division*: Quotient of two p-digit mantissas may contain more than p digits, such as nonterminating binary expansion of $1/10$
- Result of floating-point arithmetic operation may differ from result of corresponding real arithmetic operation on same operands

Example: Floating-Point Arithmetic

- Assume $\beta = 10$, $p = 6$
- Let $x = 1.92403 \times 10^2$, $y = 6.35782 \times 10^{-1}$
- Floating-point addition gives $x + y = 1.93039 \times 10^2$, assuming rounding to nearest
- Last two digits of y do not affect result, and with even smaller exponent, y could have had no effect on result
- Floating-point multiplication gives $x * y = 1.22326 \times 10^2$, which discards half of digits of true product

Floating-Point Arithmetic, continued

- Real result may also fail to be representable because its exponent is beyond available range
- Overflow is usually more serious than underflow because there is *no* good approximation to arbitrarily large magnitudes in floating-point system, whereas zero is often reasonable approximation for arbitrarily small magnitudes
- On many computer systems overflow is fatal, but an underflow may be silently set to zero

Example: Summing a Series

- Infinite series

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

is divergent, yet has finite sum in floating-point arithmetic

- Possible explanations
  - Partial sum eventually overflows
  - $1/n$ eventually underflows
  - Partial sum ceases to change once $1/n$ becomes negligible relative to partial sum

$$\frac{1}{n} < \epsilon_{\text{mach}} \sum_{k=1}^{n-1} \frac{1}{k}$$

⟨ interactive example ⟩

Floating-Point Arithmetic, continued

- Ideally, x `flop` $y = \mathrm{fl}(x \texttt{ op } y)$, i.e., floating-point arithmetic operations produce correctly rounded results
- Computers satisfying IEEE floating-point standard achieve this ideal provided x `op` y is within range of floating-point system
- But some familiar laws of real arithmetic not necessarily valid in floating-point system
- Floating-point addition and multiplication are commutative but *not* associative
- Example: if ϵ is positive floating-point number slightly smaller than ϵ_{mach}, then $(1+\epsilon)+\epsilon = 1$, but $1+(\epsilon+\epsilon) > 1$

Cancellation

- Subtraction between two p-digit numbers having same sign and similar magnitudes yields result with *fewer* than p digits, so it is usually exactly representable
- Reason is that leading digits of two numbers *cancel* (i.e., their difference is zero)
- For example,

$$1.92403 \times 10^2 - 1.92275 \times 10^2 = 1.28000 \times 10^{-1}$$

which is correct, and exactly representable, but has only three significant digits

Cancellation, continued

- Despite exactness of result, cancellation often implies serious loss of information

- Operands are often uncertain due to rounding or other previous errors, so relative uncertainty in difference may be large

- Example: if ϵ is positive floating-point number slightly smaller than ϵ_{mach}, then
$$(1+\epsilon)-(1-\epsilon)=1-1=0$$
in floating-point arithmetic, which is correct for actual operands of final subtraction, but true result of overall computation, 2ϵ, has been completely lost

- Subtraction itself is not at fault: it merely signals loss of information that had already occurred

Cancellation, continued

- Digits lost to cancellation are *most* significant, *leading* digits, whereas digits lost in rounding are *least* significant, *trailing* digits
- Because of this effect, it is generally bad to compute any small quantity as difference of large quantities, since rounding error is likely to dominate result
- For example, summing alternating series, such as

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

  for $x < 0$, may give disastrous results due to catastrophic cancellation

Example: Cancellation

Total energy of helium atom is sum of kinetic and potential energies, which are computed separately and have opposite signs, so suffer cancellation

Year	Kinetic	Potential	Total
1971	13.0	−14.0	−1.0
1977	12.76	−14.02	−1.26
1980	12.22	−14.35	−2.13
1985	12.28	−14.65	−2.37
1988	12.40	−14.84	−2.44

Although computed values for kinetic and potential energies changed by only 6% or less, resulting estimate for total energy changed by 144%

Example: Quadratic Formula

- Two solutions of quadratic equation $ax^2 + bx + c = 0$ are given by

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

- Naive use of formula can suffer overflow, or underflow, or severe cancellation
- Rescaling coefficients avoids overflow or harmful underflow
- Cancellation between $-b$ and square root can be avoided by computing one root using alternative formula

$$x = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}}$$

- Cancellation inside square root cannot be easily avoided without using higher precision

⟨ interactive example ⟩

Example: Standard Deviation

- Mean and standard deviation of sequence x_i, $i = 1, \dots, n$, are given by

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \quad \text{and} \quad \sigma = \left[\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2\right]^{\frac{1}{2}}$$

- Mathematically equivalent formula

$$\sigma = \left[\frac{1}{n-1}\left(\sum_{i=1}^{n} x_i^2 - n\bar{x}^2\right)\right]^{\frac{1}{2}}$$

avoids making two passes through data

- Single cancellation at end of one-pass formula is more damaging numerically than all cancellations in two-pass formula combined

Summary – Floating-Point Arithmetic

- On computers, infinite continuum of real numbers is approximated by finite and discrete *floating-point* number system, with *sign*, *exponent*, and *mantissa* fields within each floating-point *word*
- *Exponent* field determines range of representable *magnitudes*, characterized by underflow and overflow levels
- *Mantissa* field determines *precision*, and hence *relative accuracy*, of floating-point approximation, characterized by unit roundoff ϵ_{mach}
- *Rounding error* is loss of least significant, trailing digits when approximating true real number by nearby floating-point number
- More insidiously, *cancellation* is loss of most significant, leading digits when numbers of similar magnitude are subtracted, resulting in *fewer* significant digits in finite precision